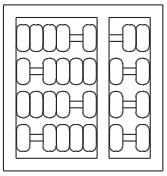

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

A First Study on Characterizing the Energy Consumption of Software Transactional Memory

Alexandro Baldassin *Felipe Klein*
Paulo Centoducatte *Guido Araujo*
Rodolfo Azevedo

Technical Report - IC-09-13 - Relatório Técnico

April - 2009 - Abril

A First Study on Characterizing the Energy Consumption of Software Transactional Memory

Alexandro Baldassin Felipe Klein Paulo Centoducatte Guido Araujo
Rodolfo Azevedo

Abstract

The well-known drawbacks imposed by lock-based synchronization have forced researchers to devise new alternatives for concurrent execution, of which transactional memory is a promising one. Extensive research has been carried out on Software Transaction Memory (STM), most of all concentrated on program performance, leaving unattended other metrics of great importance like energy consumption. This paper presents a systematic methodology to characterize a class of STMs that implement a common set of API calls. We thoroughly evaluate the impact on energy consumption due to STM by quantifying the energy costs of the primitives involved in an optimistic time-based STM implementation. This work is a first study towards a better understanding of the energy consumption behavior of STM systems, and could prompt STM designers to research new optimizations in this area, paving the way for an energy-aware transactional memory.

1 Introduction

The shift towards multicore processors and their subsequent mainstream adoption have drastically increased the need for better and effortless methods for building concurrent software. In the prevalent parallel programming model, multiple threads are executed concurrently and communicate by means of a global shared memory. Since different threads may access the same region of memory at the same time, a synchronization mechanism is necessary in order to enforce consistency. The standard synchronization primitives consist of locks and condition variables (e.g., mutexes, semaphores, and monitors).

The well-known drawbacks imposed by lock-based synchronization [1] have forced researchers to devise new alternatives of which transactional memory is a promising one [2]. From a programmer's point of view, a transaction can be seen as a group of operations that are executed atomically and in isolation from the rest of the system. In case conflicts among transactions arise, the transactional system automatically detects and resolves them (usually by aborting and rolling back one of the transactions). The implementation substrate for transactional memory can be realized entirely via software (STM), through hardware components (HTM) or as a combination of both hardware and software (hybrid approach). In this letter, we shall focus our attention specifically on STM.

Extensive research has been carried out on the design space of STM. At the algorithmic level, an STM can be regarded as non-blocking or blocking. Other key distinctions include:

nesting (flat, open, close), when and how to resolve conflicts, versioning scheme, and isolation guarantee (weak, strong). To evaluate a proposed implementation, researchers have invariably concentrated on performance, usually by measuring the system throughput in the form of transactions per second. We argue that, if STM is to become mainstream, other factors should be equally taken into account when devising a new design. In particular, energy-efficiency is of great importance and must be traded off with performance. This is specially true for embedded systems, where the energy consumption is closely related to battery lifetime. It is also of increasing interest in the non-mobile arena [3], such as data centers, where power contributes significantly to the operating costs, and even in desktop environments [4]. Hence, reducing the energy consumption is of utmost importance in most computing environments.

In this letter we introduce, for the first time, a systematic approach to characterize the energy consumption of an STM system. Our methodology allows the characterization of STMs which implement a set of common API calls. By using it, we perform a thorough evaluation of the resulting impact on energy consumption due to the adoption of STM. This initial study employs the TL2 algorithm [5] (optimistic time-based) as the STM substrate. More precisely, our contributions are: (i) the characterization methodology, which is API-oriented and, thus, applicable to several STM implementations; (ii) the fine-grained assessment of the energy costs of the STM primitives; and (iii) the quantification of the impact on the overall energy consumption due to the STM approach.

2 Methodology

The energy characterization methodology herein described attempts to profile the energy consumption of an STM system in terms of its basic components. A number of transactional memory approaches have been proposed in the literature [2] and, though distinct, all of them are built upon the same primitives, namely, *TxStart*, *TxCommit*, *TxLoad* and *TxStore*. These primitives are briefly described below:

- *TxStart* : creates a checkpoint and starts a transaction;
- *TxCommit* : ends the transaction and attempts to make the changes permanent;
- *TxLoad* : read barrier; and
- *TxStore* : write barrier.

We chose TL2 (*lazy* versioning) as the base for the experimental analysis presented in Section 3. Nevertheless, our methodology is general enough and could be applied to other STM implementations without loss of generality.

This work is a first step towards a better understanding of the energy behavior of the STM substrate, during the execution of concurrent applications. It also tries to quantify the overhead imposed, on the overall energy consumption, due to adopting the STM approach.

2.1 Simulation Platform

We adopted a cycle-accurate MPSoC simulation platform [6] which provides accurate energy and performance results. The processing units are ARMv7 processors with an 8K, direct-mapped instruction cache and a 4K, 4-way set-associative data cache. In addition to caches, the platform has also as many private memories as the number of processing units, each one with 12MB, and a single shared memory with 4MB. An important remark on this architecture's memories is that they are all SRAM-based, which are much more energy-efficient as compared to DRAM-based memories [7]. The aforementioned components are interconnected through an AMBA AHB bus.

2.2 Characterization Procedure

In order to proceed with the characterization process, we need to distinguish the energy consumed by the STM infrastructure from the energy consumed by the application itself.

This distinction is done in the profiling phase, schematically shown in Figure 1. At the top lies the application code which does not depend on the chosen programming model; the STM code, represented by its primitives, is shown circumscribed by the rectangle at the bottom. The *continuous* edges denote calls to the STM API, from within the application, while the *dashed* edges denote the return from the STM code to the application.

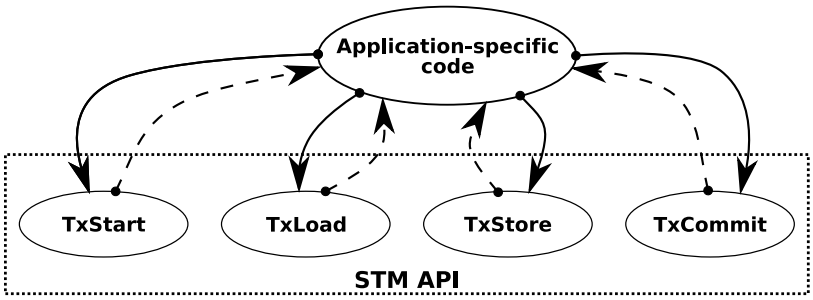

Figure 1: Energy profiling flow: continuous edges represent calls to the STM API while dashed edges represent the return to the application code

The energy profiling is performed in two steps:

1. **API profiling**: the application starts and no measurement is performed within the application code. Only after entering the STM code, through a call to its API (continuous edges), the energy measurement is activated. During this period, the energy consumed by all components of the platform (processor, caches, main memory and bus) are monitored and recorded for each primitive (*TxStart*, *TxCommit*, *TxLoad* and *TxStore*). Prior to returning to the application code (dashed edges), the measurement is deactivated. This process is repeated throughout the simulation and the resulting energy consumption is combined with the standard STM statistics (number of transactions started/committed and loads/stores performed) gathered to compute the energy consumption per primitive (i.e., their costs).

2. **Application profiling**: the application starts and measurement is initially activated. Prior to any call to the STM API (continuous edges) the measurement is deactivated and then reactivated when returning to the application code (dashed edges). Similar as above, all platform's components are taken into account and the resulting aggregate energy is recorded.

The procedure above allow us to assess individually the inherent energy costs of the application and also the overhead for each of the STM API's primitives. This facilitates the identification of possible bottlenecks of STM systems and could prompt STM designers to devise optimizations and improve their implementations, so as to reduce the energy footprint imposed by the STM approach.

There is a plethora of different applications that could benefit from the transactional memory programming model and, thus, we need to be careful when conducting the characterization process in order to cover such distinct applications and to avoid biasing the results.

Among the contrasting features that must be taken into account are: different read/write set sizes (as well as their ratio), abort rate, transaction length and others. Due to the large number of possibilities, the characterization procedure was designed to be done automatically, and to cover a large range of possible application types, within a feasible time-frame.

For that purpose, a so-called *parameterizable characterization application* has been designed, whose pseudo-code is sketched in Figure 2. This application is suitable for any

```
1: for iter ← 1 to IPC do
2:    TxStart() {the transaction starts at this point}
3:    wdw ← GetWindowRange(iter)
4:    for i ← 1 to NRD do
5:       elem ← wdw[i]
6:       x ← Consume(TxLoad(shrvar[elem]), x)
7:    end for
8:    for j ← 1 to NWR do
9:       offset ← Random(wdw.start, wdw.end)
10:      elem ← wdw[offset]
11:      y ← Produce(x + offset)
12:      TxStore(shrvar[elem], y)
13:   end for
14:   TxCommit() {the transaction ends at this point}
15:   InsertVariableDelay(ITD)
16:   x ← 0 {clear for next transaction}
17: end for
```

Figure 2: Pseudo-code for the parameterizable characterization application's core

number of cores, which perform a series of transactional reads and writes on a shared array (`shrvar`). The main parameters considered and the assumed configurations (between brackets) are summarized below:

- `VS`: *vector size*, defines the number of scalar elements of the shared array (`shrvar`) [**32, 128, 512**].
- `IPC`: *iterations per core*. Sets the number of transactions to be executed by each running core [**2K, 8K**].
- `NRD`: *number of reads* made in a single transaction [**1, 2, 4, 8, 32, 64, 128**].
- `NWR`: *number of writes* made in a single transaction [**1, 2, 4, 8, 32**] .
- `ITD`: *inter-transaction delay*. Sets the number of operations inserted between two consecutive transactions in order to emulate a variable number of computations performed outside the atomic regions [**0, 32, 1K**].

The characterization application works as follows: each core is set to execute `IPC` transactions (lines 2–14) and, within each transaction, to perform `NRD` transactional reads followed by `NWR` transactional writes on `shrvar`. Each core is responsible for reading a sliding window (`wdw`) on the array, which changes at each transaction. The function `GetWindowRange` (line 3) returns the slice of the shared array considered during the current iteration. The elements read from the window are then consumed, resulting in a distinct value (`x`).

After the value is computed, the write sequence starts by computing an `offset` with respect to the start of the sliding window. The offset is determined randomly (line 9) and is bounded to the sliding window. Next, a new value (`y`) is produced based on the current value and offset (line 11). Then, the resulting value is stored back into the sliding window.

When the sequence of reads/writes is completed, the transaction calls the commit operation (line 14). Subsequently, a variable delay (determined by `ITD`) may be requested outside the transaction boundaries, prior to the start of the next transaction.

3 Energy Characterization & Analysis

In this section we present a quantitative analysis of the results obtained through the characterization procedure described in the previous section.

We start off by showing the mean energy consumed per load and store for a system with 1, 2, 4 and 8 cores in Figure 3. In the load case (left), the horizontal axis represents the read set size in a logarithmic scale. The final energy value for a given read set size is calculated by taking the arithmetic mean of the energy consumed when the write set size is varied from 1 to 32. The figure for the store (right) is similar, except that the write set size is represented in the horizontal axis and the mean is taken over the read sets (varying from 1 to 128). It can be seen from Figure 3 that the energy consumed by both a load and store decreases as the size of the sets (read for load and write for store) increases. We found out that the larger overhead for the smaller cases is due to misses in the instruction cache (13% for load and 30% for store). Once the cache miss effect is eliminated the energy per operation seems to reach a stable value. On the other hand, further increase in the set size results in data cache misses and on a slight increment in energy consumption. Lastly, notice that a configuration with more cores consumes more energy due to a higher contention on the bus and an increasing number of wait cycles.

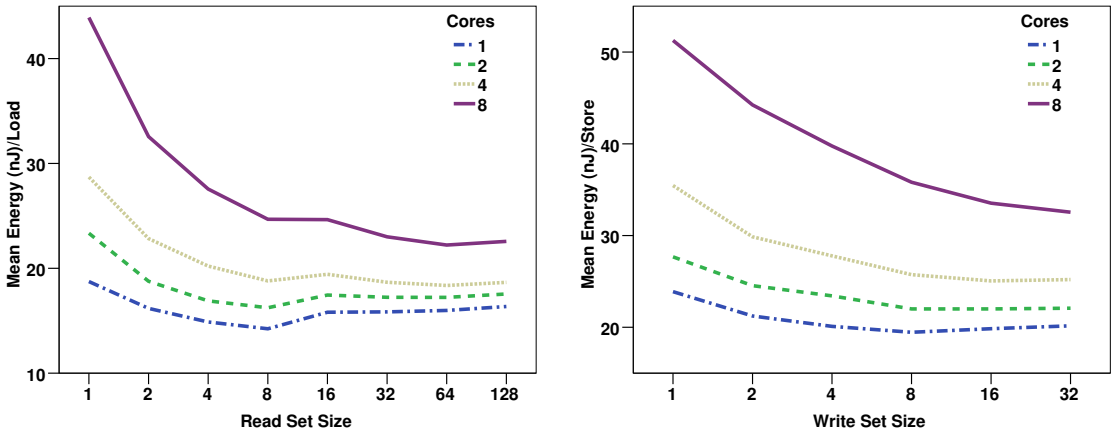

Figure 3: TM API's energy profile for *TxLoad* and *TxStore*

The results for the commit and abort operations are displayed in Figure 4. The figure shows the energy consumption per operation as both the read and write set size increase for different core configurations. Two main observations can be drawn from the commit results (top). Firstly, for a given read set size, notice that the energy cost is higher for larger write sets. This is due to the fact that the TL2 algorithm must iterate over the write set three times during the commit operation (one for acquiring the locks, another for writing back the values into main memory and a last one to release the locks). Secondly, the cost also increases with larger read set sizes, since the commit phase must validate the transaction read set for consistency. Regarding the results for the abort operation (bottom) we notice that the energy cost does not change considerably with the size of the read set. However, increasing the write set size causes a higher energy cost per abort. To explain this behavior, recall that in TL2 an abort is usually generated by either a failed load or an invalid read set. In the latter case, which is the most common form, the locks acquired partially in the commit phase must be released by the abort operation and thus result in a higher energy cost. Also, notice that the results for the commit and abort operations are as expected for a lazy STM such as TL2, i.e. the commit operation is expensive (actually, it is the most expensive among the four analyzed operations) whereas the abort operation is relatively cheap.

3.1 STM Overhead

Figure 5 presents an estimation of the STM energy overhead resulted from the characterization procedure. The horizontal axis uses the geometric mean to represent typical size values for the read and write sets. It can be noticed that, for small values (i.e., small read and write set sizes), the transactional overhead is high (ranging from ∼40% in the single-core case to ∼80% in the 8-core configuration). Once the set size increases, the overhead per operation is amortized and the overall overhead decreases. However, for larger read and write sets the overhead tends to increase again. These results suggest that there is a value

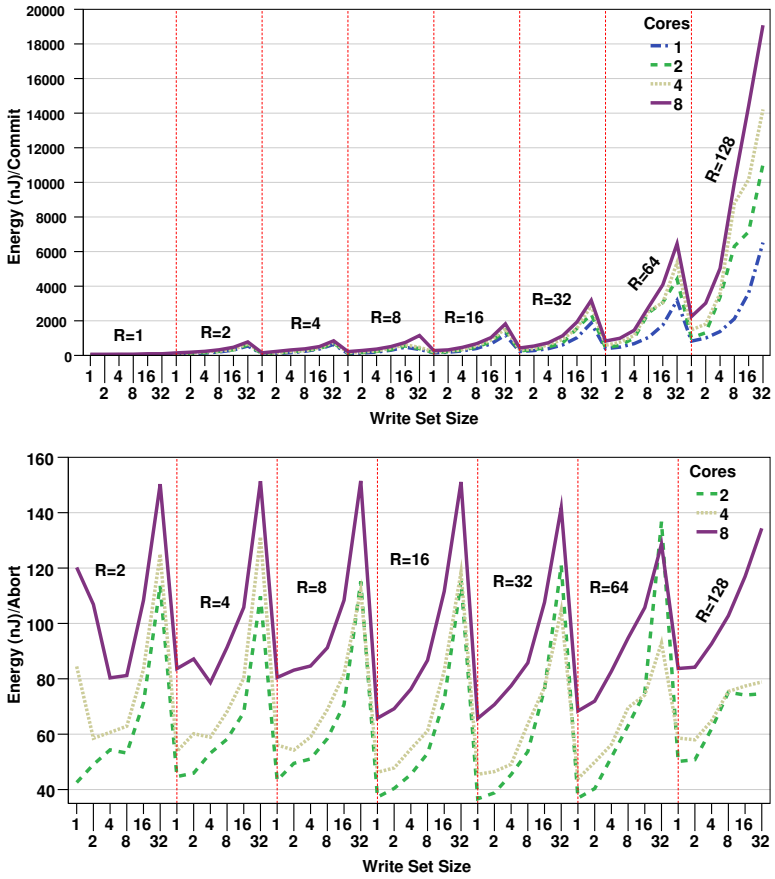

Figure 4: TM API's energy profile for *TxCommit* and *TxAbort*

for the read and write set in which the overhead tends to a minimum. This value depends on the particular core configuration (e.g., 2.83 for the single core and 22.63 for the 4-core).

3.2 STAMP Results

For the final set of experiments, we employed two of the STAMP [8] applications (*genome* and *intruder*) and analyzed the impact in energy consumption of the four major transactional calls, resulting in the energy breakdown shown in Figure 6. The input set for each application is the one suggested in the STAMP paper [8] for simulation environment. It can be seen that the major cause of energy consumption in the genome application is due to the load operation. As the number of cores is increased, the percentage of energy of each operation tends not to change much. As for intruder, the commit operation initially is responsible for roughly 55% of the total energy consumption. However, adding more cores causes the load operation to dominate. A small increase due to stores is also observed. Notice that the energy consumed due to aborts is insignificant with regard to the other operations. This is mostly because in lazy STMs the cost of an abort is cheap and, specially

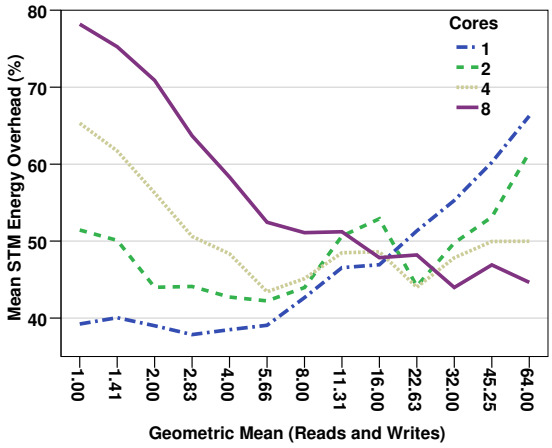

Figure 5: STM Energy Overhead

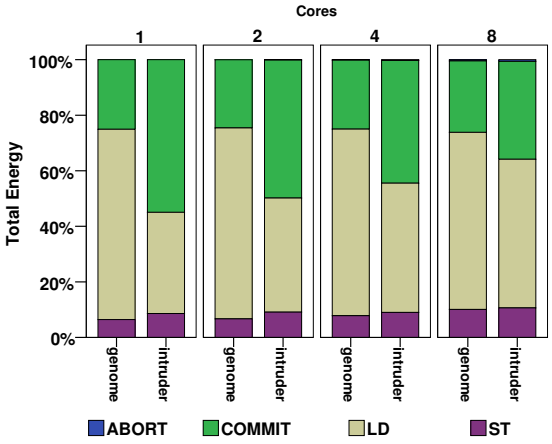

Figure 6: Energy breakdown for STAMP applications

for genome, the abort rate is low.

Notice that the conducted experiments do not consider the energy spent due to the contention management scheme, which tends to be different for each STM implementation. In this paper we focused on the STM API, which is general. If we include such schemes in our analysis, it is very likely that the contribution of the abort operation to the total energy will be higher.

4 Related Work

As previously mentioned, current evaluation of STM designs primarily addresses performance improvements over traditional locks. The usual performance metric is given by the number of transactions executed per unit of time (i.e., throughput). We are not aware of

any methodology for estimating the energy consumption of STM systems.

The works on power dissipation in HTM systems are the closest to ours. Moreshet et. al [9] initially investigated the energy consumed by transactions in a typical multiprocessor environment. While their results suggested that HTM has an advantage in terms of energy consumption over locks, one should notice that only micro-benchmarks were employed in the experiments. More recently, Ferri et. al [10] evaluated the impact of energy consumption in an embedded setting. They also proposed the use of a scratchpad memory for transaction checkpointing and a technique that shuts down the transactional cache in case of under-utilization.

5 Conclusions and Ongoing Work

This letter presented, for the first time, a systematic methodology which allows the characterization of STMs that implement a set of common API calls. By means of this methodology, we thoroughly evaluated the impact on energy consumption due to the STM approach usage and quantified the energy costs of the primitives used in an optimistic time-based STM implementation. This work is a first study towards a better understanding of the energy consumption behavior of STM systems and could prompt STM designers to provide optimizations, creating a so-called *energy-aware TM*.

As part of our current and future work we intend to evaluate and compare other STM implementations, not only in terms of energy, but also under power and EDP (energy-delay product) metrics. In addition, we plan to devise an energy macromodel for STM in order to allow energy analysis with the help of more abstract simulation models (functional), thus avoiding the long running times demanded by cycle-accurate simulation.

References

[1] Herb Sutter and James R. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[2] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[3] Stefanos Kaxiras and Margaret Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan & Claypool Publishers, 2008.

[4] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, 2007.

[5] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proc. of the 20th DISC*, 2006.

[6] Mirko Loghi, Massimo Poncino, and Luca Benini. Cycle-accurate power analysis for multiprocessor systems-on-a-chip. In *Proc. of the 14th GLSVLSI*, pages 410–406, 2004.

[7] Alberto Macii, Luca Benini, and Massimo Poncino. *Memory Design Techniques for Low Energy Embedded Systems*. 2002.

[8] Chi Cao Minh, Jae Woong Chung, C. Kozyrakis, and k. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. of the IEEE IISWC*, pages 35–46, 2008.

[9] Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. Energy reduction in multiprocessor systems using transactional memory. In *Proc. of ISLPED*, pages 331–334, 2005.

[10] Cesare Ferri, Amber Viescas, Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. Energy efficient synchronization techniques for embedded architectures. In *Proc. of the 18th GLSVLSI*, pages 435–440, 2008.