Noname manuscript No.
(will be inserted by the editor)

Analysing Software Prefetching Opportunities in Hardware Transactional Memory

Marina Shimchenko · Rubén Titos-Gil ·
Ricardo Fernández-Pascual ·
Manuel E. Acacio · Stefanos Kaxiras ·
Alberto Ros · Alexandra Jimborean

Received: date / Accepted: date

Abstract Hardware Transactional Memory emerged to make parallel programming more accessible. However, the performance pitfall of this technique is squashing speculatively executed instructions and re-executing them in case of aborts, ultimately resorting to serialization in case of repeated conflicts. A significant fraction of aborts occur due to conflicts (concurrent reads and writes to the same memory location performed by different threads). Our proposal aims to reduce conflict aborts by reducing the window of time during which transactional regions can suffer conflicts. We achieve this by using software prefetching instructions inserted automatically at compile-time. Through these prefetch instructions, we intend to bring the necessary data for each transaction from the main memory to the cache before the transaction itself starts to execute, thus converting the otherwise long latency cache misses into hits during the execution of the transaction. The obtained results show that our approach decreases the number of aborts by 30% on average and improves performance by up to 19% and 10% for two out of the eight evaluated benchmarks. We provide insights into when our technique is beneficial given certain characteristics of the transactional regions, the advantages and disadvantages of our approach, and finally, discuss potential solutions to overcome some of its limitations.

This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No 819134), the Spanish MCIU and AEI, as well as the European Commission FEDER funds, under grant RTI2018-098156-B-C53, and the Swedish VR grant number 2016-05086.

M. Shimchenko · S. Kaxiras · A. Jimborean
Department of Computing Systems, Uppsala University
E-mail: first.last@it.uu.se

R. Titos-Gil · R. Fernández-Pascual · M. E. Acacio · A. Ros · A. Jimborean
Computer Engeneering Department, University of Murcia
E-mail: rtitos@um.es, ricardof@um.es, meacacio@um.es, aros@ditec.um.es,
alexandra.jimborean@um.es

Keywords Hardware Transactional Memory · Parallel programming · Compiler · Software-prefetching

1 Introduction

The switch to multicores, following the demise of Dennard's scaling in the mid 2000's and the more recent slowing of Moore's law, brought parallelism into the center stage. Parallel programming is significantly more challenging than its serial counterpart. In large part, this is due to the intricate synchronization required in many applications. Some parallel models, e.g., for GPUs and massively-parallel accelerators, have proven very successful in exploiting parallelism found in specific application classes (e.g., graphics, embarrassingly parallel scientific computations, etc.) but, admittedly, synchronization in these approaches is trivial. For any other workload that may be unstructured and with many inter-dependencies among tasks, extracting parallelism is far more complicated.

Transactional memory (TM) in general has been proposed as a simpler alternative to the conventional parallel programming with locks [11]. TM is an optimistic form of synchronization that speculatively executes transactional regions concurrently. Whereas with traditional lock synchronization only one thread at a time can enter a critical section, TM increases concurrent execution in uncontended scenarios without burdening the programmer. Hardware implementations of the TM concept (Hardware Transactional Memory) have already been deployed in commercial processors from IBM [12], Intel [35], and more recently, ARM [3].

To ensure correctness, HTM ensures that a transaction will fail when two or more threads conflict. A conflict arises when two or more threads access the same memory location, and at least one of the accesses is a write. When there is a conflict, memory changes made by the transaction are discarded, registers are restored to their former state and the transaction re-runs until it completes successfully. Usually, to limit speculative execution and avoid infinite re-trying upon continuous mis-speculation, HTM implementations employ a fallback lock, which is acquired after a number of aborts to execute the transaction non-speculatively.

Conflicts are just one example why a transaction must abort in contemporary best-effort Hardware Transactional Memory (HTM) systems. Other sources of abort may include interruptions, page faults or lack of capacity in caches. In general, aborts are expensive both in execution time and energy due to the loss of concurrency and the need to re-execute transactions.

This paper focuses on conflict-induced aborts. We make the observation that by turning long latency loads within transactions into cache hits, we can decrease the execution time of transactional regions, and thus reduce the likelihood for conflicts. We depict this concept in Fig. 1. On the left (Fig. 1a), one can see the timeline with original transactions. They have the potential to abort each other because of their overlapping duration. In the center (Fig. 1b),

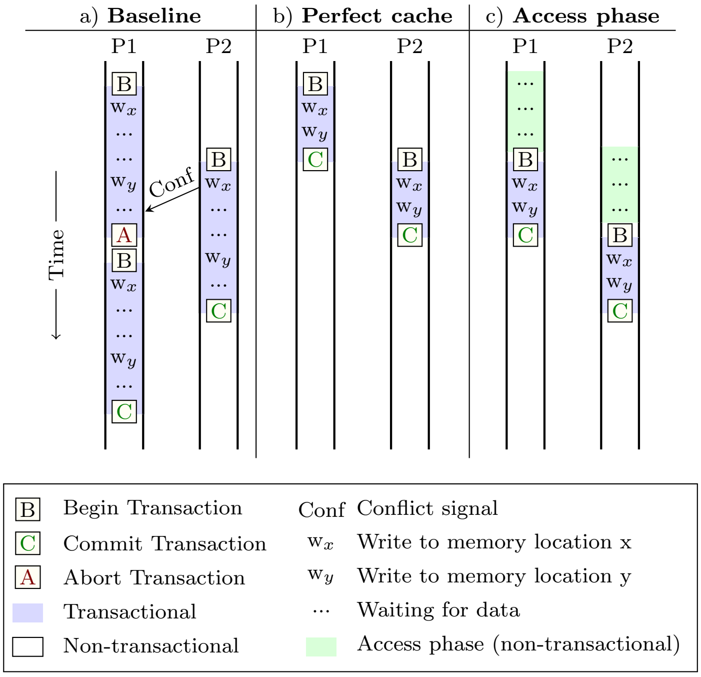

Fig. 1: Motivation for reducing cache misses in transactional regions and consequently their execution time: a) Our baseline is the original transaction, which includes long latency operations (such as read operations served from main memory) illustrated by ellipses (...). Due to the long execution time, the transactions of P1 and P2 overlap leading to a conflict, which causes P1 to abort and re-run the transaction. b) This case considers a perfect cache (i.e., all memory accesses are served directly from the cache) which significantly reduces transactional time and avoids the conflict between P1 and P2 and the resulting abort and reexecution. c) Finally, this represents our proposal in which we aim to get the benefits of a near-perfect cache, by moving the long latency operations outside the transactions, in the so called Access phases, one per transaction, aimed to prefetch the data accessed in the TM region.

these transactions execute faster since a perfect cache would transform misses into hits, which reduces their duration, and thus, the chances of a conflict. This observation gives a major insight into the correlation between the execution of transactions and the probability of conflicts. To this end, we design a compiler technique that automatically prefetches the data required by a transactional region before its execution, ensuring that data is readily available in the cache, as shown in Fig. 1c.

Our technique excels for transactions that i) abort frequently and ii) incur a large number of cache misses. We performed an analysis of the applications in STAMP benchmark suite [20], presented in Fig. 2 which shows the average number of aborts due to conflicts per committed transaction and the cache miss ratios for the accesses within a transaction, for the STAMP applications evaluated in this work. This analysis shows that the best candidates for benefiting from our technique are kmeans-h and labyrinth since both of these

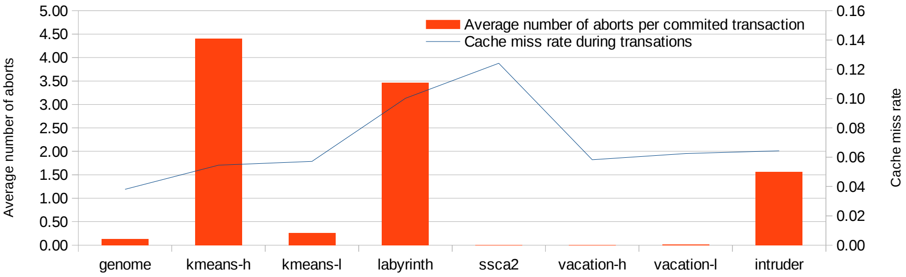

Fig. 2: The graph shows the average number of times that transactions abort due to conflicts before committing (bar and left y axis) and the cache miss ratio for all accesses within transactions (line and right y axis) for STAMP benchmarks (x axis).

benchmarks have the highest conflict rate at the time that suffer from cache misses within transactions.

Our simulation results using the Gem5 [4] simulator and applications from the STAMP benchmark suite show that our approach decreases the number of aborts by 30% on average and improves performance by up to 19% and 10% for two (kmeans-h and labyrinth) out of the eight evaluated benchmarks.

This work makes the following contributions:

- We provide a key insight regarding the correlation between the execution time of transactions and the probability of conflicts.
- We present a compiler technique that performs inter-procedural memory alias analysis (IPMAA) to detect and prefetch the data required within transactional regions. We implemented our proposal in the LLVM compiler [16].
- We provide an in-depth analysis for the analyzed benchmarks and identify the key features of transactional regions that can benefit from our approach.

The rest of the manuscript is organized as follows. Section 2 provides some background about different implementations of transactional memory. Section 3 surveys related work. Section 4 gives an overview of our proposal and details its main challenges. Section 5 describes our compiler technique in detail. Section 6 describes our hardware extensions. Section 7 introduces our experimental environment and presents the obtained results. Section 8 identifies the limitations of our technique and highlights the future improvement directions. Finally, Section 9 contains the main conclusions of this work.

2 Background on Transactional Memory

In this section we offer an overview of Hardware Transactional Memory (HTM) and its implementation in commodity processors of Intel [35], IBM [12] and ARM [3]. Stores within a transaction are executed speculatively, which means that they are only visible to the thread that executes them. When the transaction commits, the stores become non speculative and are made visible to the rest of the threads. If a transaction cannot commit due to a conflict or any other cause, it aborts and reverts architectural state to that before the beginning of the transaction: all speculative stores are discarded, and the register file checkpoint taken at transaction begin is restored. Currently available implementations of HTM are *best-effort*, which means there are no guarantees that a speculative transaction will ever succeed. In this best-effort designs, the hardware TM support must be combined with a non speculative path for transactions to make progress in spite of limited hardware resources, repeated aborts as a result of high contention, etc. A software abort handler determines whether an aborted transaction should be speculatively retried, or must take the fallback path. To maintain atomicity of non speculative transactions,the traditional lock-based synchronization scheme enforces mutual exclusion: i) no speculative transactions can begin execution when the lock is held, and ii) all speculative transactions must subscribe to the lock, so that they are aborted as a result of a conflict when the cache block that contains the lock is written upon acquisition.

In general, there are several reasons why a transaction might be aborted in a best-effort HTM design: data conflicts, insufficient speculative buffering capacity, unsupported instructions, etc. In certain implementations, such as Intel TSX, interrupts always cause the abort of a transaction, while in other implementations such as in IBM Power 8 [17] transactions can survive interrupts. A fragment of non-transactional code that executes in between transactional execution is also known as an *escape action* [21], which reduces the number of aborts of transactional programs and allows a wider use of transactions. In this work, we focus on aborts that take place due to data conflicts.

3 Related work

This section surveys related work in hardware, software, or hardware-software co-design to decrease the abort rate in transactional memory (TM) systems, since aborts are the main drawback for TM performance.

Diegues et al. [8] avoid aborting transactions that read stale data (i.e. miss the writes of a parallel transaction) and instead find a valid execution which allows such transactions to pretend they commit in the past, before their competitors. This is called a time-warp commit and guarantees correctness.

Ansari et al. [2] propose Steal-on-Abort (SOA) for software TM to change the transactions order at runtime, if a conflict occurs. Once two transactions A and B are observed to conflict, they are sequentialized in the following occurrences, such that B is always scheduled to execute after A completes, to avoid repeated conflicts. In their follow-up work [1], the authors expanded their ideas of SOA to HTM, but entail significant changes of the core.

Maldonado et al. [19] propose a kernel-level scheduler targeting repeated conflicts, as in the case of SOA. Our proposal can handle also conflicts that occur for the first time and our compiler based optimizations do not require any changes at the operating system level.

Diegues et al. [9] explore a software scheduler for HTM called Seer. It gathers statistics at runtime and supplies this information to an on-line probabilistic inference technique that identifies conflict patterns between different transactions. Seer establishes a dynamic locking scheme to serialize transactions in a fine-grain manner, yet serialization might create a bottleneck when many threads are involved. Our solution does not rely on additional locking (other than the fallback path) and therefore is not subject to increased contention.

Other proposals exploit snapshots, i.e. memory captures that always guarantee consistent reads, and allow aborting only on write-write conflicts and ignoring read-write conflicts. Litz et al. [18] in their work on snapshot isolation (SI-TM) allow multiple versions of the same data to coexist in memory, but require changes both in the memory system and in the HTM design to guarantee consistency. Their technique can handle read-write conflicts, but transactions still need to abort on write-write conflicts.

PfTouch [29] avoids resolving page faults in mutual exclusion mode, thus reducing the number of aborts. The page faults are resolved concurrently by the abort handler while other speculative transactions still run. Similarly, TAPs issue prefetch requests in parallel with the execution of other transactions. Both approaches are orthogonal as we focus on cache misses and PfTouch focuses on page faults.

Xiang and Scott [34] propose a design in which the compiler identifies and instruments potential parts of transactions that generate conflicts instead of collecting these statistics at runtime. The compiler divides the transactions in a read-mostly (planning) and a write-mostly (completion) phase and moves the planning phase out of the parent transaction. Moving part of the transactional code outside the transaction has some limitations and the authors propose some workarounds to guarantee atomicity. In our proposal, we duplicate the read-mostly part of the transaction outside the transactional code, without affecting atomicity.

Prefetching has been employed by Dash et al. [5,6] targeting distributed transactions with the goal of hiding network latency and not to reduce conflicts. In one proposal [6], the authors prefetch objects identified by the programmer using manually given hints, i.e. paths to parse the heaps and find the target objects before their address is computed. Such hints are called symbolic prefetches. In contrast, our technique is fully automated and does not require programmer's intervention. Moreover, their technique allows using stale versions of an object until the commit time, when the transaction is validated. The authors do present several workarounds to the shortcomings that can stem from using stale data (e.g. infinite loops, exceptions), but rely on time budgets allocated to each transaction and checking regularly their read sets. In contrast, ensuring that the latest version is used in each AP and transaction comes naturally with our approach, without extra overheads due to checks. The other proposal [5] focuses on compiler techniques to automatize the definition of symbolic prefetches. Yet, overall, these proposals address problems specific to the communication design of distributed transactional memory and object definition in Java. For instance, prefetching objects of the type a.m[i][k] is performed in three consecutive round trips over the network to prefetch a, then a.m, and a.m[i], which is not the case in our system. To the best of our knowledge no previous attempts have been made to use software prefetching to reduce the number of conflict-caused aborts.

Negi et al. [23] prosose transactional prefetching, which is a hardware prefetcher mechanism targeting transactional writes. In essence, they track write-set lines that are prefetched on transaction aborts. In contrast, we propose a software prefetching mechanism that targets all accesses within the transaction.

In this work, we have retargeted the software Decouple Access Execute (DAE) model [13] to handle transactional workloads. The software Decoupled Access Execute (DAE) model [13,15] was previously employed to enable effective Dynamic Voltage Frequency Scaling in order to reduce the energy expenditure. In this proposal we go beyond simply employing DAE for a new purpose (reducing conflicts in transactional memory execution), but we significantly extend the model with support for inter-procedural memory analysis and building access phases that consist in multiple functions. In contrast, DAE could previously handle only loops that contain no function calls. By adding support for following the call graph and identifying memory dependences across loop bounds and function calls, our proposal yields a more robust version of DAE that is readily applicable to any application, of high complexity, that couldn't have been targeted before.

4 Overview and challenges

To reduce the conflicts of transactional memory execution, we present a compiler technique that prefetches the shared data read by transactional regions ahead of time to speed-up their execution, reduce the overall time spent by an application executing transactions, and therefore reduce the likelihood for conflicts to occur. This section outlines the automatic code transformations performed by our compiler to turn long-latency memory accesses in local cache hits and the proposed hardware extensions, diving in the details of the encountered challenges and our solutions to address them.

The target application is transformed such that most of the read-data required by a transactional region is brought to the cache (prefetched) by a dedicated *Access Phase*, prior to its execution. We prefetch variables that escape the scope of the transaction, assuming that accesses performed through the stack are not shared. Fig. 3 shows an example of the intermediate representation (IR) code of an Access Phase (AP) automatically generated by our compiler to prefetch the data required by the transaction. The original transactional region (shown on the left) is identified as being the code executed between the TX calls: beginTX and endTX. The *tran* function delineated by these calls contains simple computations, a load, and a function call. The called function, *helper*, is therefore part of the transactional region as well.

The right-hand side of Fig. 3 shows the modified code after applying our compiler pass. Each function in the transactional code region has a corresponding access phase function (*AP_tran* and *AP_helper*). Each access phase function represents a program slice of the loads in the original function, where leaf-loads are turned into prefetch instructions [13]. The access phase functions replicate the call graph within the transaction and together they build the corresponding access phase of the transactional code region. Recall the execution flow shown in Fig. 1, where the transactional code region (shown in purple) is preceded by a dedicated access phase prefetching its data (shown in green). Thus, *AP_tran* precedes the execution of the transactional function *tran* and calls the corresponding access phase of function *helper*, *AP_helper*, as seen in Fig. 3.

Key principles of access phases. The example in Fig. 3 helps to identify the key principles upon which the access phases are built in order to guarantee the correctness of the compile-time code transformation. We must ensure that accesses phases are *side-effect free* and do not perturb the observable behavior of the program. Namely, an access phase cannot perform writes that are visible outside its scope. Access phases are merely designed to prefetch data and add the minimum number of instructions required, to avoid the runtime overhead. To this end, the compiler identifies loads of globals and variables that escape the scope of the transaction and builds the program slice of these target loads. Departing from the original transactional code region, the compiler first creates a clone of this region and then filters out all instructions that are not involved in the address computation of these loads, akin to prior work [13–15, 32, 33]. All instructions required to compute the target address and to reach the load in a complex control flow and call graph are preserved. Such instructions may be computations, control flow, function calls, instructions residing in other functions, etc. The complete algorithm of our compiler technique is presented below.

However, building lean, effective, efficient, and side-effect free access phases is challenging. This becomes a monumental task in the presence of complex control flow and call graphs, pointers, dynamic data structures or recursive functions, which are all characteristics exhibited by the transactional memory applications that we have analyzed (see Section 7). Pointer analysis is notoriously difficult for compilers and the difficulty is exacerbated by the interprocedural analysis of dependences in order to identify instructions required for building the access phase (i.e., potentially residing in a different function, deep in the call graph). We describe next the challenges of identifying the instructions required for building lean and effective access phases.

```
helper()
{
  addr1 = compute address
  load1 = load addr1
  use load1
}
tran()
{
  val  = compute value
  addr = compute address
  load = load addr
  call helper()
  use val
}
int main()
{
  ...
  beginTX
  call tran()
  endTX
  ...
}
```

a. Original IR code

```
AP_helper()
{
  addr1 = compute address
  prefetch addr1
}
AP_tran()
{
  addr = compute address
  prefetch addr
  call AP_helper()
}
helper()
{
  addr1 = compute address
  load1 = load addr1
  use load1
}
tran()
{
  val  = compute value
  addr = compute address
  load = load addr
  call helper()
  use val
}
int main()
{
  ...
  call AP_tran()
  beginTX
  call tran()
  endTX
  ...
}
```

b. Transformed IR code

Fig. 3: An example of an Access Phase prefetching data required by a transactional region.

4.1 Generating multi-function APs

Transactions often have a complex control flow graph (CFG), including many function calls. Consequently, the APs must span multiple functions, as they replicate the original control and call graph. We denote these access phases multi-function APs.

First, for each callee function, we need to generate an appropriate AP[1]. This step applies recursively. Without this step, the original functions would be called (instead of the corresponding AP), including all the unnecessary

[1] If a function is called in two different transactions, we create one AP version for each call context. AP versions are transaction specific because the selection of the instructions for each AP depends on how the memory updates performed within the function affect its callers.

```
 1  int a[3] = {0,1,2}
 2  int n = 0;
 3  func1()
 4  {
 5    n = a[1];
 6  }
 7  func2()
 8  {
 9    n = a[n];
10  }
11  int main()
12  {
13    beginTX
14    func1();
15    func2();
16    endTX
17    printf(n);
18  }
```

a. C-code

```
 1  int a[3] = {0,1,2}
 2  int n = 0;
 3  func1()
 4  {
 5    val1 = load a[1]
 6    store val1, n
 7  }
 8  func2()
 9  {
10    val2 = load a[n]
11    store val2, n
12  }
13  int main()
14  {
15    beginTX
16    call func1()
17    call func2()
18    endTX
19    call printf(n)
20  }
```

b. Original transaction

```
1  AP_func1()
2  {
3    prefetch a[1];
4  }
5  AP_func2()
6  {
7    prefetch a[n];
8    //n = ??
9  }
```

c. AP without IPMAA

```
 1  AP_func1()
 2  {
 3    val1 = load a[1];
 4    store val1, n;
 5  }
 6  AP_func2()
 7  {
 8    prefetch a[n];
 9    //n = a[1]
10  }
```

d. AP with IPMAA

Fig. 4: IPMAA is mandatory to generate representative Access Phases, that can reach out and prefetch the required data.

instructions that do not contribute to prefetching, thus increasing the size of the APs and leading to potential side effects.

Second, a multi-function structure requires inter-procedural memory alias analysis (IPMAA), as shown in Fig. 4. Fig. 4a shows a simple C-code example. The main function contains a transaction that reads and writes the shared variable n (lines 13–16). The examples in all the figures of this section assume that multiple threads execute the transactional regions in parallel. The transaction calls func1 and func2. func1 initializes an index **n** (line 5). func2 accesses the nth element of the shared array **a** (line 9) and overwrites the value of **n**. Fig. 4b shows the IR corresponding to the C-code. func1 loads the value from a[1] (line 5) and then stores it to n (line 6). func2 loads a new value from a[n] (line 10) and stores it back to n (line 11). Fig. 4c shows the APs for func1 and func2 without an inter-procedural alias analysis (IPMAA). Since these functions have no information about data dependencies between them, APs keep only instructions that are necessary for computing the addresses and for reaching the target loads that reside in the *current* function. In this case, a prefetch from AP_func2 (line 7) would access an invalid address because AP_func1 did not update **n**. However, leveraging information about data dependencies, the store to **n** in AP_func1 would be preserved as part of the AP (Fig. 4d, line 4) and the prefetch in AP_func2 would access the correct address.

There are two ways to guarantee that all necessary stores are included: to inline all callee functions within transactional regions or to use IPMAA. Inlining increases compilation time considerably and not every function is suitable for inlining. Therefore, we integrated the SVF [27,28] inter-procedural

```
 1  int a[3] = {2, 0, 1};
 2  int n = 0;
 3  tran()
 4  {
 5    n = a[n];
 6    n = a[n];
 7  }
 8  int main()
 9  {
10    beginTx
11    tran();
12    endTX
13    printf(n); // n = 0
14  }
```

a. Original c-code.

```
 1  int a[3] = {2, 0, 1};
 2  int n = 0;
 3  tran()
 4  {
 5    val = load a[n]
 6    store val, n
 7    val = load a[n]
 8    store val, n
 9  }
10  int main()
11  {
12    beginTx
13    tran();
14    endTX
15    printf(n); // n = 0
16  }
```

b. Original IR code

```
 1  int a[3] = {2, 0, 1};
 2  int n = 0;
 3  AP_tran()
 4  {
 5    val = load a[n]
 6    store val, n
 7    prefetch a[n]
 8  }
 9  tran()
10  {
11    val = load a[n]
12    store val, n
13    val = load a[n]
14    store val, n
15  }
16  int main()
17  {
18    AP_tran();
19    beginTx
20    tran();
21    endTX
22    printf(n); // n = 2
23  }
```

c. Transformed IR code

Fig. 5: The access phase non-side-effect-free problem

alias analysis. SVF is a static tool that enables scalable and precise interprocedural dependence analysis for C and C++ programs. Yet, including stores in the AP breaks the side-effect freedom property, leading us to the next challenge.

4.2 Side-effect free APs

Building side-effect free and efficient APs is a challenge, as illustrated by the example in Fig. 5. The original C-code is shown in Fig. 5a. It contains a transaction which updates the shared variable **n**. If we consider one thread execution for simplicity, the printed result of **n** should be equal to 0. Fig. 5b shows the corresponding intermediate representation (IR) code of the C-code snippet. Fig. 5c illustrates the IR with the generated AP (AP_tran). It contains one load (line 5), a load changed to a prefetch instruction (line 7), and the store to **n** (line 6) due to data dependencies between the prefetch instruction and the store. If AP_tran and the transaction execute one after another, the printed value would be incorrect because the store in the AP changed the program's behavior.

To solve this problem APs should either prohibit stores to global variables or provide a mechanism to undo their changes. After having analyzed both options, we concluded that the former would discard a significant fraction of loadsfrom being placed in an AP, since they depend on stores to global variables (or to variables that escape the scope of the function). So, this solution is only viable for very simple transactions.

Our solution consist on undoing writes. To this end, we designed a more general approach in which the compiler wraps APs in their own *doomed* transaction, creating transactional access phases (TAPs). The compiler ensures that, if not aborted for other causes, TAPs always explicitly abort at the end by executing the abort transaction instruction. In this way, all the changes performed during the execution of a TAP are discarded, keeping them side-effect free with respect to the state of the globally visible memory. Furthermore, prefetched data stays in the cache (recall that we prefetch the read sets), delivering the same benefits as the APs. The compiler provides a specific abort handler for TAPs, since they do not require a fallback path as they are never retried. Consequently, TAPs do not subscribe nor wait on the fallback lock used by regular transactions, and as a result TAPs can execute concurrently with a non speculative transaction. However, TAPs come at a cost, due to the overhead of starting and aborting a transaction, equivalent to executing three contended atomic operations (e.g., test-and-set or compare-and-swap) [26,35]. Additionally, transactional reads incur non-fixed costs as well [26]. Thus, it could be beneficial to select between the two design options on a transaction basis. We leave this investigation for future work.

4.3 Nack Access Phases

The drawback of using TAPs is a potential increase in the number of conflicts and execution time in consequence. To avoid that TAPs abort regular transactions, we propose a hardware extension inspired by *power transactions* [7], which allows transactions to acquire elevated priority by employing *requester-loses* as conflict resolution policy (instead of the default *requester-wins*) so as to survive conflicts with others. In our particular case, conflicts between TAPs and regular transactions are always resolved in favour of the latter: upon detection of conflicting accesses from TAPs, regular transactions send negative acknowledgments (*nacks*) in response, which forces the abort of the TAP. Conflicts among threads with equal priority (*i.e.*, TAP-TAP, or among regular transactions) are resolved using the default requester-wins policy.

4.4 Loops and recursions

Loops and recursions are handled differently than linear code (Fig. 6). Fig. 6a shows a simple C-code example, which contains a while loop (lines 3–6) and the linear code (line 7). This loop exhibits a loop carried dependence through **n**. Fig. 6b illustrates the corresponding IR for the C-code snippet. Each assignment to **n** is transformed into a load and a store, thus producing in total two store instructions (line 6, line 9). The store in the loop is vital to be preserved in the AP due to the loop carried data dependence. Removing it would cause the prefetch instruction (Fig. 6c, line 5) to access incorrect addresses. The store in the linear code does not contribute to the computation of the target address of the prefetch instruction and therefore, can be safely removed.

```
1  tran()
2  {
3    while(cond)
4    {
5      n = a[n];
6    }
7    n = a[n];
8  }
```

a. Original c-code.

```
1  tran()
2  {
3    while(cond)
4    {
5      val = load a[n]
6      store val, n
7    }
8    val = load a[n]
9    store val, n
10 }
```

b. Original IR code

```
1  AP_tran()
2  {
3    while(cond)
4    {
5       prefetch a[n]
6    }
7    prefetch a[n]
8  }
```

c. Corresponding AP without a proper loop handling.

```
1  AP_tran()
2  {
3    while(cond)
4    {
5      val = load a[n]
6      store val, n
7    }
8    prefetch a[n]
9  }
```

d. Corresponding AP with proper loop handling.

Fig. 6: Why loops and recursions should be handled differently than linear code

In short, in addition to identifying (i) the instructions that contribute to the computation of the target address of the load (line 5 Fig. 6d) and (ii) the control-flow instructions required to reach the load (line 3,4,7), the compiler detects (iii) data dependencies between stores and loads (read after write) and preserves in the AP the stores that feed loads that are required in order to prefetch from the correct address (line 6). When handling loops and recursive functions, for identifying the stores that feed loads, the compiler must account for loop carried dependencies and for dependencies carried by the recursions.

5 Compile-time code transformation

We implemented our technique in the LLVM 3.8 compiler framework [16]. The compile-time transformations follow the three main steps presented in Algorithm 1.

5.1 Step 1: Preparation

The first step is to prepare several aspects. The compiler parses all functions and identifies transactions, creates a clone for each transactional region (the future AP), and places a call to the clone before the corresponding transaction. At this point, all APs are merely copies of the original transactional regions, extracted in their own functions. We also temporarily extract loops into functions to process them as regular functions. Fig. 7a and Fig. 7b show the original C-code and the pseudo-code without any transformations. After

Algorithm 1: The main compiler algorithm

```
Step1: for each function do
    TX = identify_transactions();
    AP_TX = copy(TX);
    place_before_TX(AP_TX);
    for each loop in AP_TX do
        LP = extract_loop();
    end
    Step2: for each function do
        WPA = WPA();
        VECT = $\emptyset$;
        for each AP_TX do
            find_APInstr(AP_TX, WPA, VECT);
        end
        for each AP_TX do
            delete_inst_notInVECT(AP_TX, VECT);
            replace_loads_with_prefetching(AP_TX);
        end
    end
    Step3: for each function do
        for each AP_TX do
            clean_pref(AP_TX);
            for each loop in AP_TX do
                change_attribute(loop, AlwaysInline);
            end
        end
    end
end
```

step 1, transactional regions, including all the functions they call, are copied in separate functions with AP prefixes. We then insert calls to the APs just before the corresponding transaction. For example, the call to AP_tran() is inserted just before its transaction (Fig. 7c, line 21). The result of this step is shown in Fig. 7c.

5.2 Step 2: Building the AP

This step is the core of our optimizations and consists mainly in identifying the instructions required for each AP and filtering out the unnecessary ones (Algorithm 1, find_APInst).

5.2.1 Initializing helper structures

During the initialization phase, the compiler retrieves the interprocedural alias analysis (AA) information (Algorithm 1, WPA = WPA()) required to handle multi-function transactions (4.1) and creates an empty vector (set of instructions) for each function. Vectors are helper structures which will be later used to collect all the necessary instructions during the process of finding loads and

```
1  int a[3] = {0,1,2}
2  int n;
3  func()
4  {
5    n = a[1];
6  }
7  int main()
8  {
9    beginTX
10   func();
11   n = a[n];
12   endTX
13 }
```

a. C-code

```
1  int a[3] = {0,1,2}
2  int n;
3  func()
4  {
5    val = load a[1]
6    store val, n
7  }
8  int main()
9  {
10   beginTX
11   call func()
12   val = load a[n]
13   store val, n
14   endTX
15 }
```

b. Original IR code

```
1  int a[3] = {0,1,2}
2  int n;
3  func()
4  {
5    val = load a[1]
6    store val, n
7  }
8  AP_func()
9  {
10   val = load a[1]
11   store val, n
12 }
13 AP_tran()
14 {
15   call AP_func()
16   val = load a[n]
17   store val, n
18 }
19 int main()
20 {
21   call AP_tran()
22   beginTX
23   call func()
24   val = load a[n]
25   store val, n
26   endTX
27 }
```

c. After Step1

```
AP_tran's vector:
AP_func's vector:
```

d. Initial vectors state

```
AP_tran's vector:
AP_func's vector:
  val = load a[1]
```

e. Process Fig. 7c, line 15

```
AP_tran's vector:
  val = load a[n]
AP_func's vector:
  val = load a[1]
  store val, n
```

f. Process Fig. 7c, line 16

```
1  int a[3] = {0,1,2}
2  int n;
3  func()
4  {
5     val = load a[1]
6     store val, n
7  }
8  TAP_func()
9  {
10   val = load a[1]
11   store val, n
12   // aborts at the
         end
13 }
14 AP_tran()
15 {
16   call AP_func()
17   prefetch a[n]
18 }
19 int main()
20 {
21   call AP_tran()
22   beginTX
23   call func1()
24   val = load a[n]
25   store val, n
26   endTX
27 }
```

g. After Step2

Fig. 7: Compilation steps.

their dependencies. Each function has its helper vector. For example, AP_tran in Fig. 7c, calls another function. We create two initially empty helper vectors —one for the called function and one for AP_tran itself (Fig. 7d)— and use these vectors to gather the instructions that will eventually be part of the AP.

5.2.2 Selecting the AP instructions

The following functions presented in Algorithm 2 implement the selection of the instructions for the access phases, and are detailed next: `find_APInst`, `followDeps`, and `enqueueDependentInstructions`.

find_APInst The primary purpose of this function is to identify the instructions of interest (Algorithm 2, findInstToKeep), namely loads to shared data and function calls. We find loads to global variables by tracing the whole use-def chain. If the definition turns out to be a stack allocation function, we treat

Algorithm 2: Helper functions.

```
Function find_APInst(AP_TX, WPA, VECT):
    InstToKeep, Deps = empty();
    findInstToKeep(AP_TX, InstToKeep);
    followDeps(InstToKeep, Deps);

    VECT=InstToKeep + Deps;
return res

Function followDeps(InstToKeep, Deps):
    for each inst in InstToKeep do
        Q = empty();
        enqueueDependentInstructions(inst, LocalSet, Q);
        while !Q.empty() && res do
            Qinst = Q.pop();
            if Qinst.is_call() then
                find_APInst(Qinst->getAP, Qinst->getVECT);
            end
            enqueueDependentInstructions(Qinst, InstToKeep, Q);
        end
    end
return

Function enqueueDependentInstructions(Inst, InstToKeep, Q):
    enqueueInst(Inst, InstToKeep, Q);
    enqueueStores(Inst, InstToKeep, Q);
return
```

the load as local. We also keep function calls, because they might contain critical loads. If, at a later stage, we discover that a function does not contain any loads or other function calls, it will be deleted from its AP. Once all the instructions are collected into a vector (Algorithm 2, InstToKeep), we follow all the dependency chains to include the required instructions (Algorithm 2, followDeps).

followDeps For each instruction in InstToKeep, we find dependencies by calling the `enqueueDependentInstructions` function and enqueue them to Q. This process is then repeated for each instruction in Q until all required instructions are identified and the program slice is built. Furthermore, for instructions that represent function calls, in addition to collecting their required instructions, we recursively initiate the process of building the AP for the called function.

enqueueDependentInstructions This function identifies the operands of a particular instruction and selects (i) the instructions that represent these operands (using the def-use chains in LLVM) and (ii) other instructions that may affect its operands such as function calls or stores (see below).

When the operand of an instruction is an instruction itself (Algorithm 2, enqueueInst) we simply enqueue this instruction to be processed in the same manner. Next, we search for function calls that satisfy the following criteria: an operand of the instruction aliases with one of the arguments of the function and the function modifies this argument. Finally, stores may introduce dependencies if they alias with the required loads (Algorithm 2, enqueueStores). If a load and a store are in the same function, we use the standard LLVM Alias Analysis, which returns four possible answers: must-alias, may-alias, partial-alias, and no-alias. May-alias might be false-positive; therefore, by including such stores, we build heavier access phases that include potentially redundant instructions. However, by keeping only must-alias stores, we risk missing true may-alias stores. For correctness, we build safe, conservative access phases that preserve all may-alias stores.

However, using only local AA does not address the fact that instructions from different functions might have a dependency (see section 4.1). For this reason, we perform IPMAA. It is important to note that processing one instruction can modify the content of all the vectors. For example, we start by selecting the instructions for AP_tran (see Fig. 7c), but the first instruction is a call. Therefore, we proceed by building the access phase of the called function and selecting its instructions, AP_func. While processing AP_func, the compiler is only aware of the instructions in AP_tran() that have already been visited, but not of the successor instructions in the control flow graph (Fig. 7c, lines 16-17). Hence, at this point, we collect only loads and dependencies required for AP_func, namely the load in Fig. 7c, line 10, which we insert in the AP_func's vector (Fig. 7e). At this point, AP_func's vector does not contain the store (Fig. 7c, line 11), because it is not needed for any instruction residing in any of the vectors. This store will be added when returning to build AP_tran and processing the load on line 16, which has a dependency with the store from AP_func (Fig. 7c, line 11), detected through the interprocedural alias analysis.

5.2.3 Filtering out unnecessary instructions

After all the transactions are processed (Algorithm 1, find_APInst), we are ready to delete all the instructions that are not contained in the helper vectors (Algorithm 1, delete_inst_notInVect). Moreover, we transform leaf load instructions into prefetch instructions (Algorithm 1, replace_loads_with_prefetching). APs that contain stores to globally visible data are wrapped in transactions that abort at the end (see Section 4.3). The result of this step is shown in Fig. 7g. This concludes the building of the AP: selecting the required instructions for prefetching the target loads and removing the remaining, unnecessary instructions.

Table 1: Benchmarks and inputs.

Benchmark	Input
genome	-g512 -s32 -n32768
kmeans-l	-m40 -n40 -t0.05 -i inputs/random-n16384-d24-c16.txt
kmeans-h	-m15 -n15 -t0.05 -i inputs/random-n16384-d24-c16.txt
labyrinth	-i inputs/random-x48-y48-z3-n64.txt
ssca2	-s14 -i1.0 -u1.0 -l9 -p9
vacation-l	-n2 -q90 -u98 -r1048576 -t4096
vacation-h	-n4 -q60 -u90 -r1048576 -t4096
intruder	-a10 -l16 -n4096 -s1

5.3 Step 3: Cleaning

This step performs a ”cleaning” of the generated code, such as removing redundant instructions artificially introduced in Step 2 (e.g. prefetches to the same address).

6 Hardware extensions

We depart from an Intel RTM hardware transaction memory implementation, and add two key extensions to support efficient TAPs. First, we implemented support for *escape actions* to survive interrupts and page-faults (Section 2). This increases the chances that TAPs run until completion. Second, we implemented a custom policy for resolving conflicts with access phase loads (Section 4.3). In particular, TAPs never abort other transactions, but abort themselves in case of a conflict. This optimization is essential not to artificially increase the abort rate of normal transactions.

7 Evaluation and Analysis

7.1 Setup and Methodology

Our code transformations are implemented as a separate compilation pass in LLVM 3.8 [16]. The experiments run on full-system simulation with the Gem5 simulator [4] modified for adding HTM support. We use a simulator instead of a real system because of the hardware extensions required by our proposal. We model a 32-core, haswell-like multi-core running x86 code. Each core is modeled after the timing in-order Gem5 processor model.

We present results for applications from the STAMP transactional benchmark suite [20] with 32 threads. The applications along with their inputs are shown in Table 1. We excluded bayes since it exhibits high variability in its execution time as well as unpredictable behavior for different runs of the same binary [10, 24]. We used both low and high contention configurations for kmeans and vacation.

Table 2: Number of static instructions inside TAPs and binary size overhead.

	Static instructions		**Binary size overhead**	
Benchmark	**allLoads**	**profiledLoads**	**allLoads**	**profiledLoads**
genome	103	98	0.91%	0.45%
kmeans	81	70	0.06%	0.01%
labyrinth	88	84	4.08%	2.84%
ssca2	107	103	4.38%	-0.15%
vacation	127	126	10.59%	6.23%
intruder	99	95	1.29%	-1.29%

We compare the following configurations combining three versions of the programs (**orig**, **allLoads** and **profiledLoads**) with hardware with and without support for *Nack Access Phases* (Section 4.3). All the executables have been compiled with the `-O3` optimization level:

- **orig**: the original code compiled with LLVM.
- **allLoads**: TAPs include only loads (turned into prefetches) and the corresponding program slice (instructions contributing to reaching the prefetch instructions and to computing the target address). Stores are included only if they may-alias, must-alias or partially alias with the selected loads. The APs are wrapped in regular transactions.
- **allLoads-nack**: The *allLoads* version with *nack access phases* support in hardware (see Section 4.3) instead of wrapping the TAPs in regular transactions.
- **profiledLoads**: Like the *allLoads* version but, based on profiling, only loads with long miss latency are selected for prefetching, in an attempt to build lighter access phases.
- **profiledLoads-nack**: The *profiledLoads* version with *nack access phases* support.

Our versions add access phases, which means that the code includes additional instructions, and therefore, an overhead when compared to the original program. The goal is to keep this overhead low (by filtering instructions) such that it can be hidden by the benefits of reducing the abort rate, squashing and re-execution. Furthermore, the profile-based versions aim to include even fewer instructions since they target only the loads that miss in the cache the most (long latency loads). We obtained this information by profiling the original binaries in the simulator. We then annotated the selected loads, detected them with the compiler, and built the corresponding access phases targeting these loads. Our analysis shows that to reach most of the identified loads requires adding an amount of instructions close to the non-profiling versions (see Table 2). Finally, Table 2 also shows the overhead in the size of the binary when creating TAPs, which in the worse case is below 11%.

Table 3: Cache misses and aborts for the baseline configuration. All metrics refer to transactions and transactional accesses.

	genome	kmeans-h	kmeans-l	laby.	ssca2	vaca.-h	vaca.-l	intruder
Read misses	191794	34633	177859	126	92512	283080	229879	153118
Write misses	9957	8279	82096	28	151714	5604	4158	53682
Total reads	4311557	582645	3364280	1244	1499801	3761862	2839445	2366946
Total writes	976743	203942	1183569	293	468685	1194754	906972	843897
Conflict aborts	2496	192943	22655	666	484	20	48	85867
Committed	19257	16595	85092	58	93745	4078	4090	47378
Non-speculative	246	27159	2416	134	24	17	6	7527
Cache miss rate	0.04	0.05	0.06	0.10	0.12	0.06	0.06	0.06
Abort rate	0.13	4.41	0.26	3.47	0.01	0.00	0.01	1.56

7.2 Analysis of the Results

Our aim is to reduce the execution time of the transactions by eliminating cache misses with the goal of reducing conflicts and therefore aborts (see Fig. 1). As shown in the motivation, there are two STAMP applications with potential for our approach: kmeans-h and labyrinth. We include all applications in the evaluation for completion but our technique is only expected to be implemented for such applications. Table 3 elaborates on the analysis done for our motivational analysis (Fig. 1) offering abosute numbers.

First we focus on how varies the number of cache misses in the applications. Fig. 8 shows the number of cache misses normalized to our baseline configuration. Cache misses have been divided accounting for the region of code in which they happen. These regions are listed and explained in Table 4. Our technique reduces the cache misses inside transactions to a great extent for some applications (e.g., vacation). These cache misses are shifted to the TAP region of code. However, for some applications (e.g., genome) extra cache misses occur. This happens because genome is one of the benchmarks where transactions change their reading and writing sets between retries (see table 5), that is, the target address of a cache access prefetched during the TAP region changes and misses again during the execution of the transaction. Note that, as expected, kmeans-h and labyrinth reduce the misses during the transaction phases.

Then, we focus on the reduction of aborts, mostly due to conflicts. We show both the normalized number of transactions that abort (Fig. 9, showing the transaction in the code that aborts and Fig. 11, showing the reason of the abort) and the normalized number of cycles spent in the aborted transaction (Fig. 10, showing the transaction in the code that aborts, and Fig. 12, showing the reason of the abort). The legend in Fig. 9 and Fig. 10 represents the transaction identifier (applications have up to five transactions). The As we expected, the abort rate is reduced for most applications, which also affects the number of cycles spent in aborted transactions. However, sometimes we can

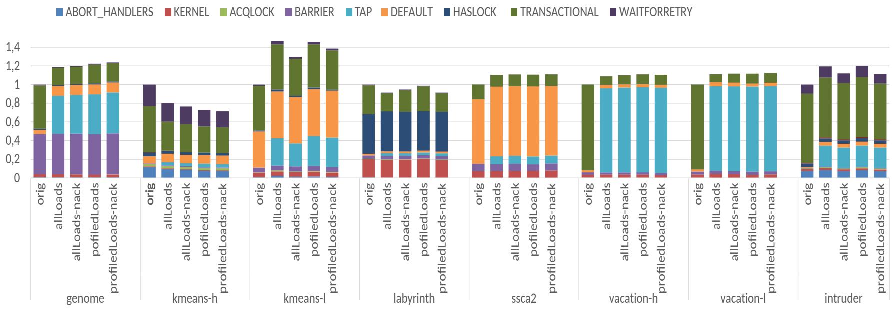

Fig. 8: Normalized number of cache misses.

Table 4: Notations for the execution phases.

Execution phases	Notation
TAP	Cycles spent inside transactional access phases
Barrier	Cycles spent waiting for all the threads to reach the barrier
HasLock	Cycles spent inside fallback paths
Default	Non transactional code
WaitForRetry	Cycles transactions wait for re-attempt
Abort_Handlers	Cycles spent inside abort handlers for each abort
Transaction_Commit	Cycles inside transactions that committed
Transaction_Aborted	Cycles inside transactions that aborted
Kernel	Cycles inside system calls, interrupts, page faults, etc.
Acqlock	Cycles spent acquiring fallback lock

Table 5: Transactions with changing reading and writing sets are marked with ✓. Grey cells mean that a benchmark does not have transactions with this ordinal number.

Benchmark	TX0	TX1	TX2	TX3	TX4
genome	✓		✓	✓	✓
intruder	✓	✓	✓		
kmeans-h					
kmeans-l					
labyrinth					
ssca2					
vacation-h	✓		✓		
vacation-l	✓		✓		

see an increase in conflicts, mostly when nack access phases (see Section 4.3) are not enabled, as TAPs can abort transactions in this case.

Finally, we show the overall application execution time in Fig. 13. Execution time is normalized to the baseline configuration, and it is also broken down into disjoint components (see Table 4). As it can be observed, the execution time of the *Transactional_Aborted* region is reduced because the number of aborts due to conflicts is lower. As a consequence, the time spent in *Abort_Handlers*, *WaitForRetry*, and *HasLock* is also reduced. The time spent in the abort handlers decreases because if fewer aborts happen, those han-

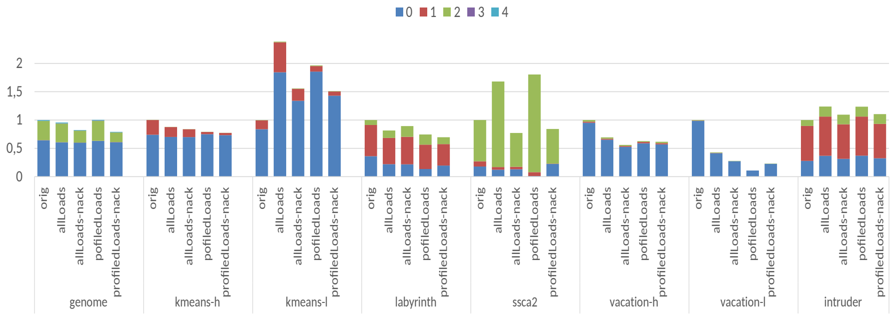

Fig. 9: Normalized number of transactions that abort per transactional region.

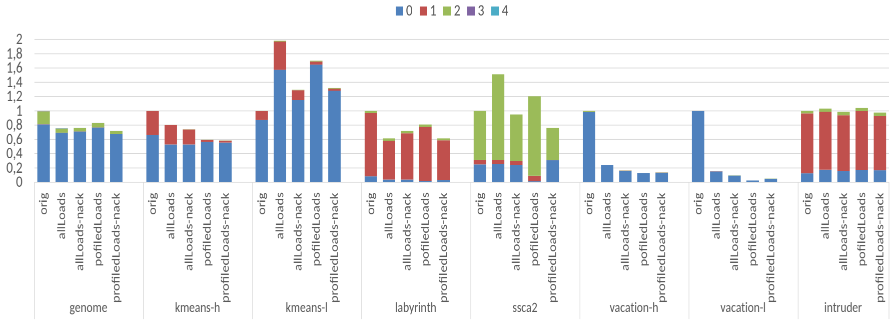

Fig. 10: Normalized number of cycles spent in aborted transactions per transactional region.

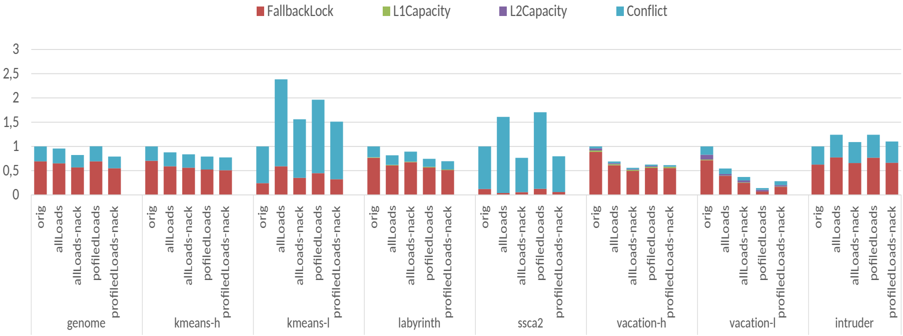

Fig. 11: Normalized number of transactions that abort per cause.

dlers execute less often. Besides, thanks to reducing the number of aborts, transactions have to retry and to resort to irrevocable execution using mutual exclusion fewer times, reducing the time that some transactions spend executing in mutual exclusion (`HasLock`) and the time that other transactions wait until the first ones finish executing (*WaitForRetry*).

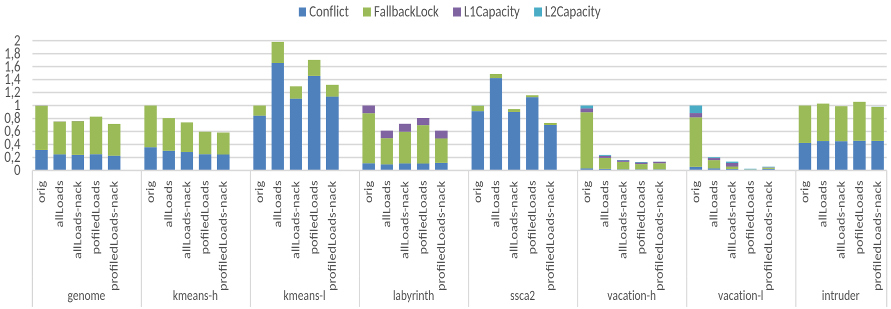

Fig. 12: Normalized number of cycles spent in aborted transactions per cause.

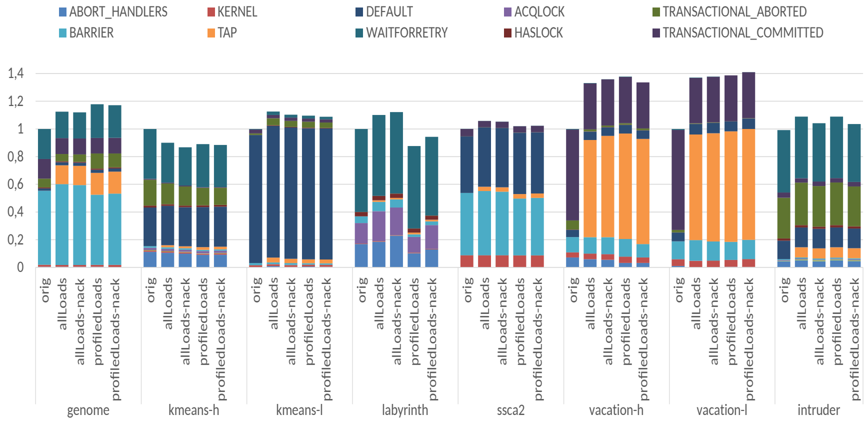

Fig. 13: Normalized execution time.

By shifting cache misses from the transactional phases to the TAPs, we also expect a decrease the *Transaction_Committed* execution time, and a corresponding increase in execution time the new category *TAP*, as reported in Fig. 13. The time spent in TAPs needs to be smaller than the decrease in all other categories to avoid increasing the total execution time. On the other hand, the time spent in non transactional code (the *Default* category) should stay the same, as we do not affect non-transactional code if TAPs prefetch only the data that transactions use.

As expected, we see performance improvements only in kmeans-h and labyrinth. The other applications degrade performance mostly due to the overhead introduced by the TAPs instructions, although the time spent in synchronization (*Barrier*) increases in some cases (genome and labyrinth, when not using profiling). The time spent executing transactions that ultimately abort (*Transactional_Aborted*) does decrease in general, but remarkably it increases for kmeans-l and ssca2, where the number of aborts increases (see Fig. 9 or Fig. 11).

We have used profiling to reduce the overhead introduced by TAPs. This profiling is beneficial for some benchmarks like labyrinth and ssca2, but does not make a great difference in most cases and is detrimental for performance in vacation-l and genome. The reason is that *profiledLoads* binaries might exclude some loads with a relatively low miss rate, but those loads give the advantage to *allLoads* binaries to prefetch more data, therefore gaining some extra performance. Also, in most cases the time spent in access phases is not reduced thanks to the profiling because to be able to calculate the addresses of the identified loads requires often adding a number of instructions similar to the non-profiling versions due to data and control flow dependencies (see Table 2).

Since TAPs execute transactionally, they can abort and can make other transactions abort, negatively affecting performance. As discussed, this explains the increase in the number of aborts and time spent executing aborted transactions (see Fig. 9 and Fig. 11). In addition to this, an aborted TAP does not completes execution and may not prefetch all data in the transaction. Note that TAPs might abort at the beginning of the execution.

When enabling Nack access phases (see Section 4.3), the performance is affected only slightly, improving the execution time in intruder and increasing it in labyrinth. This technique does reduce the number of aborted transactions, as can be seen in Fig. 9 for most applications, and the number of cycles spent executing them as can be seen in Fig. 10, but these reductions do not have a positive effect in the final execution time except in the case of intruder and kmeans-h when not using profiling.

Finally, We performed simulations when varying the number of threads and observed improvements in kmeans-h and labyrinth when running more than 8 threads. As we increase the number of threads, the performance improvements of our approach increase for kmeans-h and labyrinth. This is expected as more aborts due to conflicts appear in those applications.

8 Future work

Although our approach improves the abort rate by up to 90% for some benchmarks, the extra execution time of TAPs jeopardizes performance improvements. Finding a good trade-off between the size of TAPs and their prefetching efficiency by further reducing TAPs instructions is a promising future work. We envision that effective heuristics can be designed based on the number of dependent loads required to reach the targeted loads [15] or on the number of jumps or branches required to place the target load in the TAP [30].

Additionally, not all transactions are suitable for our technique. For example, for transactions that work on different data sets on each run, TAPs would fail to prefetch the correct data. Static analysis can help to identify the transactions that are unlikely to change their working sets.

In the evaluation performed in this work, we apply TAPs to all transactions, even though some of them have a low abort rate and miss rate. One future work direction is to monitor the abort rate and prefetch effectiveness, and dynamically disable TAPs when they are predicted to be ineffective.

Finally, we have evaluated our technique using the STAMP benchmark suite, finding only two cases for which our technique provides performance improvements. Since STAMP benchmarks present a poor scalability [22, 25], a large study considering other transactional benchmarks [31] will be interesting in order to better assess the reach of our proposal.

9 Conclusions

Improving the abort rate for applications that use HTM is extremely important, given the cost of each abort. Doing it at a hardware level requires many changes, while the software brings lightweight solutions with high flexibility and portability. In this work, we propose a new technique to improve the abort rate due to conflicts and to improve the scalability of transactional applications. Particularly, we make the observation that by turning long latency loads within transactions into cache hits, we can decrease the execution time of transactional regions, and thus reduce the likelihood for conflicts. Our approach exploits this observation by using software prefetching instructions inserted automatically at compile-time.

Our simulation results using the Gem5 simulator and applications from the STAMP benchmark suite show that our approach can significantly benefit applications with transactions that abort frequently and incur a large number of cache misses. Particularly, we obtain reductions in the number of aborts of 30% on average and performance improvements of 19% and 10% for two (kmeans-h and labyrinth) out of the eight evaluated applications. In addition to reporting on the impact that our technique has on the performance of the applications under consideration, we also provide in-depth analysis for each application and identify the key characteristics of the transactional regions that can benefit from our approach.

References

1. Ansari, M., Khan, B., Luján, M., Kotselidis, C., Kirkham, C., Watson, I.: Improving performance by reducing aborts in hardware transactional memory. In: High Performance Embedded Architectures and Compilers, pp. 35–49 (2010)
2. Ansari, M., Luján, M., Kotselidis, C., Jarvis, K., Kirkham, C., Watson, I.: Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In: Proceedings of the High Performance Embedded Architectures and Compilers, pp. 4–18 (2009)
3. ARM Ltd.: Transactional memory extension (tme) intrinsics. URL https://developer.arm.com/documentation/101028/0011/Transactional-Memory-Extension–TME–intrinsics
4. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N.,

Hill, M.D., Wood, D.A.: The gem5 simulator. Computer Architecture News **39**(2), 1–7 (2011)

5. Dash, A., Demsky, B.: Automatically generating symbolic prefetches for distributed transactional memories. In: Middleware 2010. Lecture Notes in Computer Science,, vol. 6452 (2010)
6. Dash, A., Demsky, B.: Integrating caching and prefetching mechanisms in a distributed transactional memory. IEEE Transactions on Parallel and Distributed Systems **22**(8), 1284–1298 (2011)
7. Dice, D., Herlihy, M., Kogan, A.: Improving parallelism in hardware transactional memory. ACM Transactions on Architecture and Code Optimization **15**(1), 1–24 (2018)
8. Diegues, N., Romano, P.: Time-warp: lightweight abort minimization in transactional memory. In: Proceedings of the Symposium on Principles and Practice of Parallel Programming, pp. 167–178 (2014)
9. Diegues, N., Romano, P., Garbatov, S.: Seer: Probabilistic scheduling for hardware transactional memory. ACM Transactions on Computer Systems **35**(3) (2017)
10. Dragojevic, A., Guerraoui, R.: Predicting the scalability of an stm. In: 5th ACM SIGPLAN Workshop on Transactional Computing. (2010)
11. Harris, T., Larus, J., Rajwar, R.: Transactional Memory, 2nd edn. Morgan & Claypool Publishers series (2010)
12. Jacobi, C., Slegel, T., Greiner, D.: Transactional memory architecture and implementation for IBM system Z. In: Proceedings of the International Symposium on Microarchitecture, pp. 25–36 (2012)
13. Jimborean, A., Koukos, K., Spiliopoulos, V., Black-Schaffer, D., Kaxiras., S.: Fix the code. dont tweak the hardware: A new compiler approach to voltage-frequency scaling. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 262–272 (2014)
14. Koukos, K., Ekemark, P., Zacharopoulos, G., Spiliopoulos, V., Kaxiras, S., Jimborean, A.: Daedal decoupled access-execute llvm tools repository (2016). URL https://github.com/etascale/daedal
15. Koukos, K., Ekemark, P., Zacharopoulos, G., Spiliopoulos, V., Kaxiras, S., Jimborean, A.: Multiversioned decoupled access-execute: the key to energy-efficient compilation of general-purpose programs. In: Proceedings of the 25th International Conference on Compiler Construction, pp. 121–131 (2016)
16. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 75–88 (2004)
17. Le, H.Q., Guthrie, G.L., Williams, D.E., Michael, M.M., Frey, B.G., Starke, W.J., May, C., Odaira, R., Nakaike, T.: Transactional memory support in the ibm power8 processor. IBM Journal of Research and Development **59**(1), 8:1–8:14 (2015)
18. Litz, H., Cheriton, D., Firoozshahian, A., Azizi, O., Stevenson, J.P.: Si-tm: reducing transactional memory abort rates through snapshot isolation. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 383–398 (2014)
19. Maldonado, W., Marlier, P., Felber, P., Suissa, A., Hendler, D., Fedorova, A., Lawall, J.L., Muller, G.: Scheduling support for transactional memory contention management. In: Proceedings of 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 79–90 (2009)
20. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: Proceedings of The IEEE International Symposium on Workload Characterization, pp. 35–46 (2009)
21. Moravan, M.J., Bobba, J., Moore, K.E., Yen, L., Hill, M.D., Liblit, B., Swift, M.M., Wood, D.A.: Supporting nested transactional memory in LogTM. In: Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems, pp. 359–370 (2006)
22. Nakaike, T., Odaira, R., Gaudet, M., Michael, M.M., Tomari, H.: Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture, pp. 144–157 (2015)

23. Negi, A., Armejach, A., Cristal, A., Unsal, O.S., Stenstrom, P.: Transactional prefetching: narrowing the window of contention in hardware transactional memory. In: Proceedings of the 21st international conference on Parallel architectures and compilation techniques, pp. 181–190 (2012)
24. Negi, A., Walliullah, M., Stenstrom, P.: Lv*: A low complexity lazy versioning htm infrastructure. In: Proceedings of the 25th International Conference on Embeded Computer Systems: Architectures, Modeling, and Simulation, pp. 231–240 (2010)
25. Nguyen, D., Pingali, K.: What scalable programs need from transactional memory. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 105–118 (2017)
26. Ritson, C., Barnes, F.: An evaluation of intel's restricted transactional memory for cpas. Communicating Process Architectures pp. 271–292 (2013)
27. Sui, Y., Xue, J.: SVF: interprocedural static value-flow analysis in LLVM. In: Proceedings of the 25th International Conference on compiler construction, pp. 265–266 (2016)
28. Sui, Y., Ye, D., Xue, J.: Detecting memory leaks statically with full-sparse value-flow analysis. IEEE Transactions on Software Engineering **40**(2), 107–122 (2014)
29. Titos-Gil, R., Fernández-Pascual, R., Ros, A., Acacio, M.E.: Pftouch: Concurrent page-fault handling for intel restricted transactional memory. Journal of Parallel Distributed Computing **145**, 111–123 (2020)
30. Tran, K.A., Carlson, T.E., Koukos, K., Själander, M., Spiliopoulos, V., Kaxiras, S., Jimborean, A.: Clairvoyance: look-ahead compile-time scheduling. In: Proceedings of the 2017 International Symposium on Code Generation and Optimization, pp. 171–184 (2017)
31. Wang, Q., Su, P., Chabbi, M., Liu, X.: Lightweight hardware transactional memory profiling. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, pp. 186–200 (2019)
32. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, pp. 439–449 (1981)
33. Weiser, M.: Program slicing. IEEE Transactions on Software Engineering **10**, 352–357 (1984)
34. Xiang, L., Scott., M.L.: Conflict reduction in hardware transactions using advisory locks. In: Proceedings of the Symposium on Parallelism in Algorithms and Architectures, pp. 234–243 (2015)
35. Yoo, R., Hughes, C., K.Lai, Rajwar, R.: Performance evaluation of intel transactional synchronization extensions for high performance computing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–11 (2013)