Cyclic Commit Protocol Specifications

Thomas L. Rodeheffer
Microsoft Research, Silicon Valley

September 11, 2008

Abstract

A novel *cyclic commit* protocol uses extra metadata fields to implement transactional semantics without needing an explicit commit record. We describe and present specifications of two versions of cyclic commit: Simple Cyclic Commit (SCC) and Back-Pointer Cyclic Commit (BPCC). The specifications are written in TLA+ and checked with the TLC model checker.

1 Introduction

The atomicity property offered by transactions has proven useful in building file systems and database systems that maintain consistency across crashes and reboots. For example, a file creation involves multiple write operations to update the metadata of the parent directory and the new file. Having a storage layer that provides a transactional API reduces the complexity of the higher level systems and improves overall reliability.

Existing systems typically implement transactions by logging intention records to record intended updates followed by logging a commit record to commit the transaction. Often processes for checkpointing and cleaning are needed to truncate the log and reorganize data into home pages for efficient access on a hard disk. However, modern solid-state drives implemented using flash memory provide a different performance tradeoff. A *cyclic commit* protocol [2] exploits the advantages of solid-state drives to allow an efficient implementation of transactions.

We describe and present specifications of two versions of cyclic commit: Simple Cyclic Commit (SCC) and Back-Pointer Cyclic Commit (BPCC). The specifications are written in TLA+ and checked with the TLC model checker [1].

The remainder of this paper is organized as follows. Section 2 gives an overview of cyclic commit protocols. Section 3 details the SCC protocol. Section 4 details the BPCC protocol. Listings of the full specifications are given in Appendices.

2 Overview of cyclic commit

Instead of using a separate commit record to determine whether a transaction is committed or not, a cyclic commit protocol uses per-page metadata to link a transaction's intention records into a cycle. In addition to the page identifier, version number, and new contents, each intention record also contains the page identifier and version number of the next intention in the cycle of intentions that comprise the transaction. The transaction is *committed* once all of the intention records are written. If committed, starting with any intention record, a cycle that contains all the intention records in the transaction can be found by following the next-links. Any intention belonging to an incomplete transaction is considered *uncommitted*.

We require that a page be involved in at most one current transaction at a time. For brevity, we often talk about versions of pages when what is meant is the intentions that represent those versions. There are intentions that are planned to be written, *surviving intentions* that have been written but not yet erased, and erased intentions. We say that v is a *surviving version number* for page p iff there exists a surviving intention for page p version v. Some versions are obsolete and others are exposed.

New transaction requirement: Before a new transaction can be started involving page p, there must be no current transaction involving page p.

Highest surviving version number: We define $hv(p)$ as the highest surviving version number for page p.

Obsolete version: A surviving intention for page p version v is an *obsolete version* iff v is less than the last committed version number for page p. An obsolete version is older than the last committed version.

Exposed version: A surviving intention for page p version v is an *exposed version* iff v is greater than the last committed version number for page p. An exposed version is newer than the last committed version.

In the event of a crash and reboot, a recovery process must determine the last committed version for each page by analyzing the surviving intentions. Since a transaction either commits or not, surviving intentions belonging to the same transaction are either all committed or all uncommitted. If a complete cycle can be traced through the next-links, then all intentions were written and they are all committed.

However, a break in the cycle does not necessarily mean that the intentions are uncommitted. This ambiguity results from the actions of garbage collection. When a new intention has been committed for page p, earlier intentions for page p represent storage overhead and ought to be garbage collected. These earlier intentions have been rendered obsolete by the new committed intention. However, reclaiming an obsolete intention can leave a break in the cycle it belongs to.

Figure 1 illustrates the situation. In this example, pages are indicated by the letters A through C and the version numbers are 1 and 2. Next links are shown by arrows. Versions are labeled as to whether they are committed (crosshatch fill) or uncommitted (white fill). Versions that are missing are indicated by a dotted border. We write "P_i" for version i of page P.

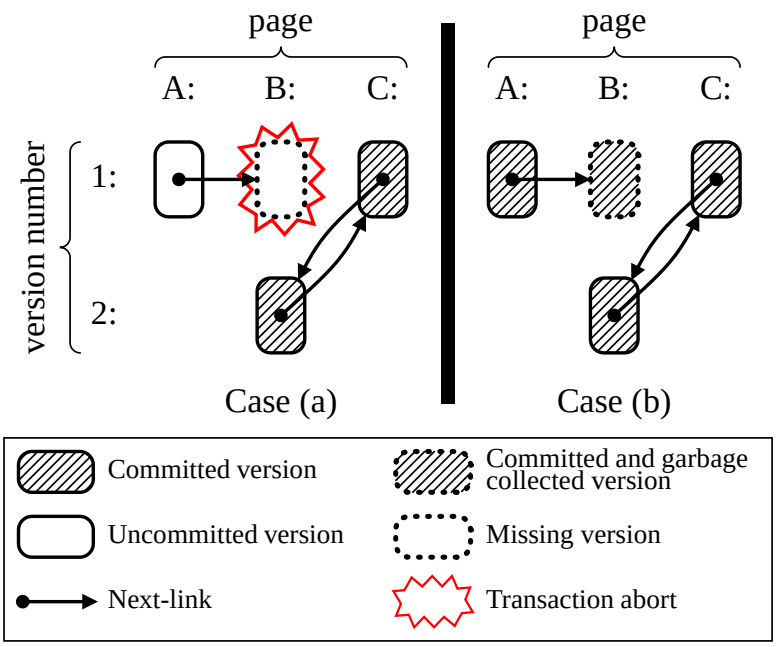

Figure 1: Ambiguous case of aborted and committed transactions.

Consider the scenario where A_1 has a next-link of B_1, but B_1 is missing. There are two cases that could lead to this scenario. In the first case, as shown in Figure 1(a), the transaction with A_1 and B_1 was aborted and B_1 was never written, then subsequently a successful transaction creates B_2 and C_1. In the second case, as shown in Figure 1(b), the transaction with A_1 and B_1 commits, followed by another successful transaction that creates B_2 and C_1, which makes B_1 obsolete and as a result B_1 is garbage collected. In the first case, A_1 belongs to an aborted transaction and should be discarded, while in the second case A_1 belongs to a committed transaction and should be preserved. If all we have are the next-links, it is impossible to distinguish these cases. Resolving the *cycle break ambiguity* is the central problem for a cyclic commit protocol.

The Simple Cyclic Commit (SCC) protocol resolves the cycle break ambiguity by requiring that an uncommitted intention be erased before it can be buried by a new intention. Under the SCC protocol, Figure 1(a) could never arise, because the uncommitted A_1 would have to be erased thus erasing the reference to B_1 before B_2 could be written.

The Back-Pointer Cyclic Commit (BPCC) protocol resolves cycle break ambiguity by recording extra information in the metadata and requiring that obsolete intentions be erased in a certain order. Under the BPCC protocol, there would be additional information in the metadata that would distinguish between Figure 1(a) and Figure 1(b).

Details of the SCC and BPCC protocols are presented next.

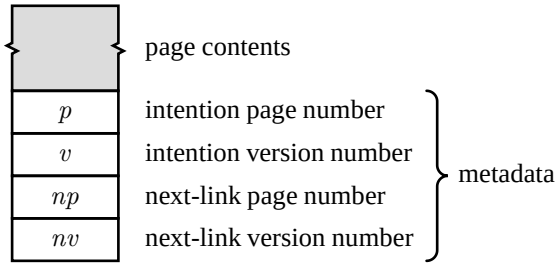

Figure 2: SCC intention format.

3 Simple Cyclic Commit (SCC)

The Simple Cyclic Commit (SCC) protocol is based on the idea that surviving uncommitted intentions should be erased before they become buried by intentions written later. We describe the protocol, the specification, and the results of model checking.

3.1 The SCC protocol

The SCC protocol uses an intention with metadata fields as illustrated in Figure 2. Given an intention I, fields $I.p$ and $I.v$ contain the page and version number of the intention and fields $I.np$ and $I.nv$ contain the page and version number of the next-link. Recall that $hv(p)$ is the highest surviving version number for page p. The SCC protocol maintains the following invariants:

SCC page invariant: Any surviving intention I with $I.v < hv(I.p)$ is committed.

SCC next-link invariant: Any surviving intention I with $I.nv < hv(I.np)$ is committed.

Observe that the SCC invariants say nothing about the commitment state of a highest surviving version whose next-link is also a highest surviving version. However, if a highest surviving version is committed then it cannot be garbage collected, because it is the last committed version of its page. So the commitment state of a highest surviving version can be determined by tracing through the next-links to see if a cycle exists. A precise description is:

SCC commitment decision process: For each highest surviving version I, trace its next-link. There are three cases:

Case 1. If $I.nv < hv(I.np)$, then I is committed, because of the SCC next-link invariant.

Case 2. If $I.nv > hv(I.np)$, then I is uncommitted, because a complete cycle could never have been written.

Case 3. If $I.nv = hv(I.np)$, then there exists a highest surviving version J such that $I.np = J.p$ and $I.nv = J.v$. Recursively determine whether J is committed and the same answer applies to I. If this leads to a cycle, then I is committed.

Whenever the highest surviving version of page p is uncommitted, from the SCC page invariant it follows that the second highest surviving version of page p must be committed. So the only question during recovery is to choose between the highest and second highest surviving versions of each page.

In the SCC protocol, there can be at most one exposed version for a given page. The SCC page invariant prevents the system from burying an uncommitted page. The SCC next-link invariant prevents the system from burying a next-link to a missing page.

The SCC page and next-link invariants are established at initialization by writing a singleton cycle of version zero for each page. The invariants are maintained by two requirements. First, the new transaction requirement prevents two or more transactions from creating new versions of the same page at the same time. Second, before a transaction can write a planned intention, there must be no uncommitted page version or next-link page version that would be buried. This second requirement is stated precisely as follows:

SCC write requirement: Before a planned intention I can be written, there must be no exposed intention J such that $I.p = J.p$ or $I.p = J.np$.

It is straightforward to see that observing the new transaction and SCC write requirements maintain the page and next-link invariants.

The SCC write requirement can be observed proactively by erasing all uncommitted intentions during the recovery process and by erasing all uncommitted intentions belonging to an incomplete transaction when that transaction is aborted. However, one could also adopt a more lazy approach and wait until a planned intention is about to be written before erasing any problematic exposed intentions. The lazy approach requires more bookkeeping and it can delay the progress of transactions, but it does permit new transactions to be started more quickly after a recovery or an abort.

3.2 Tour of the SCC specification

Appendix A.1 gives a TLA+ specification of the SCC protocol. The specification is organized into various sections. First are some preliminary definitions. Then come definitions related to the state of a metadata record, the state of a transaction in progress, and the overall state of the system. Then after a few auxiliary definitions we specify how to compute the last used version number and last committed version number from a set of metadata records. Then we specify the actions as a series of next state relations. Finally come a list of invariants.

The specification models just the metadata part of intention records, omitting any consideration of actual page contents. The metadata record contains the page and version number of the intention along with the page and version number of the next-link intention in the cycle.

Each transaction in progress consists of two parts: a map from the set of involved pages to their new version numbers and a set of metadata records that are planned to be written. When a new transaction is launched, a set of available pages is selected and arranged into a cycle, new version numbers are determined and the map constructed, the planned metadata records are all constructed, and a new transaction in progress record created.

The state consists of a set of metadata records, a set of transactions in progress, the last committed version number for each page, and the last used version number for each page. The set of metadata records models the contents of non-volatile state that survives a crash. The transactions in progress, last committed version numbers, and last used version numbers are volatile state that is erased by a crash and must be recovered from the non-volatile set of metadata records.

The operator $AvailablePages$ computes the set of pages that are not involved in current transactions. Any non-empty subset of these pages may be selected for a new transaction.

The specification models the writing of each of the metadata records from a transaction in progress to the non-volatile state as a separate action. This permits full exploration of all possible interleavings, but state explosion rapidly sets in. Planned metadata records can only be written when they will not bury an exposed intention's page or next-link page. This condition is computed by $IsWritableMeta$.

The set of intentions that is available for erasure in a garbage collection is computed by $CollectableMetas$. In the SCC protocol, any obsolete intention and any exposed intention that is not involved in a current transaction may be erased.

The specification models garbage collection as an action that erases any non-empty subset of the collectable metadata records. Flash memories can only erase entire blocks of pages, rather than individual pages. The erasure of specific garbage pages would be implemented by copying all non-garbage pages out of a subject block and then erasing the block. So it is in fact interesting to erase multiple intentions simultaneously, rather than only one at a time. This is not an issue for the SCC protocol, but it is an important consideration for the BPCC protocol, in which erasing one intention can enable the erasure of another intention.

The specification tracks the last committed version number for each page as part of volatile state. The last committed version numbers are updated when a transaction completes. The $ComputeComm$ operator specifies how to compute the last committed version numbers based on the non-volatile set of metadata records. This operator is used to recover the last committed version numbers during a crash-reboot. The invariant $InvComm$ checks that the result of $ComputeComm$ always matches the last committed version number map in volatile state.

A similar arrangement holds for the last used version number for each page. The specification tracks them as part of volatile state. They are updated when a new transaction launches and recovered by $ComputeUsed$ during a crash-reboot. The invariant $InvUsed$ checks that the result of $ComputeUsed$ is always consistent with the volatile state.

The specification permits aborting an incomplete transaction. This might leave uncommitted garbage intentions written in non-volatile storage. Similarly, a crash-reboot might leave uncommitted garbage intentions written in non-volatile storage. The specification models the lazy approach to erasing such uncommitted garbage without detailing the bookkeeping that would be required to make this approach efficient. Rather, this is incorporated into the determination of whether a planned intention is writable at the current time, as computed by $IsWritableMeta$.

pages	$MaxVers$	depth	states	time (s)
2	2	10	183	3
2	3	14	1356	4
2	4	18	8133	13
3	2	14	2199	10
3	3	20	60,263	323
3	4	26	1,127,619	5750
4	2	18	27,660	508
4	3	26	3,203,868	57,028

Table 1: SCC model checking results. Complete state space exploration.

		configuration		counterexample found at		
bug	Appendix	pages	*MaxVers*	depth	states	time (s)
BuryCheckNone	A.4	2	2	6	88	3
BuryCheckP	A.5	3	2	6	194	3
BuryCheckNP	A.6	3	2	6	203	3

Table 2: SCC model checking results for known bugs.

3.3 Model checking the SCC specification

Appendix A.2 lists an extension module $McSCC$ that provides additional definitions designed to assist in model checking. The model checking extension provides for a constraint on the maximum version number and for a symmetry on the page identifiers. Appendix A.3 shows a TLC configuration file for running the model checker. This configuration file can be modified in obvious ways to adjust the number page identifiers and the maximum version number.

Using a 2.8 GHz Intel Pentium 4 with 1 GB of memory running TLC version 2.0, we model checked the SCC specification for various configurations of number of pages and maximum version numbers. For each configuration, TLC determined the maximum depth of the state space graph as well as the total number of distinct states. No errors were found. Table 1 shows the results. The approximate running time is given in seconds.

When running the TLC model checker, it is nice to see that none of the invariants are violated for the size of the model that TLC can check. However, there is always the possibility that a bug lurks over the horizon. One way to get more confidence is to introduce a known bug on purpose and see if TLC finds a counterexample execution.

Three bugs were investigated. *BuryCheckNone* ignores the issue of buried intentions. *BuryCheckP* checks only for a conflict with the page of the written intention. *BuryCheckNP* checks only for a conflict with the next-link page of the written intention. TLC finds a counterexample for each of these bugs and reports the depth and number of distinct states searched. Table 2 shows the results. The approximate running time is given in seconds. TLC configuration files demonstrating the bugs appear in the indicated Appendices.

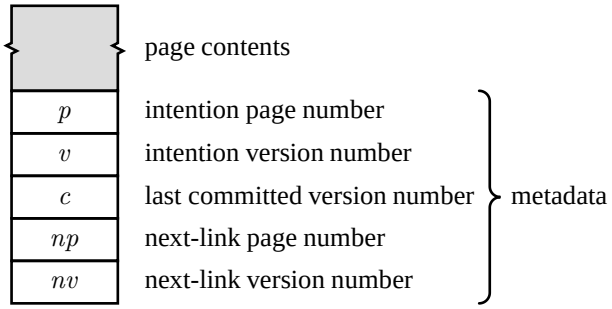

Figure 3: BPCC intention format.

4 Back-Pointer Cyclic Commit (BPCC)

The Back-Pointer Cyclic Commit (BPCC) protocol is based on the idea of augmenting the metadata so that each intention records a *back pointer* to the last committed version of the same page. This extra information enables the recovery process to distinguish between the two cases of Figure 1, as explained below. We describe the protocol, the specification, and the results of model checking.

4.1 The BPCC protocol

The BPCC protocol uses an intention with metadata fields as illustrated in Figure 3. Given an intention I, fields $I.p$ and $I.v$ contain the page and version number of the intention, field $I.c$ contains the last committed version number for page $I.p$ and fields $I.np$ and $I.nv$ contain the page and version number of the next-link.

The main concept in the BPCC protocol is the *straddle*. We say that intention K *straddles* the next-link of intention I iff $K.p = I.np$ and $K.c < I.nv < K.v$. Since $K.c$ is the last committed version of page $K.p$ prior to $K.v$, intention I could not be committed. Recall that $hv(p)$ is the highest surviving version number for page p. The BPCC protocol maintains the following invariant:

BPCC straddle invariant: Given a surviving intention I with $I.v = hv(I.p)$ and $I.nv < hv(I.np)$, then I is uncommitted iff there exists a surviving intention K such that $K.p = I.np$ and $K.c < I.nv < K.v$.

Observe that the BPCC invariant says nothing about the commitment state of a highest surviving version whose next-link is also a highest surviving version. However, if a highest surviving version is committed then it cannot be garbage collected, because it is the last committed version of its page. So the commitment state of a highest surviving version can be determined by tracing through the next-links to see if a cycle exists. A precise description is:

BPCC commitment decision process: For each highest surviving version I, trace its next-link. There are three cases:

Case 1. If $I.nv < hv(I.np)$, then I is uncommitted iff there exists a surviving intention K with $K.p = I.np$ and $K.c < I.nv < K.v$, because of the BPCC straddle invariant.

Case 2. If $I.nv > hv(I.np)$, then I is uncommitted, because a complete cycle could never have been written.

Case 3. If $I.nv = hv(I.np)$, then there exists a highest surviving version J such that $I.np = J.p$ and $I.nv = J.v$. Recursively determine whether J is committed and the same answer applies to I. If this leads to a cycle, then I is committed.

Whenever the highest surviving version of page p is uncommitted, then the back pointer tells directly what the last committed version is for page p. So the only question during recovery is to determine whether or not for each page the highest surviving version is committed.

For example, refer to the BPCC system state illustrated in Figure 4. In this example, pages are indicated by the letters A through F and the version numbers are 1 through 4. Back pointers are shown by dashed arrows. To reduce clutter, most of the back pointers are not shown. The area in which exposed versions lie is shaded.

Observe that E_3 has a next-link that points to F_3, but F_3 is missing. However, F_4 is present and it has a back pointer to F_1, indicating that the intermediate versions F_2 and F_3 could not possibly be committed. Consequently E_3 must be uncommitted. We say that the back pointer from F_4 straddles the next-link from E_3. When a version's next-link is straddled, the version must be uncommitted.

On the other hand, C_3 has a next-link that points to a missing D_3, there is a higher version D_4, but no version has a back pointer that straddles the next-link from C_3. In this case, based on the BPCC straddle invariant, the BPCC commitment decision process concludes that C_3 is committed.

The BPCC straddle invariant is established at initialization by writing a singleton cycle of version zero for each page, with the back pointer also set to version zero. None of the next-links are straddled and all of the versions are committed.

The BPCC straddle invariant must be maintained during operation as new intentions are written and obsolete intentions are garbage collected. Writing new intentions is not a problem, because the new transaction requirement guarantees that each page is involved in at most one transaction at a time. Therefore, when a new transaction is launched, we know what the last committed version is for each page involved in the transaction, and this version cannot change until the new transaction either commits or aborts. Writing new intentions automatically creates straddles as necessary to preserve the BPCC straddle invariant.

However, there is an issue with garbage collection. Refer to the example state in Figure 4. B_3 is an obsolete committed version, since the last committed version is B_4. However, B_3 is the only version that holds a straddle over the next-link from A_2. In this state it would be a mistake to erase B_3, because then the BPCC straddle invariant would be violated. Since A_2 has a next-link of B_2, which is less than the highest surviving version of page B, the absence of a straddler would lead to the erronious conclusion that A_2 was committed. Therefore, as long as they are necessary as straddlers, some obsolete versions cannot be garbage collected.

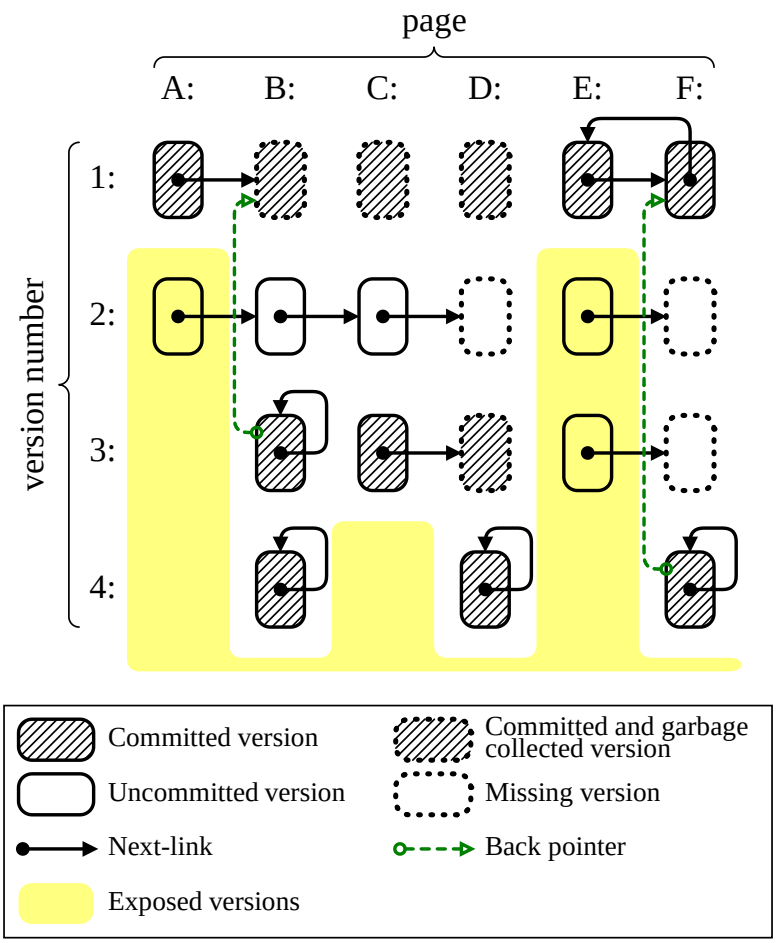

Figure 4: An example BPCC system state.

We can arrange for uncommitted versions never to be necessary as straddlers. This is good, because it means that uncommitted versions not involved in a current transaction can be garbage collected at any time. Observe that as successive uncommitted versions are written for page p, each one will have the same back pointer, because the last committed version of page p is the same in each case.

As long as the highest version does not commit, all of these uncommitted versions are exposed versions. Since the highest version straddles more than any of the others, erasing any of the others has no effect on the existence of a straddler. Erasing the highest version may cause some formerly straddled next-links now to point to a version number greater than or equal to the highest surviving version number, but this is okay because the BPCC invariant does not claim anything about such next-links. The commitment decision process will reach the same conclusion as if the next-link had been straddled, so the result is unchanged.

Once the highest version commits, all of the preceeding uncommitted versions become obsolete. The committed version has a larger straddle than all of the immediately preceeding uncommitted versions and we can preserve it in preference to them. The committed version is the *best straddler*.

So in either case, no uncommitted version is necessary as a straddler. As long as the committed version is the last committed version, it cannot be erased. However, eventually a subsequent version will commit, leaving the best straddler as an obsolete committed version.

Hence we only need to worry about preserving obsolete best straddlers. So the question is whether an obsolete best straddler might ever be needed in the future. If it won't be needed, we don't need to preserve it. For any straddler, there is a set of versions' next-links that it straddles. Because of the new transaction requirement, this set can never increase. So a straddler that does not straddle anything now will never straddle anything in the future and therefore does not need to be preserved.

The analysis can be made tighter. Observe that the commitment decision process only checks for a straddler of the next-link of a highest version. So only the next-links of highest versions are relevent. But if a highest version is left uncommitted by an aborted transaction, we would like garbage collection to be able to erase it, which would then reveal the next-link of the next lower version to the committment decision process. This can repeat, but the erasure of highest versions must stop at the last committed version, which cannot be erased. Consequently, the only next-links for which straddlers might be needed are the uncommitted versions higher than the last committed version. These are precisely the versions we call exposed versions. Since the next-link of a committed version cannot be straddled, we do not need to consider the next-link of the last committed version.

The requirement is stated precisely as follows:

BPCC necessary straddler: Intention K is a *necessary straddler* iff

- **(a)** (obsolete) K is an obsolete version and
- **(b)** (best) there is no surviving intention L such that $K.p = L.p$ and $K.c = L.c$ and $K.v < L.v$ and
- **(c)** (needed) there is a surviving exposed version I such that $K.p = I.np$ and $K.c < I.nv < K.v$.

BPCC garbage collection requirement: In order to erase an intention I, it must be the case that either

- **(a)** I is an obsolete version and I is not a necessary straddler or
- **(b)** I is an exposed version and there is no current transaction involving I.

Because of the new transaction requirement, a straddler must be written after the intention whose next-link it straddles. Therefore, a simple approach for observing the BPCC garbage collection requirement is to erase obsolete intentions in the order in which they were originally written. An easy way to keep track of this order is to number each transaction and use the transaction number as the version number of each intention planned to be written by the transaction. This approach leaves potentially

large gaps in the version number sequence of the intentions for a given page, but it works fine in all respects.

A more sophisticated approach builds additional data structures in volatile memory to keep track of precisely which exposed versions' next-links are straddled by a best straddler. The detailed bookkeeping needed to keep these data structures up to date as new intentions are written and garbage intentions are erased is not specified here.

4.2 Tour of the BPCC specification

Appendix B.1 gives a TLA+ specification of the BPCC protocol. The specification is written so as to be very similar to that of the SCC protocol, with only a few critical differences. The specification is organized into various sections. First are some preliminary definitions. Then come definitions related to the state of a metadata record, the state of a transaction in progress, and the overall state of the system. Then after a few auxiliary definitions we specify how to compute the last used version number and last committed version number from a set of metadata records. Then we specify the actions as a series of next state relations. Finally come a list of invariants.

The specification models just the metadata part of intention records, omitting any consideration of actual page contents. The metadata record contains the page and version number of the intention, the version number of the last committed version of the page, and the page and version number of the next-link intention in the cycle.

Each transaction in progress consists of two parts: a map from the set of involved pages to their new version numbers and a set of metadata records that are planned to be written. When a new transaction is launched, a set of available pages is selected and arranged into a cycle, new version numbers are determined and the map constructed, the planned metadata records are all constructed, and a new transaction in progress record created.

The state consists of a set of metadata records, a set of transactions in progress, the last committed version number for each page, and the last used version number for each page. The set of metadata records models the contents of non-volatile state that survives a crash. The transactions in progress, last committed version numbers, and last used version numbers are volatile state that is erased by a crash and must be recovered from the non-volatile set of metadata records.

The operator $AvailablePages$ computes the set of pages that are not involved in current transactions. Any non-empty subset of these pages may be selected for a new transaction.

The specification models the writing of each of the metadata records from a transaction in progress to the non-volatile state as a separate action. This permits full exploration of all possible interleavings, but state explosion rapidly sets in.

The set of intentions that is available for erasure in a garbage collection is computed by $CollectableMetas$. The specification models the required properties without detailing the bookkeeping that would be needed to maintain this set efficiently during operation.

The specification models garbage collection as an action that erases any non-empty subset of the collectable metadata records. Flash memories can only erase entire blocks of pages, rather than individual pages. The erasure of specific garbage pages would be

implemented by copying all non-garbage pages out of a subject block and then erasing the block. So it is in fact interesting to erase multiple intentions simultaneously, rather than only one at a time. In the BPCC protocol, erasing an uncommitted intention whose next-link is straddled by an obsolete intention may enable the erasure of the straddler. However, it is important to complete the erasure of the uncommitted intention before starting the erasure of the straddler. Therefore, these two intentions must be erased in separate actions.

The specification tracks the last committed version number for each page as part of volatile state. The last committed version numbers are updated when a transaction completes. The $ComputeComm$ operator specifies how to computes the last committed version numbers based on the non-volatile set of metadata records. This operator is used to recover the last committed version numbers during a crash-reboot. The invariant $InvComm$ checks that the result of $ComputeComm$ always matches the last committed version number map in volatile state.

A similar arrangement holds for the last used version number for each page. The specification tracks them as part of volatile state. They are updated when a new transaction launches and recovered by $ComputeUsed$ during a crash-reboot. The invariant $InvUsed$ checks that the result of $ComputeUsed$ is always consistent with the volatile state.

In the BPCC protocol there is no restriction on when planned metadata can be written. Consequently, $IsWritableMeta$ always returns TRUE.

4.3 Model checking the BPCC specification

Appendix B.2 lists an extension module $McBPCC$ that provides additional definitions designed to assist in model checking. The model checking extension provides for a constraint on the maximum version number and for a symmetry on the page identifiers. Appendix B.3 shows a TLC configuration file for running the model checker. This configuration file can be modified in obvious ways to adjust the number page identifiers and the maximum version number.

Using a 2.8 GHz Intel Pentium 4 with 1 GB of memory running TLC version 2.0, we model checked the BPCC specification for various configurations of number of pages and maximum version numbers. For each configuration, TLC determined the maximum depth of the state space graph as well as the total number of distinct states. No errors were found. Table 3 shows the results. The approximate running time is given in seconds.

When running the TLC model checker, it is nice to see that none of the invariants are violated for the size of the model that TLC can check. However, there is always the possibility that a bug lurks over the horizon. One way to get more confidence is to introduce a known bug on purpose and see if TLC finds a counterexample execution.

Two bugs were investigated. *StradGcCheckNone* ignores the issue of straddlers during garbage collection. *StradGcCheckHigh* checks only the highest version's next-link. TLC finds a counterexample for each of these bugs and reports the depth and number of distinct states searched. Table 4 shows the results. The approximate running time is given in seconds. TLC configuration files demonstrating the bugs appear in the indicated Appendices.

pages	$MaxVers$	depth	states	time (s)
2	2	10	403	7
2	3	14	8599	18
2	4	18	186,657	275
3	2	14	11,783	76
3	3	20	2,992,030	26,400
4	2	22	430,727	14,700

Table 3: BPCC model checking results. Complete state space exploration.

		configuration		counterexample found at		
bug	Appendix	pages	*MaxVers*	depth	states	time (s)
StradGcCheckNone	B.4	2	3	9	1339	7
StradGcCheckHigh	B.5	2	4	12	21,745	20

Table 4: BPCC model checking results for known bugs.

Acknowledgements

Thanks to Vijayan Prabhakaran and Lidong Zhou for starting the development of cyclic commit protocols, helping to clarify the descriptions, and commenting on the specifications.

References

[1] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[2] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional flash. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, December 2008. To appear.

A SCC specification

A.1 Specification

MODULE SCC

EXTENDS $Naturals$, $Sequences$, $FiniteSets$

VARIABLE $state$

CONSTANT $Page$ page identifier
$Vers \triangleq Nat$ version number

HELPFUL TLA+ DEFINITIONS

The set of all maps from subsets of D to R . This is like $[D \to R]$ except it includes all subsets of D , that is, partial maps.

$PMAP(D,\ R) \triangleq$ UNION $\{[d \to R] : d \in$ SUBSET $D\}$

Arrange a set into all possible sequences in which each element of the set appears exactly once.

$AllExactOnceSeq(S) \triangleq$
  LET $X \triangleq 1\ ..\ Cardinality(S)$
  IN $\{q \in [X \to S] : \forall\, s \in S : \exists\, x \in X : q[x] = s\}$

Select the maximum from a non-empty set.

$MaxOf(S) \triangleq$ CHOOSE $s0 \in S : \forall\, s1 \in S : s0 \geq s1$

Definition of a metadata record.

$Meta \triangleq$ [
  $p : Page,\ \ v : Vers,$ page and version of this intention
  $np : Page,\ \ nv : Vers$ next page and version in cycle
]

Initial metadata for page p . A singleton cycle of version zero.

$InitMeta(p) \triangleq$ [
  $p \mapsto p,\ \ v \mapsto 0,$
  $np \mapsto p,\ \ nv \mapsto 0$
]

Definition of the state of an in-progress transaction. The state consists of a map from the set of pages involved in the transaction to their new version numbers and the set of metadata records planned but not yet written.

$Tran \triangleq$ [
  $pv : PMAP(Page,\ Vers),$ map involved pages to new versions
  $remm :$ SUBSET $Meta$ metadata yet to be written
]

Make a transaction record for the cycle of pages given by the sequence pq and the current last used version numbers given by $used$. We examine the sequence to determine the set of involved pages pp . Then we compose the map pv from the involved pages to their new version numbers. Then we construct the metadata that is to be written for page index i in the cycle. Finally put it all together to compose the transaction record.

$$
\begin{array}{l}
MakeTran(pq,\ used) \triangleq \\
\quad \text{LET} \\
\quad\quad pp \triangleq \{pq[i] : i \in \text{DOMAIN } pq\} \quad \text{set of pages} \\
\quad\quad pv \triangleq [p \in pp \mapsto used[p] + 1] \quad \text{map to new version numbers} \\
\quad\quad mi(i) \triangleq \quad \text{metadata planned for index } i \\
\quad\quad\quad \text{LET} \\
\quad\quad\quad\quad p \triangleq pq[i] \\
\quad\quad\quad\quad np \triangleq pq[\text{IF } i = Len(pq) \text{ THEN } 1 \text{ ELSE } i + 1] \\
\quad\quad\quad \text{IN } [\\
\quad\quad\quad\quad p \mapsto p, \quad v \mapsto pv[p], \\
\quad\quad\quad\quad np \mapsto np, \quad nv \mapsto pv[np] \\
\quad\quad\quad] \\
\quad \text{IN } [\\
\quad\quad pv \mapsto pv, \\
\quad\quad remm \mapsto \{mi(i) : i \in \text{DOMAIN } pq\} \\
\quad]
\end{array}
$$

Get the set of pages involved in a transaction. The domain of the map specifies the set of involved pages.

$$TranPP(t) \triangleq \text{DOMAIN } t.pv$$

Determine if a given metadata record is part of a given transaction.

$$IsMetaPartOfTran(m,\ t) \triangleq m.p \in TranPP(t) \land m.v = t.pv[m.p]$$

Definition of the state of the system. The state consists of a set of metadata records, a set of transactions in progress, the last committed version number for each page, and the last used version number for each page. Note that we do not model the allocation of pages or specifically where the metadata records are stored. Instead, the metadata records are modelled simply as a set.

$$
\begin{array}{l}
State \triangleq [\\
\quad meta : \text{SUBSET } Meta, \quad \text{written metadata records} \\
\quad tran : \text{SUBSET } Tran, \quad \text{transactions in progress} \\
\quad comm : [Page \to Vers], \quad \text{last committed version} \\
\quad used : [Page \to Vers] \quad \text{last used version} \\
]
\end{array}
$$

Initial state.

$$
\begin{array}{l}
InitState \triangleq [\\
\quad meta \mapsto \{InitMeta(p) : p \in Page\}, \\
\quad tran \mapsto \{\}, \\
\quad comm \mapsto [p \in Page \mapsto 0], \\
\quad used \mapsto [p \in Page \mapsto 0] \\
]
\end{array}
$$

The set of all metadata records written or planned to be written in state s .

$$AllMeta(s) \triangleq s.meta \cup \text{UNION } \{t.remm : t \in s.tran\}$$

Given a set mm of metadata records, the subset that have p as their page or as their next page, respectively.

$$MetaP(mm,\ p) \triangleq \{m \in mm : m.p = p\}$$
$$MetaNP(mm,\ p) \triangleq \{m \in mm : m.np = p\}$$

Pages that are available for a new transaction in state s . Although multiple transactions may be in progress at the same time, the sets of pages must not overlap. Hence a page that is involved in a current transaction is not available for launching in a new transaction.

$$AvailPages(s) \triangleq \{p \in Page : \forall\, t \in s.tran : p \notin TranPP(t)\}$$

Determine if metadata m is obsolete in state s .

$$IsObsoleteMeta(m,\ s) \triangleq m.v < s.comm[m.p]$$

Determine if metadata m is exposed in state s .

$$IsExposedMeta(m,\ s) \triangleq m.v > s.comm[m.p]$$

Determine if metadata $m0$ would bury metadata $m1$. We check if $m0.p$ matches either $m1.p$ or $m1.np$.

$$WouldBuryMeta(m0,\ m1) \triangleq$$
$$\lor\ m0.p = m1.p$$
$$\lor\ m0.p = m1.np$$

Determine if a planned metadata m may be written in state s . A planned metadata may not be written if there is an exposed metadata that it would bury. Before the planned metadata can be written, the exposed metadata must be erased.

$$IsWritableMeta(m,\ s) \triangleq$$
$$\forall\, m1 \in s.meta : IsExposedMeta(m1,\ s) \Rightarrow \neg\, WouldBuryMeta(m,\ m1)$$

Written metadata records that are garbage collectable in state s . Any obsolete intention may be collected. Any exposed intention may be collected if it is not part of a current transaction.

$$CollectableMetas(s) \triangleq$$
$$\{m \in s.meta :$$
$$\lor\ IsObsoleteMeta(m,\ s)$$
$$\lor\ IsExposedMeta(m,\ s) \land \forall\, t \in s.tran : \neg IsMetaPartOfTran(m,\ t)$$
$$\}$$

Compute the last used version per page based on the set mm of metadata records. Note that a page p can appear as the page and as the next page in a metadata record. In either case, the corresponding version number is in use for that page.

$$ComputeUsed(mm) \triangleq$$
LET
$$vv(p) \triangleq$$ all versions of page p mentioned in any metadata
$$\{m.v \ : m \in MetaP(mm,\ p)\} \cup$$
$$\{m.nv : m \in MetaNP(mm,\ p)\}$$
IN
$$[p \in Page \mapsto MaxOf(vv(p))]$$

Compute the last committed version per page based on the set mm of metadata records. Because of the SCC invariant, the committed or uncommitted status of any given intention can be determined by comparing its next-link version number nv against the highest version of its next-link page $hm(np).v$.

If $nv < hm(np).v$ then the given intention is committed.

If $nv > hm(np).v$ then the given intention is uncommitted.

If $nv = hm(np).v$ then the status is recursively the same as for the next-link intention, with a cycle meaning committed.

$ComputeComm(mm) \triangleq$

LET

Metadata of highest version for page p .

$hm(p) \triangleq$ LET $M \triangleq MetaP(mm,\ p)$
IN CHOOSE $m0 \in M : \forall\, m1 \in M : m0.v \geq m1.v$

Metadata of second highest version for page p .

$h2m(p) \triangleq$ LET $M \triangleq MetaP(mm,\ p) \setminus \{hm(p)\}$
IN CHOOSE $m0 \in M : \forall\, m1 \in M : m0.v \geq m1.v$

Having started with page $p0$, determine if version nv of page np is committed by recursively tracing the commit cycle through the metadata records.

$IsComm[np \in Page,\ nv \in Vers,\ p0 \in Page] \triangleq$
IF $np = p0$ THEN TRUE ELSE — cycle
IF $nv < hm(np).v$ THEN TRUE ELSE — link is less than highest
IF $nv > hm(np).v$ THEN FALSE ELSE — link is more than highest
$IsComm[hm(np).np,\ hm(np).nv,\ p0]$

Determine the last committed version of page p .

$comm(p) \triangleq$
IF $IsComm[hm(p).np,\ hm(p).nv,\ p]$
THEN $hm(p).v$ — highest version is committed
ELSE $h2m(p).v$ — otherwise must be second highest version

IN

$[p \in Page \mapsto comm(p)]$

NEXT STATE RELATIONS

Launch a new transaction. We select a non-empty subset of the available pages and arrange them into any cycle. Then we construct a transaction in progress record based on the cycle and the current last used version numbers. We update the last used version number of pages involved in the cycle.

$NextLaunchTransaction \triangleq$
$\exists\, pp \in$ SUBSET $AvailPages(state)$: — any set of available pages
$\exists\, pq \in AllExactOnceSeq(pp)$: — arrange in any cycle
$\wedge\ pp \neq \{\}$ — must be non-empty
$\wedge$ LET
$t \triangleq MakeTran(pq,\ state.used)$
IN

$state' = [state \text{ EXCEPT}$
$!.tran = @ \cup \{t\},$
$!.used = [p \in Page \mapsto \text{IF } p \in pp \text{ THEN } t.pv[p] \text{ ELSE } @[p]]$
$]$

Perform an intermediate step in a transaction. We select some unwritten metadata record from a transaction in progress and write it, provided that it is not the final unwritten metadata for that transaction.

$NextInterStepTransaction \triangleq$
$\exists t \in state.tran :$
$\exists m \in t.remm :$
$\land \{m\} \neq t.remm$ not the last unwritten metadata
$\land IsWritableMeta(m, state)$ is writable at this time
$\land$ LET
$t1 \triangleq [t \text{ EXCEPT } !.remm = @ \setminus \{m\}]$
IN
$state' = [state \text{ EXCEPT}$
$!.tran = @ \setminus \{t\},$ remove t
$!.tran = @ \cup \{t1\},$ put in $t1$
$!.meta = @ \cup \{m\}$ put in m
$]$

Perform the final step in a transaction. We add the last unwritten metadata record from a transaction in progress. Since the transaction is complete the transaction record goes away and each page involved in the transaction has to have its last committed version number updated.

$NextFinalStepTransaction \triangleq$
$\exists t \in state.tran :$
$\exists m \in t.remm :$
$\land \{m\} = t.remm$ the last unwritten metadata
$\land IsWritableMeta(m, state)$ is writable at this time
$\land$ LET
$pp \triangleq TranPP(t)$
IN
$state' = [state \text{ EXCEPT}$
$!.tran = @ \setminus \{t\},$ remove t
$!.meta = @ \cup \{m\},$ put in m
$!.comm = [p \in Page \mapsto \text{IF } p \in pp \text{ THEN } t.pv[p] \text{ ELSE } @[p]]$
$]$

Abort a transaction. We delete a transaction in progress. Note that this may leave some uncommitted garbage that will have to be collected before a new intention can be written that would bury it.

$NextAbortTransaction \triangleq$
$\exists t \in state.tran :$
$state' = [state \text{ EXCEPT } !.tran = @ \setminus \{t\}]$

Model a crash reboot. We discard all transactions in progress and compute the last committed version number and last used version number of each page from the written metadata records. Note that this may leave some uncommitted garbage that will have to be collected before a new intention can be written that would bury it.

$NextCrashReboot \triangleq$
$state' = [$
$meta \mapsto state.meta,$
$tran \mapsto \{\},$
$comm \mapsto ComputeComm(state.meta),$
$used \mapsto ComputeUsed(state.meta)$
$]$

Collect some garbage. We can collect any subset of garbage in one action.

$NextCollectGarbage \triangleq$
$\exists\, mm \in \text{SUBSET } CollectableMetas(state) :$ any subset
$\land\, mm \neq \{\}$ non-empty
$\land\, state' = [state \text{ EXCEPT } !.meta = @ \setminus mm]$

Initial state.

$Init \triangleq state = InitState$

Next state relation.

$Next \triangleq$
$\lor\, NextLaunchTransaction$
$\lor\, NextInterStepTransaction$
$\lor\, NextFinalStepTransaction$
$\lor\, NextAbortTransaction$
$\lor\, NextCrashReboot$
$\lor\, NextCollectGarbage$

INVARIANTS

The state must always be of the proper type.

$InvType \triangleq state \in State$

A given page and version number may appear at most once in the metadata.

$InvMetaUniq \triangleq$
$\forall\, m1,\, m2 \in AllMeta(state) :$
$(m1.p = m2.p \land m1.v = m2.v) \Rightarrow m1 = m2$

What we think is the last used version per page must always be consistent with that computed from the metadata records written and planned to be written. The volatile last used version may be greater than that computed from the metadata because of aborted transactions. However, that is still consistent.

$InvUsed \triangleq$
LET
$mm \triangleq AllMeta(state)$ all metadata
$used \triangleq ComputeUsed(mm)$
IN
$\forall\, p \in Page : state.used[p] \geq used[p]$

What we think is the last committed version per page must always be identical to that computed from the metadata records written.

$InvComm \triangleq$
  LET
    $mm \triangleq state.meta$ written metadata
    $comm \triangleq ComputeComm(mm)$
  IN
    $\forall\, p \in Page : state.comm[p] = comm[p]$

A.2 Model specification

MODULE $McSCC$

EXTENDS $SCC,\ TLC$

CONSTANT $MaxVers$

Model checking definition of $Vers$. We have to permit one higher than the maximum value because the type invariant $InvType$ is checked before the constraint is applied.

$McVers \triangleq 0\ ..\ (MaxVers + 1)$

Maximum version number constraint.

$MaxVersConstraint \triangleq \forall\, m \in AllMeta(state) : m.v \leq MaxVers$

Page symmetry.

$PageSym \triangleq Permutations(Page)$

VARIOUS BUGGY DEFINITIONS

Ignore it.

$WouldBuryMeta\_CheckNone(m0,\ m1) \triangleq$ FALSE

Only check $m0.p$ against $m1.p$.

$WouldBuryMeta\_CheckP(m0,\ m1) \triangleq$
  $\lor\ m0.p = m1.p$

Only check $m0.p$ against $m1.np$.

$WouldBuryMeta\_CheckNP(m0,\ m1) \triangleq$
  $\lor\ m0.p = m1.np$

A.3 Model configuration

```
CONSTANT Page = { a, b, c }
CONSTANT MaxVers = 3

CONSTANT Vers <- McVers

INIT Init
NEXT Next

INVARIANT InvType
INVARIANT InvMetaUniq
INVARIANT InvUsed
INVARIANT InvComm

SYMMETRY PageSym

CONSTRAINT MaxVersConstraint
```

A.4 Model configuration - BuryCheckNone

```
CONSTANT Page = { a, b }
CONSTANT MaxVers = 2

CONSTANT Vers <- McVers
CONSTANT WouldBuryMeta <- WouldBuryMeta_CheckNone

INIT Init
NEXT Next

INVARIANT InvType
INVARIANT InvMetaUniq
INVARIANT InvUsed
INVARIANT InvComm

SYMMETRY PageSym

CONSTRAINT MaxVersConstraint
```

A.5 Model configuration - BuryCheckP

```
CONSTANT Page = { a, b, c }
CONSTANT MaxVers = 2

CONSTANT Vers <- McVers
CONSTANT WouldBuryMeta <- WouldBuryMeta_CheckP

INIT Init
NEXT Next

INVARIANT InvType
INVARIANT InvMetaUniq
INVARIANT InvUsed
INVARIANT InvComm

SYMMETRY PageSym

CONSTRAINT MaxVersConstraint
```

A.6 Model configuration - BuryCheckNP

```
CONSTANT Page = { a, b, c }
CONSTANT MaxVers = 2

CONSTANT Vers <- McVers
CONSTANT WouldBuryMeta <- WouldBuryMeta_CheckNP

INIT Init
NEXT Next

INVARIANT InvType
INVARIANT InvMetaUniq
INVARIANT InvUsed
INVARIANT InvComm

SYMMETRY PageSym

CONSTRAINT MaxVersConstraint
```

B BPCC specification

B.1 Specification

MODULE $BPCC$

EXTENDS $Naturals$, $Sequences$, $FiniteSets$

VARIABLE $state$

CONSTANT $Page$ page identifier
$Vers \triangleq Nat$ version number

HELPFUL TLA+ DEFINITIONS

The set of all maps from subsets of D to R . This is like $[D \to R]$ except it includes all subsets of D , that is, partial maps.

$PMAP(D,\ R) \triangleq$ UNION $\{[d \to R] : d \in$ SUBSET $D\}$

Arrange a set into all possible sequences in which each element of the set appears exactly once.

$AllExactOnceSeq(S) \triangleq$
  LET $X \triangleq 1\ ..\ Cardinality(S)$
  IN $\{q \in [X \to S] : \forall\, s \in S : \exists\, x \in X : q[x] = s\}$

Select the maximum from a non-empty set.

$MaxOf(S) \triangleq$ CHOOSE $s0 \in S : \forall\, s1 \in S : s0 \geq s1$

Definition of a metadata record.

$Meta \triangleq$ [
  $p : Page,\ \ v : Vers,$ page and version of this intention
  $c : Vers,$ last committed version of this page
  $np : Page,\ \ nv : Vers$ next page and version in cycle
]

Initial metadata for page p . A singleton cycle of version zero. Back pointer to the last committed version is initialized to zero. This is an exception to the general rule that the back pointer refers to a version earlier than this intention. However it works for initialization, because there can be no earlier version to straddle.

$InitMeta(p) \triangleq$ [
  $p \mapsto p,\ \ v \mapsto 0,\ \ c \mapsto 0,$
  $np \mapsto p,\ \ nv \mapsto 0$
]

Determine that $m0$ is a better straddler than $m1$, one for the same page with the same back pointer but a newer version.

$IsBetterStraddlerThan(m0,\ m1) \triangleq$
  $\land\ m0.p = m1.p$ same page
  $\land\ m0.c = m1.c$ same back pointer

$\land\ m0.v > m1.v$ newer version

Determine that the next-link of $m0$ is straddled by $m1$.

$IsStraddledBy(m0,\ m1)\ \triangleq$

$\land\ m0.np = m1.p$ next page is his page

$\land\ m0.nv > m1.c$ next version after his last committed version

$\land\ m0.nv < m1.v$ next version before his version

Definition of the state of an in-progress transaction. The state consists of a map from the set of pages involved in the transaction to their new version numbers and the set of metadata records planned but not yet written.

$Tran\ \triangleq\ [$

$pv\ \ : PMAP(Page,\ Vers),$ map involved pages to new versions

$remm$: SUBSET $Meta$ metadata yet to be written

$]$

Make a transaction record for the cycle of pages given by the sequence pq , the current last used version numbers given by $used$, and the current last committed version numbers given by $comm$. We examine the sequence to determine the set of involved pages pp . Then we compose the map pv from the involved pages to their new version numbers. Then we construct the metadata that is to be written for page index i in the cycle. Finally put it all together to compose the transaction record.

$MakeTran(pq,\ used,\ comm)\ \triangleq$

LET

$pp\ \triangleq\ \{pq[i] : i \in$ DOMAIN $pq\}$ set of pages

$pv\ \triangleq\ [p \in pp \mapsto used[p] + 1]$ map to new version numbers

$mi(i)\ \triangleq$ metadata planned for index i

LET

$p\ \triangleq\ pq[i]$

$np\ \triangleq\ pq[$IF $i = Len(pq)$ THEN 1 ELSE $i + 1]$

IN $[$

$p\ \mapsto p,\quad v\ \mapsto pv[p],\quad c \mapsto comm[p],$

$np \mapsto np,\ \ nv \mapsto pv[np]$

$]$

IN $[$

$pv\ \ \mapsto pv,$

$remm \mapsto \{mi(i) : i \in$ DOMAIN $pq\}$

$]$

Get the set of pages involved in a transaction. The domain of the map specifies the set of involved pages.

$TranPP(t)\ \triangleq$ DOMAIN $t.pv$

Determine if a given metadata record is part of a given transaction.

$IsMetaPartOfTran(m,\ t)\ \triangleq\ m.p \in TranPP(t) \land m.v = t.pv[m.p]$

Definition of the state of the system. The state consists of a set of metadata records, a set of transactions in progress, the last committed version number for each page, and the last used version number for each page. Note that we do not model the allocation of pages or specifically where the metadata records are stored. Instead, the metadata records are modelled simply as a set.

$State \triangleq [$
$meta$: SUBSET $Meta$, written metadata records
$tran$: SUBSET $Tran$, transactions in progress
$comm$: $[Page \to Vers]$, last committed version
$used$: $[Page \to Vers]$ last used version
$]$

Initial state.

$InitState \triangleq [$
$meta \mapsto \{InitMeta(p) : p \in Page\}$,
$tran \mapsto \{\}$,
$comm \mapsto [p \in Page \mapsto 0]$,
$used \mapsto [p \in Page \mapsto 0]$
$]$

The set of all metadata records written or planned to be written in state s .

$AllMeta(s) \triangleq s.meta \cup$ UNION $\{t.remm : t \in s.tran\}$

Given a set mm of metadata records, the subset that have p as their page or as their next page, respectively.

$MetaP(mm,\ p) \triangleq \{m \in mm : m.p = p\}$
$MetaNP(mm,\ p) \triangleq \{m \in mm : m.np = p\}$

Pages that are available for a new transaction in state s . Although multiple transactions may be in progress at the same time, the sets of pages must not overlap. Hence a page that is involved in a current transaction is not available for launching in a new transaction.

$AvailPages(s) \triangleq \{p \in Page : \forall\, t \in s.tran : p \notin TranPP(t)\}$

Determine if metadata m is obsolete in state s .

$IsObsoleteMeta(m,\ s) \triangleq m.v < s.comm[m.p]$

Determine if metadata m is exposed in state s .

$IsExposedMeta(m,\ s) \triangleq m.v > s.comm[m.p]$

Determine if a planned metadata m may be written in state s . In the $BPCC$ protocol, there is no restriction on writing planned metadata.

$IsWritableMeta(m,\ s) \triangleq$ TRUE

Determine if metadata m is not needed for a straddle in state s . There are two reasons either of which could justify why an intention m would not be needed as a straddle. First, if there is a better straddler than m . Second, if there is no exposed intention that m straddles.

$UnneededForStraddle(m,\ s) \triangleq$
$\lor \exists\, m1 \in s.meta : IsBetterStraddlerThan(m1,\ m)$
$\lor \forall\, m1 \in s.meta : IsExposedMeta(m1,\ s) \Rightarrow \neg IsStraddledBy(m1,\ m)$

Written metadata records that are garbage collectable in state s . Any obsolete intention may be collected if it is not needed as a straddle. Any exposed intention may be collected if it is not part of a current transaction.

$CollectableMetas(s) \triangleq$

$\{m \in s.meta :$
$\quad \lor\ IsObsoleteMeta(m,\ s) \land UnneededForStraddle(m,\ s)$
$\quad \lor\ IsExposedMeta(m,\ s) \land \forall\, t \in s.tran : \neg IsMetaPartOfTran(m,\ t)$
$\}$

Compute the last used version per page based on the set mm of metadata records. Note that a page p can appear as the page and as the next page in a metadata record. In either case, the corresponding version number is in use for that page.

$ComputeUsed(mm) \triangleq$
LET
$vv(p) \triangleq$ all versions of page p mentioned in any metadata
$\{m.v : m \in MetaP(mm,\ p)\} \cup$
$\{m.nv : m \in MetaNP(mm,\ p)\}$
IN
$[p \in Page \mapsto MaxOf(vv(p))]$

Compute the last committed version per page based on the set mm of metadata records. Because of the $BPCC$ invariant, the committed or uncommitted status of any given intention can be determined by comparing its next-link version number nv against the highest version of its next-link page $hm(np).v$.

If $nv < hm(np).v$ then the given intention is committed iff there is no straddler.

If $nv > hm(np).v$ then the given intention is uncommitted.

If $nv = hm(np).v$ then the status is recursively the same as for the next-link intention, with a cycle meaning committed.

$ComputeComm(mm) \triangleq$
LET
Metadata of highest version for page p .
$hm(p) \triangleq$ LET $M \triangleq MetaP(mm,\ p)$
IN CHOOSE $m0 \in M : \forall\, m1 \in M : m0.v \geq m1.v$

Metadata of lowest version $> v$ for page p .
$mG(p,\ v) \triangleq$ LET $M \triangleq \{m \in MetaP(mm,\ p) : m.v > v\}$
IN CHOOSE $m0 \in M : \forall\, m1 \in M : m0.v \leq m1.v$

Having started with page $p0$, determine if version nv of page np is committed by recursively tracing the commit cycle through the metadata records.

If $nv < hm(np).v$ then we know that at least one version $> nv$ for page np exists. Therefore we can compute the least such, that is, $mG(np,\ nv)$. If there is any straddler, this version must be one.

$IsComm[np \in Page,\ nv \in Vers,\ p0 \in Page] \triangleq$
LET
$nostrad \triangleq mG(np,\ nv).c \geq nv$ fails to be a straddler
IN
IF $np = p0$ THEN TRUE ELSE cycle
IF $nv < hm(np).v$ THEN $nostrad$ ELSE link is less than highest
IF $nv > hm(np).v$ THEN FALSE ELSE link is more than highest
$IsComm[hm(np).np,\ hm(np).nv,\ p0]$

Determine the last committed version of page p .

$comm(p) \triangleq$
  IF $IsComm[hm(p).np,\ hm(p).nv,\ p]$
   THEN $hm(p).v$ highest version is committed
   ELSE $hm(p).c$ otherwise what was last committed then
IN
  $[p \in Page \mapsto comm(p)]$

NEXT STATE RELATIONS

Launch a new transaction. We select a non-empty subset of the available pages and arrange them into any cycle. Then we construct a transaction in progress record based on the cycle and the current last used version numbers. We update the last used version number of pages involved in the cycle.

$NextLaunchTransaction \triangleq$
  $\exists\, pp \in$ SUBSET $AvailPages(state)$: any set of available pages
  $\exists\, pq \in AllExactOnceSeq(pp)$: arrange in any cycle
  $\land\ pp \neq \{\}$ must be non-empty
  $\land$ LET
      $t \triangleq MakeTran(pq,\ state.used,\ state.comm)$
    IN
      $state' = [state$ EXCEPT
        $!.tran = @ \cup \{t\}$,
        $!.used = [p \in Page \mapsto$ IF $p \in pp$ THEN $t.pv[p]$ ELSE $@[p]]$
      $]$

Perform an intermediate step in a transaction. We select some unwritten metadata record from a transaction in progress and write it, provided that it is not the final unwritten metadata for that transaction.

$NextInterStepTransaction \triangleq$
  $\exists\, t \in state.tran$:
  $\exists\, m \in t.remm$:
  $\land\ \{m\} \neq t.remm$ not the last unwritten metadata
  $\land\ IsWritableMeta(m,\ state)$ is writable at this time
  $\land$ LET
      $t1 \triangleq [t$ EXCEPT $!.remm = @ \setminus \{m\}]$
    IN
      $state' = [state$ EXCEPT
        $!.tran = @ \setminus \{t\}$, remove t
        $!.tran = @ \cup \{t1\}$, put in $t1$
        $!.meta = @ \cup \{m\}$ put in m
      $]$

Perform the final step in a transaction. We add the last unwritten metadata record from a transaction in progress. Since the transaction is complete the transaction record goes away and each page involved in the transaction has to have its last committed version number updated.

$NextFinalStepTransaction \triangleq$
  $\exists\, t \in state.tran$:

```
∃ m ∈ t.remm :
∧ {m} = t.remm                    the last unwritten metadata
∧ IsWritableMeta(m, state)        is writable at this time
∧ LET
    pp ≜ TranPP(t)
  IN
    state' = [state EXCEPT
      !.tran  = @ \ {t},          remove t
      !.meta  = @ ∪ {m},          put in m
      !.comm = [p ∈ Page ↦ IF p ∈ pp THEN t.pv[p] ELSE @[p]]
    ]
```

Abort a transaction. We delete a transaction in progress.

```
NextAbortTransaction ≜
  ∃ t ∈ state.tran :
  state' = [state EXCEPT !.tran = @ \ {t}]
```

Model a crash reboot. We discard all transactions in progress and compute the last committed version number and last used version number of each page from the written metadata records.

```
NextCrashReboot ≜
  state' = [
    meta ↦ state.meta,
    tran ↦ {},
    comm ↦ ComputeComm(state.meta),
    used ↦ ComputeUsed(state.meta)
  ]
```

Collect some garbage. We can collect any subset of garbage in one action.

```
NextCollectGarbage ≜
  ∃ mm ∈ SUBSET CollectableMetas(state) :   any subset
  ∧ mm ≠ {}                                 non-empty
  ∧ state' = [state EXCEPT !.meta = @ \ mm]
```

Initial state.

```
Init ≜ state = InitState
```

Next state relation.

```
Next ≜
  ∨ NextLaunchTransaction
  ∨ NextInterStepTransaction
  ∨ NextFinalStepTransaction
  ∨ NextAbortTransaction
  ∨ NextCrashReboot
  ∨ NextCollectGarbage
```

INVARIANTS

The state must always be of the proper type.

$InvType \triangleq state \in State$

A given page and version number may appear at most once in the metadata.

$InvMetaUniq \triangleq$
$\forall\, m1,\, m2 \in AllMeta(state) :$
$(m1.p = m2.p \wedge m1.v = m2.v) \Rightarrow m1 = m2$

What we think is the last used version per page must always be consistent with that computed from the metadata records written and planned to be written. The volatile last used version may be greater than that computed from the metadata because of aborted transactions. However, that is still consistent.

$InvUsed \triangleq$
LET
$mm \triangleq AllMeta(state)$ all metadata
$used \triangleq ComputeUsed(mm)$
IN
$\forall\, p \in Page : state.used[p] \geq used[p]$

What we think is the last committed version per page must always be identical to that computed from the metadata records written.

$InvComm \triangleq$
LET
$mm \triangleq state.meta$ written metadata
$comm \triangleq ComputeComm(mm)$
IN
$\forall\, p \in Page : state.comm[p] = comm[p]$

B.2 Model specification

MODULE $McBPCC$

EXTENDS $BPCC,\ TLC$

CONSTANT $MaxVers$

Model checking definition of $Vers$. We have to permit one higher than the maximum value because the type invariant $InvType$ is checked before the constraint is applied.

$McVers \triangleq 0\,..\,(MaxVers + 1)$

Maximum version number constraint.

$MaxVersConstraint \triangleq \forall\, m \in AllMeta(state) : m.v \leq MaxVers$

Page symmetry.

$PageSym \triangleq Permutations(Page)$

Determine that m is a highest version number metadata.

$IsHighestMeta(m,\ s) \triangleq \forall\, m1 \in s.meta : m1.p = m.p \Rightarrow m1.v \leq m.v$

VARIOUS BUGGY DEFINITIONS

Consider that all intentions are unneeded for straddle. This is a bug, because an obsolete intention can be needed for a straddle.

$UnneededForStraddle\_CheckNone(m,\ s) \triangleq$ TRUE

Consider that the only next-links that need to be straddled are those from the highest numbered version. This is a bug, because the highest numbered version could be erased, exposing the next highest version.

$UnneededForStraddle\_CheckHigh(m,\ s) \triangleq$
$\quad \vee \exists\, m1 \in s.meta : IsBetterStraddlerThan(m1,\ m)$
$\quad \vee \forall\, m1 \in s.meta : IsHighestMeta(m1,\ s) \Rightarrow \neg IsStraddledBy(m1,\ m)$

B.3 Model configuration

```
CONSTANT Page = { a, b, c }
CONSTANT MaxVers = 3

CONSTANT Vers <- McVers

INIT Init
NEXT Next

INVARIANT InvType
INVARIANT InvMetaUniq
INVARIANT InvUsed
INVARIANT InvComm

SYMMETRY PageSym

CONSTRAINT MaxVersConstraint
```

B.4 Model configuration - StradGcCheckNone

```
CONSTANT Page = { a, b }
CONSTANT MaxVers = 3

CONSTANT Vers <- McVers
CONSTANT UnneededForStraddle <- UnneededForStraddle_CheckNone

INIT Init
NEXT Next

INVARIANT InvType
INVARIANT InvMetaUniq
INVARIANT InvUsed
INVARIANT InvComm

SYMMETRY PageSym

CONSTRAINT MaxVersConstraint
```

B.5 Model configuration - StradGcCheckHigh

```
CONSTANT Page = { a, b }
CONSTANT MaxVers = 4

CONSTANT Vers <- McVers
CONSTANT UnneededForStraddle <- UnneededForStraddle_CheckHigh

INIT Init
NEXT Next

INVARIANT InvType
INVARIANT InvMetaUniq
INVARIANT InvUsed
INVARIANT InvComm

SYMMETRY PageSym

CONSTRAINT MaxVersConstraint
```