UPPSALA
UNIVERSITET

IT 15030
Examensarbete 30 hp
Juni 2015

Employing Hardware Transactional Memory in Prefetching for Energy Efficiency

Georgios Zacharopoulos

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Employing Hardware Transactional Memory in Prefetching for Energy Efficiency

Georgios Zacharopoulos

Energy efficiency is becoming a highly significant topic regarding modern hardware. The need for decreased energy consumption in our computers and more battery life in our laptops and smart-phones is increasing, without sustaining performance loss in our machines. Much work is being conducted towards that cause and as a result our lives could become more convenient. For serving the purpose of this project, we have investigated the implementation of Hardware Transactional Memory (HTM) in the prefetching phase of Decoupled Access/Execute (DAE) model [1]. The challenge posed by using DAE model is to make sure the memory state remains intact while prefetching data. We propose a solution to overcome this challenge by employing HTM that is supported by Intel's latest processors. An innovative approach of HTM was carried out, in order to achieve the final implementation of it in the Access phase of the DAE model. Evaluation proved that benefits resulting from the DAE model utilization can be maintained by our approach. Furthermore, we are able to extend the use of the model to more applications that was previously not possible.

Handledare: Alexandra Jimborean
Ämnesgranskare: Stefanos Kaxiras
Examinator: Edith Ngai
IT 15030
Tryckt av: Reprocentralen ITC

To my parents, Areti and Dimitris.

For being always there for me.

Contents

List of Figures

List of Tables

List of Algorithms

1 Background for the project

Energy consumption of modern hardware is one of the most important issues, when it comes to research related to computer architecture. For a more efficient execution of tasks to be reached, the Decoupled Access/Execute (DAE) model was introduced by Konstantinos Koukos et al [1]. The DAE approach divides the execution of a task into two parts, the Access and the Execute phase. DAE has been proven to be efficient at saving energy while maintaining performance. Nevertheless, DAE can not possibly be applied in applications where pointers are present. Due to pointer-aliasing, the compiler cannot guarantee statically that the Access phase will be side-effect free. This calls for a solution, which is why Hardware Transactional Memory is proposed in this thesis' scope.

1.1 Decoupled Access Execute (DAE) Model

DAE is a task-based model which can be applied to both sequential and parallel processing applications. In this project we will be focusing on sequential code. The philosophy of the DAE model is that each task is divided into two phases [1]: the access phase, during which data are being prefetched into the cache and the execute phase, where the original computation of the data takes place. Since the access phase is memory-bound, we can run at low frequency without hurting the performance and save energy at the same time. The execute phase starts right after the access phase and can now run at high frequency, having prefetched the data, and thus result into fewer cache misses.
The DAE model has been extended with the use of a compiler methodology to automatically generate the access phases for a task-based programming system by Alexandra Jimborean et al [2]. The problem that needs to be addressed is to make sure that we prefetch the data during the access phase, without changing the memory state. In more detail, we want to allow stores to local variables in the access phase so as to achieve control flow and compute the memory addresses that are prefetced, without having any stores to values visible outside the task. Making sure that the memory state does not change is an issue that needs to be carefully handled, as pointers can be a problem, due to pointer aliasing.
The solution that is proposed to overcome that is the use of Hardware Transactional Memory. The access phase can be run transactionally and get aborted on demand when its execution is over. The outcome would be to force globally visible data coming from stores be invalidated, which keeps the memory state unmodified. Data accessed for reading however will be in the cache, thus having prefetching effects.

1.2 Hardware Transactional Memory

Hardware Transactional Memory (HTM) is an **optimistic** way to execute a critical section, which is a specific area of code which two or more threads should not execute at the same time. This is due to data race issues that might come up as a result.

It is worth stating that terms such as hardware transactions and optimistic concurrency are not something new in modern processors. In every processor with a branch predictor, speculative execution of instructions takes place after a branch, either by committing the results or aborting and discarding the changes. This is being done based on whether the branch is really taken or not [3].
Transactional memory could be seen as a simple extension of this design. The state of the cache as well as the state of the registers can be conceived as being part of the processor's internal state, rather than being viewed globally. IBM's Blue Gene/Q has supported HTM since 2011 [4], and Sun's Rock CPU had also intended to do so, before Oracle cancelled it. Furthermore a large number of research CPUs over the years have also been proposed. For HTM to be beneficial, however, we need a coherence protocol and large caches, which is something that has been available only since the past decade. Having large caches, a CPU is able to buffer a large number of memory writes without having to store them to main memory. A multicore CPU already provides cache coherency (MOSI/MOESI protocol). This ensures that cache lines representing the same memory address in different cores are synchronized. The way this is achieved is by maintaining a state for each cache line and requiring it to be in the exclusive state, which means that it is not present in any other cores' caches, before it can be written. In other words, it is being monitored whether a core has tried to modify a memory address that any other core might have in its cache.
The idea behind HTM is to allow two cores to execute code speculatively if they touch global data. Both should succeed, unless they touch the same bit of a global state. As an example, let us suggest two or more cores updating entries in a tree. All cores will read from the root to a leaf and then each one will modify a leaf. In a regular locking approach, we might have a single lock for the whole tree, so that accesses would be serialized. With an HTM approach, each one starts a transaction, reaches the desired node, and then updates it. These transactions will always succeed, unless two cores try to update the same node.
This is the reason hardware keeps track of which cache lines have been read from where and which have been written to. If one core tries to read the same cache line address as another core reads, then there is no conflict. But in case one writes to a cache line

that another has already read or written to, then there is a conflict, and one of the transactions should be aborted. Selecting the transaction to be aborted depends on the implementation.

To summarise, the main concept behind HTM is to **execute critical sections optimistically**, instead of using unnecessary locking. The way to achieve that is by keeping track of read/write sets and abort on read/write or write/write conflict. The cache-coherency protocol is used to detect those conflicts. **L1 cache serves as a transactional buffer** and the tracking is done at cache line granularity (64 bytes).
It is clear that since we are going to execute a critical section optimistically, different kinds of aborts are imminent. Some examples of aborts that should be taken into consideration while using HTM are capacity aborts, conflict aborts, aborts due to false-sharing, aborts caused by hardware interrupts and explicit aborts that the developer may want to implement, as in the case that we are going to investigate.

Even though HTM has been designed for parallel processing applications, we are going to apply it in sequential code and in a **different way** than it is generally used. The way we are going to implement HTM in order to serve the purposes of the project is explained in detail in the 3 Methodology section.

1.3 Intel's Hardware Transactional Memory

It wasn't until recently that Intel included hardware support of Transactional Memory in the Haswell architecture. Intel's processors support two interfaces regarding HTM. Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM) [5].

Intel's TSX (Transactional Synchronization Extensions) instructions let the processor expose and exploit concurrency that may be hidden in an application due to dynamically unnecessary synchronization [6]. Areas of code that are executed transactionally are called Transactional Regions. Any architectural updates, due to memory operations inside transactions become visible to other logical processors only on a successful commit, which can also be regarded as an atomic commit. Intel TSX maintains read and write-set at the granularity of a cache line. Therefore, conflicts within transactions are detected at the granularity of a cache line. [1]

[1] Intel recently announced an erratum and according to that, Transactional Synchronization Extensions (TSX-NI) are going to be disabled in its future products that support Hardware Transactional Memory [7]. A software developer discovered the erratum through testing and Intel later confirmed it.

1.3.1 Hardware Lock Elision (HLE)

HLE provides two instruction prefix hints: XACQUIRE and XRELEASE. XACQUIRE is inserted before the instruction that is used to acquire the lock that protects the critical section. The write associated with the lock acquire operation is then elided by the processor.

The lock acquire has an associated write operation to the lock, but the processor does not add the lock's address to the write-set of the transaction and also does not issue a write request to the lock. The address of the lock is added to the read-set and the logical processor resumes to transactional execution.

All other processors will continue to see the lock as available, if it was available before the XACQUIRE instruction. Since the processor that executes transactionally did not add the address of the lock to its write-set and no externally visible write operations were performed to it, other processors can read the lock without data conflicts being caused. So other processors can also enter and concurrently execute the critical section protected by the lock. Moreover, a transactional abort will be performed if necessary, as the processor monitors data conflicts that might occur during the transactional execution.

As we can see, no external write operations to the lock take place, but even so the hardware ensures program order of operations on the lock. If the processor that elides the lock, reads the value of the lock in the critical section, it will appear as if the processor had acquired the lock, and so the read will return the non-elided value. This makes an HLE execution functionally equal to an execution without the HLE prefixes.

The XRELEASE instruction is used before the instruction that would release the lock protecting the critical section. This also means a write to the lock. In the case that the instruction restores the value of the lock to the value it had before the XACQUIRE operation on the same lock, then the processor elides that external write request associated with the lock release and the address of the lock is not added to the write-set. Next the processors attempts to commit the transactional execution.

Intel disabled the TSX instructions via a microcode update, so Hardware Lock Elision (HLE) and Restricted Trsansactional Memory (RTM) will not be available any more. This of course will slow down the development of TSX-enabled software. Nevertheless, there is the option to enable TSX via the BIOS/-firmware menu for those who need to experiment, even though Intel recommends waiting for Haswell-EX. So it can be supported but only for software development reasons. In Intel's official Errata document it is listed as HSW136. Its description of the problem is that the Intel TSX instructions may result in unpredictable system behavior under software using a complex set of internal timing conditions and system events. It is also stated that it is possible for the bios to contain a workaround for this erratum. For all our experiments we used Intel's i7-4790K processor that does support TSX instructions.

HLE allows multiple threads to execute inside critical sections protected by the same lock, given that they do not perform any conflicts on each others data. So the threads can execute concurrently and with no need of serialization. The software lock acquisition operations on a common lock are recognized by the hardware. As a result it elides the lock and executes the critical sections on the two threads for instance, without requiring any unnecessary communication through the lock if such communication was indeed dynamically unnecessary.
In the case that the processor is not able to execute the region transactionally, then it will execute the region non-transactionally and without elision. HLE software guarantees forward-progress, the same as the non-HLE lock-based execution does. HLE is backwards compatible in the sense that hardware without HLE support will just ignore XACQUIRE and XRELEASE prefixes, as well as they can easily be implement to existing lock-based code without causing any bugs.

For HLE execution to successfully commit transactionally, the lock must satisfy certain properties and access to the lock must follow certain guidelines.

- An XRELEASE prefixed instruction has to restore the value of the lock that was elided to the value it had before the lock was acquired. That way, hardware safely elides locks by not adding them to the write-set. The data size and data address of the lock release (XRELEASE) instruction must match that of the lock acquire (XACQUIRE) and the lock must not go over a cache line limit.

- Software must not write to the elided lock inside a transactional region with any instruction other than XRELEASE, otherwise a transactional abort may take place. Moreover, recursive locks (where a thread gets the same lock multiple times without first releasing the lock) may also result to a transactional abort. Software may also observe the result of the elided lock acquire inside a critical section. So a software read operation will return the value of the write to the lock.

1.3.2 Restricted Transactional Memory (RTM)

Main difference between RTM and HLE is that RTM allows us to have better control over the transactional regions and thus over the overall code. This is achieved by implementing a fall-back path in case of any abort that might happen and this way the forward-progress of our software is ensured.
RTM provides three instructions: XBEGIN, XEND and XABORT, to start, commit and explicitly abort a transactional execution.

XBEGIN instruction specifies the start of a transactional region and makes the logical processor to start a transactional execution. A value is returned that states whether a transaction has successfully started or the abort status, in the case the transactional execution got aborted.
It takes an operand which provides a relative offset to the fall-back instruction address, given that a region could not be executed transactionally. An RTM transactional execution can get aborted for various reasons (capacity, conflict, hardware interrupts, explicit abort instruction, false sharing etc). Hardware detects a transactional abort and restarts execution from the instruction address with the architectural state associated with the one at the start of XBEGIN and the EAX register updated to the specific abort status.

```
__inline unsigned int _xbegin()
{
  unsigned status;
  __asm {
   move    eax, 0xFFFFFFFF
   xbegin _txnL1
 _txnL1:
   move    status, eax
 }
 return status;
}
```

If a transaction is created successfully, then the function returns 0xFFFFFFFF. In the case that transaction gets aborted then the logical processor discards any register and memory updates and restores the state to that of the outermost xbegin instruction. The EAX register gets updated with the status of the aborted transaction.

XEND instruction specifies the end of a transactional region. If this is the outermost instruction, then the processor will try to commit its state. In the case it fails to commit, the processor will rollback all memory and register updates performed during the execution and resume execution at the fall-back address from the outermost xbegin instruction. EAX register is also updated to the current abort.

```
__inline void _xend()
{ __asm {
xend
  }
}
```

XABORT allows for explicit abort of the execution. It forces a transaction to abort and the logical processor resumes execution at the fall-back address of the outermost xbegin instruction. It takes an 8-bit argument that is loaded into the EAX register, that makes it available to the software following an RTM abort.

```
__inline void _xabort()
{ __asm {
    xabort
  }
}
```

On an abort, execution is rolled back and all updates performed in the Transactional Region are discarded. Moreover, architectural state of the processor is restored, as if the optimistic execution never occurred.
If an alternate fall-back path is not provided, then no forward-progress is guaranteed, as an RTM region might not commit successfully.

2 Related Work

The last years a lot of research has been conducted regarding Hardware Transactional Memory [8] [9] [10] and particularly on Intel's latest processors that support it [12].

According to previous work regarding synchronization overhead by using HTM [8], lock elision is found to improve scalability when: a) There is little contention between the work different threads do in a critical section and b) There is contention for the lock protecting the critical section. Moreover a sufficient increase in throughput (20-25 %) has also been reached, comparing TSX modified and coarse grained locking, while testing LevelDB benchmark suite for write-only workloads.

In a different piece of work, regarding use of HTM in memory databases [9], a summary of significant limitations were concluded after testing:

- The size (32KB) and associativity (8-way) of L1 cache can limit transaction size and therefore cause capacity aborts.
- Interrupts and context switches can limit the duration of the transaction.
- Certain (fairly uncommon) instructions, such as XABORT, CPUID and PAUSE always cause an abort.
- No forward-progress is guaranteed using RTM, unless a fall-back path is implemented.

Nevertheless, based on experiments ran on high-performance database systems, HTM has been proven to perform better, both compared to **coarse-grained locking**, which ensures synchronization but trades performance and to **fine-grained locking** mechanisms that are more complex and can lead to deadlocks due to differences in locking order.

Dai Wang, Mike, et al [10], showed that performance can also be affected by the following factors:

a) Transaction Size

b) Retry count

c) Write ratio of the application

d) Randomness in the access pattern of the application

e) Intel's interface. (HLE or RTM)

Some really interesting findings were reached, after experiments were run on a micro-benchmark, specifically a one dimensional integer array, where its elements are accessed through either reads or writes based on two different access patterns. (Random or contiguous) The findings can be summarized in the following 4 points:

1. Smaller transactions, as expected, results to higher likelihood of committing.
2. Lower abort rate does not necessarily mean higher overall performance.
3. Regarding the access mode, contiguous access has better performance than random access.
4. By retrying aborted hardware transactions many times, abort rate reduces.

More experiments, regarding performance and energy analysis of RTM interface on Haswell [11] confirm some of the previous findings. Performance depends on workload characteristics and in general suffers less overhead than Software Transactional Memory. Moreover, capacity tests show that the write set of a transaction should not exceed 32 KB and the read set can be as big as L3 cache (8 MB), before aborting due to capacity limitations.

To sum up, in most recent research papers, Intel's HTM was found to achieve better performance, good parallelism and simplicity compared to other locking techniques. Moreover, HTM offers great scalability, given that transaction regions are small.
We underline that this work targets sequential code, thus the **abort rate** varies mostly with respect to the **transaction size** and the number of loads and stores within the transaction.

3 Methodology

Our final goal is to achieve data prefetching during the access phase without changing the memory state. So, in order to accomplish that, we wish to put the Access phase inside a Transaction and then explicitly abort it. Right after the abort takes place, the Execute phase is going to take over, resuming to the actual execution of the task.

3.1 Identification and printing of Unsafe Store Instructions

To begin with, we need to take a look at the kind of input we are going to have for our final implementation and testing. We retrieve the Llvm code which will be used as input and follows the Decoupled Access/Execute scheme [13]. In one example of an Llvm file, for each load instruction we detect all store instructions that may store the value that the load instruction reads. Furthermore, we measure the level of alias between the address of the load and and the address of each store instruction. The code has annotations marking different level of unsafe write instructions. The compiler automatically generates the access-execute versions and marks the write accesses that are executed speculatively in the access phase.

There are four categories of instruction alias, according to which we find out the unsafe ones.

1. MustAlias, which translates to a certain match between a load and a store instruction.
2. PartialAlias, meaning that the memory areas of each address refer to overlap.
3. MayAlias, where there is no proof that two addresses overlap, nor that they do not. This is the default description.
4. NoAlias, where there is no way that addresses point to the same memory location.

The first step is to write an Llvm pass analysis to print the unsafe writes [14] . In this case our interest focuses on the first two categories: **Must Alias** and **Partial Alias**. The above annotations are marked in the Llvm files as metadata attached to the store instructions. We wrote an Llvm pass analysis to identify all unsafe store instructions and print them out to standard output. We achieved that by iterating over each function

in a module, then over each basic block in the function and identify the store instructions that had those specific metadata attached. Moreover we wish to retrieve the exact number of these unsafe store instructions for each given application.

An example of such **unsafe store instruction** is the following:

```
store i32 %add, i32* %loadgep_bsLive, align 4, !GlobalAlias !1
```

We want these unsafe stores to be wrapped inside transactions during the Access phase, so as to preserve the memory state.

3.2 Investigate HLE and RTM

Moving on, we investigate which of Intel's HTM interfaces suits best our cause. We wish to wrap the unsafe store instructions into transactions and manually abort the transactions before they commit. Practically, the access phase will get aborted as soon as it finishes execution and the execute phase will take over and benefit from the prefetching positive effects of HTM.

To better investigate that, we created **3 versions** of the same micro benchmark written in C that use the same critical sections.
The first version uses a simple locking mechanism (coarse grained locking), whereas the other two use Restricted Transactional Memory and Hardware Lock Elision respectively. We compare these micro-benchmarks in terms of performance (Computer Cycles). Furthermore we monitor the abort-rate of a transaction in respect to the size of the critical section and number of Load and Store instructions in it.

Algorithm 1 RTM implementation for multi-threaded applications

```
while 1 do
    XBEGIN
    if lock is taken() then
        XABORT
    Critical Section(SIZE)
    XEND
    if ! Reasons to retry then
        break

/ * Fallback Path * /
lock()
Critical Section(SIZE)
unlock()
```

In a general basis, regarding multi-threaded execution, RTM is implemented as shown above. We enter the transactional area and we first check whether the lock is occupied or not. (This safe-check is implemented additionally as as an optimization). If it is, then we abort and retry the transaction. In the most likely case that it is not, we go ahead, execute the critical section and then we commit. In case of an abort prior to commit instruction, we are going to check the reason for aborting. Given that it is not a restrictive one (e.g. abort due to capacity limitations) we are going to retry the transactional execution. In any other case, we are going to quit and take the fall-back path. This is going to lead to a regular (coarse-grained) locking execution and practically means that we failed to execute the critical section area transactionally. On the bright side we make sure that the execution of the critical section will take place under any circumstances, thus being able to guarantee the forward-progress of the application.

Results in Figure 3.1 showed that for a small critical sections (1/8 of the L1$ for this experiment), RTM had **better performance** than HLE and regular locking when we use 4 threads comparing to just 2. This is because HTM benefits from a higher contention for acquiring the lock. On the other hand we can see that for a 2-threaded application we have similar performance to coarse-grained locking and even a bit worse for RTM for a higher number of iterations. Every iteration tries to start a new transaction in order to execute the critical section and starting a transaction is costly. Moreover, we tested a sequential (single-threaded) version of RTM, by setting a fixed number of iter-

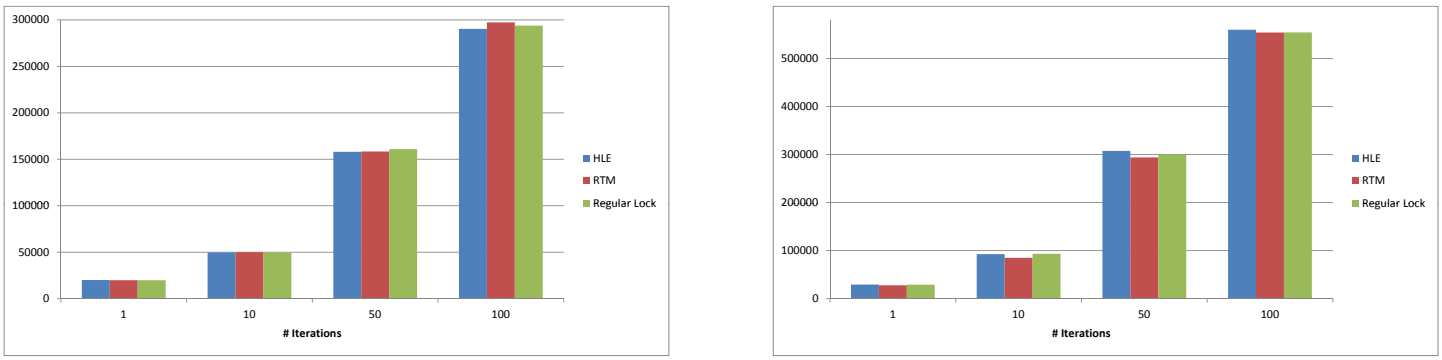


Figure 3.1: Performance of HLE vs RTM vs Regular Locking in multithreaded applications. Having 2 Threads and 4 Threads respectively.

ations to 1 million and increasing the size of the transaction. In Figure 3.3 we can see that **abort ratio** rises sharply and reaches almost 100% as we approach the L1D cache threshold. This is exactly what we expected, as L1 $ serves as a **transactional buffer** and its capacity for this case is 32 KB, regarding store instructions. Finally, another interesting point is the way the number of load and store instructions varies as the size of the transaction increases. Load instructions rise rapidly as we get more aborts and need to restart a transaction in order to execute the critical section or take the fall-back path and execute non-transactionally. Store instructions get affected too, but not in the exact same way. There is a gradual increase in writes, as the size of the transaction gets larger and larger and this rise of course gets sharper when aborts start taking place.

Restricted Transactional Memory is in general harder to implement, but provides more flexibility to use and gives better control on the overall execution. Moreover, it gives us the possibility to control the fallback path that will be taken, either by a regular abort or an explicit abort, which is exactly what we want to achieve for this project.

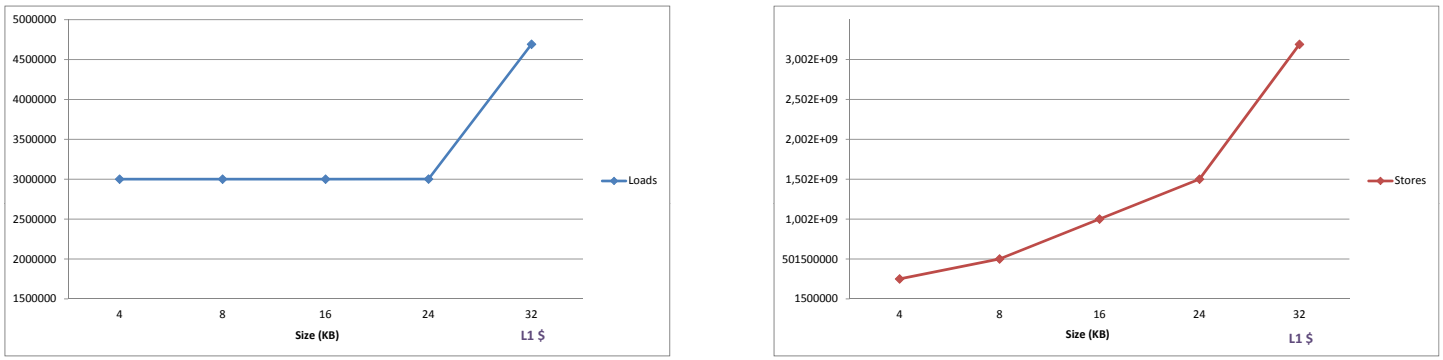


Figure 3.2: Load and Store instructions for Sequential RTM

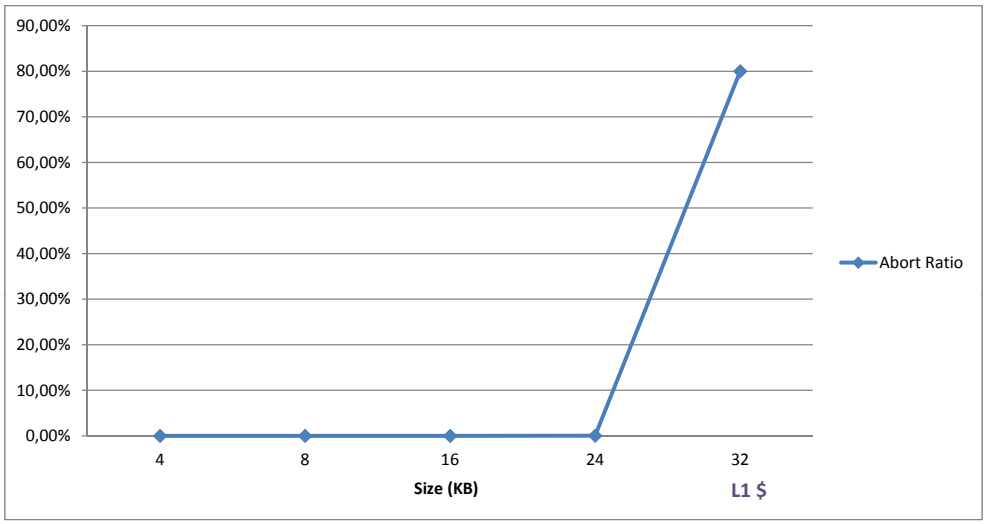


Figure 3.3: Abort Ratio for RTM

3.3 Compare RTM and original version in LLVM IR

As far as our investigation is concerned regarding DAE model, code will be executed sequentially and there will be no locking mechanisms.
We created a sequential micro-benchmark (single-threaded) that executes a critical section, similar to the one used in previous experiments, but with the exception of not using any locks, since it shall not be necessary any more. We use that micro benchmark as input to generate the DAE version of it. In a copy of that file, we are placing the critical section of the micro benchmark inside a transaction using RTM. We compile both files with clang and generate their respective DAE llvm files. We compared them, in order to study and understand how RTM is implemented in the Lllvm IR level [15].

Figure 3.4: Llvm IR

For our critical section in both files, the Access phase and the Execute phase are now taking place. The access phase is where the prefetching is being done and is followed by the computation, as described by the DAE model. The reason we performed the comparison described above, is to manage to wrap the Access phase inside a transaction by using RTM instructions. Apart from achieving that, we wished also to maintain the control flow graph of our program, as described in later sections.
Prior to proceeding and demonstrating the implementation, there are some interesting questions that are raising in order to study and evaluate the behaviour of our fusion between DAE model and RTM inside Access phase. In prefetching phase we wish to bring as much data as possible to the first level cache in order to exploit that in the execute phase. But the access phase is now going to be executed transactionally, so we need to take a look at how read and write sets will be maintained in this newly formed situation.

3.4 Memory behaviour of Hardware Transactional Memory

While executing inside critical sections using Hardware Transactional Memory on an Intel's processor and in this case RTM, we keep track of read and write sets and abort on read/write or write/write conflict. Intel's TSX detects conflicts on cache line granularity (64 bytes). In our implementation of RTM in the Access phase of DAE model, we will not have such conflicts, as we are going to execute sequentially.

3.4.1 Read and Write sets

We keep track of stores and loads in the write-set and read-set respectively. The addresses and the data of the stores for the write-set are kept in the store buffer [16]. L1 cache serves as our store buffer and its size is 32 KB. Likewise we keep the addresses of loads in the load buffer, but we also compare them with the addresses of the stores for aliasing. If there is a match, then the load must wait for the older store to complete in order to maintain dependency. The capacity of the read-set is much bigger and reaches the size of L3 cache, which is 8 MB.

The biggest challenge is dealing with stores. The write-set data must be buffered until the end of the transaction. In case of a successful commit, all stores in the write-set become globally visible in an atomic fashion, typically by writing the buffered data to the cache. Alternatively, if a transaction aborts, then the buffered stores must be discarded without changing the memory. Loads are simpler, in the sense that they do not change memory, only the architectural registers. On a successful commit, all loads in the read-set are written to the register file. Aborting a transaction means restoring the register file, which is easier than undoing changes to the memory system.

In more detail, when a store is issued to the out-of-order core for renaming and scheduling, an entry in the store buffer is allocated (in-order) for the address and the data. The store buffer will then hold the address and data until the instruction has retired and the data has been written to the L1D cache.

Likewise, when a load is issued, an entry in the load buffer is reserved for the address. As also mentioned above, loads must also compare their load address to the addresses of the store buffer to check for aliasing with older stores.

As dictated by x86 ordering model, the load buffer is snooped by coherency traffic. That means that a remote store must invalidate all other copies of a cache line. If a cache line is read by a load, and then invalidated by a remote store, the load must be cancelled, since it probably has read invalid data. However the x86 memory model does not require snooping the store buffer.

Hardware Transactional Memory on Intel's processors is most likely a deferred update system that use the caches of each core for transactional data and register checkpoints. In each cache line in the L1D and L2 there are bits that indicate whether the line belongs to the read-set or write-set for the two threads that can execute on each core. A store during a transaction will simply write to the cache line, and the shared L3 cache holds the original data (which may require an update to the L3 for the first write in a transaction to a Modified line). To avoid data corruption, any data generated inside a transaction must stay in the L1D or L2 and not be evicted to the L3 or memory. The architectural registers have saved their previous state and stored it on-chip, so that the transaction can read and write from registers freely.

3.4.2 Commit and Abort

To commit a transaction in the system, the following actions are applied [17]. The L1D and L2 cache controllers make sure that every cache line which is in the WS is also in the Modified state and the WS bits are going back to zero. In a similar way, any cache line that is in the RS (but not the WS) must be in the Shared state and the RS bits are going back to zero. Moreover, the old register file checkpoint is removed and the existing contents of the register file become the new architectural state.

The coherency-based conflict detection model can identify conflicts early, as soon as they occur (as opposed to at commit time). Once the conflict has been detected early, it is most likely that the transaction is aborted immediately. However, other problems might cause aborts in some implementations. Apart from capacity issues, there are aborts depending for example on the time we spend inside a transaction. The more time we spend, the more like is for a hardware interrupt to occur and abort our transaction. In our case we will be aborting explicitly, as we want to preserve the processor's architectural state while benefit from the prefetching that takes place.

Aborting transactions is simpler than committing. The cache controllers change all the WS lines to the Invalid state and the WS bits are returned to zero. The controllers zero out also the RS bits, but it is not necessary to invalidate the cache lines (although the processor might do this). The architectural register file is restored from the old checkpoint that was saved.

3.5 Implementation of RTM in DAE

After having identified the RTM instructions in the Llvm IR level, we can proceed to the implementation of RTM in the access phase of DAE model. We want to perform it in a way that would allow us to achieve what we have described previously and maintain the control flow of the application at the same time.

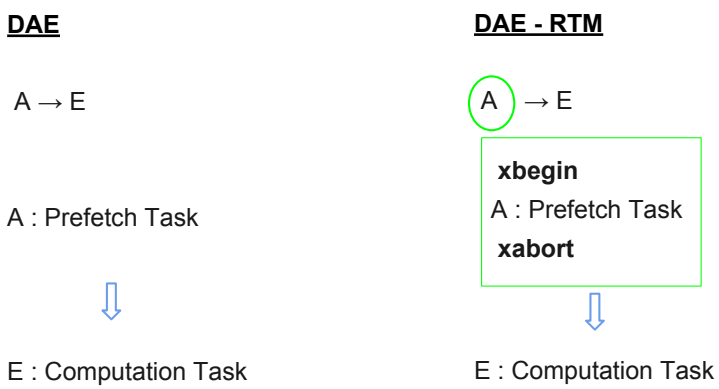


Figure 3.5: DAE vs RTM-DAE

In order to execute the Access phase transactionally we start a new transaction. We go ahead and perform the prefetching of the task, only this time we are inside a transaction. Right after the prefetching is over, we abort the transaction explicitly. As soon as we abort there will be no retries and we are going to continue execution with the computation task taking place (Execute phase). In case of any abort, other than our explicit abort, the computation task is also specified as the fall-back path. This way we guarantee the correct control flow of the application, as well as the forward progress of it.

Write set lines generated during the Access phase are getting invalidated right after prefetching is done and the Write set bits are zeroed out. On the other hand the Read set cache lines are not necessarily invalidated, but the Read set bits are zeroed out, the same as Write set bits. The register file is getting restored to the previous state as well. Since we **do not** want to alter the memory state during the prefetching, having the stores invalidated on abort can assure us that we will leave the memory state intact after the

Access phase is over, which is exactly the problem that wanted to tackle in the first place.

3.5.1 LLVM transformation pass

In order to place the Access phase in a transaction automatically, as it is described above, we need to create an Llvm transformation pass that receives the DAE version of an instance as input and transforms it into a DAE-RTM version that follows the scheme shown in Figure 5.

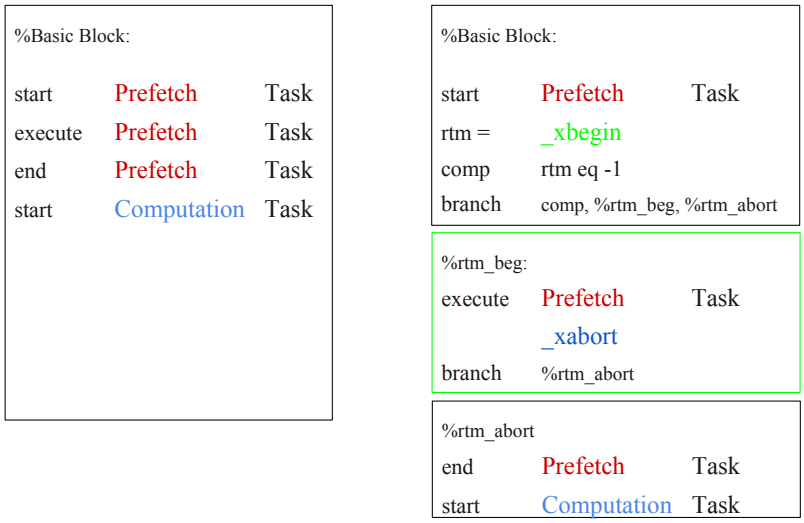


Figure 3.6: Transformation Pass: DAE vs RTM-DAE

The Llvm pass is going to iterate over each of the module's function and over every basic block of a function [18]. The goal is to detect the execution of the prefetching task, namely the Access phase in the targeted function, whose data are going to be prefetched. In the basic block of the module that prefetching gets detected, the **xbegin** call is inserted before the execution of the prefetching task, in order to start a transaction. Moreover the current basic block gets split into three parts, as shown in Figure 6. The first one includes the instructions of the original block, until the Access phase call. The second one is where the Access phase takes place and is followed by the explicit abort call, **xabort**. Finally the third part is where the computation task takes over, namely the Execute phase.

The Llvm pass that we constructed performs all that automatically. It basically detects whether there is a prefetching taking place in a function's basic block and inserts an xbegin call in assembly language. It splits the basic block twice in order to form three new blocks that will contain the appropriate instructions, as well as taking care of the control flow. The second block is executed inside a transaction and an xabort call is inserted, in assembly as well, right after the prefetching instruction call. In the third block the actual execution of the task is resumed. Right after the xbegin call, a compare instruction is inserted to make sure the transaction started correctly. If it is, then a branch instruction redirects the execution flow to the second block, otherwise the second blocked is skipped and the execution continues in the third block. In case of an unexpected abort or if the transaction did not start, then the execution is transferred from the first block to the third. We can see that the Execute phase is serving as the fall-back path, which ensures the forward progress of our application.

4 Evaluation

4.1 Environment and tools used

All our experiments were run on Intel's i7-4790K Haswell processor that supports Intel TSX instructions. It supports 4 Cores and 8 Threads. Its L1 cache size is 256 KB (4 x 32 KB 8-way set associative instruction caches and 4 x 32 KB 8-way set associative data caches). L2 cache is 256 KB (4 x 256), one for each core. L3 cache that is shared among all cores has capacity of 8MB. Its base clock speed is set to 4 GHz and its turbo frequency up to 4.4 GHz.

In order to achieve an accurate profiling to test our implementation, the Performance API (PAPI) [19] framework was used in our instances, so as to access the hardware performance counters of our machine. We used native events supported from our Haswell processor that helped us monitor abort rate, as well as identifying different abort types, while executing a transaction. Execution time and Instruction per Cycle (IPC) counters were also used, helping us calculate the overall performance and energy consumption.

As we stated earlier, L1 cache serves as the transactional buffer, so in this case the capacity of the buffer will be 32 KB. Furthermore, as it was deducted from related work results and our own experiments, the Write-set size that is withstood inside a transaction is equal to L1 cache and the Read-set size can be as big as L3 cache size (256 x Write set size).

In order to perform our experiments and compare the new RTM version of DAE model on our hardware, we had to choose a number of applications to evaluate. Benchmarks from the **SPEC CPU2006** suite have been selected for this cause [20]. The selection of benchmarks varies and can be divided roughly into two groups: Memory bound and Compute Bound applications. We expect that the memory bound applications should benefit more from our implementation. That is due to the fact that overhead caused by the extra execution time of the Access phase, can be compensated by the fewer L1 data cache misses in the Execute phase, which in turn should be able to execute faster. Moreover the overall energy consumption for these applications is expected to be less according to our study. On the other hand, as far as compute bound applications are concerned, it is hard for any improvement to get noticed.

The benchmarks tested are Mcf, Lbm, Soplex, Milc, Bzip2, Libquantum, Hmmer and

Astar. We tried to classify these benchmarks as memory or compute bound. This classification has been in respect to L1 Data Cache misses of these benchmarks as well as the number of Instructions Per Cycle (IPC)[21] [22]. The more cache misses and the lowest IPC a benchmark has, indicates a memory bound behaviour. On the contrary, compute bound benchmarks show a high IPC number and few cache misses.

Memory Bound	**Compute Bound**
Mcf	Bzip2
Lbm	Libquantum
Soplex	Hmmer
Milc	Astar

Table 1: Characterization of SpecCPU 2006 Benchmarks

In order to run and test the benchmarks, SPEC CPU2006 provides appropriate input for all of them. For our experiments, we chose the reference input for each benchmark. Soplex, Hmmer and Astar were also tested for two different reference version inputs each.

Prior to running the performance and energy consumption experiments, we monitored the abort rate in the access phase of these benchmarks to get a better view on how our newest version is going to behave in the prefetching phase. Heavy prefetching may benefit a memory bound application, but it adds overhead, as the time spent in access phase is more. We try to find the right balance in prefetching in order to get the most out of it, while paying the least cost for it.

In the following graphs we can get a rough idea on what kinds of aborts we are going to withstand in the prefetching phase of RTM-DAE. The red bar (Misc 3) indicates that we are resulting into explicit aborts which is what we wish, as it means that we executed the access phase without any unexpected aborts. Misc 1 aborts show the number of RTM aborts due to various memory events and Misc 5 due to other events, such as hardware interrupts [23]. Aborts due to capacity constraints are part of Misc 1 category and are shown in separate graphs in purple to get a better view on our benchmarks' behaviour.

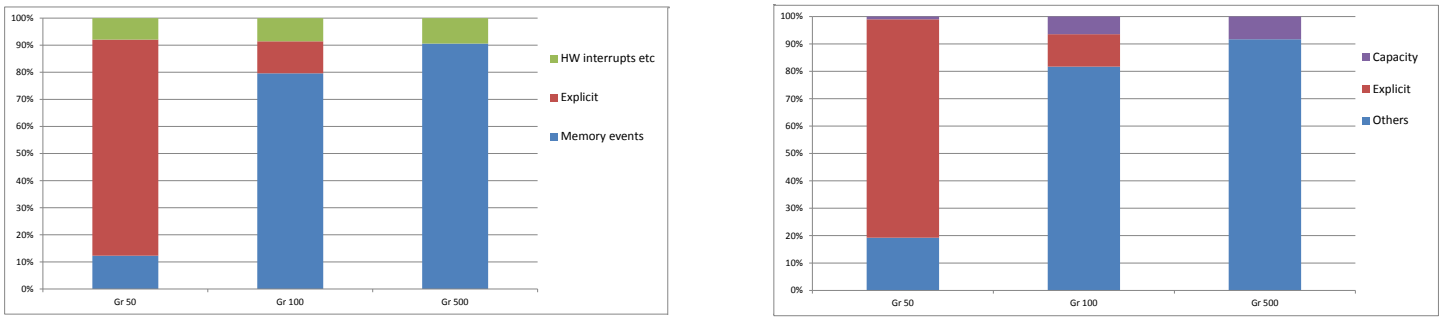


Figure 4.1: Mcf - RTM abort rate

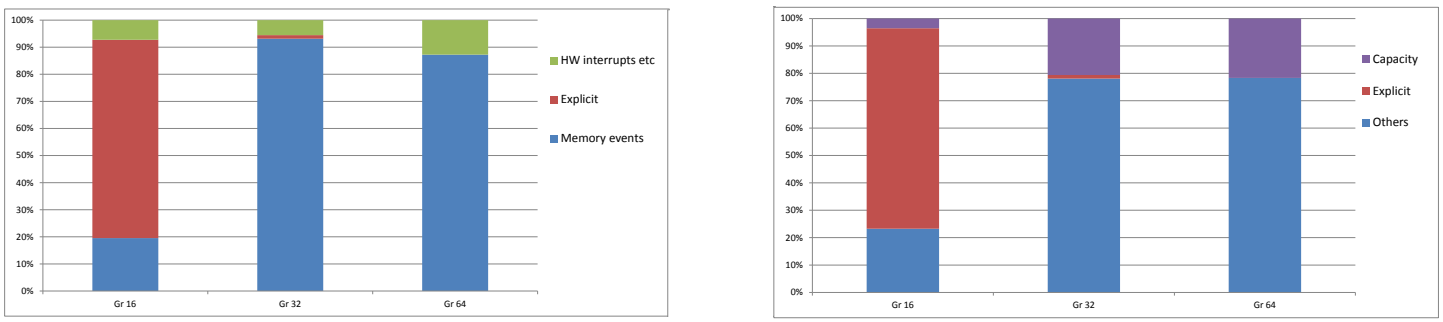


Figure 4.2: Lbm - RTM abort rate

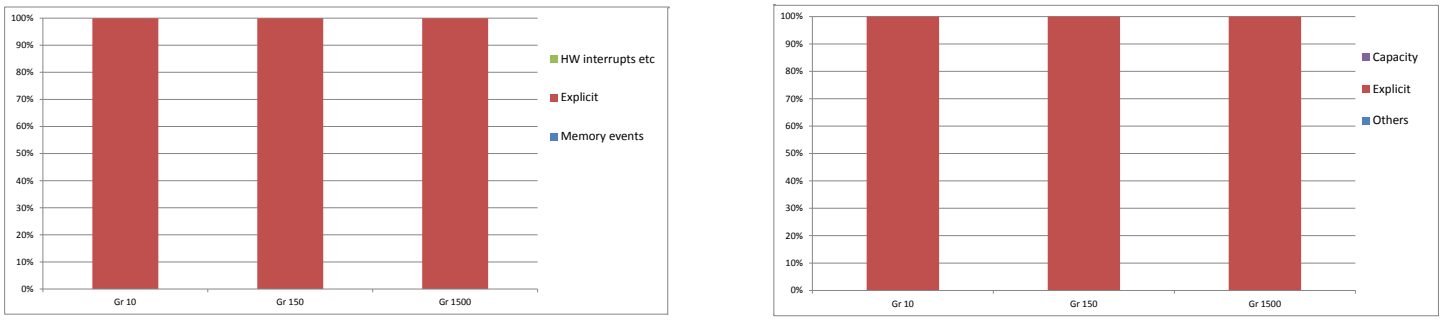


Figure 4.3: Milc - RTM abort rate

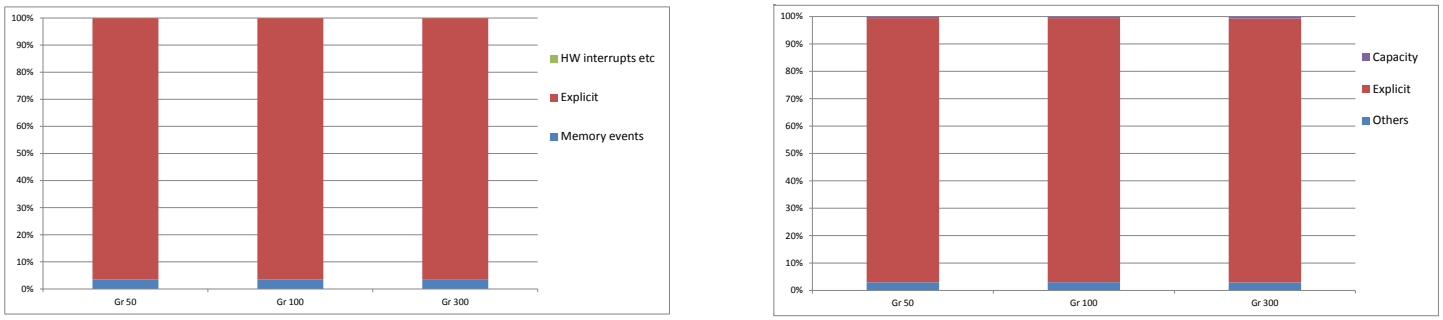


Figure 4.4: Soplex - RTM abort rate

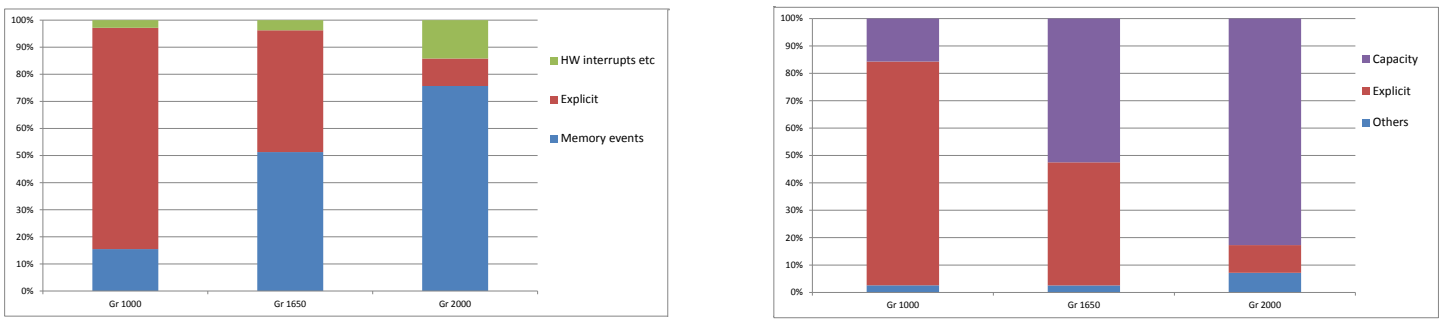


Figure 4.5: Libquantum - RTM abort rate

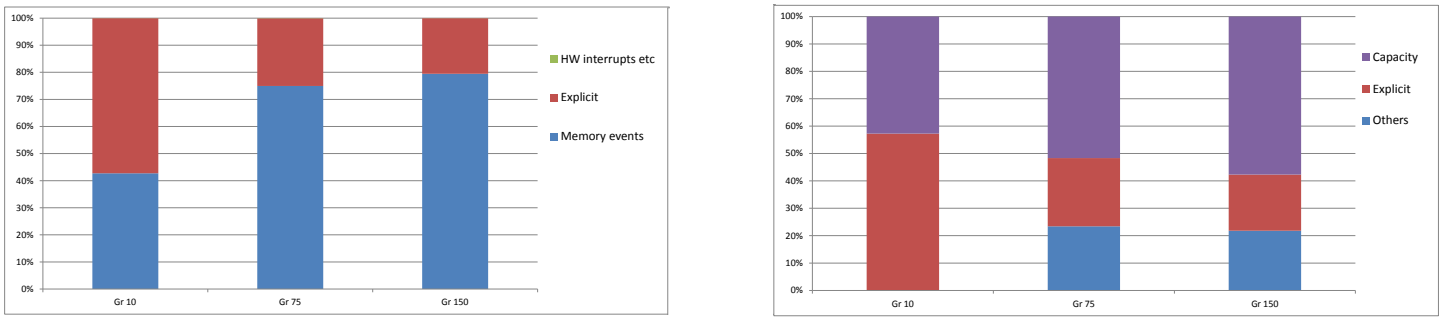


Figure 4.6: Hmmer - RTM abort rate

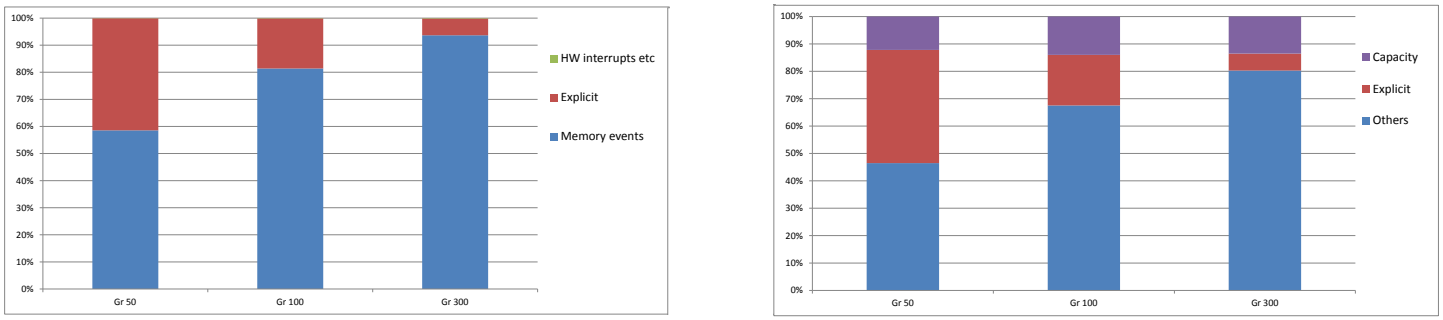


Figure 4.7: Astar - RTM abort rate

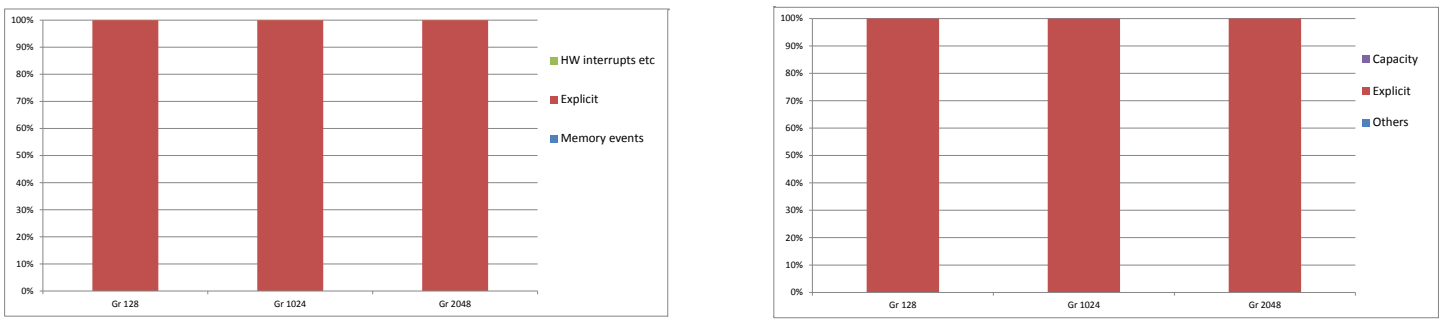


Figure 4.8: Bzip2 - RTM abort rate

4.2 Results

4.2.1 DAE and RTM-DAE - Initial pass

The way to find the sweet spot in prefetching for each benchmark is to experiment in terms of **granularity**, which is the size of the chunking of the outermost loop in a targeted function and the level of depth or in other words, the number of **indirections**, as described by Per Ekemark et al [13]. The indirections are a measure of how many intermediate loads are required to compute an address in the Access phase.
For the following experiments we chose a specific version and granularity for each benchmark for the DAE and DAE-RTM versions and compared them with the original Coupled Access-Execute version. We kept the granularity low in order to minimize the abort rate in the DAE-RTM version.

Benchmarks	Granularity	Indirections
Mcf	50	2
Lbm	16	2
Soplex-pds	300	11
Soplex-ref	100	11
Milc	1650	2
Libquantum	1000	3
Hmmer-nph3	10	5
Hmmer-retro	10	5
Astar-lakes	50	6
Astar-rivers	50	6

Table 2: Granularity and Indirections' level

An important point to keep in mind in DAE model, as also stated in the beginning, is to take care of the memory state. We wish to keep the memory state intact while prefetching, so we should discard any store instructions which we cannot prove statically that are **not side-effect free**.

Benchmarks	Unsafe Stores
Bzip2	247
Mcf	0
Lbm	0
Soplex	0
Libquantum	0
Milc	0
Hmmer	0
Astar	0

Table 3: Unsafe store instructions in initial DAE

As we can see, at this stage we cannot run a DAE version of Bzip2, as there are unsafe store instructions that, if allowed in access phase, cannot ensure correct program behaviour. We ran experiments on these benchmarks in order to compare performance (Execution Time) and the amount of energy consumed, according to the following formulas:

$$P_{dynamic} = CV^2 f$$

$$Energy = Power \times Time$$

The frequency set for the Execute phase, while running our experiments, was 4 GHz and for the Access phase was set to 1.7 GHz.

The group of the **memory-bound** applications, namely mcf, lbm, milc and soplex presented us with the most positive results regarding energy efficiency, as was expected prior to running the experiments. They exhibit an **improvement in energy consumption** in the DAE version, compared to the respective original version, but on the other hand the overall execution time is worse, as more time is needed to execute both phases. The original version spends more time running in high frequency, waiting for data to become available from memory. On the other hand, the DAE version runs the access phase first, in order to bring the data needed in L1 cache, while running at the lowest frequency. Following that, the execute phase takes place running in highest frequency and faster time-wise, benefiting from the previous prefetching.

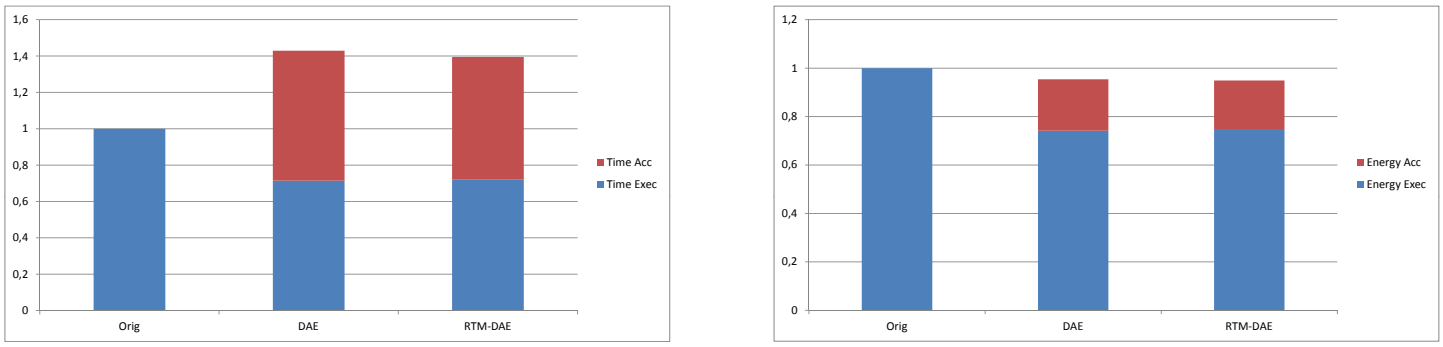


Figure 4.9: 429.mcf

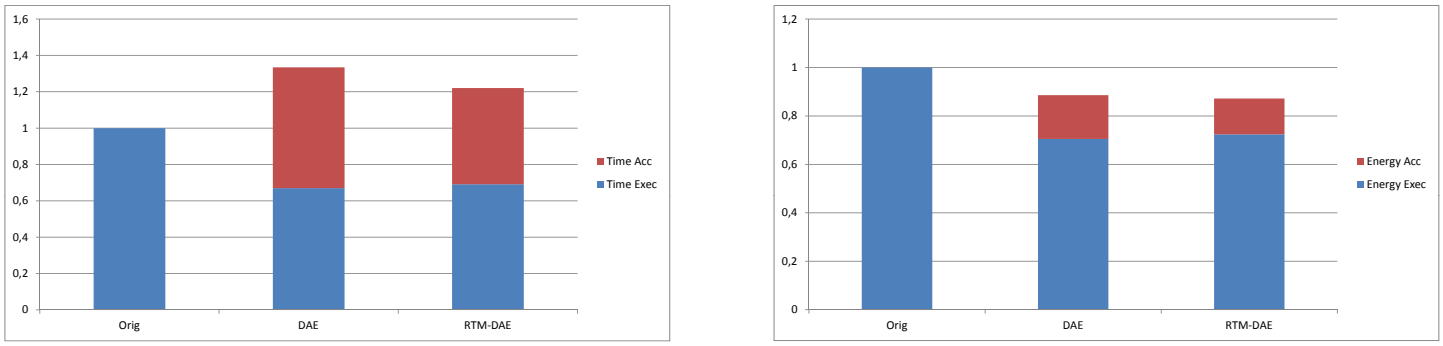


Figure 4.10: 470.lbm

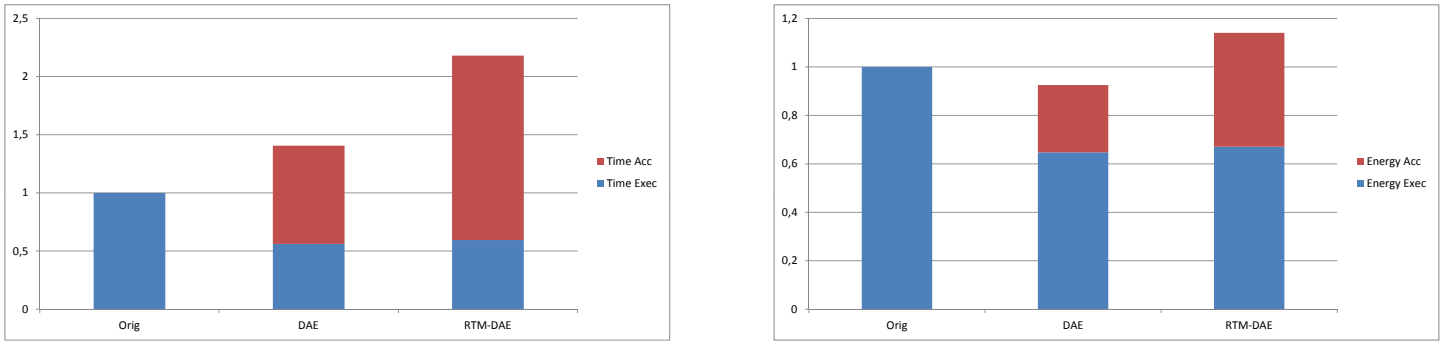


Figure 4.11: 433.milc

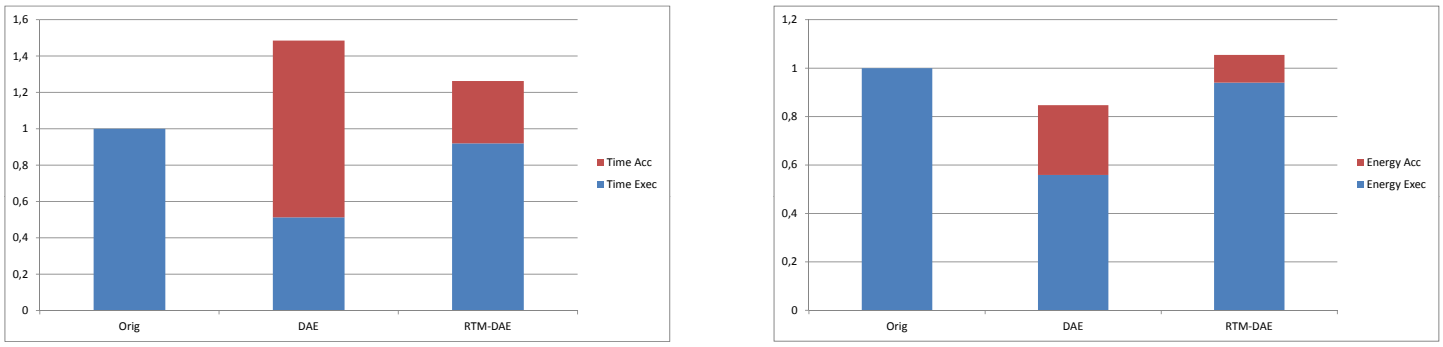


Figure 4.12: 450.soplex - pds

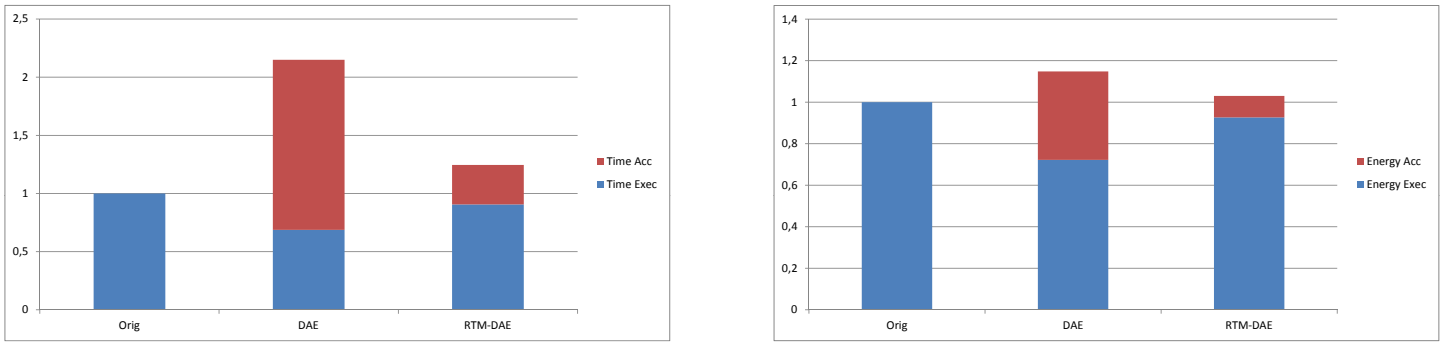


Figure 4.13: 450.soplex - ref

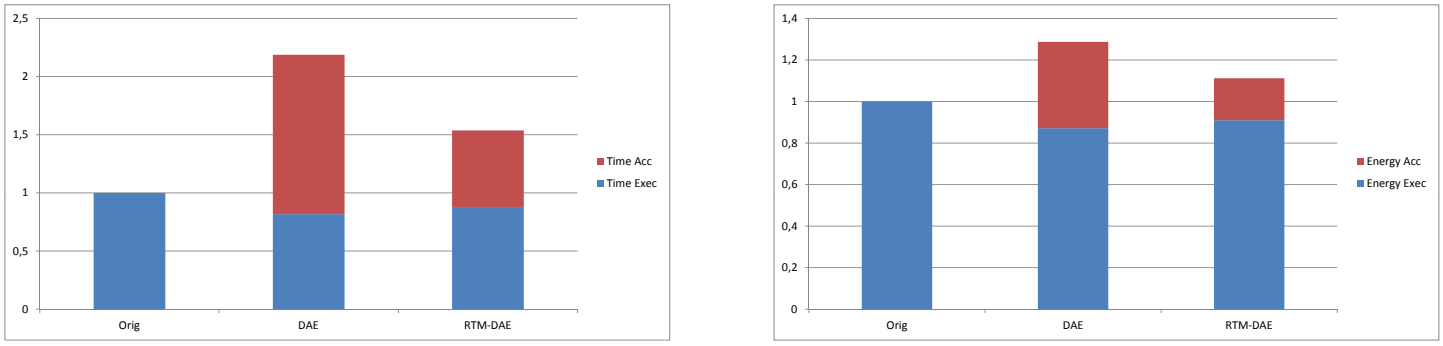
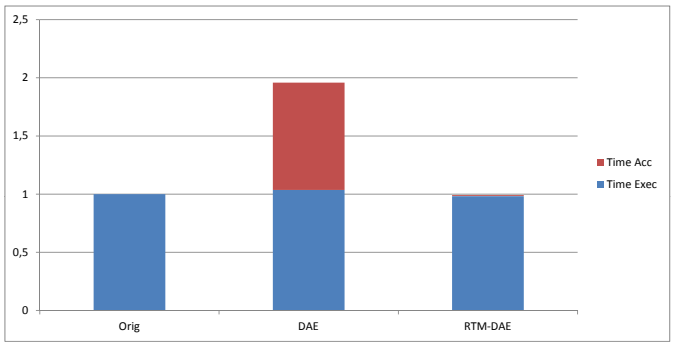


Figure 4.14: 462.libquantum



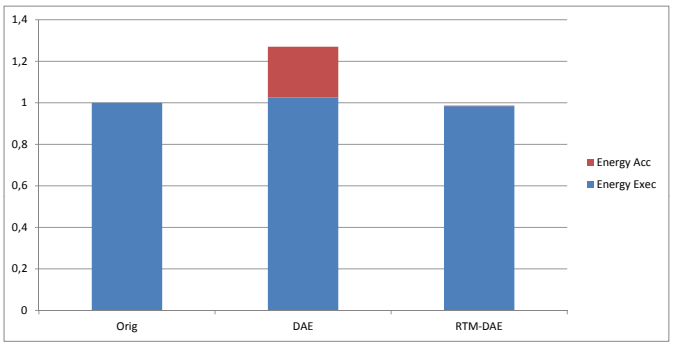
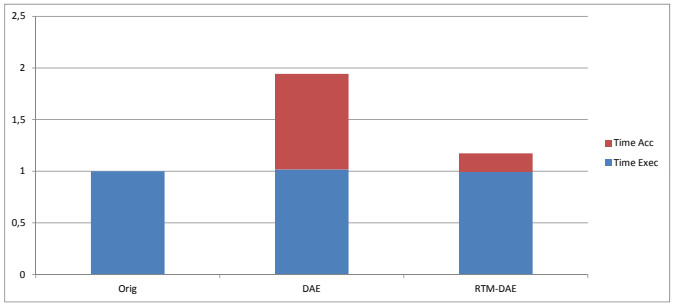


Figure 4.15: 473.hmmer - nph3



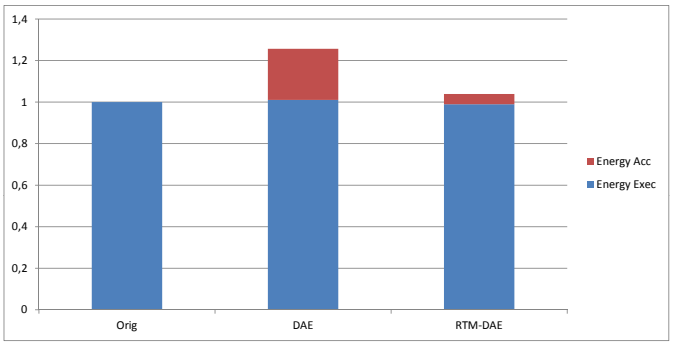
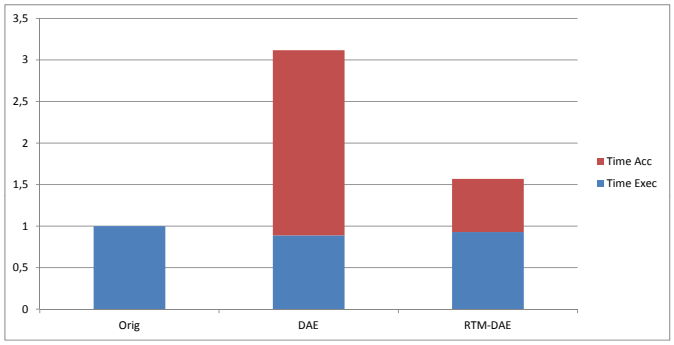


Figure 4.16: 456.hmmer - retro



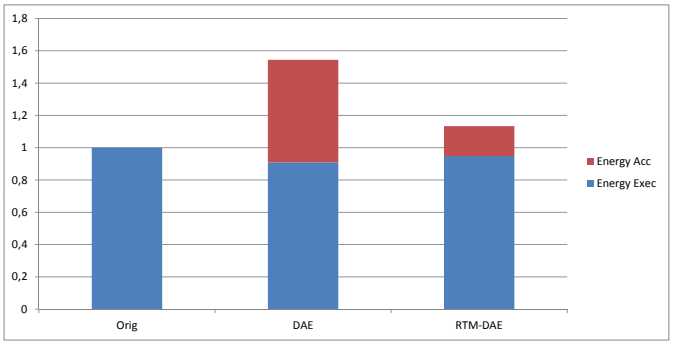


Figure 4.17: 456.astar - lakes

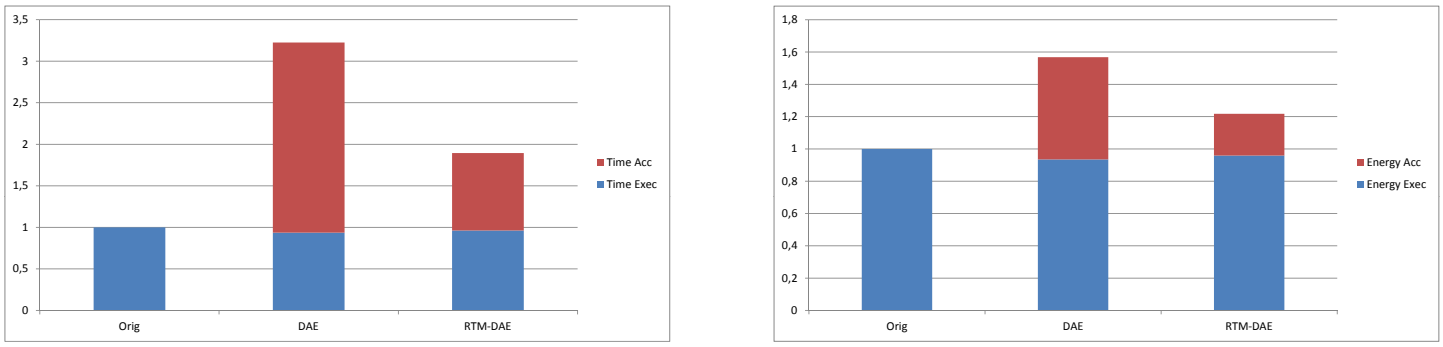


Figure 4.18: 473.astar - rivers

RTM-DAE version, in respect to the abort rate in the access phase, exhibits a similar improvement in energy consumption in Lbm and Mcf benchmarks, while spending slightly less time in the prefetching phase due to a certain abort rate, according to each benchmark. Milc benchmark shows that for an RTM-DAE version of a full access phase with almost no aborts, too much overhead is being introduced in the access phase, as the time spent in prefetching is more comparing to DAE version and the energy consumption as well. In Soplex benchmark, having as input the pds reference version, the DAE version exhibits a similar improvement to the previous benchmarks, but the RTM-DAE version, due to increased aborts, fails to save energy as the prefetching phase is a lot smaller and results into no benefit. This behaviour of transactional memory version can also be observed in the other reference input of Soplex.

The benchmarks that we classified as being more **compute bound**, Libquantum, Hmmer and Astar showed no improvement in performance or energy. The execute phase of DAE model shows very little or no improvement in these benchmarks compared to the original both in terms of performance and energy use. This behaviour is expected, as the data in these applications fit in the L1 cache to begin with, which eliminates any prefetching benefit. Simply put the execute phase remains the same while the prefetching adds overhead in terms of both time execution and energy consumption. The DAE-RTM implementation confirms that, even though we can see a smaller prefetching phase, resulting from aborts, this leaves the execute phase the same, both in terms of time execution and energy consumption.

4.2.2 RTM-DAE - Stores enabled pass

For the second round of our experiments we modified the llvm pass that would generate the llvm files for each benchmark which follow the DAE model. One of the issues of DAE model is that, in order to ensure correct program behaviour, we had to exclude store instructions in the access phase of the model that we are not able to prove statically that they are side-effect free. Since we are able now to use transactional memory in the access phase and abort it as soon as it finishes, we can overcome this problem and allow these previously *unsafe* store instructions.

Benchmarks	Unsafe Stores
Bzip2	495
Mcf	30
Lbm	3
Soplex	37
Libquantum	6
Milc	3
Hmmer	11
Astar	14

Table 4: Unsafe store instructions in DAE after enabling stores

After enabling store instructions in the prefetching phase, we monitored the number of those instructions that alias with any global instruction in the application. In Table 4 we can see that number for each of the selected benchmarks. Bzip2 benchmark can this time also be included in our experiments, as correct program behaviour can this time be achieved in the new version of DAE model in which we are using RTM.

One quick observation of the following graphs reveals that in this stores-enabled version of DAE-RTM an extra overhead regarding time execution is added in all memory-bound benchmarks. This is of-course expected, as stores are now also allowed in access phase. In terms of energy consumption, latest version of DAE-RTM shows a slight improvement in Mcf benchmark, which in comparison to original version reaches a total 5-6 % benefit. For Lbm benchmark the results exhibit a worse outcome both in energy and in time execution, which unfortunately indicates that allowing stores in the access phase do not benefit the execute phase.

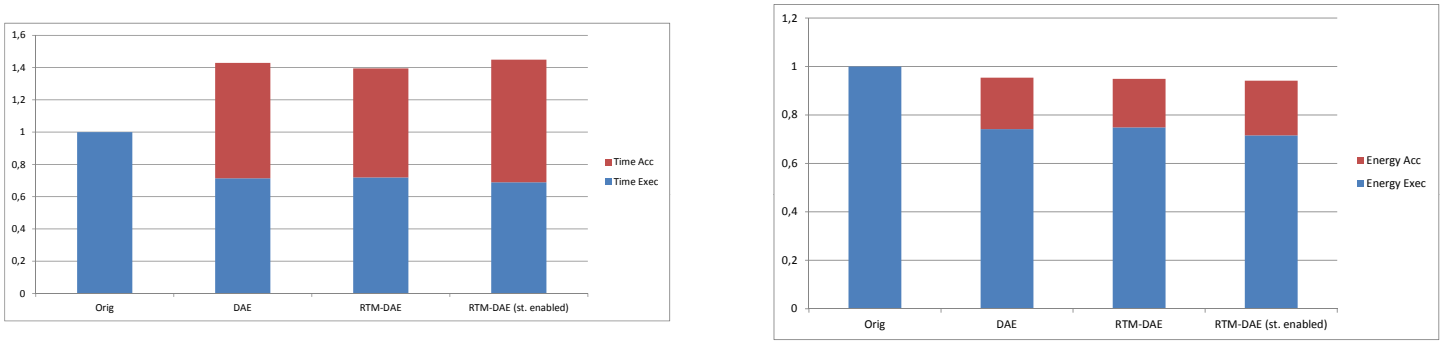


Figure 4.19: 429.mcf

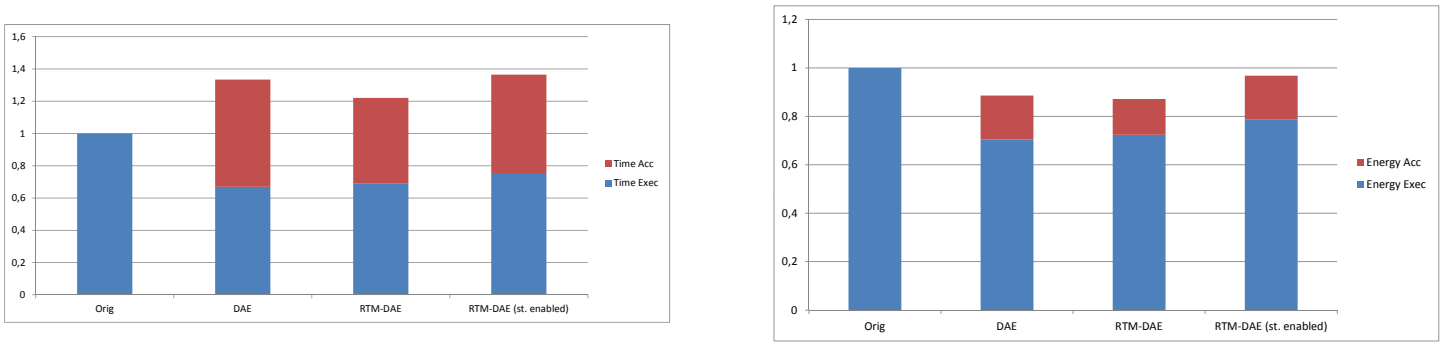


Figure 4.20: 470.lbm

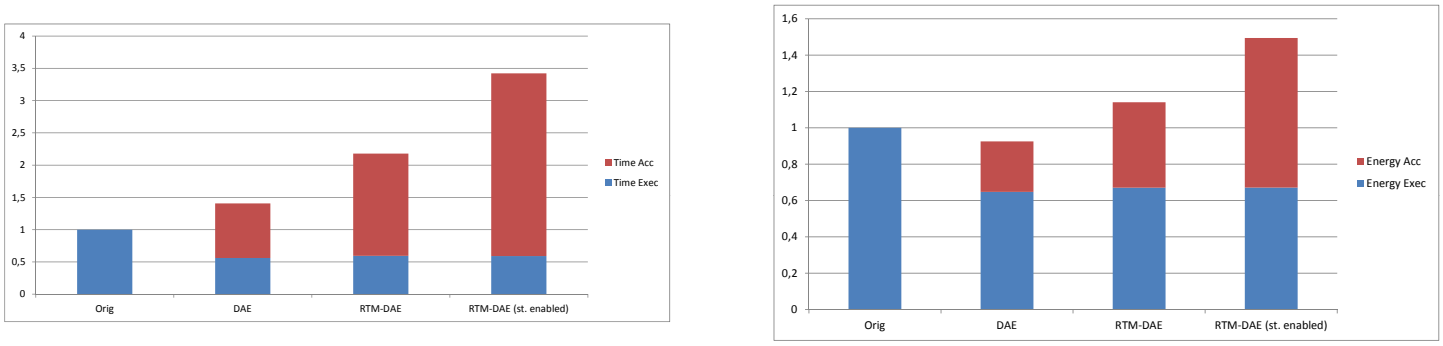


Figure 4.21: 433.milc

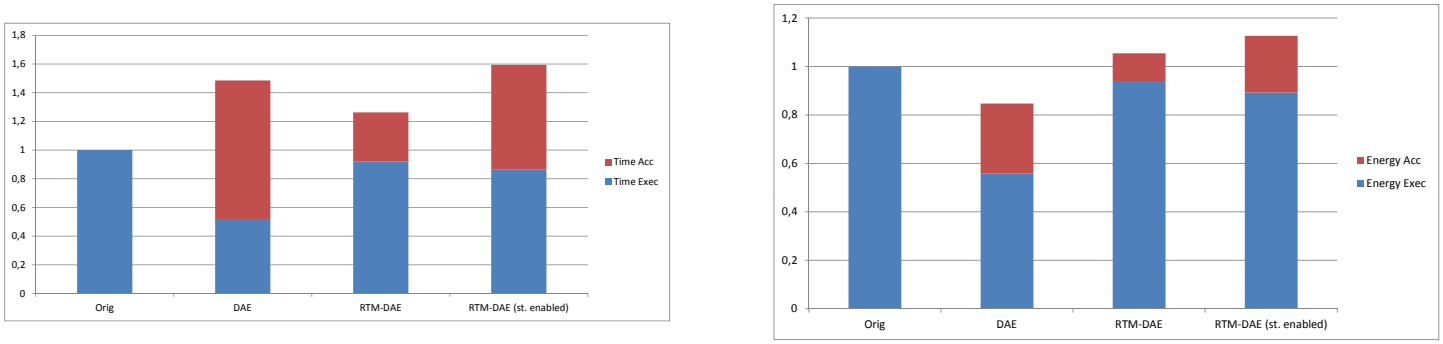


Figure 4.22: 450.soplex - pds

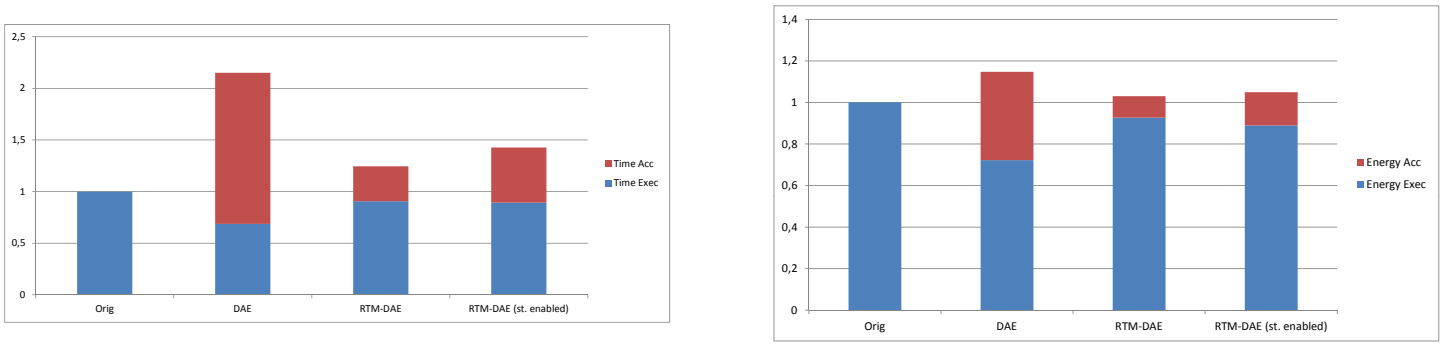


Figure 4.23: 450.soplex - ref

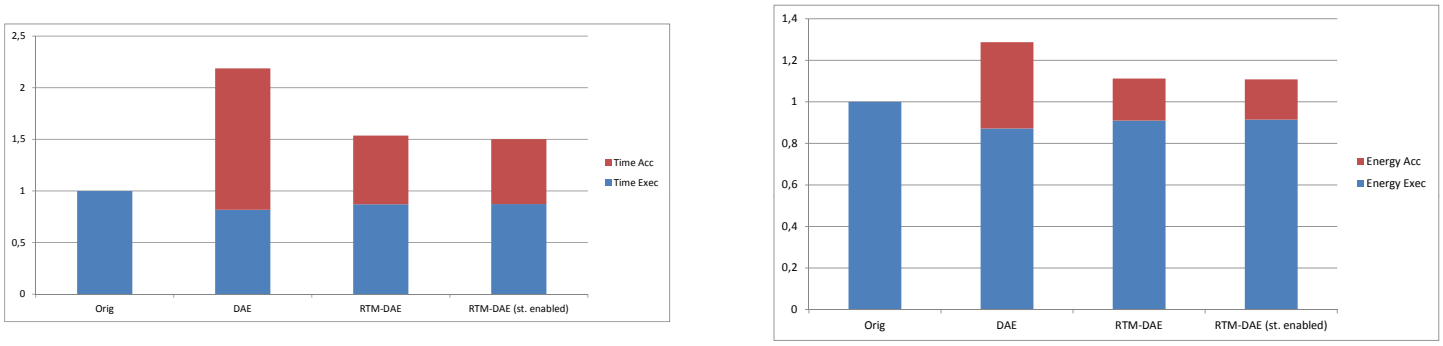
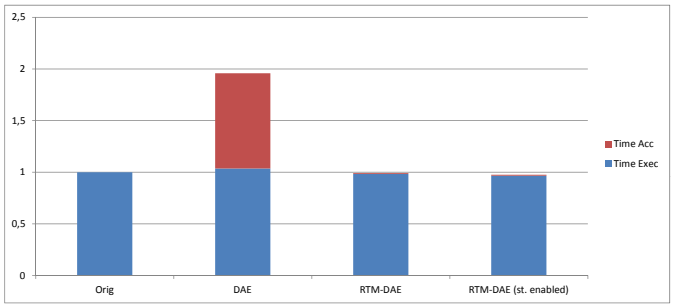


Figure 4.24: 462.libquantum



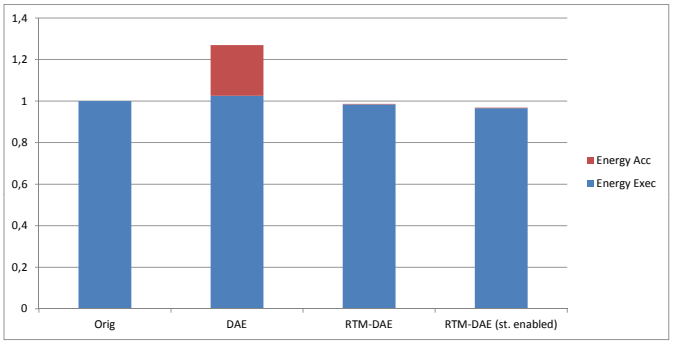
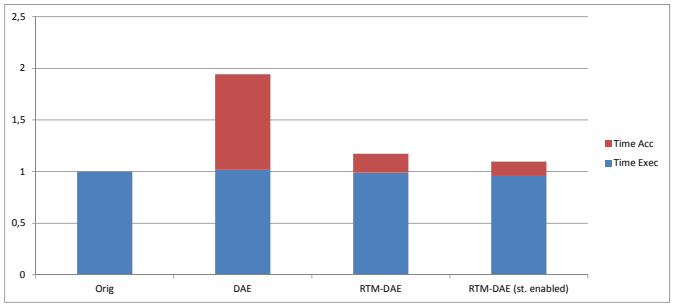


Figure 4.25: 473.hmmer - nph3



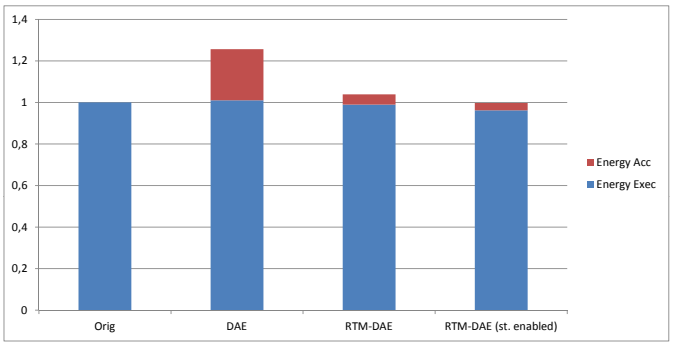
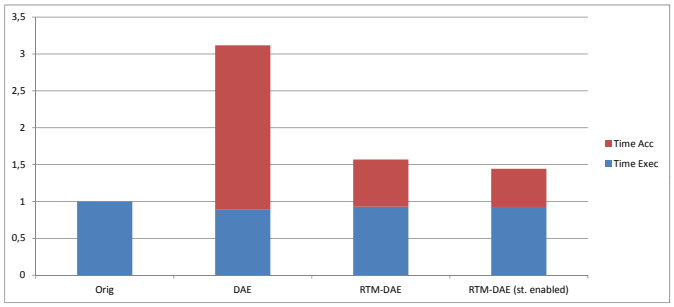


Figure 4.26: 456.hmmer - retro



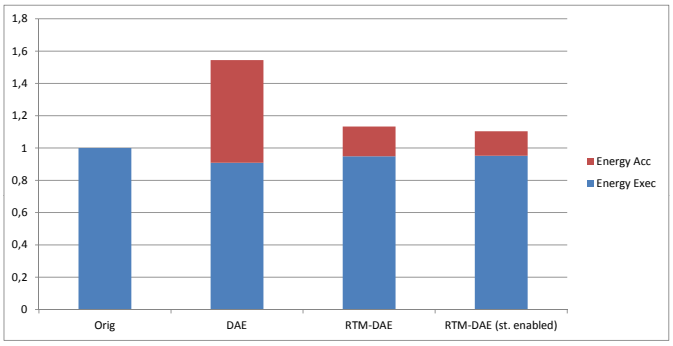


Figure 4.27: 456.astar - lakes

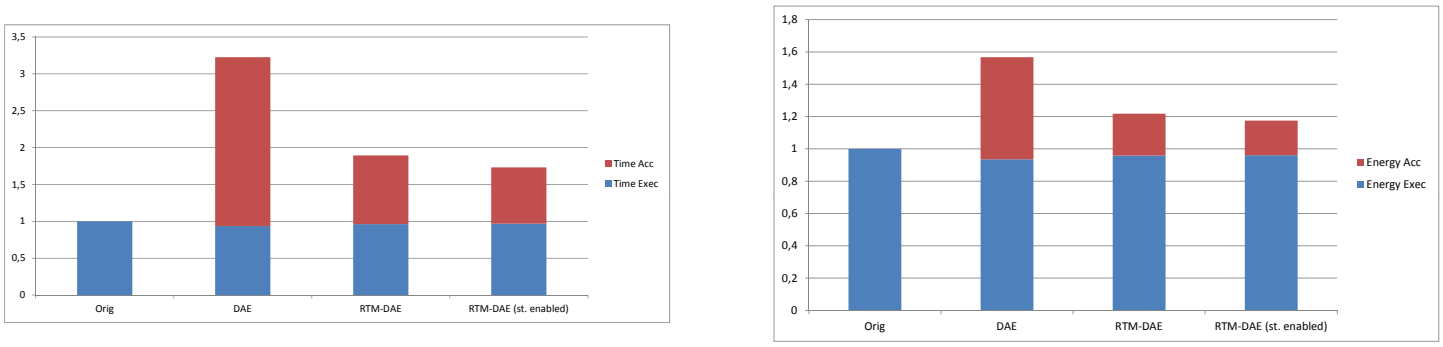


Figure 4.28: 473.astar - rivers

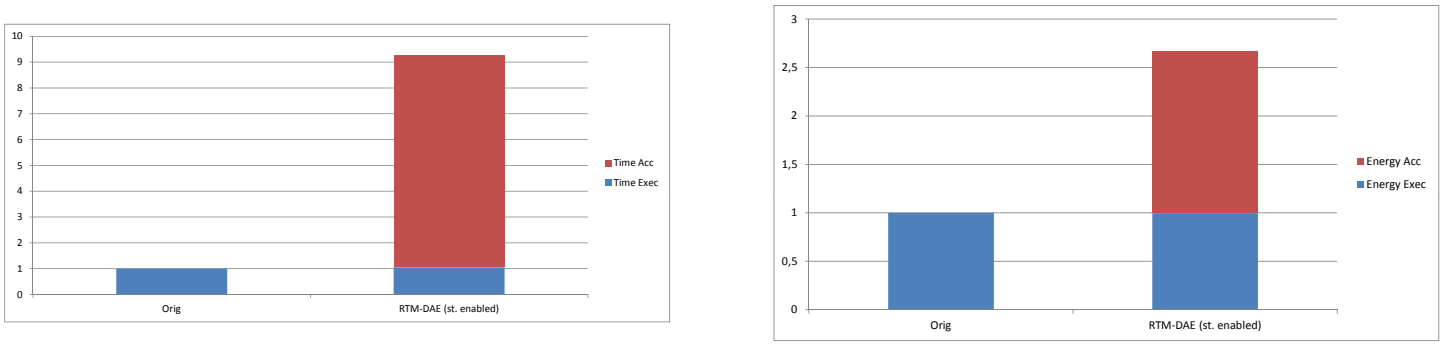


Figure 4.29: 401.bzip2

This can also be seen in Milc application, where more overhead is added, without gaining any profit in execute phase. On the other hand, for both reference input versions of Soplex we can get a small benefit in execute phase, even though the larger cost of the access phase makes it unable to compensate for it and worse overall.

Regarding the group of compute-bound applications and starting with libquantum, we can see that there was no effect in energy use or performance in the latest DAE-RTM version. Interestingly enough, for Hmmer benchmark, while having retro reference version as input, there is a slight improvement time-wise and energy-wise in the execute phase, which equalised the overall energy consumption of original version. In Astar, the prefetching phase is a bit smaller smaller this time, but no actual benefit is resulted into

the respective execute phase. Finally the Bzip2 shows a large overhead in our DAE-RTM implementation and seems that we are not able to get benefited by employing a prefetching phase for this benchmark.

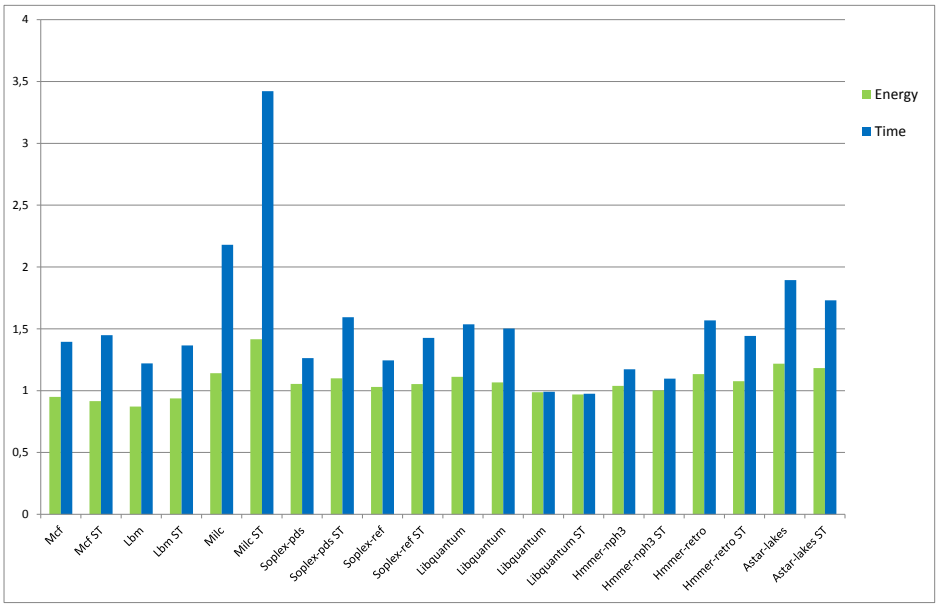


Figure 4.30: RTM -DAE Benchmarks

5 Conclusions and Future work

The main challenge that we were called to overcome regarding this thesis, is whether we could allow store instructions taking place in the access phase of the DAE model while keeping the memory state intact at the same time. By the use of Hardware Transactional Memory and namely RTM in the prefetching phase, we managed to achieve that goal.

We ran our experiments on real hardware, using applications with different characteristics. Results showed that by placing the access phase inside a transaction, we can still get the benefits of the DAE model in memory-bound benchmarks. An overhead is added due to RTM, but careful adjustment of granularity and indirection level can result into decreased energy consumption of 6-15 % in certain memory-bound applications, with the cost of performing a bit worse.

Enabling store instructions that previously would have side-effects in the access phase of the initial DAE implementation, unfortunately did now show encouraging results for the applications that we tested. On the bright side we were able to see the way bzip2 benchmark behaves on the presence of the DAE model. This was previously not possible, as correct program behaviour could not be ensured.

Hardware Transactional Memory did not generate great results in terms of energy efficiency, but nevertheless gives us the ability to employ the DAE model in applications that previously could not have been possible. The current results do not exhibit an improvement, but in future implementations on other applications, given this new possibility which extends the DAE model use, might present us with more energy efficient ways to run modern software.

References

[1] Konstantinos Koukos, David Black-Schaffer, Vasileios Spiliopoulos, Stefanos Kaxiras. Towards More Efficient Execution: A Decoupled Access-Execute Approach, Uppsala University, 2013.

[2] Alexandra Jimborean, Konstantinos Koukos, Vasileios Spiliopoulos, David Black-Schaffer, Stefanos Kaxiras. Fix the code. Don't tweak the hardware: A new compiler approach to Voltage-Frequency scaling, Uppsala University, 2014.

[3] Understanding Hardware Transactional Memory, *http* : *//www.quepublishing.com/articles/article.aspx?p* = 2142912, Last accessed: June 6, 2015.

[4] IBMs new transactional memory *http://arstechnica.com/gadgets/2011/08/ibms-new-transactional-memory-make-or-break-time-for-multithreaded-revolution/* Last accessed: June 6, 2015.

[5] Intrinsics for Intel Transactional Synchronization Extensions (Intel TSX), *https* : *//software.intel.com/sites/products/documentation/doclib/iss/*2013*/compiler/cpp−lin/GUID* − *BAFE*2*B*5*E* − 6*AA*3 − 4283 − *BAB*2 − *F*36*B*00725*EB*3.*htm*, Last accessed: June 6, 2015.

[6] Intel Architecture Instruction Set Extensions Programming Reference, 319433-012A, FEBRUARY 2012.

[7] Intel Xeon Processor E3-1200 v3 Product Family Specification Update, February 2015, Revision 010.

[8] Chitters, Vamsi, Adam Midvidy, and Jeff Tsui. "Reducing Synchronization Overhead Using Hardware Transactional Memory."

[9] Leis, Viktor, Alfons Kemper, and Thomas Neumann. "Exploiting hardware transactional memory in main-memory databases." Data Engineering (ICDE), 2014 IEEE 30th International Conference on. IEEE, 2014.

[10] Dai Wang, Mike, et al. "Exploring the performance and programmability design space of hardware transactional memory." ACM SIGPLAN Workshop on Transactional Computing (TRANSACT). 2014.

[11] Goel, Bhavishya, et al. "Performance and energy analysis of the restricted transactional memory implementation on haswell." Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. IEEE, 2014.

[12] List of Intel CPU 4th generation (Haswell) with TSX (RTM/HLE) support $http://www.trans-mem.com/haswell-tsx$, Last accessed: June 6, 2015.

[13] Per Ekemark, Alexandra Jimborean, David Black-Schaffer
Static Multi-Versioning for Efficient Prefetching, Uppsala University, 2015.

[14] Writing an LLVM pass, $http://llvm.org/docs/WritingAnLLVMPass.html$, Last accessed: June 6, 2015.

[15] Llvm IR, $https://idea.popcount.org/2013-07-24-ir-is-better-than-assembly/$, Last accessed: June 6, 2015.

[16] Haswell Transactional Memory Alternatives, $http://www.realworldtech.com/haswell-tm-alt/2/$, Last accessed: June 6, 2015.

[17] Analysis of Haswells Transactional Memory, $http://www.realworldtech.com/haswell-tm/$, Last accessed: June 6, 2015.

[18] LLVMs Analysis and Transform Passes, $http://llvm.org/docs/Passes.html$, Last accessed: June 6, 2015.

[19] PAPI, $http://icl.cs.utk.edu/papi/overview/index.html$, Last accessed: June 6, 2015.

[20] SPEC CPU 2006, $https://www.spec.org/cpu2006/$, Last accessed: June 6, 2015.

[21] Prakash Tribuvan Kumar, Peng Lu. Performance characterization of spec cpu2006 benchmarks on intel core 2 duo processor. ISAST Trans. Comput. Softw. Eng, 2008, 2.1: 36-41

[22] Bird, Sarah, et al. "Performance characterization of SPEC CPU benchmarks on intels core microarchitecture based processor." SPEC Benchmark Workshop. 2007.

[23] Intel 64 and IA-32 Architectures, Software Developers Manual, Volume 3B:System Programming Guide, Part 2. March 2013.