State Checkpointing in Main-Memory Database Systems
Matthew Antoszyk

Abstract

In the modern era, databases are a critical and inescapable component of commerce. While most businesses must guarantee the durability of the data they collect, the rapid internationalization of the world’s economies often means that there is never a convenient time to quiesce the database in order to take a checkpoint of the state. We also notice a second trend in the database community: as the price of storage decreases, it has become increasingly common for entire databases to fit within main memory. We leverage this second trend to provide a solution to the first. This paper introduces Checkpointing Asynchronously Using Logical Consistency (CALC), a checkpointing algorithm for main memory resident systems that makes use of a virtual point of consistency to capture a checkpoint without quiescing the database. We present performance data for CALC in Calvin, and we compare our results to an implementation of the Naïve Wait-Free Ping-Pong (NPP) algorithm suggested by Cao et al.¹ Our results demonstrate CALC’s ability to take a relatively short checkpoint without introducing latency or significantly reducing throughput of the system.

1. Introduction

This paper concerns a checkpointing algorithm optimized for main memory database systems, the CALC algorithm, which was originally conceived in Professor Daniel Abadi’s lab at Yale University. CALC utilizes a 5-phase process to engineer a virtual point of consistency that requires at most two copies of each database record, the live version, which reflects the current state of the record, and the stable version, which reflects the state of the record that the checkpointing thread should see only if it differs from the live version.

Initially, the database system begins in the rest phase, during which all transactions write to each record’s live version. At some point, a checkpoint is signaled for, which causes the system to transition into the prepare phase. In the prepare phase, transactions write to the live versions of records but store the overwritten values in transaction local arrays. If transactions that started in the prepare phase commit during the prepare phase, no further course of action is taken. If they commit during the resolve phase, they write the contents of their array to the stable versions of the records that they modified.

The prepare phase ends when all transactions that began in the rest phase have committed, marking the virtual point of consistency. Only transactions that

commit before the virtual point of consistency have their updates reflected in the checkpoint. After the prepare phase, the resolve phase begins. Transactions that begin during the resolve phase write to live versions of records but copy the previous live values to the stable versions when stable versions do not already exist. Once all transactions that began in the prepare phase have ended, the capture phase begins. In the capture phase, a background thread writes to disk either the stable version of each record if it exists or the live version otherwise. Updates during the capture phase occur in the same manner as in the resolve phase. Finally, after the checkpoint has been taken, stable record values are pruned in the cleanup phase. Once all stable records are pruned, CALC returns to the rest phase.

2. Implementation

We implemented the CALC algorithm in Calvin, a deterministic database system with a key-value store. Calvin provides an interface to implement key-value stores that requires four basic operations: put, delete, get, and exists. We found the interface provided by Calvin to be insufficient for the needs of the CALC algorithm. In particular, the put and delete functions for this interface only took a key and a value as parameters, whereas the CALC algorithm needs the transaction ID of the transaction making the update in order to appropriately update the live and stable versions of records. Furthermore, puts and deletes that occur in the prepare phase need a way of returning the previous live versions of updated records if they exist.

In order to create a key-value store that is capable of taking a checkpoint through the CALC algorithm, we created a new key-value store, which uses an unordered map. This CALC store associates each key with a calc-pair. A calc-pair contains a pair of values, the live and stable versions for a record as a pair for its first value and a boolean as its second. The boolean indicates to the checkpointing thread whether or not a particular record was inserted after the virtual point of consistency. This boolean value is true if a record was inserted at any point after the start of the resolve phase but before the end of the capture phase. All records have this boolean set to false and have their stable versions erased by a background thread during the cleanup phase.

A CALC checkpoint can be signaled for at any point. Once a checkpoint has been requested, the current phase transitions from rest to prepare. Then, the scheduler routinely calls a method to check if the requirements have been met to transition to the next phase in the checkpoint. If so, the method updates the phase.

At the beginning of each phase, the highest transaction ID such that no transaction with an ID greater than that value has begun is recorded. This value marks the initial high watermark for the phase. The prepare and resolve phases each end when the safe transaction ID, the highest transaction ID such that all transactions with lower IDs have committed, is greater than the initial high watermark for the phase—that is, when all transactions that began in the previous phase have completed.

3. Experimental Setup
The objective of our experiment was to measure the effect of CALC and NPP checkpoints on the throughput of a system running at maximum CPU usage. We obtain all of our results using a microbenchmark testing framework. In the microbenchmark tests, we initialize a database of two million records. Each record consists of a calc pair and a boolean. The values in the database are all single characters.

Each transaction performs a write on ten records. Nine of these records are randomly selected, and one record, called a hot record, is selected from the set of hot keys that contains 10,000 keys. There can be at most 2,000 actions in our system at one time, which means the maximum chance of lock contention on a hot record is 20%. However, during average execution there are no more than 300 active transactions so this percentage is typically much lower.

In order to see an effect of the checkpoint our transactions need to use a sufficient amount of CPU such that a checkpointing thread cannot complete using only spare CPU. To fine-tune the amount of CPU used by each transaction, we add code that performs arbitrary mathematical calculations in each transaction. This code performs its calculations repeatedly. We are able to scale the CPU intensity of the write by growing or shrinking the range of the for-loop that dictates how many times the computation is performed. The range of the for-loop is zero to \( l \). The experiments were run on a single EC2 node, of type c1.xlarge. The c1.xlarge AMI provides four cores, each with two hyper threaded virtual CPUs, for a total of eight vCPUs. The c1.xlarge uses a 64-bit processor architecture and has 20 ECUs.

Our database, Calvin, requires eight threads to run. We have a communicator thread, a lock manager thread, a sequencer reader thread, a sequencer writer thread, and four worker threads. The sequencer writer thread creates new transactions and the sequencer reader assigns them to a node. While Calvin is a distributed database system and is capable of running across several nodes, all of our experiments involve only one node, so the sequencer reader always assigns transactions to our single node. The communicator is responsible for passing messages between nodes. Again, since we have only one node this thread spends the duration of the experiment busy-waiting. It constantly looks for messages that it never finds. The lock manager receives transactions from the sequencer writer and orders them for receiving locks. It also is responsible for releasing the locks of completed transactions. The worker threads execute our write operations.

Because we have eight vCPUs and eight threads, we are able to create a one-to-one assignment of threads to vCPUs that is shown in figure 1. Figure 2 shows the arrangement of vCPUs. For both CALC and NPP we create a ninth thread to capture the checkpoint (and to do the cleanup, in the case of CALC), and this requires a decision of which of the eight vCPUs we ought to schedule it on. We discuss our choice and its implications in the results section below. Finally, in our experiments, we recorded data...
every quarter of a second and used this value to determine per second throughput.

4. Results

4.1 Expected Results

As we discussed above, in order to take a checkpoint we are required to create a ninth thread and assign it to one of the vCPUs in our system. We expect that adding a second thread to a vCPU should halve the resources available to the thread that was already executing. Using this knowledge, we are able to make predictions about the drop in throughput we should see when scheduling the checkpointing thread on each of the eight vCPUs. For example, we know that with four worker threads, each worker is responsible for one quarter of the throughput. If we halve the CPU resources available to one worker, giving the other half to the checkpointing thread, we should see at most a one eight decrease in overall throughput.

We expect to see a one half drop in throughput when we schedule the thread on the lock manager’s vCPU because we have designed our test so that in normal execution, before the checkpoint is signaled, the lock manager is the bottleneck in the system. That is, the workers are capable of executing about as many transactions as the lock manager can provide them. If we halve the resources of the lock manager, it provides half as many transaction to the workers and the throughput drops by half.

We expect to see no drop in throughput if we schedule the ninth thread on the sequencer reader, sequencer writer or the communicator (vCPUs, 2, 6, 3, respectively) because none of these threads are bottlenecks of the system. They should all be able to share their resources without affecting the lock manager, which is the current bottleneck, or the workers, which could be turned into the bottleneck if adversely affected.

We will only see these expected results if we can find a value of $I$ which will create a CPU intensive transactions. A transaction should cost enough CPU that our four workers are just barely able to complete all of the transactions passed to them by the lock manager. The lock manager should be the bottleneck, but we ought to be very close to a threshold at which the workers become the bottleneck. If we are far away from this threshold, we expect to see a less than one eight drop in throughput if we schedule the checkpoint thread on a worker. This is because some of the spare CPU can be given to the checkpointing thread without negatively impacting throughput. Our experiments show 1100 to be an appropriate value for $I$.

When we run our tests, however, we find three unexpected results. First, we notice that when we schedule the checkpoint thread on any of the worker threads (0, 1, 4, 5), in addition to the expected 50% drop of throughput from the shared worker, the other three workers drop their throughputs by 5%, resulting in a 16% reduction of overall throughput. Second, we notice that when the ninth thread is scheduled on the sequencer writer or the communicator (threads 6 and 3), the throughput drops by 50% during the cleanup phase. Third, we do not see this same drop in the cleanup phase when we schedule the checkpoint thread on the
sequencer writer (thread 2). Although we did not originally expect to see the drop, its absence is noteworthy given its presence in the cases of threads six and three.

All three of these anomalies can be explained by examining the hardware we are using. While understanding these strange occurrences is not strictly necessary to make a comparison between CALC and NPP, it is necessary in order to understand the system in which we are running our experiments. It would be irresponsible of us to try to make comparison without explaining how our algorithm interacts with the hardware, so before we make our comparison we take a look under the hood of our EC2 instance.

4.2 Explaining the Anomalies

It is important to note that one of the fundamental principles of Amazon’s EC2 is that it is built on commodity hardware. They build their instances so that hardware can be changed at any time without notice to the user. As such, it is impossible to guarantee the exact hardware configuration we are using for our experiment. Amazon guarantees that a c1.xlarge instance uses a processor from the Intel Xeon Family, which could include any of a number of processors. Nonetheless, Intel’s processors in the Xeon family that are capable of running four hyperthreaded cores share certain characteristics, and relying on these we are able to explain our three mysteries.

Figure 3 is a diagram of the Profusion Chipset Multi-Ported System Architecture, built with an Intel Xeon III microprocessor. This diagram shows the Dual-Ported Architecture of the Profusion Chipset.²

---

The Profusion chipset divides the processors into two groups of four and provides each its own access to memory through a controller. This dual-ported system was used in the Profusion and other early chipsets making use of the Intel Xeon III Processor.\textsuperscript{3,4} It is also used in the most recent Intel Xeon Processors\textsuperscript{5} so it is likely that whichever Intel Xeon Processor we are using has the same dual-ported architecture.

If we recall the layout of our processor from figure 2, we can see that in a dual ported architecture, the workers (0, 1, 4, 5) would share one memory port, and threads 2, 3, 6 and 7 would share the other. We suggest then, that when we add ninth thread to any of the workers, we are creating a competition for memory bandwidth that affects all of the workers.

While we have been thinking of the checkpointing thread as one ninth thread, in CALC, it is actually two consecutive threads, first the thread that takes the

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{Dual -ported Memory Architecture}
\end{figure}

checkpoint in the capture phase and, second, the cleanup thread in the cleanup phase. Each of these threads reads every record in the database, a process that requires a large amount of memory bandwidth. As a result, we see a uniform 5% decrease in throughput from the other workers, each of which must now wait extra time to receive its data.

Another issue that all multi-core systems must deal with is cache coherency. Each of the eight processes has its own cache, and therefore, possibly, its own copy of any record. The system needs a way to ensure both that any process that reads a record is reading the most up-to-date version and that any process that writes to a record has its result recognized as the most up-to-date version. Figure 4 adds another layer to the Profusion chipset picture by showing the Left and Right Cache Coherency Filers. These are used in what is called the Snooping Protocol to maintain cache coherency.

While the complexity of a snooping protocol depends on the number of levels of cache the particular hardware has, the basic concept is the same in all cases. In our simple case presented above, the Cache Coherency Filters sit above the individual caches from each processor. On the left side, the Coherency Filter sits above processors zero, one, two and three. The filter then keeps track of all address lines in those processor’s individual caches. If processor zero wants to use its copy of a record, the filter knows whether or not processor zero has the most up-to-date version. If it does, and a write occurs, the filter must also send a message across the bus to the right side filter to inform it that zero has written to that record. If the Right Side Coherency Filter finds a version of the record in one of its processors caches, it marks the record as invalidated. Now let us imagine that processor zero has invalidated a record in the cache of processor five. The next time processor five wants the record, through snooping, processor zero should notice this and automatically send processor five the record.6

---

The “snooping filter,” as the cache filter is also called, is valuable because it takes a memory bandwidth operation and moves some of the work to the CPU. When processor five wants to read an invalid record, it does not force a valid copy to be written to disc so that it can access it. Instead, the other snoop filter is listening and at the appropriate time it finds its own copy, performing a CPU operation, and sends it over the bus, bypassing main memory.\(^7\)

In our cases in which we create our ninth thread on processors three and six, the cleanup phase invalidates almost all records in the workers’ cache. Every record that has a stable version is overwritten. As a result, when all of our workers want to write, they are depending on processor three or six to send them the most updated copy. Three or six, however, are precisely the vCPUs with two threads and therefore with the fewest CPU resources. Because three and six are currently CPU constrained the snooping protocol works against them forcing the worker threads to wait until their valid data arrives, reducing the overall throughput.

We can see evidence of the worker threads waiting by checking the number of times a worker thread goes through its command structure. A worker thread is a while-loop that, in each pass, looks for an available transaction. If it finds one, it executes it. If we run the experiment with \(I\) equal to 1500 so that the worker threads are the bottleneck, we will see that that number of loops in which a transaction is executed drops during the cleanup phase. In Figure 5 below, we present one set of data captured in each phase. ‘Loops’ is the number of times the thread goes through the while loop, and ‘actionLoops’ is the number of times it goes through the while-loop and executes a transaction.

In the cleanup it is immediately apparent that the transactions are taking significantly longer as we see that the number of loops has decreased 54% from the rest phase. The transactions in the cleanup phase, therefore, are taking nearly twice as long. The numbers look the same for thread six, though they are not shown here.

<table>
<thead>
<tr>
<th></th>
<th>Loops</th>
<th>ActionLoops</th>
<th>% Action Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rest</td>
<td>1193</td>
<td>1193</td>
<td>100%</td>
</tr>
<tr>
<td>Capture</td>
<td>1112</td>
<td>1112</td>
<td>100%</td>
</tr>
<tr>
<td>Cleanup</td>
<td>548</td>
<td>548</td>
<td>100%</td>
</tr>
<tr>
<td>Rest</td>
<td>1180</td>
<td>1180</td>
<td>100%</td>
</tr>
</tbody>
</table>

*Figure 5: Worker Thread Control Loop Data with Checkpoint on Thread Three*

The immediate follow up question is why we see a different behavior when we schedule the checkpointer on thread two. The answer is that thread two has a large amount of spare CPU. Every other process in Calvin other than thread two always runs at 100% CPU. Even when they are doing no significant work they busy-wait. Thread two, though, never uses more than 30% CPU. Recall that thread two is the sequencer reader, which is assigning transactions to nodes. However since we only have one node this is a simple and non-CPU consuming task. vCPU two, even

---

\(^7\) U. Drepper.
with a second thread added to it, is able to snoop much more assiduously and the workers rarely have to wait for the data.

Keeping these system effects in mind, we decide to schedule our checkpoint on one of the worker threads, specifically thread zero. We accept the additional memory-bandwidth penalty for sharing with a worker thread because scheduling on a worker thread allows us to analyze directly the effect of CPU and I/O costs of a checkpoint on our systems ability to process transactions. If we schedule the ninth thread on any of the other processors we will see no effect or at most an indirect effect.

4.2 The NPP Algorithm

Here we give a brief, high-level overview of the NPP algorithm to provide context for our comparison. The NPP algorithm is designed to reduce latency spikes at the end of checkpointing by investing in extra main memory and work per update. NPP maintains three copies of the database state, named AS, Odd, and Even. AS always maintains the complete state. Updates are then made to AS and either Odd or Even. Let's suppose updates are being made to AS and Odd. When a checkpoint is signaled we then switch and begin making updates to Even. We can then write the contents of Odd out to disc. The next checkpoint proceeds in an identical manner with the roles of Odd and Even inverted.8

The advantage of NPP is that capturing the checkpoint is a simple task of flushing an array to out to disk. The checkpoint ought to be short as it requires a single pass over the array and writes out only the records that have been updated since the last checkpoint (using a dirty bit to keep track). It is simple to swap a single pointer and have writes occur on the opposite array in the meantime. There is no need for a cleanup phase to make a second pass through the database and perform extra work. NPP has two main disadvantages. First, it requires extra work per update to write new values to two parallel data structures, AS and Odd or Even. Second, the checkpoint can only be initiated at a physical point of consistency. When we switch from Odd to Even (or vice versa), we must be able to guarantee that all committed transactions' and only those transactions' work will be reflected in the checkpoint. The checkpoint can only be initiated at a moment when there are no active transactions in the system. A physical point of consistency is less disruptive than quiescing the database, since we need only a moment of consistency rather than remaining quiesced for the duration of the checkpoint. Since physical points of consistency are unlikely to occur in a database system, NPP must manufacture one before signaling a checkpoint. This will degrade performance of the system.

4.3 NPP vs. CALC

In our first test, we want to measure CALC against NPP by a few standards. We will measure the drop in throughput during the course of the checkpoint, the length of the checkpoint, and the total number of transactions lost over the duration

8 Cao et al.
as a result of taking the checkpoint. These three measures taken together give an accurate picture of the costs of the checkpoint: the effect of the disruption, the duration of the disruption and the cumulative cost of the disruption over the entire checkpoint.

In order to test NPP and CALC on even footing, we need \( I \) values for each that ensure that the lock manager is just barely the bottleneck. We have already determined that \( I \) equal to 1100 is appropriate for CALC. Our experiments show that an appropriate value for \( I \) in NPP is 800.

We expect a lower \( I \) value for NPP because its updates are more CPU intensive, and its does not need as much additional CPU load added to its transactions. Figure 6 shows the throughput results for the microbenchmark test with a CALC and IPP checkpoint scheduled on thread zero. In both experiments the checkpoint is called for after five seconds.

The resolve and prepare phases are shown as one line at five seconds since they are nearly instantaneous. The capture phase lasts 5.06 seconds and the cleanup phase lasts 3.49 seconds. The total duration of the CALC checkpoint is 8.55 seconds. The NPP checkpoint signals for a physical point of consistency at five seconds and we see a drop that lasts 0.25 seconds which is immediately followed by creation of the background checkpointing thread. The checkpoint takes 6.33 seconds to complete so the entire NPP checkpoint lasts 6.58 seconds.

CALC has an average throughput of 22,205 transactions per second during the rest phase. During the capture phase, CALC has an average throughput of 18,756 transactions per second and during the cleanup phase an average throughput of 18,993 transactions per second, a 15.5% and 14.5% drop respectively. The difference between throughput in cleanup and throughput in capture is not statistically significant. Theses percentages are greater than one-eighth because of the memory bandwidth constraint introduced by adding another thread to the bus connecting processors responsible for the workers.
Figure 6: Throughput for CALC and NPP Checkpoints at $I = 1100$ and $I = 800$

Before the checkpoint, NPP has an average throughput of 21,262. While the checkpoint is being taken, its throughput drops to an average of 16,323, a 23.8% drop from its initial throughput. In the ‘drop’ period during which we are preparing for a physical point of consistency by completing all active transactions, the average throughput is 689.0 transactions per second.

We define transactions lost as the difference in throughput from rest and during the checkpoint multiplied by the duration of the checkpoint. For CALC, we make this calculation twice, once for the capture phase and once for the cleanup phase, and we sum the two results. For NPP we also make this calculation twice, once for the ‘drop’ period and once for the checkpoint duration. CALC loses 28,657 transactions over the course of its checkpoint while NPP loses 47,136 transactions. Despite having a checkpoint that lasts over two seconds longer, 64% more transactions are lost during a NPP checkpoint than during a CALC checkpoint.

There are two reasons for this result. The first is that the NPP checkpoint includes one incredibly costly quarter of a second – the ‘drop’ period. Ten percent of all lost transactions are lost during a period that makes up only 3.8% of the total checkpoint. CALC, on the other hand, does not suffer this penalty since it needs only a virtual point of consistency, not a physical point of consistency. The second reason NPP loses more transactions than CALC is that NPP must make two writes for every put operation. While we have chosen $I$ values so that NPP and CALC start on even ground, NPP suffers twice as much as CALC from the memory bandwidth competition introduced by the ninth thread. On account of this double
penalty for writes during capture, NPP has a more significant drop in throughput during the checkpoint than CALC. This result demonstrates that another benefit of CALC over NPP is that it is less susceptible to degraded performance from memory bandwidth competition. Although this result is specific only to our particular system, in the general case CALC is able to maintain only a single data structure whereas NPP must maintain three, so this advantage should carry over across systems.

4.4a One Long Transaction

In this section we investigate the performance of CALC and NPP in systems with long transactions. Because NPP achieves a physical point of consistency by waiting for all transactions in the system to commit, the longer the transactions are the more costly it should become to achieve a physical point of consistency. We introduce long transactions into the system by forcing thread one to busy-wait for two seconds during every transaction. We then run the same experiment as before, the results of which are shown in figure 7.

![Throughput for CALC and NPP Checkpoint with One Long Transaction](image)

Figure 7: Throughput for CALC and NPP Checkpoint with One Long Transaction

The prepare phase lasts slightly more than one second now. This is because we signal for the prepare phase at second five. Having stared a long transaction at second zero, at second five there will be a long transaction halfway completed. Upon completion of this transaction we move to the resolve phase, but not before the
worker thread immediately starts on another long transaction. The resolve phase then must last two seconds, as it waits for this transaction to complete. The capture and cleanup phases take 4.78 and 3.48 seconds respectively, which is comparable to what we saw in our previous experiment.

CALC rest throughput averages 17,927 transactions per second. We expected to see this drop as we have reduced the throughput of worker zero to one half a transaction per second, cutting total throughput by one quarter. In the prepare phase, throughput drops to 14,508 transactions per second. This is because transaction in the prepare phase have to do the extra work of copying all of the values they overwrite to their transaction-local arrays.

The capture and cleanup phases have throughputs of 12,950 transactions per second, a 27.7% decrease from resting throughput. The decrease is greater in this experiment because there are only three workers responsible for the throughput. When we split one of those workers with the checkpointing thread we now can see a one sixth drop in throughput rather than a one eighth drop. The CALC checkpoint results in a total of 45,665 transactions lost.

NPP starts with a throughput of 14,730 transactions per second. This is again the result of dedicating one worker to long transactions. The ‘drop’ duration is now a full two seconds, however we reach zero transactions per second after 0.2 seconds, and for the remaining 1.8 seconds no transactions are completed. The checkpoint itself lasts only 5.69 seconds, nearly a second shorter than in the prior experiment. The checkpoint duration is reduced because, with reduced throughput, fewer records have been touched prior to taking the checkpoint. Since NPP only writes out records that have been updated since the last checkpoint, there are fewer writes to do during the checkpoint. The throughput during the checkpoint drops to 10,962 transactions per second. NPP causes a total loss of 48,884 transactions.

Surprisingly, CALC ties NPP in transactions lost when we introduce a long transaction into the system, and this result bears some additional explanation. NPP does suffer from the need to take a physical point of consistency. In fact, 53% of the transactions lost occurred in the 1.8 seconds spent waiting for the long transaction to finish. However, NPP loses far fewer records in the checkpointing phase due to the shortened length of the checkpoint.

This result encourages two avenues of further research. The first is additional testing in a system that allows us to add a fifth thread as a long transaction with hurting overall throughput. If NPP did not have the advantage of a faster checkpoint it would not perform as well. The second avenue is in developing the partial CALC or pCALC algorithm. This version of CALC is designed to, like NPP, only write out to disc those records that have been updated since the last checkpoint. This data is strong evidence that pCALC would be a valuable development of the algorithm.

Although CALC and NPP have similar numbers for transactions lost, CALC does have at least one significant advantage over NPP when we introduce long transactions. NPP has to introduce latency into the system in order to achieve the physical point of consistency. CALCs virtual point of consistency avoids this problem. Keeping in mind the motivation for CALC to provide checkpointing for
systems without quiescing the database for any duration, CALC is clearly a superior choice than NPP for this purpose in systems with long transactions.

4.4b Multiple Long Transactions

The pattern for multiple long transactions is identical to that of when there is one long transaction. CALC performs on par with NPP for all of the reasons mentioned above. However, there is one complication in dealing with multiple transactions that is worth mentioning. The simplest way to achieve a physical point of consistency is to allow the workers to complete all transactions currently in the system – those that have all of their locks and those that are waiting for locks. When we introduce multiple long transactions into the system, there is a scenario in which NPP is severely outperformed by CALC. If a long transaction $t1$ is executing and a second long transaction $t2$ is waiting for a lock held by $t1$ when the checkpoint is called, the following occurs. All other transactions in the system finish and we wait with zero throughput for $t1$ to complete. Once it commits, $t2$ is granted its lock and then begins. We wait for $t2$ to complete, still with zero throughput, before achieving a physical point of consistency and beginning the checkpoint. Depending on how a system has implemented its locking mechanism, achieving a physical point of consistency while avoiding this scenario may or may not be a trivial task.

5. Conclusion

Our results show that CALC can be a viable and effective checkpointing algorithm. In systems with short transactions, we were able to show that while CALC lasts slightly longer than the NPP algorithm, it suffers a 15% drop in throughput as opposed to a 24% drop in throughput. As a result, a CALC checkpoint is less costly in terms of transactions lost. Furthermore, we have shown that CALC is more robust to contention for memory bandwidth. When long transactions are present, NPP benefits from its partial checkpointing. For this reason, the pCALC optimization is worth developing, and pCALC will serve as an even more effective checkpointing algorithm than the algorithm implemented and tested here.