COMP 60621

Three hours

UNIVERSITY OF MANCHESTER
SCHOOL OF COMPUTER SCIENCE

Designing for Parallelism and Future Multi-core Computing

Date: Tuesday 15th January 2019

Time: 14:00 - 17:00

Please select either Paper A or Paper B and then answer ALL Questions for that paper

© The University of Manchester, 2019

Two academic papers are attached for use with the examination.
Otherwise this is a CLOSED book examination.

The use of electronic calculators is NOT permitted.

[PTO]

1. Provide an analysis of **one** of the following two papers**:**

   a) DHTM: Durable Hardware Transactional Memory. ISCA 2018

   or

   b) GraphGrind: Addressing Load Imbalance of Graph Partitioning. ICS 2017

   by answering the following questions:-

   a) What is the problem being addressed? (10 marks)
   b) What is the proposed solution? (12 marks)
   c) What are the assumptions? (6 marks)
   d) How is it evaluated? (12 marks)
   e) What are the limitations? (6 marks)
   f) Overall assessment of paper and possible improvements? (4 marks)

   (Total 50)

END OF EXAMINATION

2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

DHTM: Durable Hardware Transactional Memory

Arpit Joshi
University of Edinburgh
arpit.joshi@ed.ac.uk

Vijay Nagarajan
University of Edinburgh
vijay.nagarajan@ed.ac.uk

Marcelo Cintra
Intel, Germany
marcelo.cintra@intel.com

Stratis Viglas
Google
sviglas@google.com

***Abstract*—The emergence of byte-addressable persistent (non-volatile) memory provides a low latency and high bandwidth path to durability. However, programmers need guarantees on what will remain in persistent memory in the event of a system crash. A widely accepted model for crash consistent programming is ACID transactions, in which updates within a transaction are made visible as well as durable in an atomic manner. However, existing software based proposals suffer from significant performance overheads.**

In this paper, we support both atomic visibility and durability in hardware. We propose DHTM (durable hardware transactional memory) that leverages a commercial HTM to provide atomic visibility and extends it with hardware support for redo logging to provide atomic durability. Furthermore, we leverage the same logging infrastructure to extend the supported transaction size (from being L1-limited to LLC-limited) with only minor changes to the coherence protocol. Our evaluation shows that DHTM outperforms the state-of-the-art by an average of 21% to 25% on TATP, TPC-C and a set of microbenchmarks. We believe DHTM is the first complete and practical hardware based solution for ACID transactions that has the potential to significantly ease the burden of crash consistent programming.

I. Introduction

The emergence of byte-addressable non-volatile memory technologies [1], [2], [3], [4], also known as *persistent memory*, is fast blurring the divide between memory and storage. Being directly attached to the memory bus, persistent memory provides a high-bandwidth and low-latency alternative for durability. However, merely providing a fast non-volatile medium will not suffice. Programmers need guarantees about what will remain in persistent memory upon a crash or a failure.

In a recent study, Marathe et al. [5] highlight the numerous challenges in designing crash consistent programs and advocate for systematic programming models such as transactions that provide ACID guarantees (Atomicity, Consistency, Isolation and Durability). ACID essentially implies that updates within a transaction are made visible (to other transactions) as well as durable (to a non-volatile medium), in an atomic manner. While the database community has developed a plethora of techniques to guarantee ACID efficiently, these techniques have predominantly been developed with slow block-based media in mind. When applied to in-memory settings, such techniques tend to spend a significant amount of time on concurrency control [6], [7], [8] and logging [6], [9], [10]. This leads us to ask the question: How fast can we enforce ACID in the presence of fast persistent memory?

Related Work. Recently, there have been multiple proposals for providing ACID updates to persistent memory. These proposals are classified in Table I based on how they enforce atomic visibility and atomic durability. The first class of designs [11], [12], [13], [14] support atomic durability via software logging by employing flushing and ordering instructions. Ensuring atomic durability in software, however, comes at a significant performance cost [15], [16], [17], [18], [19] which motivated the development of the second class of designs that either employ hardware support for atomic durability [15], [17], [20], [21], [22], [23] or leverage hardware support for ordering to guarantee atomic durability [16], [18], [24], [25]. However, both of these classes enforce atomic visibility in software using software transactional memory (STM) or locks.

Another approach to ACID is to leverage commercially available Hardware Transactional Memory (HTM) to support atomic visibility, which is the focus of the remaining classes of designs. However, current commercially available HTM systems have two limitations. First, they efficiently support only small transactions [26], [27], [28], [29], [30]; if a cache line written within a transaction is evicted from the L1 cache, the transaction must abort. The severity of the problem has been highlighted by a recent study which finds that transactions whose write-set size is larger than 128 cache lines (quarter of the L1 size) are highly likely to abort [31]. This L1 limitation can significantly limit usability and efficiency for ACID transactions, which tend to have relatively large write working-set sizes (Section V). Second, HTM systems only provide ACI guarantees, i.e., atomic visibility but not atomic durability. To guarantee ACID, the third class of designs [13], [32], [33] leverages the HTM for atomic visibility and integrates it with software support for atomic durability. The latter requires the writing of a log entry for every modified object within the transaction, thereby increasing the transaction's write set (and the abort rate). The fourth class supports ACID by integrating HTM with hardware support for durability. However, PTM [34] (the only proposal in this class) not only introduces significant changes to the cache hierarchy, but also continues to suffer from the L1 limitation.

Our Approach. Our primary goal is to design an HTM that

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00045

IEEE computer society

Designs	Atomic Visibility	Atomic Durability	Trans. Size	LLC Extensions
Atlas [11], REWIND [12], DudeTM [13], Mnemosyne [14]	Locks or STM	Software	Not limited	None
WrAP [15], DPO [16]*, LOC [17], HOPS [18]*, ATOM [20], [21], [22], Kiln [23], NVHeaps [24]*, DCT [25]*, [35]	Locks or STM	Hardware	Not limited	[15], [17], [21], [23], [24], [35]
DudeTM [13], PHyTM [32], cc-HTM [33], [36]	HTM	Software	L1 limited	None
PTM [34]	HTM	Hardware	L1 limited	Yes
DHTM	HTM	Hardware	LLC Limited	None

Table I: Classification of techniques supporting ACID updates on persistent memory. (* Leverage hardware support for ordering to provide atomic durability.)

can support ACID transactions efficiently. A secondary goal is to extend the supported transaction size by supporting overflows from the L1 cache to the last level cache (LLC) without adding significant complexity to the coherence protocol or the LLC.

One way of achieving these goals is to leverage existing unbounded HTM designs [37], [38], [39] that rely on logging to support overflows and make those logs durable [20]. However, such an approach, where durability is treated as a secondary consideration, will have poor performance as persisting the log and/or the data will be in the critical path.

We advocate an alternative approach in which durability is a first class design constraint. We propose Durable Hardware Transactional Memory (DHTM) in which we integrate a commercial HTM like RTM [28] with hardware support for redo logging. DHTM achieves atomic visibility by leveraging RTM. Whereas for achieving durability, DHTM provides architectural support for transparently and efficiently writing redo log entries to a durable transaction log maintained in persistent memory; the key efficiency enabler here is our novel mechanism for collating and flushing log entries without consuming excessive memory bandwidth. The redo log based design allows us to commit a transaction as soon as all the log entries have been written to persistent memory, without waiting for data to be made durable. DHTM then extends the supported transaction size, by leveraging the same logging infrastructure for also supporting L1 overflows. When the write set of a transaction overflows from the L1 cache, DHTM logs the address of the overflowed cache line and leverages the log to commit (or abort) the transaction. DHTM supports this with minor changes to the coherence protocol and without adding any additional transaction tracking hardware to the LLC. In summary, our key contributions are:

- We propose DHTM, the first complete hardware solution for an ACID compliant transactional memory system which is not bound by the size of the L1 cache.
- We enforce ACID efficiently by leveraging RTM [28] for atomic visibility and by providing atomic durability via hardware support for redo logging. We also propose a mechanism for coalescing log entries to reduce the required memory bandwidth.
- We extend the supported transaction size by allowing for the transaction's write set to overflow from the L1 to the LLC by leveraging the same logging infrastructure for handling these overflows. We accomplish this with only minor changes to coherence protocol.
- Our evaluation shows that DHTM outperforms the state-of-the-art [20] by 21% to 25% on average across TATP, TPC-C and a set of micro-benchmarks.

II. Background

A. Hardware Transactional Memory

From idea inception [40] to mainstream commercial adoption, HTMs have come a long way. Although prior work has explored unbounded transactions, current commercial HTMs predominantly provide only a best effort service, with transaction sizes being limited by the size and associativity of the L1 cache. Below, we briefly describe an HTM system which is similar to state-of-the-art commercial HTM designs [26], [27], [29] and is specifically modelled on Intel's *Restricted Transactional Memory* (RTM) [28] design. Later we briefly discuss designs that support overflow from private caches. For a broader perspective, the reader is referred to Harris et al.'s book [41].

Commercial HTMs. HTMs primarily provide support for three functionalities: buffering the speculative state, tracking read and write sets and detecting conflicts. Commercial HTMs typically buffer speculative state in private caches (typically L1). Each L1 cache line is associated with a *write bit* to keep track of the write set of a transaction. If a cache line belonging to the write set of a transaction is evicted from the L1, the transaction is aborted. Thus, the supported write-set size is limited by the size and the associativity of the L1 cache. Commercial HTMs avoid supporting overflows from the private L1 caches to reduce the design complexity, and in particular that of the LLC.

Similar to the write bit, a *read bit* is also associated with each cache line in the L1 cache. This bit is set when the corresponding cache line is read within a transaction. When such a cache line is evicted, the transaction is typically not aborted, but the address of the cache line is added to a *read-set overflow signature* (also maintained in the L1 cache).

Thus, the read set of a transaction is tracked using both the read bits in the L1 cache and the read-set overflow signature.

Conflict detection happens at the L1 cache, with help from the cache coherence substrate. Specifically, when the L1 receives an invalidate request for a cache line in the read set, or an invalidate/data forwarding request for a cache line in the write set, a conflict is detected, triggering an abort of one of the transactions. What transaction must abort is determined by the conflict resolution policy. Two of the commonly used policies are the requester wins policy [28] and the (first) writer wins policy [29].

Overflow Support. Multiple techniques [37], [38], [39] have been proposed to support write set overflows from private caches. Techniques with lazy version management [38], [39] allow the write set to overflow into a redo log. On a commit, these values need to be copied in-place. Consequently, these techniques stall any transaction that conflicts with a committed transaction that is still copying its updates in-place. Techniques with eager version management, such as LogTM [37], allow the write set to overflow in-place in memory but maintain an undo log that is applied in case of an abort. Therefore, they have to stall transactions that conflict with an aborting transaction that is applying its undo log. Stalling adds significant design complexity as it requires support for retrying requests using a NACK based coherence protocol. Our goal with DHTM is to support overflows from the L1 cache to the LLC while maintaining the simplicity of an RTM like protocol (§III-C).

DHTM performs data updates in the cache and eager conflict detection in the same way as RTM. However, it additionally maintains a redo log in memory for atomic durability and also supports write set overflows to the LLC with minor modifications to the coherence protocol.

B. Crash Consistency

Storage systems [42] and more recently systems with persistent memory [12], [14], [24] have employed write-ahead logging to provide crash consistency. Write-ahead logging operates on the principle that the log entries be made persistent before data values can persist. Software implementations rely on instructions such as *non-temporal store*, *cache-line write-back*, *sfence* and *pcommit* to ensure the required ordering, but suffer from a significant performance overhead in the process [15], [16], [17], [18], [19]. To mitigate this overhead, prior work has proposed hardware support for accelerating ordering [15], [16], [18], [25], [35] and techniques for transparently performing logging in hardware [17], [20], [21]. With the former, programmers still need to insert appropriate barriers between log writes and data writes. In contrast, with hardware logging, the programmer is relieved from the burden of writing log entries. Instead, an interface is provided to demarcate the region of code that needs to execute in a crash consistent manner. The hardware ensures that log entries are transparently written to persistent memory in the correct order.

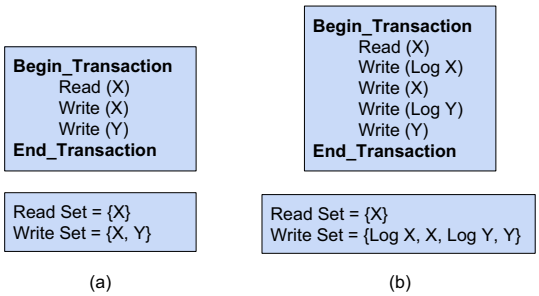


Figure 1: Working set sizes for transactions (a) without including durability log and (b) with durability log.

III. DHTM Design

In this section we present the design of our *durable HTM* (DHTM), that adds support for durability on top of a commercial RTM-like HTM design.

System Model. For the following discussion, we assume a multicore processor with a two level cache hierarchy consisting of private L1 caches and a shared last level cache (LLC). The private L1s are kept coherent using a MESI directory based coherence protocol with forwarding (similar to the one in section 8.2 in [43]). We assume the directory is held in the LLC with each cache line maintaining the coherence state and sharing vector. We assume a baseline HTM similar to Intel's RTM. We assume that the HTM supports strong isolation. Finally, we assume that memory is non-volatile and byte addressable. It is worth noting that the above model is mainly to help anchor our description and as such, none of these choices are fundamental to DHTM.

Overview. At a conceptual level, adding durability to an HTM requires some form of logging. Figure 1a shows a volatile transaction at the top and the corresponding read and write sets at the bottom. The transaction reads X (read set) and writes to X and Y (write set). One way for this transaction to be made durable is by executing the code sequence shown on the top in Figure 1b, which additionally writes log entries for the data being modified. The resultant read set of the persistent transaction remains the same, but the write set consists of $Log\ X$, X, $Log\ Y$ and Y. Thus, adding support for durability essentially doubles the write-set size of transactions. This is a challenge on current RTM-like HTM designs which already limit the write-set size. To compound matters, applications that demand ACID tend to have relatively large transaction sizes. Therefore, one of our goals is to support transactions with a larger write-set size relative to those supported by current commercial HTMs. But in the quest for larger transactions, we do not want to introduce significant hardware complexity; in particular, we do not want to introduce changes to the shared LLC like

adding transaction tracking hardware or searching the LLC for cache lines belonging to the write set – something that current HTM designs avoid.

Our approach is to integrate hardware based redo logging to an RTM-like HTM. For atomic visibility, DHTM leverages the RTM-like HTM and for atomic durability, it employs hardware redo logging. Since logging is performed transparently, DHTM's programming interface is similar to that of volatile transactions (Figure 1a). DHTM's redo logging mechanism leverages the L1 cache write-back interface to dynamically write redo log entries to persistent memory for cache lines being modified within a transaction. Furthermore, DHTM allows dirty cache lines to overflow from the L1 cache into the LLC without causing an abort. This increases the transaction size with minor changes to the coherence protocol and without adding significant design complexity (in particular, without adding transaction tracking hardware to the LLC). Below, we first describe DHTM's hardware logging mechanism. Then, we describe how logging integrates with the HTM, followed by the description on how DHTM manages overflow.

A. Logging for Durability

We ensure atomic durability using *write-ahead logging*. The idea is to maintain a persistent copy of the old and new versions at all times during the transaction, so that the state can be recovered to either of the versions. This persistent copy is maintained in the form of *log entries* which consist of the address and the old or new version of data. In this section we provide a design for a redo-log based implementation to work in conjunction with HTM.

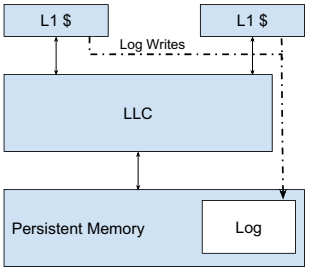
Why Redo Logging? We choose a redo-log based design as it allows us to have both fast commits as well as fast aborts. In volatile transactions, undo logging supports faster commits because, on transaction completion all the in-place updates would have already taken place (in the cache); commit therefore only requires two simple steps: discarding the undo log and flash-clearing the speculative write bits to make the write set visible to other threads. Durable transactions, however, impose additional constraints. Both the undo log and the write set (data) have to be written to persistent memory – only then, can the transaction be committed. While techniques have been proposed for minimizing the fine grained ordering overheads while writing log entries [20], flushing the write set can significantly increase commit time.

Redo logging, in contrast, requires only the redo log to be written to persistent memory at commit time. This is because the redo log, in addition to serving as a recovery log in case of a failure, can also provide the up-to-date values on commit. This allows for the data updates to be written to persistent memory in the background, and out of the commit critical path. One traditional drawback of redo logging is that, because writes are not allowed to overwrite previous values, subsequent reads to those addresses need to be redirected to the redo log. Our proposed hardware based redo-logging mechanism overcomes this limitation by allowing writes to overwrite previous values in the cache. A subsequent read can therefore directly read the updated value from the cache. It is worth noting, however, that the writes do not overwrite the old values in memory but are written to a separate log area. Lastly, aborts are also faster with redo logging and only require two simple steps: discarding the redo log and invalidating the modified lines in the cache.

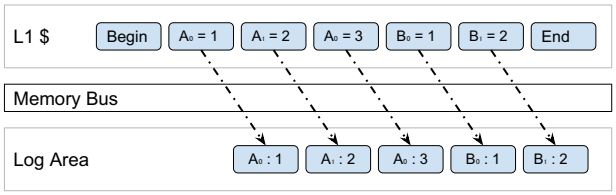
Log management. In the DHTM design, the transaction log space is thread private and is allocated by the operating system (OS) when the thread is spawned. The OS keeps track of all the logs it has allocated so that it can recover transactions from logs in case of a system crash. This per thread transaction log is organized as a circular log buffer similar to Mnemosyne [14]. On a log overflow, DHTM aborts the transaction with an indication that the abort is because of log overflow. The OS in this case allocates a larger log space for the thread and the transaction is retried.

Hardware Support. One of the design goals of DHTM is to write log entries to the transaction log in persistent memory without adding them to the write set. To this end, logging is performed in hardware in DHTM, allowing DHTM to differentiate between log writes and data writes. The L1 cache controller is modified to enable it to write log entries to persistent memory by bypassing the LLC as shown in the Figure 2a. The L1 cache controller creates these log entries on the fly at a word granularity for every store within a transaction. Figure 2b shows the log writes that the L1 cache controller performs.

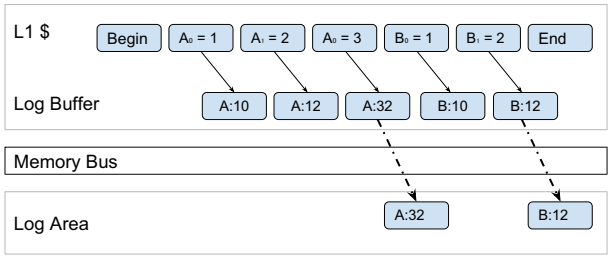
Log coalescing. Writing a word-granular redo-log entry for every store can generate a large number of log entries which can consume significant amounts of memory write bandwidth. Figure 2b highlights this with an example. Let us assume that each cache line consists of two words (all words belonging to cache lines A and B are initially 0); the subscript for each cache line refers to the word in the cache line that is being modified. Performing word-granular logging generates 5 log writes across the memory bus for 5 store requests to different words in cache lines A and B. The bandwidth consumed can be mitigated to some extent by coalescing multiple log entries into one cache line before writing them to memory. Nonetheless, creating a log entry for every store request is problematic. Recall that each log entry is composed of the data and the address (metadata). The finer the granularity of logging, the greater the amount of metadata, which in turn translates into higher bandwidth consumption. Second, logging for every store request might miss opportunities for coalescing multiple stores to the same word via a single log entry. For example, in Figure 2b the word A_0 gets written to twice which leads to the creation of 2 log entries, however only the second log entry would have sufficed.



(a) Log Write Path



(b) Hardware redo log at word granularity. Each redo-log entry consists of (address, new value) pair.



(c) Hardware redo log at cache line granularity using a log buffer.

Figure 2: Redo logging in hardware.

An alternative is to perform logging at cache line granularity. But naively creating a log entry for every store request will only worsen the memory bandwidth consumption. At the same time, the final state of a cache line (at the end of a transaction) must be logged for correctness. If we can predict the final store to a cache line, that would be an opportune moment to log that cache line, since that would minimize the number of entries logged for that cache line. It is important to note that the prediction must be conservative, in that, it must not miss the last store under any circumstance.

We conservatively predict the final store to a cache line via a simple structure called log buffer that is added to the L1 cache. The log buffer is a fully associative structure with a small number of entries that keeps track of cache lines with their cache line addresses. When a store is performed, the corresponding cache line address is added to the log buffer (if not already present). A log entry is written to persistent memory only when an entry is evicted from the log buffer. An entry is evicted from the log buffer under two situations: (a) when the log buffer is full, an eviction has to happen in order to make space for a new cache line address; (b) when an L1 cache line is replaced and the log buffer holds the corresponding address, the address is evicted from the log buffer. Thus, we use eviction from the log buffer as a proxy for predicting the last store to a cache line; in practice, this simple policy works well because write reuse distance (when there is reuse) is typically low for transactional workloads. When an entry is evicted from the log buffer, the redo-log entry for that cache line is created as usual by composing the address with the contents of that cache line from the L1 cache. Then, the redo-log entry is written to persistent memory – in doing so, the stores to one cache line are temporally coalesced, such that all these coalesced stores get only one log write. Finally, at the end of the transaction, all of the cache lines being tracked in the log buffer are logged to persistent memory. It is important to note that this log buffer is different from the log buffer used in LogTM [37]. LogTM uses a buffer to reduce the contention for the L1 cache port and to hide L1 cache miss latency whereas the buffer in DHTM is to coalesce log writes to the same cache line and to predict the last write to a cache line.

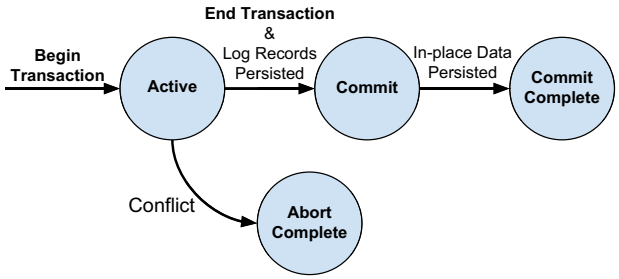
Figure 2c shows the previous example in the presence of a single entry log buffer. Initially the buffer holds cache line A while it is being modified. When cache line B has to be modified, the updated value of cache line A is written to the log area and the buffer now holds cache line B. Eventually when the transaction ends, a redo log entry for B is also written to the log area. In this example, 5 store requests require only 2 log writes over the memory bus.

B. Integrating Logging with HTM

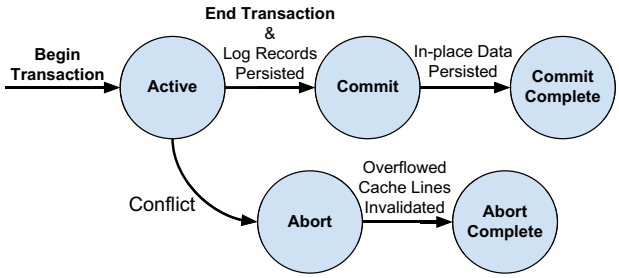
In this section, we will describe how to integrate our logging mechanism with an RTM-like HTM. This section will assume that the transaction will abort on a write-set overflow from the L1; we will handle write-set overflows in the next section.

Overview. Committing a volatile transaction requires that the read-/write-set tracking structures be cleared and that the speculative state be made visible to other threads. In addition to the above steps, in order to commit a durable transaction (with redo logging), the redo-log entries must be written to persistent memory. Recall that the data updates (write set) can be written to persistent memory lazily and out of the commit critical path. Conflict detection works identically to a volatile transaction. Non-transactional accesses also safely integrate with DHTM, similar to RTM, by aborting an ongoing transaction if it conflicts with a non-transactional access. Aborts also are largely identical, with an added step of (logically) clearing the redo log for the transaction. Thus, a durable transaction can be expressed in the form of a state diagram as shown in Figure 3a, with the following states: *Active*, *Commit*, *Commit Complete* and *Abort Complete*. Below, we discuss these in more detail.

Commit. Upon reaching the end of the transaction, and having written all redo-log entries to persistent memory, the



(a) States of a transaction (without overflows).



(b) States of a transaction (with overflows).

Figure 3: Transaction States. A core can start executing subsequent non-transactional instructions after reaching *Commit/Abort* and can start a new transaction after reaching *Commit Complete/Abort Complete*.

transaction effectively *commits*. To mark that the transaction has committed, DHTM writes a *commit log record* to the log area. The L1 cache controller then starts writing back the cache lines belonging to the write set of the committed transaction via the cache write-back interface. DHTM does not flash clear the write bit associated with cache lines on a commit, instead it clears those bits once a write-back is issued for the corresponding cache line. After writing back all the modified cache lines to persistent memory, DHTM marks the transaction as *completed* by writing a *complete log* record to the log area. Writing a complete log record is not a correctness requirement but reduces recovery time on a failure (as we shall see in the section on recovery). Once a transaction has committed, DHTM can start executing non-transactional code following the transaction. But since DHTM has only one set of write bits per cache line, it cannot start executing a new transaction until the previous transaction has completed. This is because, in order to complete a transaction, DHTM relies on these write bits to identify the modified cache lines that need to be written back to persistent memory.

In the DHTM design, there is a window between the commit point of a transaction and its completion point (when the cache lines modified in the transaction are being written back to persistent memory and are being marked as non-speculative) during which a conflict might be detected incorrectly. For example, consider that a transaction T_A tries to modify a cache line X. But X has already been modified by a committed but not yet complete transaction T_B, and has not yet been marked as non-speculative. In such a scenario a conflict will be detected incorrectly. DHTM sidesteps this problem by also consulting the state of the transaction during conflict detection; as the transaction status of T_B indicates that it has committed, DHTM does not raise a conflict in this situation. Additionally, DHTM inserts a sentinel log entry in the transaction log of both T_A and T_B indicating that transaction T_A is dependent on the updates of transaction T_B. This sentinel log entry enables the recovery manager to decide the correct order of replay for transactions with conflicting updates.

Abort. Aborting a volatile transaction requires that the read-/write-set tracking structures be cleared and that the speculative state be invalidated. To abort a durable transaction, in addition to the above steps, the log entries need to be cleared. DHTM logically clears the log entries by writing an abort log record, effectively marking the log entries as being part of an aborted transaction.

Recovery. At the time of failure, a durable transaction can be in one of the following states: *Active*, *Commit*, *Commit Complete*, or *Abort Complete*. The recovery manager does not have to do anything for transactions in Active or Abort Complete state as none of the updates of the transactions would have been written back in-place in persistent memory. In other words, persistent memory has the pre-transaction state for those transactions. For committed but not completed transactions (transactions in *Commit* state), the recovery manager reads the log entries and writes the updated values in-place in persistent memory, thus recovering the updates of the transaction. Finally, for completed transactions (in *Commit Complete* state) the recovery manager does not have to do anything, since all of their updates would have already been written back in-place in persistent memory. In the absence of a complete log record, the recovery manager would have had to copy all the updates from the log area to in-place in persistent memory. Thus the writing a complete log record helps reduce the recovery time.

The replay order of committed but not complete transactions does not matter as long as they do not have conflicting updates. For transactions with conflicting updates, the recovery manager infers the required replay order by looking at the sentinel log entries in the relevant transaction logs.

The recovery manager is implemented as an operating system service which is invoked upon system re-start. As described earlier, the OS keeps track of all the logs it has allocated which it also registers with the recovery manager on creation and de-registers when the the log is deallocated. When the recovery manager is invoked, it scans all the registered logs and restores all the committed but not completed transactions.

C. Handling Overflow

The design described above continues to suffer from the transaction size limitation that is typical of an RTM design. We now describe an extension that allows the write set of a transaction to overflow the L1 cache without aborting the transaction. Consistent with current commercial HTM designs, our proposed extension also does not require expensive operations at the LLC (e.g. searching for cache lines belonging to a transaction), making it amenable to commercial adoption. We first summarize the challenges in supporting overflow efficiently and then describe our approach.

Challenges. Commercial HTM designs like RTM allow the read set to overflow the L1 cache. Conflicts are detected with the help of the overflow signature maintained in the L1 cache, which tracks the addresses of cache lines that have overflowed. On a transaction commit or an abort, the overflow signature is cleared; importantly, nothing needs to be updated in the LLC. In contrast, RTM-like designs do not support write-set overflows from the L1 cache. This is because, aborting a transaction requires that the HTM invalidate all the cache lines belonging to the write set. Whereas this can be done in private L1 caches by flash invalidating the cache lines, doing this for a shared structure as large as the LLC is expensive and involves non-trivial changes (indexing and searching operations). With durable transactions, a commit would also require a similar operation at the LLC: all the cache lines that have overflowed must be identified and written back to persistent memory.

Overview. Our DHTM design allows for the write set to overflow from the L1, with minor changes to the coherence protocol and without requiring any structural changes to the LLC. Our key idea is to leverage the redo log (which holds the speculative state of the transaction) for handling write-set overflows, thus obviating the need for expensive changes to the shared LLC.

DHTM handles write-set overflow by allowing for cache lines belonging to the write set to be replaced from the L1 cache to the LLC; in order to enable conflict detection the coherence state of the cache line in the LLC is kept unchanged, however. This ensures that the LLC continues to show the cache line as being owned by the core executing the transaction. Therefore, any coherence message will continue to be forwarded to the owner's L1, wherein a potential conflict can be detected. It is worth noting that our idea of using stale coherence state for conflict detection is similar to the *sticky state* solution used in LogTM [37]. Therefore, the resulting coherence protocol extensions in DHTM are similar to the sticky state extensions of LogTM.

While maintaining stale state in the LLC helps in conflict detection, we also need a mechanism to identify all the cache lines that have overflowed from the L1 cache to the LLC for versioning. To this end, DHTM maintains an *overflow list* along with the redo log in memory. When a dirty cache line overflows from the L1 cache to the LLC, DHTM writes the address of the overflowed cache line to the overflow list. On a commit or an abort, DHTM uses the overflow list to identify cache lines belonging to the write set that have overflowed, and writes them back to persistent memory (in case of a commit), or invalidates the corresponding LLC cache line (in case of an abort).

The recovery procedure remains the same with overflows, since the cache lines belonging to the write set that overflowed the L1 cache are already present in the redo log. In summary, a durable transaction with write-set overflows can be expressed in the form of a state diagram as shown in Figure 3b with the following states: *Active*, *Commit*, *Commit Complete*, *Abort* and *Abort Complete*. One key difference with overflows is that like commits, aborts now require a completion phase. Below, we discuss this in more detail.

Commit. After writing the commit log record to the log area and issuing write-backs for all the cache lines belonging to the write set in the L1 cache, DHTM reads the overflow list corresponding to the committing transaction (recall that the overflow list contains addresses of all the dirty cache lines that have overflowed from the L1 cache). It then sends write-back messages for those cache lines to the LLC. On receiving a write-back request, the LLC writes back the relevant cache line in-place in persistent memory and also transitions the cache line to a clean state and clears its sharer vector. After writing back all the cache lines in-place in persistent memory, DHTM writes a *complete* log record to the transaction log and then transitions the transaction status to *Commit Complete*. This completion operation ensures that the LLC correctly reflects the status of overflowed write-set cache lines belonging to the transaction and eliminates any need for LLC modifications to clear such state.

Conflict Detection. Recall that conflicts are detected at the L1 controller, by checking coherence requests against the read/write bits associated with cache lines or the read overflow signature. In order to enable conflict detection in the presence of write-set overflows, we need to ensure that coherence requests for the overflowing cache lines continue to reach the L1 cache. To this end, when a dirty block overflows from the L1, the coherence state of the LLC is kept unchanged. Specifically, when an L1 cache receives a Fwd-GetM request or a Fwd-GetS request for a cache line that is not present in the cache, DHTM infers that the request corresponds to a cache line that has overflowed from the L1. Therefore, a conflict is detected and one of the transactions is aborted based on the conflict resolution policy.

Abort. To abort a transaction, DHTM first invalidates the cache lines belonging to the write set in L1 as described in Section III-B. But additionally, the cache lines in the LLC that overflowed from the L1 will also need to be invalidated. Therefore, in the presence of overflows, abort also has a completion phase as shown in Figure 3b.

In the completion phase, DHTM reads the cache line addresses from the overflow list in the transaction log and issues invalidate requests for those cache lines to the LLC. The LLC invalidates the cache lines on receiving an invalidate message. Similar to the completion phase of commit (Section III-B), the completion phase for abort can continue in parallel with the execution of other non-transactional instructions, but a subsequent transaction cannot begin until this phase completes. However, differently from a commit, DHTM does not (need to) write an abort complete log record for an aborted transaction as the state that needs to be invalidated is in volatile caches and will anyway be lost on a system crash.

One corner case concerns cache lines that have been reread back into the L1 during the transaction, after overflowing from the L1 to the LLC. When such a transaction aborts, these reread cache lines must be identified as belonging to the write set and invalidated by the L1. Such reread cache lines are correctly identified by DHTM as follows. On an LLC read, DHTM will look at the state of the cache line and the sharer vector; if the cache line is dirty and its state is in *modified* state with the requester marked as its owner, DHTM will identify the cache line as belonging to the write set and will set the write bit in the L1. This ensures that such reread cache lines are invalidated on an abort.

It is worth noting that, as opposed to existing proposals for supporting overflow (such as LogTM) DHTM does not stall requests from other transactions. Consider the case where transaction T_A has modified cache line X which then overflowed to the LLC. T_A is subsequently aborted because of a conflict. While T_A is in the process of aborting, another transaction T_B issues a read for cache line X. Because of the eager version management of LogTM, the read for X cannot be completed until X has been reverted to a non-speculative state from the undo log. Therefore, LogTM would NACK the read request for X while waiting for the abort process of T_A to end and T_B will have to subsequently re-issue the read. This adds significant complexity to the coherence protocol. DHTM on the other hand has non-speculative data in memory because it maintains a redo log for atomic durability. Therefore, it can immediately complete the read for X by fetching it from memory. In summary, DHTM maintains the simplicity of an RTM like design while allowing for overflows from the L1 to the LLC.

IV. Putting it Together

In this section we first explain through detailed examples, the life cycle of a transaction. We also quantify the overall hardware overhead and finally describe a software fallback mechanism for transactions that do not fit in DHTM.

Transaction Lifecycle. Figure 4 shows the life cycle of a transaction for both commits (e and f) and aborts (g and h). For this example, let us assume a dual-core processor. Each figure shows three views: (i) L1 view from the perspective of core 1, showing L1 cache lines with their read/write bits, a single entry log buffer, the read overflow signature and transaction status register; (ii) LLC view, showing for each cache line, its coherence state, sharer vector and dirty bit; and (iii) Persistent memory view, showing the overflow list, log area and in-place values of cache lines in memory.

(a) Initial state. The transaction is in *Active* state and has already modified cache line A with a value of 15 and has already read cache line B. No transactional data has overflowed from the L1. The LLC has cache lines A and B owned by the core 1 in *modified* state.

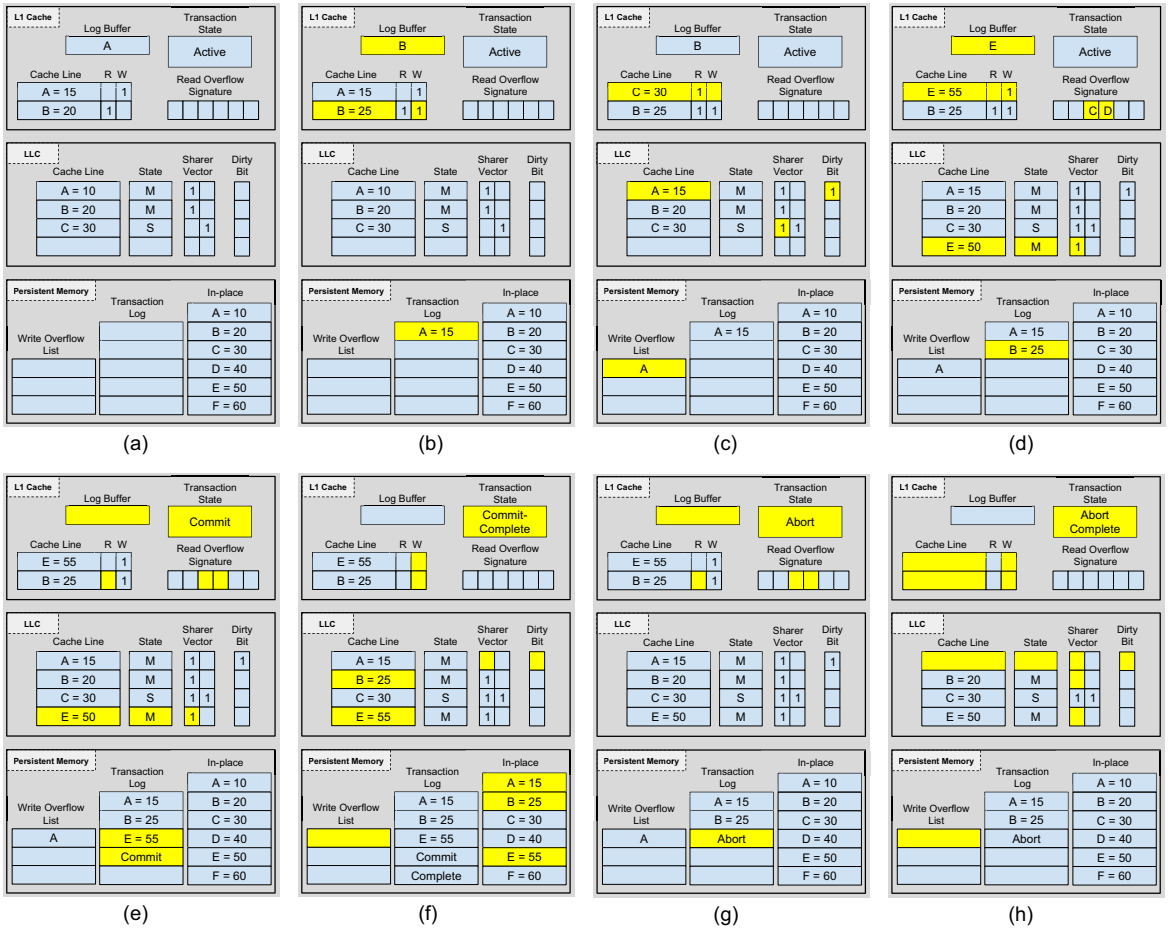
(b) Write B. The transaction modifies the value of cache line B to 25. Therefore, DHTM sets the write bit for cache line B. Also, cache line A needs to be evicted from log buffer (to make space for cache line B), so its updated value of 15 is written to the redo log area.

(c) Read C. The transaction reads cache line C, because of which cache line A gets replaced from the L1 cache. Therefore, cache line A is written back to the LLC and its dirty bit is set, but the coherence state of cache line A is not changed. Also, the address of cache line A is written to the overflow list in memory. Cache line C is present in the LLC in shared state, its sharer vector is updated to add core 1 as a sharer and the cache line is brought to the L1 with its read-bit set.

(d) Write E. The transaction writes 55 to cache line E. This leads to cache line C being replaced from the L1. Before being replaced, the address of cache line C is added to the read overflow signature (but because of inherent imprecision, let us assume that the signature conservatively shows both C and D as its members). Since cache line E is not present in the cache hierarchy, it is brought to the LLC in *modified* state with the sharer vector showing core 1 as the owner. The cache line is also added to the L1 cache where it is updated and its write bit is set. Since cache line E also needs to be added to the log buffer, cache line B is removed from the buffer and its updated value is written to the transaction log.

(e) Commit. When the transaction commits, cache line E is written from the L1 to the transaction log and a commit log record is also written to the log. Simultaneously, the read bits and the read overflow signature in the L1 are cleared and the transaction state is updated to *Commit*. The transaction commits at this point and core 1 may continue executing non-transactional instructions.

(f) Commit Complete. In the commit completion stage, the L1 writes back cache lines B and E to the LLC and clears their respective write bits. On receiving the write-backs, the LLC updates them and also writes them back to persistent memory. Then the memory controller reads the overflow list and issues a write-back request to the LLC for cache line A. The LLC on receiving the request, clears the sharer vector and dirty bit for cache line A and writes it back in-place in persistent memory. Finally, a complete log record is written to the log area, the overflow list is cleared and



(a) **Initial state** of the transaction. The figure shows cache lines with read/write bits in the L1 cache along with the log buffer, the read overflow signature and the transaction state (*Active*). In the LLC it shows cache lines with their state and sharer vector. Finally, in persistent memory it shows the write overflow list, the transaction log and in-place values for various cache lines. The read set of the transaction consists of cache line B and the write set consists of cache line A which is also being tracked in the log buffer.

(b) **Write B**. The transaction modifies the value of cache line B to 25. Therefore, DHTM sets the write bit for cache line B. Cache line B is added to the log buffer by evicting cache line A, for which a log entry consisting of its updated value (15) is written to the redo log area.

(c) **Read C**. The transaction reads cache line C which is present in a shared state in the LLC. Its sharer vector is updated to add core 1 as a sharer. The cache line is added to the L1 cache with its read bit set. Moreover, cache line A is evicted from the L1 cache and is updated in the LLC where its dirty bit is set but its coherence state is not updated. Also, the address of cache line A is written to the overflow list in memory.

(d) **Write E**. Transaction writes 55 to cache line E which is brought from memory and added to LLC in an exclusive state and is also added to the L1 cache with the write bit set. Moreover, it is also added to the log buffer in L1 by evicting B for which a log entry consisting of its updated value (25) is written to the transaction log. Also, since C is replaced from the L1 cache its address is added to the read overflow signature, but because of its inherent imprecision let us assume that it shows both C and D as its members.

(e) **Commit**. When the transaction commits, a log entry for cache line E, tracked in the log buffer, is written to the transaction log followed by a commit record. Simultaneously, the read bits in the L1 cache and the read overflow signature are cleared and the transaction status is changed to *Commit*.

(f) **Commit complete**. After commit, the L1 cache updates the value of cache line B and E in the LLC and persistent memory and clears their write bits. Then, the memory controller reads the write overflow list and issues a write back request to the LLC for cache line A. The LLC clears the sharer vector and dirty bit of cache line A and writes it back in-place in persistent memory. Finally, DTM writes a complete log record to the transaction log, the overflow area is cleared and the transaction status is updated to *Commit Complete*.

(g) **Abort**. If instead the transaction is aborted after step (d), the read bits, the read overflow signature and the log buffer in L1 are cleared. An abort log record is written to the transaction log and the transaction status is changed to *Abort*.

(h) **Abort complete**. L1 invalidates cache lines B and E and sends an invalidate message for them to the LLC. The LLC then clears the sharer vector for those cache lines. Subsequently the memory controller reads the write overflow list and sends invalidate message for cache line A to the LLC. The LLC then invalidates cache line A and clears its sharer vector and dirty bit. Finally, the transaction status is changed to *Abort Complete*.

Figure 4: Flow of a transaction

Register	Description
Log Buffer	Tracks cache lines pending log writes
Transaction State	Identify the state of a transaction
Log Area	
Start Pointer	The start address of the log space
Next Pointer	Address to write the next log entry
Size	Size of the log space
Overflow List	
Start Pointer	The start address of the overflow list
Next Pointer	Address to write the next entry
Size	Size of the overflow list

Table II: Hardware Overhead

the transaction state is updated to *Commit Complete*. At this point the transaction has completed and core 1 may begin a new transaction.

(g) Abort. Shows the state of the system if the transaction were to abort after (d). An abort log record is written to the log area, the read bits and the read overflow signature in the L1 are cleared, and the transaction status is updated to *Abort*. The transaction has aborted at this point and core 1 may continue executing subsequent non-transactional instructions.

(h) Abort Complete. In the abort completion phase, the L1 invalidates cache lines B and E belonging to the write set and sends invalidate messages to the LLC which then clears the sharer vector for cache lines B and E. Then the memory controller reads the overflow list and issues invalidate message for cache line A to the LLC. On receiving the invalidate message, the LLC invalidates cache line A and clears its sharer vector and dirty bit. Finally, the transaction status is updated to *Abort Complete* and at this point, core 1 may begin a new transaction.

Hardware Overhead. Table II shows the hardware overhead that DHTM adds on top of an RTM-like HTM design. DHTM adds to the L1 cache a fully-associative structure called the *log-buffer*, for keeping track of cache lines for which redo log entries need to be written to persistent memory. It also adds a *transaction state* register to identify the current state of the transaction. DHTM also adds two sets of registers to keep track of the log area and the overflow list. The registers in each set consist of a *start pointer* to identify the start address of the corresponding area, a *next pointer* to identify the address where the next entry can be written to and finally a *size* register to keep track of the size of each area so that an overflow can be detected.

Fallback Path. DHTM increases the limit for the transaction size from the L1 cache to the LLC. However, if a transaction aborts continually because of a overflow from the LLC (or due to any other reason) then it might not be able to make forward progress. Therefore, a fallback path must be provided. In principle integrating a software fallback path to DHTM is no different from the ones proposed for RTM [44] because both employ a similar mechanism for atomic visibility. In particular, this software fallback path does not interact with hardware logging because, before initiating the fallback path DHTM would abort the transaction (taking it to abort complete state) which clears the log. The only difference is that our fallback will provide both atomic visibility and durability similar to Mnemosyne [14].

Cores	8 In-order cores @ 2GHz
L1 I/D Cache	32KB 64B lines, 4−way
L1 Access Latency	3 cycles
L2 Cache	1MB×8 tiles, 64B lines, 16−way
L2 Access Latency	30 cycles
MSHRs	32
NVM Access Latency	360 (240) cycles write (read)

Table III: System Parameters

Workload	Description	Write Set
TPC-C	Online transaction processing	590
TATP	Mobile carrier database	167
Queue	Insert/delete entries in a queue	52
Hash	Insert/delete entries in a hash table	58
SDG	Insert/delete edges in a scalable graph	56
SPS	Random swaps between entries in an array	63
BTree	Insert/delete nodes in a b-tree	61
RBTree	Insert/delete nodes in a red-black tree	53

Table IV: Benchmarks used in our experiments along with their descriptions and write-set sizes (# cache lines).

V. Experimental Setup

We now describe our simulation infrastructure, system configuration, benchmarks and designs that we evaluate. We implemented DHTM on the gem5 [45] simulator with Ruby. We extend the Ruby memory model to implement DHTM functionality, with a log buffer size of 64 entries. We evaluate DHTM on an 8-core multicore (one thread per core) with each core containing a 32 KB private L1 and a multi-banked LLC. The local L1s are kept coherent using a MESI based directory protocol. DHTM is built on top of HTM that is based on an RTM-like implementation. Conflicts are detected by piggybacking on top of the coherence protocol and the HTM employs a first-writer wins conflict resolution policy similar to IBM POWER8 [29]. However, it is important to note that the choice of conflict resolution policy is not fundamental to DHTM design and it can be implemented with other policies (like requester wins) as well. Table III shows the main parameters of our system. The peak memory bandwidth in our setup is 5.3 GB/s.

Workloads and their Characteristics. We considered two classes of workloads for our study. TPC-C and TATP (the first two rows from Table. IV) are traditional online transaction processing (OLTP) workloads that require ACID guarantees. We use in-memory implementations of these

workloads [12]. It is worth noting that the OLTP workloads have write working-set sizes exceeding or comparable to the size of the L1 cache. Indeed, the write-set size of TPC-C (37 KB) exceeds the L1 cache size (32 KB) and can cause both capacity and conflict L1 misses which in turn can cause aborts when run on an HTM. Although TATP has a write-set size of around 10 KB, we find that there are significant conflict misses, which can lead to aborts.

The second class of workloads (the last six rows from Table. IV) are micro-benchmarks that perform atomic search, insert and delete operations on the corresponding data structure. The micro-benchmarks are similar to those in the benchmark suite used by NVHeaps [24]. We evaluate each of these micro-benchmarks with a data set size of 3 KB, similar to ATOM [20]. It is worth noting that the write-set sizes of the micro-benchmarks are significantly smaller than the L1 size, in contrast to the OLTP workloads.

Evaluated Designs:

- **SO**: This *software only* design uses locks for atomic visibility and software logging for atomic durability. For the OLTP workloads, we use the default software concurrency control mechanism which uses fine-grained locking. For the micro-benchmarks, we partition the data-structure into coarse-grained partitions with a lock associated with each partition, to allow for concurrency across the different partitions. We use a software logging mechanism similar to Mnemosyne [14], wherein log entries are flushed synchronously as soon as their values are finalized (thus benefiting from coalescing as well).
- **sdTM**: This design (software durability + hardware transactional memory) is based on PHyTM [32], which uses HTM similar to Intel's RTM for atomic visibility. We disable the software concurrency control mechanism in the benchmarks and instead enclose each transaction within a hardware transaction. We use software logging similar to Mnemosyne for atomic durability.
- **ATOM**: This design uses locks (similar to the **SO** design) for atomic visibility. It uses the state-of-the-art hardware undo logging mechanism for atomic durability [20].
- **LogTM-ATOM**: This design uses LogTM [37] like HTM for atomic visibility and integrates it with ATOM [20] for atomic durability. It is worth noting that this represents a new design that has not been studied previously.
- **DHTM**: This is our proposed design which supports atomic visibility by using HTM similar to Intel's RTM and atomic durability by using hardware based redo logging. It also allows the write set to overflow from the L1 to the LLC.

VI. Results

In this section, we quantitatively compare the performance of the evaluated designs on the micro-benchmarks. We then present studies to better understand where the DHTM gains are coming from. One important parameter that could affect the efficacy of DHTM is the size of the log-buffer. Therefore, we quantify its impact. We also evaluate the efficacy of the designs on TPC-C and TATP workloads. Finally, we analyze the overheads of persistence by comparing our design with a non-persistent design.

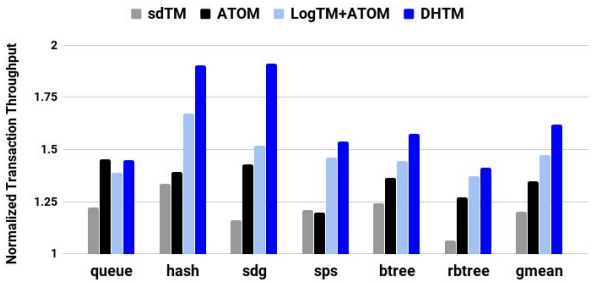


Figure 5: Transaction throughput normalized to SO.

	queue	hash	sdg	sps	btree	rbtree	Ave.
sdTM	68	19	23	27	37	46	37
DHTM	46	5	13	16	18	26	21

Table V: Abort rates for sdTM and DHTM designs.

A. Transaction Throughput

Figure 5 shows the transaction throughput of all the evaluated designs normalized to the software-only (SO) design, on the micro-benchmarks. As we can see, sdTM provides an average throughput improvement of 20% over SO. Recall that sdTM uses HTM for concurrency control which can potentially uncover more concurrency, especially in workloads where locking is coarse-grained. On the other hand, sdTM can suffer the negative effects of rollbacks in situation where the HTM aborts frequently. Table V shows the abort rates for the workloads for the sdTM design. In general, we can observe a correlation between the abort rates experienced by various workloads and the throughput improvement over SO. In particular, for the *rbtree* workload which experiences a significant 46% abort rate, sdTM provides only a minimal 5% improvement over SO.

We can also observe than ATOM provides a more robust average improvement of 35% over SO. Recall that ATOM uses the same concurrency control mechanism as SO (locks), but provides faster atomic durability by performing undo logging in hardware. Interestingly, we can see that ATOM has a comfortable 15% advantage over sdTM because of the aborts experienced by the latter.

Our DHTM design provides the best throughput improvement amongst all competing designs. On average, DHTM improves transaction throughput by 61% compared to SO. In comparison to SO, it can achieve faster durability by way of hardware logging and can also uncover more concurrency. In comparison with sdTM, DHTM improves transaction

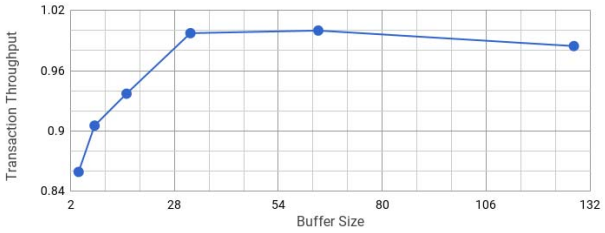


Figure 6: Normalized transaction throughput sensitivity towards log-buffer size for hash benchmark.

throughput by 41%. It not only benefits from faster durability, but also benefits from fewer aborts because DHTM supports write-set overflows from the L1. This is evidenced in Table V, where we can see that DHTM suffers from a relatively lower 21% abort ratio in comparison with the 37% abort ratio of sdTM. In comparison with ATOM, DHTM provides a 26% higher improvement in transaction throughput. DHTM not only benefits from better concurrency (because of HTM), but also because logging is faster in DHTM. Indeed, because ATOM uses undo logging, it suffers from the overhead of persisting data in-place in the commit critical path. In contrast, because DHTM uses redo logging, data can be persisted out of the critical path.

Finally, we look at LogTM-ATOM which implements eager version management for both atomic visibility and atomic durability. On average, DHTM provides 17% higher improvement in throughput compared to LogTM-ATOM. Since both designs leverage HTM for atomic visibility, the difference in performance between the two designs is primarily because of difference in atomic durability mechanisms. In other words, the low transaction commit latency in DHTM enabled by the redo log leads to performance improvement over LogTM-ATOM. From this we can also infer that more than half of the DHTM's 26% improvement over ATOM is because of faster durability (and the rest owing to higher concurrency). In summary, DHTM provides a significant performance improvement, because of faster logging and/or integration with HTM, over the state-of-the-art design (ATOM [20]) and over a (novel) design combining LogTM [37] with ATOM.

B. Sensitivity to the size of the log-buffer

Figure 6 shows the impact of the size of log buffer on the performance of DHTM for the hash benchmark (other benchmarks show similar trends). We run the benchmark with buffer sizes ranging from 4 through 128 entries. As the number of entries are increased, the throughput increases, saturates at the size of 64 entries (default configuration in DHTM) and then marginally reduces upon further increase in the size. A small persist buffer size leads to creation of multiple redo log entries which consumes higher amount of memory bandwidth and adversely impacts other memory requests. On the other hand, a larger buffer delays log writes which results in those log writes happening in the critical path of commit. Recall that a transaction cannot commit until all the redo log entries have been made persistent. In summary, we find that a 64-entry log-buffer provides the best coalescing effect.

	SO	ATOM	DHTM
TPC-C	1	1.67	1.88
TATP	1	1.27	1.53

Table VI: Transaction throughput for ATOM and DHTM normalized to SO for TPC-C and TATP benchmarks.

	1×	2×	10×
NP	2.9	3.0	3.3
DHTM	1.9	2.4	3

Table VII: Transaction throughput for NP and DHTM normalized to SO for hash benchmark with varying memory bandwidth.

C. TPC-C and TATP Throughput

Table VI shows the transaction throughput of ATOM and DHTM normalized to the throughput of SO for the TPC-C and TATP workloads. We have not shown sdTM results because it performs quite poorly. Because of the significant number of HTM aborts (owing to the large write working-set size of the OLTP workloads), these conventional HTM designs revert to the software concurrency mode often.

As we can see, DHTM continues to provide impressive speedups, not only over the SO baseline, but also over ATOM. Specifically, for the TPC-C workload, DHTM provides an 88% improvement over SO and 21% higher improvement compared to ATOM. For the TATP workload, DHTM provides a 53% improvement over SO and 26% higher improvement compared to ATOM.

D. The Cost of Atomic Durability

In this section, we wanted to see how close DHTM is compared to a non-persistent (volatile) HTM design that we call NP. For micro-benchmarks, we find that NP provides 2.2× higher transaction throughput compared to SO which is 59% better than DHTM.

Next, we wanted to understand better the reason for the performance gap. Are their inefficiencies in DHTM or is it fundamentally limited by the cost of durability? There are two primary sources of overheads in DHTM compared to NP. First, the overhead of log writes and data writes that are in the critical path. These include log writes that are pending when a transaction execution completes and is waiting to commit and data writes from the committed but yet to complete transaction pending when a core encounters the next transaction. To evaluate the performance impact of

these overheads we implemented a DHTM design where these writes happen instantaneously. This design is able to improve performance over DHTM by 16% for micro-benchmarks Therefore log/data writes in the critical path of execution appear not to be the major source of overhead.

The second source of overhead corresponds to a fundamental difference in memory write bandwidth requirements for DHTM and NP. Recall that, in comparison with NP, DHTM needs to flush cache lines for atomic durability. To analyze the impact of this overhead, we performed experiments by varying the available memory bandwidth. Table VII shows the transaction throughput for NP and DHTM designs normalized to SO design for the hash micro-benchmark with varying memory bandwidth. With the baseline bandwidth (5.3 GB/s) the difference between NP and DHTM designs is 100% whereas with $10\times$ the baseline bandwidth the difference is only 30%. Thus in a system with higher memory bandwidth DHTM can achieve performance similar to that of a volatile only (NP) design.

VII. Conclusion

ACID transactions are a well-understood and widely adopted programming model. How fast can we achieve ACID in the presence of fast persistent memory? We have proposed DHTM, a HTM design in which durability is treated as a first class design constraint. It extends a commercial HTM like RTM with hardware support for atomic durability. It supports atomic visibility by employing an RTM like HTM and atomic durability by employing a hardware logging infrastructure which transparently and efficiently writes redo log entries to persistent memory. A redo-log based design allows us to commit a transaction as soon as all the log entries have been made persistent, without waiting for the data to persist.

One of our design goals was to support larger transactions, since ACID transactions tend to be considerably larger than those supported by current L1-limited RTM like HTM designs. But in supporting larger transactions we did not want to introduce significant hardware complexity. In particular, we did not want to introduce changes to the shared LLC – something that current HTM designs avoid. Our key insight here is to reuse the logging infrastructure that is necessary for durability for also supporting L1 overflows. Our experimental results showed that our proposal outperforms the state-of-the-art ACID design by an average of 21% to 25% on TATP, TPC-C and a set of micro-benchmarks.

Acknowledgment

We would like to thank our shepherd, Daniel Sanchez, and the anonymous reviewers for their helpful comments. This work is supported by the Intel University Research Office and by EPSRC grants EP/M001202/1 and EP/M027317/1 to the University of Edinburgh.

References

[1] H. Akinaga and H. Shima, "Resistive Random Access Memory (ReRAM) Based on Metal Oxides," *Proceedings of the IEEE*, vol. 98, no. 12, 2010.

[2] Intel Corporation and Micron, "Intel and Micron Produce Breakthrough Memory Technology," 2015, http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology.

[3] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. M. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno, "2Mb Spin-Transfer Torque RAM (SPRAM) with Bit-by-Bit Bidirectional Current Write and Parallelizing-Direction Current Read," in *Proceedings of the International Solid-State Circuits Conference*, 2007.

[4] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. Chen, H. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4-5, 2008.

[5] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris, "Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory," in *USENIX Workshop on Hot Topics in Storage and File Systems*, 2017.

[6] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "OLTP Through the Looking Glass, and What We Found There," in *Proceedings of the International Conference on Management of Data*, 2008.

[7] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner, "Improving in-memory database index performance with Intel® Transactional Synchronization Extensions," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2014.

[8] V. Leis, A. Kemper, and T. Neumann, "Scaling HTM-Supported Database Transactions to Many Cores," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 2, 2016.

[9] J. Huang, K. Schwan, and M. K. Qureshi, "NVRAM-aware Logging in Transaction Systems," *Proc. VLDB Endow.*, 2014.

[10] T. Wang and R. Johnson, "Scalable Logging Through Emerging Non-volatile Memory," *Proc. VLDB Endow.*, 2014.

[11] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging Locks for Non-volatile Memory Consistency," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications*, 2014.

[12] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "REWIND: Recovery Write-Ahead System for In-memory Non-volatile Data-Structures," *Proc. of VLDB Endow*, vol. 8, no. 5, 2015.

[13] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "DUDETM: Building Durable Transactions with Decoupling for Persistent Memory," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[14] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[15] K. Doshi, E. Giles, and P. Varman, "Atomic Persistence for SCM with a Non-intrusive Backend Controller," in *Proceed-*

ings of the International Conference on High Performance Computer Architecture, 2016.

[16] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated Persist Ordering," in *Proceedings of the International Symposium on Microarchitecture*, 2016.

[17] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *Proceedings of the International Conference on Computer Design*, 2014.

[18] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An Analysis of Persistent Memory Use with WHISPER," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[19] J. Zhao, O. Mutlu, and Y. Xie, "FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems," in *Proceedings of the International Symposium on Microarchitecture*, 2014.

[20] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2017.

[21] E. L. M. Matheus A. Ogleari and J. Zhao, "Relaxing persistent memory constraints with hardware-driven undo+redo logging," University of California, Santa Cruz, Tech. Rep., 2016. [Online]. Available: https://users.soe.ucsc.edu/~jzhao/files/HardwareLogging.pdf

[22] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM," in *Proceedings of the International Symposium on Microarchitecture*, 2017.

[23] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support," in *Proceedings of the International Symposium on Microarchitecture*, 2013.

[24] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories," in *Proceedings of the International Conferenceon Architectural Support for Programming Languages and OperatingSystems*, 2011.

[25] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-Performance Transactions for Persistent Memories," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[26] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman, "ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory," in *Proceedings of the International Symposium on Microarchitecture*, 2010.

[27] R. Cypher, A. Landin, H. Zeffer, S. Yip, M. Karlsson, M. Ekman, S. Chaudhry, and M. Tremblay, "Rock: A High-Performance Sparc CMT Processor," *IEEE Micro*, 2009.

[28] Intel Corporation, *Intel® Architecture Instruction Set Extensions Programming Reference*, 2014.

[29] H. Q. Le, G. Guthrie, D. Williams, M. M. Michael, B. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike, "Transactional memory support in the IBM POWER8 processor," *IBM Journal of Research and Development*, 2015.

[30] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of blue gene/q hardware support for transactional memories," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[31] V. Leis, A. Kemper, and T. Neumann, "Exploiting hardware transactional memory in main-memory databases," in *International Conference on Data Engineering*, 2014.

[32] H. Avni and T. Brown, "PHyTM: Persistent Hybrid Transactional Memory," *Proceedings of VLDB Endowment*, 2016.

[33] E. Giles, K. Doshi, and P. Varman, "Continuous Checkpointing of HTM Transactions in NVM," in *Proceedings of the International Symposium on Memory Management*, 2017.

[34] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen, "Persistent Transactional Memory," *IEEE Computer Architecture Letters*, 2015.

[35] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient Persist Barriers for Multicores," in *Proceedings of the International Symposium on Microarchitecture*, 2015.

[36] E. Giles, K. Doshi, and P. Varman, "Brief Announcement: Hardware Transactional Storage Class Memory," in *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, 2017.

[37] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: log-based transactional memory," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2006.

[38] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2005.

[39] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *Proceedings of the International Symposium on Computer Architecture*, 2005.

[40] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Proceedings of the International Symposium on Computer Architecture*, 1993.

[41] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, 2Nd Edition*, 2nd ed. Morgan and Claypool Publishers, 2010.

[42] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, 1992.

[43] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, 1st ed. Morgan & Claypool Publishers, 2011.

[44] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy, "Improved single global lock fallback for best-effort hardware transactional memory," in *Workshop on Transactional Computing*, 2014.

[45] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, 2011.

GraphGrind: Addressing Load Imbalance of Graph Partitioning

Jiawen Sun
The Queen's University of Belfast
jsun03@qub.ac.uk

Hans Vandierendonck
The Queen's University of Belfast
h.vandierendonck@qub.ac.uk

Dimitrios S. Nikolopoulos
The Queen's University of Belfast
d.nikolopoulos@qub.ac.uk

ABSTRACT

We investigate how graph partitioning adversely affects the performance of graph analytics. We demonstrate that graph partitioning induces extra work during graph traversal and that graph partitions have markedly different connectivity than the original graph. By consequence, increasing the number of partitions reaches a tipping point after which overheads quickly dominate performance gains. Moreover, we show that the heuristic to balance CPU load between graph partitions by balancing the number of edges is inappropriate for a range of graph analyses. However, even when it is appropriate, it is sub-optimal due to the skewed degree distribution of social networks. Based on these observations, we propose GraphGrind, a new graph analytics system that addresses the limitations incurred by graph partitioning. We moreover propose a NUMA-aware extension to the Cilk programming language and obtain a scale-free yet NUMA-aware parallel programming environment which underpins NUMA-aware scheduling in GraphGrind. We demonstrate that GraphGrind outperforms state-of-the-art graph analytics systems for shared memory including Ligra, Polymer and Galois.

CCS CONCEPTS

•Computing methodologies →Shared memory algorithms; *Parallel programming languages;* **•Computer systems organization →Multicore architectures;**

KEYWORDS

graph analytics, graph partitioning, Non-Uniform Memory Access (NUMA)

ACM Reference format:
Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. GraphGrind: Addressing Load Imbalance of Graph Partitioning. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3079079.3079097

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICS '17, Chicago, IL, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. 978-1-4503-5020-4/17/06...$15.00
DOI: http://dx.doi.org/10.1145/3079079.3079097

1 INTRODUCTION

Many important problems in social network analysis, artificial intelligence, business analytics and computational sciences can be solved using graph-structured analysis. There is increasing evidence that large-scale shared-memory machines with terabyte-scale main memory are well-suited to solve these graph analytics problems as they are characterized by frequent and fine-grain synchronization [1, 15, 19, 21, 23, 28]. Recently, graph partitioning has been proposed to isolate memory accesses to specific parts of the graph data. Graph partitioning allows to stage graph data in main memory from backing disk [15] and allows to direct memory accesses to the locally-attached memory node in Non-Uniform Memory Access (NUMA) machines [28]. Moreover, graph partitioning is essential in distributed memory systems to spread the computation evenly across all nodes [9].

Several studies have proposed efficient heuristic partitioning techniques for social network graphs [9, 15], as near-optimal partitioning is excessively time-consuming. A common approach is to partition the edge set with the aim to place an equal number of edges in each partition. This results in balanced computation per partition as many graph analyses perform work proportional to the number of edges [9].

While graph partitioning is a crucial building block for graph analytics, little is known about the various ways in which it affects performance. This paper analyzes heuristic graph partitioning in detail and identifies side effects that limit achievable performance. In particular, we show that graph partitioning incurs an innate performance overhead, which stems from increased control flow and from the decreased connection density of the partitions.

Moreover, we find that partitioning the edge set results in an imbalance in the number of vertices appearing in each partition. Alternatively, partitioning the vertex set results in an imbalance in the number of edges. Thus, significant load imbalance exists between partitions, either for loops iterating over vertices, or for loops iterating over edges.

This paper makes the following contributions:

- We analyze the characteristics of graph partitions and identify how these limit performance.
- We present GraphGrind, a NUMA-aware graph analytics framework that reduces the performance impact of graph partitioning. Key highlights of GraphGrind are an improved graph representation, tuning the partitioning to the characteristics of the algorithm and improving the NUMA memory mapping of key data structures.
- We develop an extension to the Cilk parallel programming language [8, 12] that allows expression of NUMA affinity for parallel loops. Our extension simplifies the design of

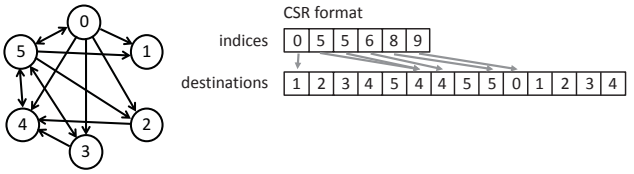


Figure 1: A graph with skewed degree distribution and its representation in CSR format.

GraphGrind and is generally applicable to enforce NUMA-aware scheduling in parallel programs.

- We experimentally evaluate the performance of GraphGrind on 6 real-world graphs and 3 synthetic graphs. We show that GraphGrind improves performance by up to 82% over Polymer and up to 326% over Ligra.

The remainder of this paper is organized as follows. Section 2 introduces the background of graph analytics. Section 3 motivates this work through analyzing the adverse impact of graph partitioning. Section 4 describes the design and implementation of GraphGrind, a graph processing system that significantly reduces the overhead and load imbalance of traversing partitioned graphs. Section 5 presents an experimental evaluation of GraphGrind. Section 6 discusses further related work.

2 BACKGROUND

Graph analytics provide abstract, *vertex oriented* and/or *edge oriented* programming models that iteratively calculate a value associated to a vertex.

2.1 Graph Representation

The two key data structures are graphs and frontiers. A graph $G = (V, E)$ has a set of vertices V and a set of directed edges $E \subset V \times V$ represented as pairs of end-points. A frontier is a subset of the vertices which are active. Graph algorithms visit the destination vertices of the active edges ($\{v \in V : (u, v) \in E \wedge u \in F\}$) and apply an algorithm-specific function to update the value computed for v taking into account the current value for u. This operation is repeated until all values have converged.

Figure 1 illustrates a graph with skewed degree distribution and its representation in the Compressed Sparse Rows (CSR) format [22]. The CSR format stores two arrays: an edge array with IDs of the destination vertices and an index array storing for each vertex the index into the edge array where the destinations of its edges are recorded. The index array has length $|V|$ and the edge array has length $|E|$.

The Compressed Sparse Columns (CSC) representation is analogous and stores the incoming edges to each vertex as opposed to the outgoing edges.

2.2 Edge Traversal

The efficient implementation of graph algorithms is sophisticated and requires deep knowledge of the characteristics of the algorithms. First, the frontier is a set of vertices and may be implemented either as a bitmap or as an array storing vertex IDs. The most efficient implementation depends on the *density* of the frontier [11]. In a *dense frontier* more *edges* are active, while a *sparse frontier* has few active *edges*. The threshold is typically set at 5% active edges.

Secondly, edges may be traversed in *forward* or *backward* manner. In each case, the goal is to traverse the destination vertices of active edges. A *forward* traversal first traverses source vertices $u \in V$ and checks if they are active ($u \in F$). If they are, then their out-going edges are traversed. A *backward* traversal iterates over destination vertices $v \in V$ as well as their incoming edges $(u, v) \in E$. Only then can it check that the source vertex u is active.

Some algorithms execute faster with forward traversal, while others with backward traversal. The distinction is to a large extent motivated experimentally [23]. Beamer et al. motivate the distinction by the number of visited edges [2].

The graph representation is designed for efficient forward *and* backward iteration. Hereto, a dual representation is used for directed graphs (incoming and out-going edges are equal for undirected graphs), i.e., the graph is stored once in CSC format and once in CSR format [23].

ALGORITHM 1: Partitioning by destination

```
input  : Graph G = (V, E); number of partitions P
output : Graph partitions G_i = (V, E_i) for i = 0, ..., P − 1
1 avg = |E|/P;                          // target edges per partition
2 i = 0;
3 for v : V do
4     if |E_i| >= avg and i < P − 1 then
5         ++i;                          // i has exceeded target edges
6     E_i = E_i ∪ in-edges(v);          // i is home partition of v
```

3 MOTIVATION

A low-overhead algorithm to partition the edge set is listed in Algorithm 1 [15, 28]. The graph is partitioned as $G_i = (V, E_i)$ where E_i is a partitioning of E: $\cup_i E_i = E$ and all E_i are non-overlapping. The algorithm assigns each vertex to a home partition such that (i) each partition is home to a range of subsequent vertex IDs and (ii) an edge $(u, v) \in E$ is assigned to the home partition of v. It follows that $E_i \subset V \times V_i$: each partition only has edges pointing to its own home vertices, but the sources may be any vertex.

An often-used criterion for balancing CPU load is to equalize the number of edges per partition, as many graph analytic algorithms perform an amount of work that is proportional to the number of edges. We refer to this partitioning technique as *partitioning by destination* as edges are assigned to the home partition of the destination vertex. Alternatively, *partitioning by source* assigns an edge (u, v) to the home partition of u. Both algorithms achieve nearly the same number of edges in each partition [28].

Figure 2 shows how Algorithm 1 partitions the graph in Figure 1 in two parts. Partition 0 contains 7 edges and is home to vertices 0, 1, 2 and 3. Partition 1 also contains 7 edges and is home to vertices 4 and 5. Figure 1 furthermore

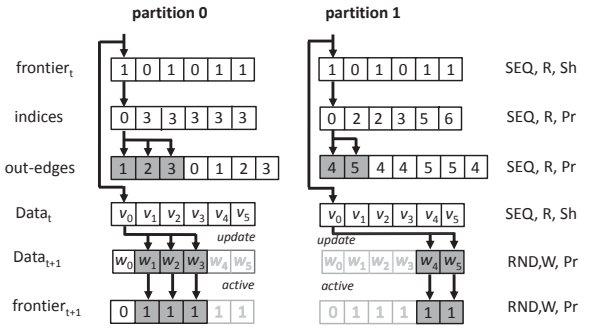


Figure 2: Traversal of the graph in Figure 1 partitioned by destination.

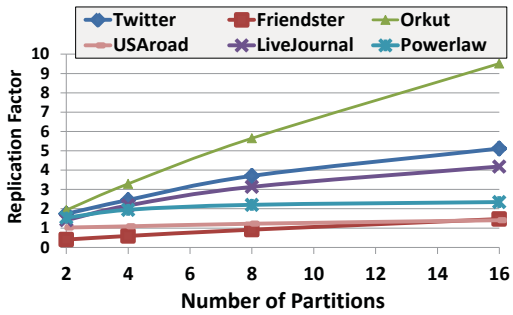


Figure 3: Compressed vertices replication factor varying partition number

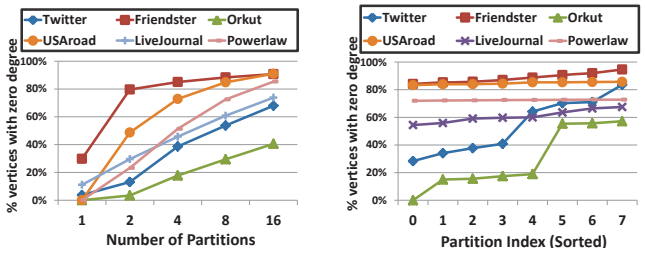


Figure 4: Percentage of vertices with zero out-degree averaged across all partitions (left) and variation across each of 8 partitions (right).

shows how a single traversal of the graph proceeds, assuming a *dense forward* traversal. This traversal first checks whether each vertex is active, i.e., it has a 1 value in the *frontier* array. This is the case for vertex 0, so it traverses the out-edges of vertex 0 in each partition in parallel. It computes updated values for the vertices 1, 2 and 3 in partition 0 and for vertices 4 and 5 in partition 1. It updates the frontier accordingly. Note that each partition updates distinct values as edges with the same destination appear in the same partition.

3.1 Extra Work Induced by Partitioning

When partitioning the edge set, the list of edges of a vertex is split with parts of the list appearing in different partitions. As such, the edges for some vertices are stored in distinct partitions. Graph traversal must thus visit the vertex once for each replication. The additional cost of this is a small amount of control flow, lookups in the graph representation and checking whether the vertex is active. While these actions require only a few dozen assembly instructions, it is important to keep in mind that graph analytics perform little computation, typically less than a dozen assembly instructions per edge. Moreover, the overhead involves several main memory accesses as these algorithms are memory intensive.

Figure 3 shows the average replication factor of vertices for various degrees of partitioning. The graphs are described in Section 5. We show data for 6 of the 9 graphs as the remaining 3 behave similarly. Graphs with few edges per vertex (USARoad and Friendster) have the lowest replication factors while highly skewed graphs (Twitter and Orkut) have the highest. Assuming 4 partitions, replication factors are often in the range 2–3, which implies that the control flow overhead of graph traversal is repeated 2 to 3 times. This results in an instruction count increase of up to 18%.

Figure 3 moreover shows that the graph partitioning algorithm studied in this paper achieves a comparable replication factor for the Twitter graph as the more elaborate algorithm in [9]. We may thus assume that the conclusions of this paper are independent of the partitioning algorithm used, as our conclusions build on the observation that the replication factor is larger than one.

3.2 Sparsity of Graph Partitions

If vertices are not replicated across all partitions, then by necessity vertices will not have incoming or out-going edges in several of the partitions. Figure 4 (left) shows the average number of vertices with zero degree for varying degrees of partitioning by destination. Similar results hold for partitioning by source. The fraction of vertices with zero out-going edges shoots up quickly as more partitions are introduced, exceeding in many cases 50% for 4 partitions. Moreover, real-world social networks have strongly imbalanced partitions (Figure 4 (right)). In contrast, the partitions of synthetic graphs, intended to model real-world graphs, have equal numbers of unconnected vertices in each partition. Interestingly, the Friendster graph has fairly equal partitions. The sparsity of graph partitions leads to an opportunity: if we can avoid iterating over the absent vertices in a partition, then the instruction count increase for these vertices can be restricted only to the partitions where the vertex occurs. To this end, GraphGrind uses a variation of the CSR representation where zero-degree vertices are not recorded.

3.3 Balancing Edges vs. Vertices

It is hard to partition a social network graph in a balanced way due to its skewed degree distribution. Figure 5 shows the relative number of vertices per partition for various graphs and numbers of partitions. Social network graphs like Twitter and Friendster have highly different numbers of vertices per partition when balancing the number of edges.

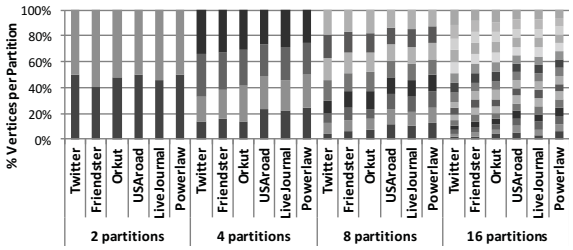


Figure 5: Relative sizes of partitions for varying degree of partitioning.

The imbalance of the number of vertices per partition has an important impact on performance. First, many graph algorithms make passes over vertices apart from passes over the edges. As such, the work performed per graph partition is not only proportional to the number of edges, but also depends on the number of vertices.

Secondly, not all algorithms perform a fixed amount of work per edge. Instead, algorithms such as BFS, betweenness-centrality, Bellman-Ford and K-Core visit at most one active edge per active vertex. For them, balancing the edges between partitions does not result in a balanced CPU load.

Thirdly, an imbalance in the number of vertices per partition results in a skewed utilization of memory and creates hotspots for certain partitions. This unnecessarily drives to scale-out distributed systems to higher degrees of parallelism to drive the worst-case partition size down, even if the computation does not warrant scaling out. In shared memory systems the memory imbalance may be combated by storing data in a sub-optimal NUMA node, which results in the lesser evil of remote NUMA accesses.

Increasing the number of partitions may seem to avoid skewed partitions. This is however not true. As Figure 5 shows, the presence of highly-connected vertices remains an issue with higher degrees of partitioning as some partitions have twice as many vertices as others. We conclude that the graph partitioning needs to balance CPU load and should be adapted to characteristics of the algorithm.

4 GRAPHGRIND: DESIGN AND IMPLEMENTATION

GraphGrind is a NUMA-aware graph analytics framework that builds on the characteristics of graph partitions to optimise the memory layout of graphs and to reduce load imbalance. GraphGrind contains all the required features of graph analytics systems, including hierarchical parallel decomposition of the computation, NUMA-aware data placement and code scheduling [28], balanced vertex-cut partitioning [9] and adapting data structures [11] and search direction [2] to the size of the frontier. We discuss its key features below.

4.1 Application Programming Interface

GraphGrind is compatible with the Ligra programming model. It provides two data types: graphs and frontiers. A frontier is a subset of the vertices in a graph. The key functions apply operations to edges or vertices and calculate new frontiers in the process. They are defined as follows:

- *size()*: For a frontier F, $size(F)$ returns $|F|$.
- The *edge-map()* operator is the main work-horse. It applies an algorithm-specific function to every active vertex in the graph. Its arguments are a graph $G = (V, E)$, a frontier F, a function Fn and a condition C. An edge $(u, v) \in E$ is active if $u \in F$ and $C(v) = true$. The argument Fwd determines whether a forward or a backward traversal is likely to be faster. *Edge-map* returns a new frontier consisting of all visited vertices v for which $Fn(u, v)$ returned a true value.
- *vertex-map()* applies a function Fn to every vertex in the frontier F. It returns a new frontier consisting of all visited vertices u for which $Fn(u, v)$ returned a true value.

We extend the programming interface with a *cache* for ***backward*** *edge-map* traversals. While *edge-map* may execute in parallel, it traverses the incoming edges of a vertex sequentially when the number of vertices is not very large (less than 1000). Compilers should, in principle, be able to hold the intermediate updates for the destination vertex's value in registers. However, the complexity of control flow and pointer aliasing prohibits this in practice. GraphGrind allows the programmer to specify how to cache intermediate updates for the function Fn. This explicit notation allows compilers to allocate them to registers and involves a cache type definition and 3 functions to initialize the cache, to update it and to commit it to the main state.

4.2 Frontier Representation

We adapt the representation of frontiers between bitmaps and arrays of vertex IDs on-the-fly, depending on their density [11]. Frontiers are created either by constructors, or by the *edge-map* and *vertex-map* functions. From the users point of view, frontiers are immutable. One of the constructors creates a frontier containing all vertices. We explicitly record this property in the frontier to omit checks of the frontier and speed up graph traversal. Remember that graph analytics typically perform little work per edge. As such, any reduction in instruction count has a measurable impact.

This optimization affects traversal with dense frontiers. The backward traversal benefits much more from this optimization as it performs more lookups in the frontier, namely once per edge vs. once per vertex in the case of the forward traversal. We similarly optimize the *vertex-map* operation and any auxiliary loop iterating over the frontier.

4.3 Compressed Graph Representation

We modify the CSR and CSC representation to combat efficiency issues with zero-degree vertices. We compress the index array by storing only information for vertices with non-zero degree and store the vertex ID with it. Figure 6 shows the modified CSR format for the graph partitions of Figure 2. In Partition 0, vertex 0 and 5 have out-edges which are home to vertex 0, 1, 2 and 3. In Partition 1, only vertex 1 has zero degree, so it is not stored. The representation reduces the

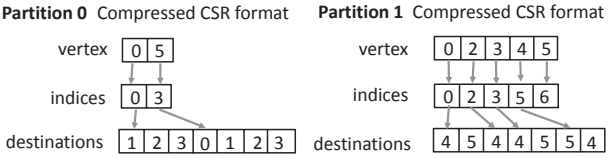


Figure 6: Compressed CSR format.

size of the index array due to the high number of zero-degree vertices. The main benefit, however, is that a sequential edge traversal becomes more efficient as iteration over the index array automatically skips all zero degree vertices.

GraphGrind stores each graph partition in the CSR and CSC representations in order to support the direction-reversing technique. I.e., a *dense forward* traversal uses CSR while a *dense backward traversal* uses CSC. This representation is, however, not efficient for traversals with sparse frontiers as these are dominated by control flow, which is aggravated by the replication of vertices. As such, we store a *non-partitioned* copy of the original CSR representation of the graph specifically for sparse traversals. As such, GraphGrind stores three copies of the graph for undirected graphs, and two copies for directed graphs (as the CSR and CSC representations are equal for directed graphs).

4.4 Partition Balancing Criterion

We have argued that balancing the number of edges across partitions does not necessarily result in the best balancing of CPU time. Instead, some algorithms observe better CPU load balancing when the number of vertices in each partition is about equal. GraphGrind adds a parameter to the algorithm specification that shows its preference for a balanced edge partitioning vs. a balanced vertex partitioning. This parameter is checked during graph ingress in order to select the balancing criterion for graph partitioning. Our balanced vertex partitioning is similar to Algorithm 1, except that we strive for $|V|/P$ destination vertices in each partition.

Balancing vertices is appropriate for 3 of the 8 algorithms that we use in the experimental evaluation. The algorithms are commonly used in prior work. As such, this property is sufficiently important to ask programmers to record it. The property is easily derived from the algorithm specification.

4.5 NUMA Optimization

The state-of-the-art in NUMA-aware programming requires two coordinated actions: (i) data placement and (ii) thread placement. Common data placement strategies are to allocate data in a specific NUMA node or to distribute the data across nodes. Thread placement is optimized such that the thread has a low latency/high bandwidth connection to the NUMA domain holding its most frequently accessed data. This two-pronged strategy allows for many optimizations, such as co-locating threads with data and spreading data and threads across NUMA domains to enhance memory bandwidth.

Graph partitions can enforce NUMA-local access as each partition can be stored and processed within the confines of one NUMA node. Prior work has advocated to replicate frontiers and algorithm-specific data arrays on each NUMA node [28]. Accordingly, memory accesses are NUMA-local, except when interchanging data across nodes.

Table 1: NUMA allocation and binding strategy

Data structure	**NUMA allocation**
full graph	interleaved
graph partition	allocate on one node
vertex arrays	match home partition
Operation	**NUMA binding**
edge-map (sparse)	none
edge-map (dense)	bind to holding node
vertex-oriented loops (e.g., vertex-map)	equally distribute loop iterations over NUMA nodes

GraphGrind follows a different route, which is summarized in Table 1. The full graph is stored in an interleaved fashion over the NUMA nodes. As the full graph is used with sparsely populated frontiers only, the memory accesses are few and hard to schedule optimally. Interleaved allocation provides a good compromise.

Graph partitions are spread over NUMA nodes in such a way that each partition is stored on one NUMA node and all NUMA nodes hold the same number of partitions. A graph traversal over a partition is scheduled on the NUMA node that holds that partition. This ensure that the majority of memory accesses are issued against the local NUMA node.

We distribute *vertex arrays* over NUMA nodes, storing the element for each vertex on the same NUMA node as its home partition. As such, the *edge-map* operation that is *writing* data to a vertex element performs NUMA-local accesses. This placement incurs some *false sharing*, as NUMA placement works on the granularity of virtual memory pages. As such, a small fraction of the vertices will be placed on a remote NUMA node. E.g., assuming 1M vertices, at most 1 in 10,000 will be stored in a different node. The distribution of vertex arrays may be highly skewed due to the imbalance of vertices in each partition. Loops iterating over the vertex arrays, such as *vertex-map* and loops that analyze frontiers, are however scheduled such that the loop iterations are equally spread across NUMA nodes. While this induces some remote NUMA accesses, it is far more important to load-balance these loops than it is to optimize NUMA-awareness.

An alternative strategy is to replicate the vertex arrays on each NUMA node [28]. We found this to be sub-optimal due to the additional memory traffic that is required to replicate and to merge vertex arrays. In contrast, our NUMA placement and scheduling rules guarantee that an *edge-map* operation on a graph partition only writes to vertex array elements stored on the local NUMA node. Read operations may be remote, but these have lower impact on performance. As such, we obtain good NUMA locality without incurring the overhead of replicating data.

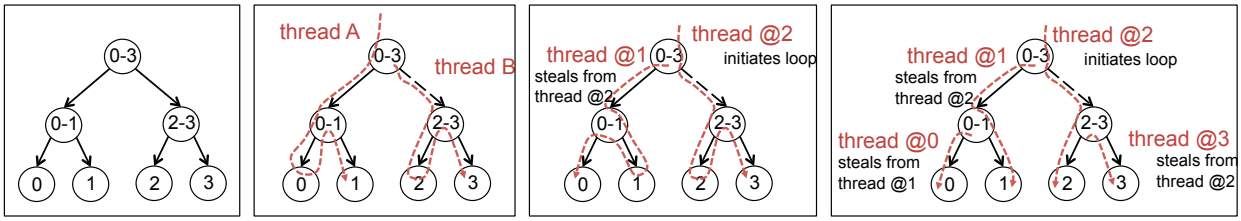


Figure 7: NUMA-aware work-stealing. "Thread @n" represents any thread executing on socket "n".

4.6 A NUMA-Aware Cilk Extension

GraphGrind is built on Cilk [8], an efficient work-stealing scheduler for parallel programs. Cilk, however, is agnostic of the memory hierarchy as it promotes cache-obliviousness [7, 27]. We modify the Cilk language and runtime system to support NUMA-aware scheduling and work stealing. We have deliberately searched for a minimalistic modification as to not affect space- and time-efficiency [3] and implement this in Intel Cilkplus version 1.2 [12]. We delegate a proof of the space and time bounds to future work.

We focus exclusively on parallel loops, which in Cilk are expressed with the `cilk_for` keyword, asserting that all iterations of the loop may execute in parallel. We extend the programming language with a pragma "`#pragma cilk numa(strict)`" that can be supplied immediately preceding a `cilk_for` loop, similarly to the existing *grainsize* pragma. The NUMA pragma indicates that loop iteration i should preferably be executed on cores associated to NUMA domain i. The assumption that the number of loop iterations does not exceed the number of NUMA domains is a pragmatic one. Programmers may split loops over a NUMA-aware outer loop and a normal **cilk_for** innner loop that executes only on the NUMA domain encoded in its calling context.

Cilk implements parallel loops using a helper function that recursively splits the iteration range of the loop in half. Once the iteration range is shorter than a heuristically determined threshold the helper function executes the loop sequentially over this part of iteration range.

Figure 7 (a) shows the call tree of the helper function for a loop with 4 iterations. Each node represents an invocation of the helper function. Edges indicate a parent-child relationship between function calls. Nodes in distinct subtrees are independent and may execute concurrently. Cilk uses a work-first scheduler [3] which translates into a depth-first traversal of the tree (Figure 7 (b)). Idle threads attempt to steal work from a randomly selected victim thread. Threads steal the continuation of the oldest function on their victim's call stack, i.e., the one nearest to the root of the call tree. E.g., if thread A starts execution of the range 0-3 in depth-first order it will first execute the sub-range 0-1. Meanwhile, thread B may steal the continuation of the oldest function and execute the sub-range 2-3.

We provide a NUMA-aware helper function that changes the execution order of loop iterations. The thread that executes an instance of the helper function checks its current NUMA domain and first executes the sub-range that matches its NUMA domain. E.g., if a thread on NUMA domain 2 initiates execution of the loop, it executes the range 2-3 before the range 0-1 (Figure 7 (c)). This strategy is applied recursively: a thread on NUMA domain 3 will first execute loop iteration 3. This way, work is distributed to the correct NUMA domain with a minimal work stealing (Figure 7 (d)).

Work stealing is modified to respect the NUMA constraints. Every dynamic function call is marked by the helper function with the range of NUMA nodes where the function may execute. This range reflects the iteration sub-range of the loop. The range is copied over to recursively called functions. A worker that selects a victim thread inspects the NUMA range of the victim's oldest function and aborts the work stealing attempt if the NUMA range does not contain its own NUMA node. By default, NUMA ranges are not set and work stealing proceeds as normal.

The algorithm is robust against anomalous conditions such as absence of active threads on a NUMA domain and a mismatch between the number of NUMA domains specified by the program and those in hardware. In both cases, pending iterations are executed on sub-optimal NUMA domains.

The NUMA extension supports non-commuting reductions [6] and pedigrees [16]. Both constructs depend on the execution order of function calls, which the helper function disrupts. The solution is beyond the scope of this paper.

5 EXPERIMENTAL EVALUATION

We evaluate GraphGrind on a 4-socket 2.6GHz Intel Xeon E7-4860 v2, totaling 96 threads, with 256 GB of DRAM. We compile all codes using our modified version of the Clang compiler which implements the NUMA extension to Intel Cilkplus [12]. We evaluate 8 graph algorithms (see Table 2) using 9 widely used graph data sets (see Table 3). All reported results are averaged over 5 executions.

5.1 Performance Comparison

We compare the performance of GraphGrind against leading graph analytics systems for shared-memory, namely Ligra[1] [23], Polymer [2][28] and Galois [19] version 2.0 (Table 4). GraphGrind and Polymer both use 4 partitions to match the NUMA characteristics of our hardware. All systems use 96 threads.

[1] https://github.com/jshun/ligra.git
[2] http://ipads.se.sjtu.edu.cn:1312/opensource/polymer.git

Table 2: Graph algorithms and their characteristics. Frontiers: S=sparse, D=dense.

Algorithm	Description	Edge traversal	Frontiers	Cache	Balance
BC	betweenness-centrality [23]	backward	SDS	Yes	Vertices
CC	connected components using label propagation [23]	backward	DS	Yes	Edges
PR	simple Page-Rank algorithm using power method (10 iterations) [20]	backward	D	Yes	Edges
BFS	breadth-first search [23]	backward	SDS	No	Vertices
PRDelta	optimized Page-Rank forwarding delta-updates between vertices [23]	forward	DS	No	Edges
SPMV	sparse matrix-vector multiplication (1 iteration)	forward	D	No	Edges
BF	Bellman-Ford algorithm for single-source shortest path [23]	forward	SDS	No	Vertices
BP	Bayesian belief propagation [28] (10 iterations)	forward	D	No	Edges

Table 3: Characterization of real-world and synthetic graphs used in experiments.

Graph	Vertices	Edges	Type
Twitter [14]	41.7M	1.467B	directed
Friendster [26]	125M	1.81B	directed
Orkut [18]	3.07M	234M	undirected
LiveJournal [26]	4.85M	69.0M	directed
Yahoo_mem [25]	1.64M	30.4M	undirected
USAroad [28]	23.9M	58M	undirected
Powerlaw ($\alpha = 2.0$)	100M	1.5B	directed
RMAT24	16.8M	168M	directed
RMAT27	134M	1.342B	directed

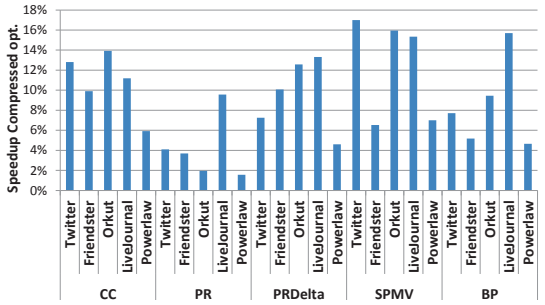


Figure 8: Speedup of compressed graph compared to visit zero-degree vertices.

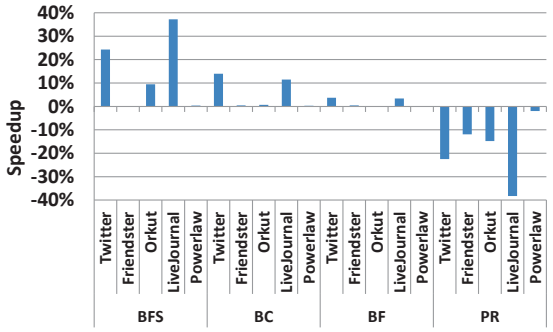


Figure 9: Speedup of balancing vertices compared to balancing edges in graph partitions.

We show the backward PageRank algorithm for Polymer as the forward version, presented in [28], contains errors. The absolute execution times depend on our hardware, compiler version and randomly generated graphs. Moreover, some algorithms are sensitive to the start vertex, which in our experiments is vertex 100 for all graphs. The reported trends match previously reported results.

Overall, GraphGrind outperforms the other systems for all algorithms and all graphs, except for CC and BF on the USAroad graph. In these cases, Galois is faster. This results from using different algorithms [19, 28]. Nonetheless, GraphGrind makes progress over Polymer and Ligra for these cases. In a few cases, GraphGrind performs on par with other systems. These are labeled in bold-face as well.

The performance improvements are significant: up to 326% faster than Ligra (SPMV with Orkut graph) and up to 82.2% faster than Polymer (BP with USAroad graph). The smallest speedups appear for BFS, as there is already little computation going on. The superior performance of GraphGrind results from a combination of optimizations. Next, we will tease out the main contributing factors.

5.2 Compressed Graph Representation

GraphGrind's graph data structure prunes vertices with zero degree from the representation. We will show later that this saves significant spaces compared to the CSC and CSR representations used by Polymer. Moreover, by not storing these vertices, *edge-map* traversals no longer need to visit them. Figure 8 shows the speedup resulting from the graph representation for 5 algorithms, which ranges between 2% and 16%. Twitter and LiveJournal benefit most due to the high sparsity of graph partitions.

5.3 Adapting Graph Partitioning

We remove CPU load imbalance through selecting an appropriate criterion to balance the graph partitions. We identified through code inspection that 3 of the evaluated algorithms (BFS, BC and BF) prefer an equal number of vertices in each partition. The others prefer a uniform number of edges. Figure 9 shows the speedup obtained by balancing vertices over balancing edges for these 3 algorithms and PR. We show results for a subset of the graphs, the remaining graphs behave similar to the ones shown. The partitioning has negligible impact for Friendster and PowerGraph, which have a balanced number of vertices per partition in either case (see Figure 4). Graphs with unbalanced partitions see important improvements with vertex-balanced partitions, with up to 37% speedup for LiveJournal.

Table 4: Runtime in seconds of GraphGrind, Polymer, Ligra and Galois. The fastest results are indicated in bold-face. Execution times that differ by less than 1% are both labeled. Missing results occur as not all systems implement each algorithm. GraphGrind and Polymer use 4 partitions.

Algoritm	Graph	GG	Polymer	Ligra	Galois
CC	Twitter	**1.810**	2.580	2.878	16.660
	Friendster	**5.924**	8.030	7.330	6.210
	Orkut	**0.122**	0.180	0.138	0.311
	LiveJournal	**0.111**	0.177	0.125	0.206
	Yahoo_mem	**0.042**	0.049	0.063	0.046
	USAroad	35.348	36.730	38.910	**20.110**
	Powerlaw	**1.168**	2.110	1.680	3.113
	RMAT24	**0.455**	0.522	0.601	1.440
	RMAT27	**2.305**	3.220	2.444	10.120
BC	Twitter	**1.771**		4.130	4.160
	Friendster	**3.394**		5.490	6.110
	Orkut	**0.149**		0.160	0.178
	LiveJournal	**0.197**		0.334	0.388
	Yahoo_mem	**0.091**		0.110	0.150
	USAroad	**4.402**		5.174	6.010
	Powerlaw	**2.118**		2.300	2.860
	RMAT24	**0.482**		0.503	1.110
	RMAT27	**2.073**		2.360	15.110
PR	Twitter	**15.979**	20.400	23.660	20.120
	Friendster	**38.249**	41.8	43.300	61.200
	Orkut	**1.596**	1.660	2.240	2.120
	LiveJournal	**0.652**	0.688	0.708	0.700
	Yahoo_mem	**0.234**	0.262	0.278	0.255
	USAroad	**0.933**	1.220	1.582	1.180
	Powerlaw	**10.394**	12.716	13.600	11.614
	RMAT24	**2.730**	2.970	3.660	3.110
	RMAT27	**17.517**	23.21	28.600	30.220
BFS	Twitter	**0.254**	0.298	0.319	0.449
	Friendster	**0.896**	**0.899**	1.210	1.330
	Orkut	**0.039**	0.043	0.044	0.051
	LiveJournal	**0.050**	0.068	0.078	0.103
	Yahoo_mem	**0.025**	0.026	0.033	0.363
	USAroad	**1.750**	1.855	2.009	5.180
	Powerlaw	**0.595**	0.601	**0.599**	0.993
	RMAT24	**0.104**	0.119	0.118	**0.104**
	RMAT27	**0.412**	0.421	0.429	0.631

Algoritm	Graph	GG	Polymer	Ligra	Galois
PRDelta	Twitter	**20.560**	24.120	29.890	
	Friendster	**36.097**	36.600	62.100	
	Orkut	**1.244**	1.310	3.472	
	LiveJournal	**1.013**	1.110	1.138	
	Yahoo_mem	**0.831**	1.094	1.640	
	USAroad	**2.124**	2.260	2.905	
	Powerlaw	**10.659**	14.100	16.900	
	RMAT24	**1.845**	2.230	2.911	
	RMAT27	**8.645**	12.120	14.500	
SPMV	Twitter	**2.251**	2.860	4.610	
	Friendster	**3.624**	5.220	9.010	
	Orkut	**0.148**	0.208	0.630	
	LiveJournal	**0.060**	0.096	0.151	
	Yahoo_mem	**0.033**	0.045	0.063	
	USAroad	**0.077**	0.128	0.166	
	Powerlaw	**0.655**	**0.661**	0.707	
	RMAT24	**0.197**	0.221	0.288	
	RMAT27	**1.963**	2.210	2.830	
BF	Twitter	**1.489**	1.618	2.213	12.810
	Friendster	**6.498**	7.193	7.690	9.220
	Orkut	**0.213**	0.310	0.354	2.100
	LiveJournal	**0.258**	0.293	0.284	0.530
	Yahoo_mem	**0.146**	0.200	0.173	0.288
	USAroad	21.992	24.110	26.310	**16.330**
	Powerlaw	**10.326**	11.112	12.600	15.110
	RMAT24	**1.366**	1.390	1.410	1.880
	RMAT27	**1.665**	1.933	2.180	5.310
BP	Twitter	**38.896**	**38.900**	56.980	
	Friendster	**58.704**	66.210	129.000	
	Orkut	**2.223**	3.110	5.538	
	LiveJournal	**1.026**	1.420	1.940	
	Yahoo_mem	**0.448**	0.455	1.124	
	USAroad	**1.024**	1.660	1.462	
	Powerlaw	**15.264**	15.530	19.500	
	RMAT24	**4.788**	7.030	9.310	
	RMAT27	**32.994**	43.320	58.230	

Vertex-balanced partitioning is appropriate only for algorithms with fixed amount of work per vertex. Other algorithms, like PR, have a strong preference for edge-balanced partitioning. We conclude that it is crucial to balance partitions appropriately to the algorithm.

5.4 NUMA Optimization

Various choices can be made for the placement of vertex arrays, i.e., arrays storing frontiers or per-vertex application-specific data. GraphGrind places the vertex arrays such that each vertex is co-located with its home partition. Vertex-oriented loops, such as those in *vertex-map*, are typically short and have well-balanced work per iteration. As such, GraphGrind distributes the iterations equally across threads, even though this results in remote NUMA accesses.

We compare two variations on the NUMA policy (Figure 10): (i) placing vertex data and scheduling iterations on their home partition; (ii) equally spreading vertex data and iterations across all NUMA nodes. Option (i) aims to avoid remote NUMA access during vertex-oriented loops. This is however uniformly worse than GraphGrind's policy. It shows that CPU load balance is simply more important than NUMA locality for the vertex-oriented loops.

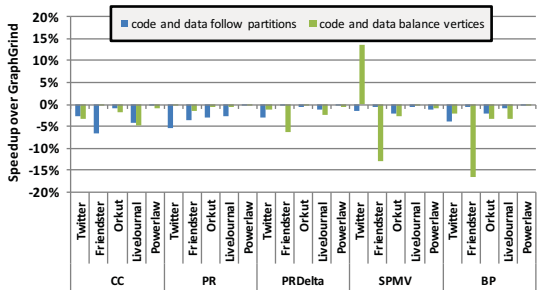


Figure 10: Impact of NUMA decisions for vertex arrays. GraphGrind may be described as data follow partitions, code balances iterations.

Option (ii) load-balances vertex-oriented loops and tries to minimize remote NUMA accesses by spreading vertex arrays to match the distribution of iterations. This results in worse performance in nearly all cases as the placement decision is sub-optimal for the *edge-map* operator. This operator performs the majority of main memory accesses and will incur excess remote memory accesses when vertices are not co-located with their home partition.

An interesting effect occurs when SPMV processes the Twitter graph, as in this case an increase in remote memory

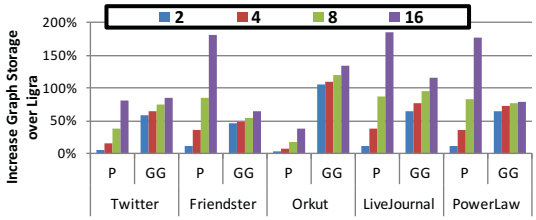


Figure 11: Increase of graph storage for Polymer (P) and GraphGrind (GG) compared to Ligra.

accesses during *edge-map* results in improved performance. We contrast this against Friendster, where the same effect results in performance degradation. We measured the local and remote memory accesses incurred and observe that both GraphGrind and option (ii) incur the same total number of memory accesses and that option (ii) incurs an increased number of remote accesses for both graphs.

The performance difference between the graphs, however, results as Twitter has highly skewed partitions: The number of elements of vertex arrays accessed on one NUMA node is much higher than on other NUMA nodes. Where GraphGrind directs those accesses to the local NUMA node, option (ii) spreads them across nodes. This way, option (ii) can share the unused memory bandwidth on one NUMA node with the computation on another node. On Friendster, GraphGrind is faster than option (ii) because Friendster has relatively uniform partitions and performs more memory accesses per unit of time. As such, all NUMA nodes are equally stressed and there is no benefit in making remote accesses.

These results show that a careful trade-off is required to optimize NUMA placement, as option (i) incurs fewer remote memory accesses than GraphGrind, yet has worse performance. In rare cases can remote accesses result in performance improvement due to imbalance in memory traffic.

5.5 Peephole Optimizations

GraphGrind marks frontiers that are initialized to contain all vertices such that an optimized *edge-map* can avoid memory accesses and control flow related to frontier access. Only algorithms that initialize frontiers this way can benefit. The algorithms using backward traversal (CC and PR) benefit most, up to 8%, as the backward traversal queries the frontier once for every edge, while the forward traversal queries it only once per vertex. The speedup is modest, but consistently positive. It moreover requires no user intervention.

GraphGrind allows programmers to define a cache, which allows the compiler to store intermediate values in registers (the cache) and avoid memory accesses. This optimization is relevant only during backward traversal. When applicable, the cache results in a speedup between 2 and 15%.

5.6 Memory Usage

Figure 11 shows the additional memory used on graph data for Polymer and GraphGrind compared to Ligra. Polymer stores each graph partition in CSR and CSC format (as in Ligra) using index arrays of length $|V|$. Because of this, the memory consumption of Polymer grows as $P|V|$ for P partitions. As GraphGrind stores only vertices with non-zero degree in the index arrays, its memory usage grows more slowly and follows the vertex replication factor (Figure 3). However, as GraphGrind stores an additional copy of the graph for sparse traversal, it starts at a 50% increase compared to Ligra for directed graphs. Overall, GraphGrind's memory consumption is more scalable than Polymer's.

6 FURTHER RELATED WORK

It has been documented that generic tools such as METIS [13] to partition graphs by vertex or edge cut do not produce good partitions for social network graphs. Moreover, they take much more time to compute than many graph algorithms. Sheep [17] is a distributed graph partitioner that produces high quality edge partitions an order of magnitude faster than METIS. Alternatively, linear-time heuristics have been proposed. The vertex cut is a greedy edge partitioning algorithm that minimizes the number of cut vertices [9].

Bourse et al. [4] target distributed memory systems as it minimizes the number of edges crossing partitions, which involve messages. This is not immediately relevant to the performance of shared memory systems. It is not immediately clear that the algorithm would perform well in the context of our system. The algorithm moreover approximates edge and vertex balancing. Experimental evaluation shows deviations in the vertex balance up to 50%, which would have a prohibitively high impact on GraphGrind.

GraphChi [15] streams graph data from disk. It uses partitioning to obtain small vertex sets that fit in the main memory. It uses partitioning by destination with an equal number of edges per partition. The vertex data must be made to fit in memory by tuning the number of partitions.

X-Stream [21] uses what we call partitioning by source, but does not required edges to be pre-sorted. It aims for a uniform number of vertices per partition as it wants to keep only vertex data in fast memory (e.g., CPU cache), whereas edges are streamed in from slower memory (e.g., main memory).

GraphX [10] is a library for graph analytics. It partitions edge lists using Spark's resilient distributed datasets (RDD) and supports user-defined partitioning schemes.

Our observations are relevant for each of the systems discussed above. E.g., the reduced connectivity of partitions implies that memory locality is poor in a system like X-stream. A large variation in vertices per partition implies that partitions with few vertices will leave a large portion of main memory unutilized in GraphChi.

Frasca et al. [5] design NUMA-aware work queues for betweenness centrality. The work queues first execute locally generated work prior to stealing work from other queues. Work queues are visited in order of increasing NUMA distance. They demonstrate a 51.2% performance improvement compared to an OpenMP implementation.

Agarwal et al. [1] study the execution of breadth-first-search on NUMA systems. They too organize the computation around work queues, spread over multiple sockets. They use efficient spinning locks and lock-free channels to synchronize threads and they introduce peephole optimizations, e.g., avoiding atomic operations by first checking if they will fail.

Graph compression can significantly reduce memory requirements and with it memory bandwidth. Shun *et al* [24] compress the destination IDs of vertices stored in the edge array of the CSR and CSC representations. They reduce memory usage up to 56%. These techniques are orthogonal to the compressed representation of the CSC and CSR index arrays proposed in this work, as they pertain to edges only.

7 CONCLUSION

Graph partitioning is an important technique to efficiently orchestrate the execution of graph analytics. In this paper, we study graph partitioning in the context of NUMA-aware data placement and code scheduling. We analyze the performance issues that graph partitioning inadvertently introduces, including load imbalance, increased work per vertex, and a significantly reduced connection density. Combined, these problems imply that graph partitioning is inherently unscalable to large partition counts.

We propose several techniques to counter-act the identified performance issues and implement these in GraphGrind, a novel NUMA-aware graph analytics framework that is compatible with the Ligra API. We moreover extend the Cilk language, in which GraphGrind is implemented, to enable NUMA-aware scheduling.

GraphGrind achieves significant speedup compared to prior work, out-performing Polymer, the most recent contender, by as much as 82%. We moreover show that fully minimizing remote memory accesses is not optimal in irregular computations. Instead, one needs to strike a careful trade-off between remote accesses and CPU load balancing.

We believe that this work makes important progress in making graph partitioning scalable. In future work, we will explore how to apply graph partitioning at much higher scales and translate these into enhanced performance.

ACKNOWLEDGMENTS

This work is supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under the ASAP project, grant agreement no. 619706, and by the United Kingdom EPSRC under grant agreement EP/L027402/1.

REFERENCES

[1] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. 2010. Scalable Graph Exploration on Multicore Processors. In *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[2] S. Beamer, K. Asanović, and D. Patterson. 2012. Direction-optimizing Breadth-first Search. In *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis* Article 12, 10 pages.

[3] R. D. Blumofe and C. E. Leiserson. 1994. Scheduling multithreaded computations by work stealing. In *Proc. of the Annual Symp. on Foundations of Computer Science*. 356–368.

[4] B. Florian, L. Marc, and V. Milan. 2014. Balanced graph edge partition. In *Proc. of the 20th SIGKDD Intl. Conf. on Knowledge discovery and data mining*. 1456–1465.

[5] M. Frasca, K. Madduri, and P. Raghavan. 2012. NUMA-aware Graph Mining Techniques for Performance and Energy Efficiency. In *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis* Article 95, 11 pages.

[6] M. Frigo, P. Halpern, C. E Leiserson, and S. Lewin-Berlin. 2009. Reducers and other Cilk++ hyperobjects. In *Proc. of the Annual Symp. on Parallelism in algorithms and architectures*. 79–90.

[7] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. 1999. Cache-Oblivious Algorithms. In *Proc. of the Annual Symp. on Foundations of Computer Science* 285–.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. of the Conf. on Programming Language Design and Implementation* 212–223.

[9] J. E Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.. In *Proc. of the Intl. Symp. on Operating System Design and Implementation*, Vol. 12. 2.

[10] J. E Gonzalez, R. S Xin, A. Dave, D. Crankshaw, M. J Franklin, and I. Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *Proc. of the Intl. Symp. on Operating System Design and Implementation*. 599–613.

[11] S. Hong, T. Oguntebi, and K. Olukotun. 2011. Efficient parallel graph exploration on multi-core CPU and GPU. In *Parallel Architectures and Compilation Techniques (PACT), 2011 Intl. Conf. on*. 78–88.

[12] Intel 2013. *Intel Cilk Plus Language Extension Specification* (version 1.2. 324396-003us ed.). Intel.

[13] G. Karypis and V. Kumar. 1998. Multilevel k-way Partitioning Scheme for Irregular Graphs. *J. Parallel and Distrib. Comput.* 48, 1 (1998), 96 – 129.

[14] H. Kwak, C. Lee, H. Park, and S. Moon. 2010. What is Twitter, a social network or a news media?. In *Proc. of the international conference on World wide web*. 591–600.

[15] A. Kyrola, G. E Blelloch, and C. Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC.. In *Proc. of the Intl. Symp. on Operating System Design and Implementation*, Vol. 12. 31–46.

[16] C. E. Leiserson, T. B. Schardl, and J. Sukha. 2012. Deterministic Parallel Random-number Generation for Dynamic-multithreading Platforms. In *Proc. of the Symp. on Principles and Practice of Parallel Programming*. 193–204.

[17] D. Margo and M. Seltzer. 2015. A Scalable Distributed Graph Partitioner. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1478–1489.

[18] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *Proc. of the Internet Measurement Conf.*

[19] D. Nguyen, A. Lenharth, and K. Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proc. of the Symp. on Operating Systems Principles*. 456–471.

[20] L. Page, S. Brin, R. Motwani, and T. Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Stanford InfoLab.

[21] A. Roy, I. Mihailovic, and W. Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proc. of the Symp. on Operating Systems Principles*. 472–488.

[22] Y. Saad. 1990. *SPARSKIT: A basic tool for sparse matrix computations*. Technical Report NASA-CR-185876. NASA.

[23] J. Shun and G. E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proc. of the Symp. on Principles and Practice of Parallel Programming*. 135–146.

[24] J. Shun, L. Dhulipala, and G. E Blelloch. 2015. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *Data Compression Conf.* 403–412.

[25] Y. Vigfusson. 2010. *Affinity in distributed systems*. Ph.D. Dissertation. Cornell University.

[26] J. Yang and J. Leskovec. 2012. Defining and Evaluating Network Communities based on Ground-truth. *CoRR* abs/1205.6233.

[27] K. Yotov, Tom R., K. Pingali, J. Gunnels, and F. Gustavson. 2007. An experimental comparison of cache-oblivious and cache-conscious programs. In *Proc. of the Annual Symp. on Parallel algorithms and architectures*. 93–104.

[28] K. Zhang, R. Chen, and H. Chen. 2015. NUMA-aware graph-structured analytics. In *Proc. of the Symp. on Principles and Practice of Parallel Programming*. 183–193.