SAPIENZA
UNIVERSITÀ DI ROMA

SCHOOL OF INFORMATION ENGINEERING, COMPUTER SCIENCE AND STATISTICS

Master of Science in ENGINEERING IN COMPUTER SCIENCE

EFFICIENT SOFTWARE TRANSACTIONAL MEMORY VIA THREAD SCHEDULING AND DYNAMIC VOLTAGE AND FREQUENCY SCALING

Advisor

Francesco Quaglia

Co-Advisor

Pierangelo Di Sanzo

External Reviewer

Massimo Mecella

Candidate

Stefano Conoci

January 2017

Academic Year 2015/2016

Stefano Conoci

Efficient Software Transactional Memory via thread scheduling and Dynamic Voltage and Frequency Scaling

Master's Thesis in Engineering in Computer Science

Sapienza Università di Roma

January 2017

To my father,

for consistently being everything he could ever be.

Contents

Abstract

Transactional memory is a interesting parallel programming paradigm that offers the scalability of fine-grained locking without the need of handcrafted synchronization. It relies on the concept of atomic transactions that might commit or abort depending on the interleaving of operations on shared data. However, an excessive number of aborts could lead to performance degradation and wasted energy. Recently, there has been interest in the performance and energy optimization of transactional memory with techniques like thread scheduling [1, 2]. However, there are not yet studies that explore the energy efficiency and performance trade-offs obtainable when running transactional applications at lower energy CPU states. In this work we investigate the performance and energy efficiency of two current generation systems executing transactional applications with different configurations of parallel threads and CPU frequency and voltage. The results of this investigation are exploited to develop an architecture for the efficient execution of transactional applications. It is based on exploration heuristics that can efficiently select at run-time the configuration that provide the highest performance while operating withing user defined constraints on power and energy consumption. This thesis is organized as follows. In Chapter 1 we provide an overview of concurrent programming, transactional memories and we characterize the energy consumption of modern computing system. Chapter 2 contains a brief summary of the present state of the art of the performance and energy optimization of transactional memories. In Chapter 3 we perform an in-depth analysis of the performance and energy efficiency of transactional applications running with different configurations of parallel threads and CPU energy states. In Chapter 4 we present the proposed architecture, the exploration heuristics and we show the trade-offs obtainable with different constraints on power and energy consumption. Chapter 5 concludes this work with a brief summary of the achieved results.

1 Parallel computing and power consumption

1.1 Concurrent and parallel computing

Concurrent computing is a paradigm in which multiple computations are executed during overlapping time periods concurrently. This overlapping is made possible by time sharing operating systems developed in the 60s to overcome the limitations of batch computing. Computers used to be big, expensive and most importantly shared. Inspired by the idea that no average single user could utilize a computer's full load, time sharing operating systems allow multiple units of execution to be running at the same time in an interleaved fashion. The scheduler is in charge of deciding which program is being executed on the processor at any time. Each running task gets assigned a time slice, usually expressed as a number of clock cycles, which defines the amount of processor time left until the scheduler switches to another task. In concurrent programming there are two units of execution: processes and threads.

Threads, also known as lightweight processes, are an execution unit that lives inside a process and share the same memory space. The use of time sharing, even on single processor machines, gives to the user an illusion of actual parallelism of operations and allowed the development of intuitive means of interactions such as the use of the mouse as well as responsive graphical user interfaces. One of the most influential early time sharing operating system is Multics (Multiplexed Information and Computing Service) [3] which influenced all of the modern systems.

Concurrent programming is not only used for running multiple independent programs at the same time but can also be exploited by a single applications made of multiple processes or threads that can communicate through memory portions used as shared memory. This can be an effective way of optimizing the usage of a single processor considering that I/O operations on secondary storage or network interfaces could take multiple order of magnitudes more time to complete compared to read and write operations on main memory. The interactions of multiple traces of execution on the same data, even if at different times, might create different problems such as deadlocks, race conditions or resource starvation. These problems only appear in a portion of the program executions due to the non-deterministic nature of the ordering of execution defined by the scheduler which is influenced by the state of the whole system. The fragment of a program that performs operations on shared data is called critical sections and requires means of synchronization to avoid the contingency of the problems listed above. As identified by Edsger W. Dijkstra, this problems should be solved by guaranteeing the mutual exclusion of tasks in the critical section [4].

In 1965 Moore's law predicted that the number of transistors in a dense integrated circuit will double each year [5]. Ten years later it was revised as doubling each two years. Until only a few years ago the evolution of processors confirmed the pace of this prediction but it changed its manifestation over time. Up until early 2000, the increased density of transistors was used to increase the clock frequency of processors which resulted in a proportional speedup in the execution of both sequential and concurrent programs. However in the last 15 years, the thermal effect known as the power wall limited the possibility of increasing the core frequency even further. The ever increasing amount of transistors that could be put on the same chip area were already used to replicate some of the processing elements with the goal of increasing the performance of single core processors. The most used techniques were pipelining, which created the possibility of Instruction Level Parallelism, and Simultaneous Multithreading as a manifestation of Thread Level Parallelism which became the starting point for the development of multi-core processors.

Unlike a simple increase in core frequency, this radical change in hardware forces a radical mindset change on how the software is developed in order to fully exploit the processor capabilities. Albeit often confused, there are relevant differences between concurrent computing and parallel computing. We can define parallel computing as the execution of multiple processes or threads simultaneously on a single multi-core processor or multiple processors.

Concurrency is related to the software structure, namely the division in multiple processes or threads, while parallelism is about the actual execution. “Concurrency provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable” [6]. In order to have an application scale to multiple cores developers must find ways to partition the data allowing each core to perform its processing independently of others. However, this partitioning can often be a complex task such as for irregular problems which can only be “divided into subproblems in a manner not predictable a priori. This implies that the applications used to solve this kind of problems have an highly unpredictable behavior in terms of computing power and memory used” [7]. Moreover, problems have different level of intrinsic parallelism that can be exploited from a parallel execution. The possibility of executing on multiple cores at the same time makes the synchronization of threads and processes more critical. While in the single processor concurrent case the synchronization was mostly required for avoiding unpredictable behaviors of the application, in parallel computing synchronization also has a major effect on the performance of the whole application.

1.2 Parallelization and scalability

The simplest way of coordinating the access of multiple tasks that try to access the same portion of shared memory is by guaranteeing mutual exclusion. This can be achieved by locking portions of shared memory for exclusive usage. Parallel threads that attempt to access a locked memory item should wait until the lock is released. The most common blocking primitives are semaphores and mutex. The idea of locking portions of shared memory directly is only theoretical. In real implementations, lock acquisition and release should delimit all the operations on shared memory objects .

The computational effect of locking is a sequentialization of threads which has a negative impact on the performance of parallel applications. The weight of this performance limitation is related to the average percentage of the running time for a given thread spent waiting for another thread to release a lock. Unfortunately this percentage is likely to increase as the number of cores in the processor increases.

Amdahl's law defines the theoretical speedup of an application with fixed workload when system resources are improved. It can also be applied to the speedup of parallel applications with respect to their sequential implementation by computing the serial fraction of the program [8].

$$S_{Amdahl} = \frac{T_s}{T_p} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$$

where α is the serial fraction of the program as a number in the range from 0 to 1, p is the number of processors or cores, T_s is the serial execution time and T_p is the parallel execution time. With an infinite number of cores the function converges to $1/\alpha$: a program with a serial fraction of 20% cannot have a speedup higher than 5.

In literature some authors critiqued Amdahl's law considering the relation it defines as incomplete [9]. Gustafson proposed a revised iteration after extensive empirical study of a 1024 multiprocessor system inspired by the evidence that "when given a more powerful processor, the problem generally expands to make use of the increased facilities" [10]. When increasing the workload, either by increasing its size or computational complexity, the sequential portion can often remain constant. The researchers Sun and Ni in their paper "Another view on parallel speedup" [11] synthesize both the aforementioned laws in a single formula that considers an increase in workload bounded by memory. This is a relevant constraint considering that by increasing the number of parallel cores there is also an higher frequency of main memory requests which might lead to the memory subsystem becoming the system bottleneck.

1.3 Read-modify-write and synchronization

Read-Modify-Write is a class of operations that allow to read a memory location, compute a new value based on the memory content and then write to the same memory location atomically. Some operations also allow to evaluate a condition based on the read data and write back the new value only if a given condition is true. An operation is considered atomic when if it succeeds it changes the state of the system according to the operation semantic while if it fails the operation doesn't affect the system state in anyway. Atomicity can be considered a guarantee of isolation of concurrent processes. In single core machines it is enough to block system interrupts to achieve atomicity of a process execution. However, in multi-core parallel systems achieving this property is much more complex and requires hardware support. In x86 machines this operations can have a negative effect on the performance of the whole systems as it requires many operations to be performed by the cache controllers of all cores [12].

Read-Modify-Write operations are used to implement blocking synchronization primitives based on locks such as semaphores. However they are also the building blocks of non-blocking synchronization data structures and algorithms . The most relevant operations of this class are:

- *compare-and-swap*: reads the value of a memory location and compares it with a passed parameter that represents the expected value. If both values are the same a new value passed as parameter is written on the same memory location. This primitive is often used in a sequence composed of an initial read of the memory value, the computation of the new value that wants to be written and finally the compare-and-swap operation. This sequence is used as a guarantee that no other thread wrote on the memory location in the interval between reading an old value and writing the new value. Unfortunately this suffers of the ABA problem which requires the definition of a policy to be effectively prevented [13];

- *fetch-and-add*: reads the value of a memory location, adds up a passed parameter and writes the result in the same memory location;
- *test-and-set*: reads a memory location, checks if it respects a condition, writes a new value to the memory location and returns the old one. This is often used to swap bits.

All of the this primitives can be used to implement a locking mechanism. In different systems some of the presented operations can be directly implemented in hardware which can highly reduce their performance cost.

1.4 Correctness property of parallel programs

One of the most relevant difficulties introduced by parallel programming compared to the sequential paradigm is the undeniably increased complexity in the process of debugging and validation of the program behavior. Race conditions are difficult to recreate and debug as they arise from the non-deterministic ordering of threads working on shared data. In this scenario, the definition of parallel programs properties and conditions allow to fight this unpredictability through a formal approach. We can define two primary parallel program properties: *Correctness* and *Liveness*.

Correctness, also referred to as *Safety*, can be informally defined as the property that nothing wrong happens in the execution of the parallel program with respect to developer expectations. It can be defined more precisely by restricting to programs with inputs and outputs: "a correct program always finishes its computation with outputs related in the desired way to the original inputs" [14]. It is a property that excludes possible inter-leavings that might lead to incorrect behaviors but does not define which inter-leavings might occur. In the study of possible conditions that can generalize the correctness property we can roughly distinguish between two classes of conditions: those that are willing to isolate single operations on single shared objects and those that are willing to isolate transactions which can be defined as a sequence of operations on some form of shared data which require some properties to be guarantee in their execution.

1.4.1 Linearizability

Linearizability “provides a real-time (i.e., wall-clock) guarantee on the behavior of a set of single operations (often reads and writes) on a single object (e.g., distributed register or data item)” [15]. The condition is built from two main idea “first, each operation should appear to take effect instantaneously, and second, the order of non-concurrent operations should be preserved” [16].

The formal definition is based on the concept of history which is a sequence of call and return events on objects by a set of threads or processes. A sequential history is an history where all operations take effect instantly, namely all invocations have an immediate response. We can consider an history linearizable if:

- pairs of invocations and responses can be reordered as a sequential history;
- this sequential history is correct with respect to the sequential definition of the object;
- if in the original history a response precedes an invocation then it must also precede it in the sequential history.

An object is linearizable if all its valid histories are linearizable. As interesting property of this correctness condition is that it is a composable property: a system entirely composed of linearizable objects is also linearizable.

1.4.2 Serializability

Serializability is a condition that provides guarantees of correctness to the parallel execution of transactions. Informally, it is based on the idea that the outcome of the execution of a set of transactions should be the same as the outcome of the same transactions executed serially. While *linearizability* has its roots in the areas of study of distributed and parallel programming, the concept of *serializability* was initially introduced by researchers of data management and parallel databases. A transaction is a sequence of operations whose processing must provide guarantees on the properties of Atomicity, Consistency, Isolation and Durability (ACID properties). "In an atomic group of operations, either every operation in the group must succeed, or the effects of all of them must be undone" [17]. This distinction determines two possible outcomes for a transaction: the commit of all the actions performed by the transaction or an abort which may require a rollback of some the operations already performed on the system. *Serializability* is a condition used to guarantee isolation, namely that each transaction should be executed independently from other transactions and that the effects of the abort of a transaction should not affect the execution of other running transactions. The definitions of the other ACID properties can be found in this publication [18].

The formal definition of *serializability*, similarly to *linearizability*, relies on the notions of history and sequential history albeit defined in the context of transactions. An history models the execution of a set of transactions on a set of objects as a total order of operations, commits and aborts. Two transactions of an history are considered sequential if one invokes its first operations after the other either commits or aborts; if that is not the case they are considered concurrent. Consequently, if an history contains only sequential transactions it is a sequential history, otherwise it is a concurrent history. An history H of committed transactions is serializable if exists an history S(H) such that:

- it contains exactly the same transactions as H;
- it is a sequential history;
- every read returns the last value written.

Unfortunately, the problem of deciding if an history of transactions is serializable is NP-complete [19], making it unfeasible to be used in any Concurrency Control Protocol: a set of rules enforced at run-time by systems such as a Database Management

Systems and Software Transactional Memory frameworks to guarantee that the parallel execution is preserving some desired properties. Conversely, real systems use derivations of the concept of *serializability* which define subsets of serializable histories which can be verified in polynomial time. One of the most relevant differences with linearizability is that it doesn't impose any real time constraint on the ordering. The combination of *serializability* and *linearizability* generates the concept of *strict serializability*: transaction behavior is equivalent to some serial execution, and the serial order corresponds to real time.

1.4.3 Opacity

Unfortunately both the aforementioned conditions are not sufficient in the context of Transactional Memory. A running transaction that reads an incomplete state during its execution might lead to unpredictable situations such as a divide by 0 or looping inside a cycle. In a database system each transaction is executed in its own thread, thus in case of a runtime error the DBMS can simply abort the transaction and restart it in a new thread. Unfortunately that's not viable for Transactional Memory systems where both transactional and non transactional operations are interleaved in the same execution unit. In order to account for this more specific requirements, Rachid Guerraoui and Michał Kapałka formalized a new correctness criterion called *Opacity* [20] which provides stronger guarantees about the consistency of the values read by transaction. The formal definition requires multiple definitions and can be found directly in the cited paper. Intuitively it is a form of *strict serializability* that considers both running and aborting transactions: every operations should have its read-set consistent during its execution, even if the transaction ends up aborting.

1.5 Liveness property of parallel programs

Liveness, also referred to as *Progress*, is a parallel programs property that defines guarantees on the evolution of the system state. A procedure accessing shared objects can be considered blocking if it suspends the threads that want to enter a critical section while it is already being executed by another thread; otherwise it is considered non-blocking. Consequently, in a non-blocking procedure each critical section can only contain atomic operations to ensure that a process cannot be suspended during its execution. Blocking procedures can lead to an under-utilization of resources for ″asynchronous, fault-tolerant systems: if a faulty process is halted or delaying in a critical section, non-faulty processes will also be unable to progress″ [21].

We can distinguish between different progress conditions for either blocking or non-blocking procedures. The conditions for the former class are:

- *Deadlock-free*: some thread acquires a lock eventually;
- *Starvation-free*: every thread acquires a lock eventually.

These conditions are dependent on the scheduler because it is required that each thread can eventually complete the execution inside a critical section and release the lock. Differently, in the class of non-blocking procedures we can distinguish between:

- *Lock-free*: any process can complete an infinite number of operations in any infinite execution;
- *Wait-free*: "any process can complete any operation in a finite number of steps, regardless of the execution speed on the other processes" [21];
- *Obstruction-free*: a process executed in isolation can complete any operation in a finite number of steps;

The first two are independent from the scheduler, while the last requires that threads execute in isolation. Obstruction-freedom is weaker than lock-freedom which is weaker than wait-freedom. A very basic example of a *lock-free* procedure is the increment of a shared memory variable through the atomic operation *fetch-and-add*.

1.6 Software Transactional Memory

In 1977, Lomet observed that an abstraction similar to database transactions might be an effective programming language mechanism to ensure the consistency of shared data between processes [22]. However, Lomet did not propose any practical implementation. "An Architecture for Mostly Functional Languages" [23] published in 1986 introduced the idea of providing hardware support for transactions. In 1995 Nir Shavit and Dan Touitou proposed the first concept of a software only transactional memory [24]. In the last decade, the concept of transactional memory, both software and hardware, gained a lot of interest due to the increased relevance of parallel computing.

Software Transactional Memory(STM) systems provide an abstraction for coordinating concurrent reads and writes to shared data in a concurrent program. It is an interesting alternative to lock based synchronization that allows to hide the synchronization issues from the programmer by replacing locks with atomic transactions.

The use of locks introduces several software engineering problems. An erroneous handling of locks might lead to problems of deadlocks and races which may only appear under rare circumstances thus making them particularly hard to debug. Error recovery can also be tricky as it is required to release locks in exception handlers. In addition, locks do not compose which means that correctly synchronized fragments could fail when combined in a single procedure, hence reducing the possibility of software reuse. Synchronization is usually achieved using coarse-grained locks which increase the contention on shared data reducing the scalability of applications when increasing the number of parallel threads. However, developing an application with fine-grained locks is very complex and makes all the aforementioned problems even more glaring.

The goal of transactional memory systems is to achieve the scalability of fine-grained locks without the increased complexity and problems introduced by hand-crafted lock-based synchronization. The STM paradigm is particularly effective in situations such as graph algorithms where the set of accessed nodes depends on the encountered values. Transactional memory allows to ″exploit concurrency that would otherwise be hidden due to dynamically unnecessary synchronization″ [25]. An interesting usage of this programming paradigm is the transactional implementation of the Quake game server [26] where the fine grained lock-based implementation is simplified by not requiring the initial phase of simulation used to identify ″the list of objects on the map that the player is likely to interact with″ which are then locked to prevent concurrent access.

STM systems are developed as a framework that operates on top of the operating system and can be used by applications directly through API or, in some implementations, through ad hoc programming language extensions. For each executing transaction the STM system must keep track of its read-set and write-set to enforce correctness in different point of the execution.

The two main approaches used by STM systems to obtain an automated concurrency control are pessimistic concurrency control and optimistic concurrency control. The pessimistic approach is based on the idea of locks which are requested by threads either at encounter time or at commit-time [27]. It can be an over conservative technique which can have a negative impact on scalability. Optimistic concurrency control is based on non-blocking primitives and multi-versioning of shared memory items. The general idea is that conflict detection and resolution can be performed even after a conflict occurs. There is not a clear winner between this two different approaches: pessimistic concurrency control can have better performance in workloads with high contention and long transactions; conversely the optimistic approach can be preferred in workloads with short transactions that can complete most of the times without conflicts.

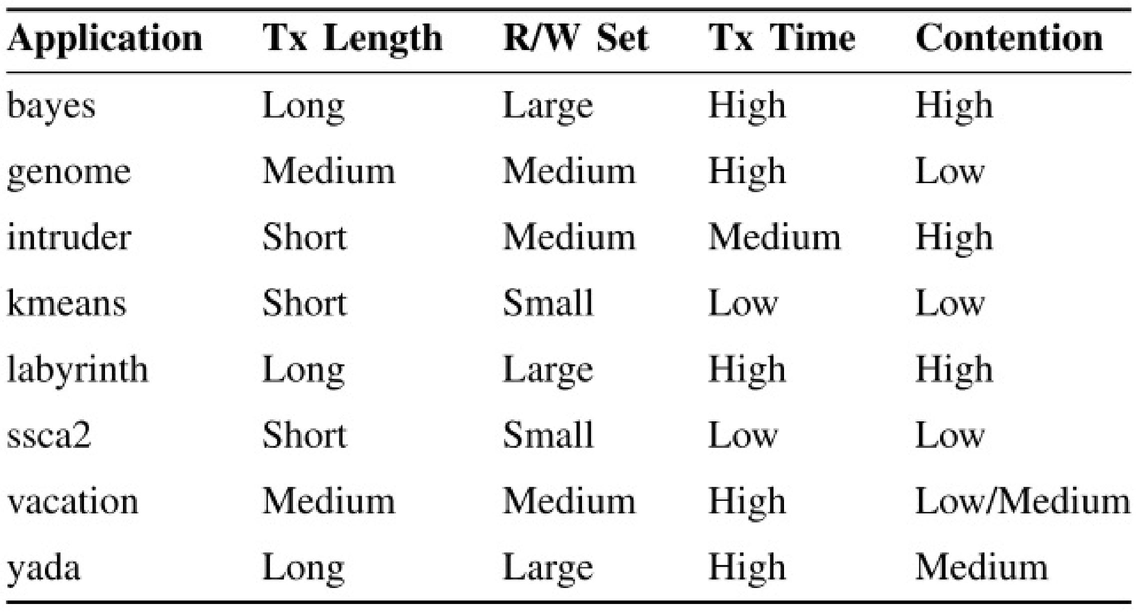
As anticipated in paragraph 1.4.3, transactions are protected portions of code executed by regular application threads: an inconsistent read of a shared item could crash or loop the whole program. To avoid this problem most STM implementations use Opacity as the correctness condition which however introduces a relevant performance cost [20]. Regarding liveness conditions, obstruction-freedom is a viable condition used by most implementations [19]. Unfortunately, wait-freedom is not achievable in an asynchronous system because aborts cannot be avoided [28].

The contention manager is a fundamental module of any STM system that has the role of maximizing the number of commits. It decides which transaction should be aborted in case of a conflict and when to restart an aborted transaction. Contention managers are particularly relevant in implementations that use an optimistic concurrency control approach which can suffer high abort rates in workloads with high contention resulting in a severe degradation of performance and energy efficiency. In literature we can find multiple policies optimized for different workloads [29].

1.7 STAMP benchmark suite

Stanford Transactional Applications for Multi-Processing (STAMP) is an extensive benchmark suite for evaluating transactional memory systems" [30]. It defines eight different applications and thirty different input parameters that cover a broad range of transactional applications. This benchmark suite is used for all the experimental analysis of transactional applications in this thesis as it contains workloads with varying degree of contention, various read-set and write-set sizes and different transaction length. The eight benchmarks used by this suite are:

- *bayes*: machine learning application that learns the structure of a Bayesian network from observed data;
- *genome*: bioinformatics application that performs genome assembly of DNA segments;
- *intruder*: simulation of the execution of a signature-based network intrusion detection that scans network packets and matches them against a known set of intrusion signatures;



Application	Tx Length	R/W Set	Tx Time	Contention
bayes	Long	Large	High	High
genome	Medium	Medium	High	Low
intruder	Short	Medium	Medium	High
kmeans	Short	Small	Low	Low
labyrinth	Long	Large	High	High
ssca2	Short	Small	Low	Low
vacation	Medium	Medium	High	Low/Medium
yada	Long	Large	High	Medium

Figure 1: STAMP Benchmark Transactional Profiles from [30]

- *kmeans*: data mining algorithm that groups objects of a N dimensional space in K different clusters;
- labyrinth: implements a variation of the Lee's algorithm [31] that computes a path between an user defined start point and end point inside a three-dimensional maze;
- *ssc2*: Scalable Synthetic Compact Applications 2 (ssca2) [32] is a set of four different kernels that perform operations on directed, large, weighted graphs. They are used in multiple scientific applications such as computational biology and security. STAMP implementation only focuses on the first kernel;
- *vacation*: implements an online transaction processing system that emulates a system for travel reservations;
- *yada*: stands as Yet Another Delaunay Application. It is the transactional implementation of an algorithm for Delaunay Mesh Refinement [33].

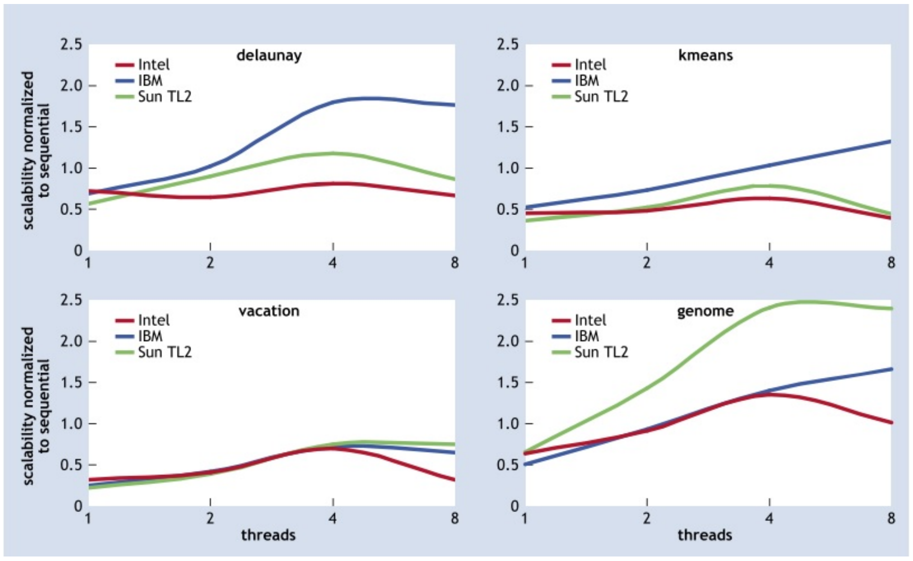


Figure 2: Results for three STM Runtimes on a Quad-Core Intel Xeon Server from [34]

Figure 1 shows the transactional behavior of each of this applications by defining respectively: the length of transactions, read and write set sizes, transactions execution time compared to the overall execution time and the degree of contention. This workload profiles are only relative to executions with the suggested input parameters; the transactional profile of this applications can vary significantly if user defined input parameters are used.

1.8 Performance of STM systems

The automated concurrency control of STM systems introduces a relevant overhead in the execution of transactional application. It is particularly relevant in systems with a limited number of cores which cannot exploit the effects of the increased scalability compared to coarse grained locking or sequential implementations. In the article ”Software Transactional Memory: why is it only a research toy?” [34] the authors study the overhead and performance of different STM systems (Intel [35], IBM [36], TL2 [27]) on multiple workloads.

Figure 2 presents the scalability of transactional applications normalized to their

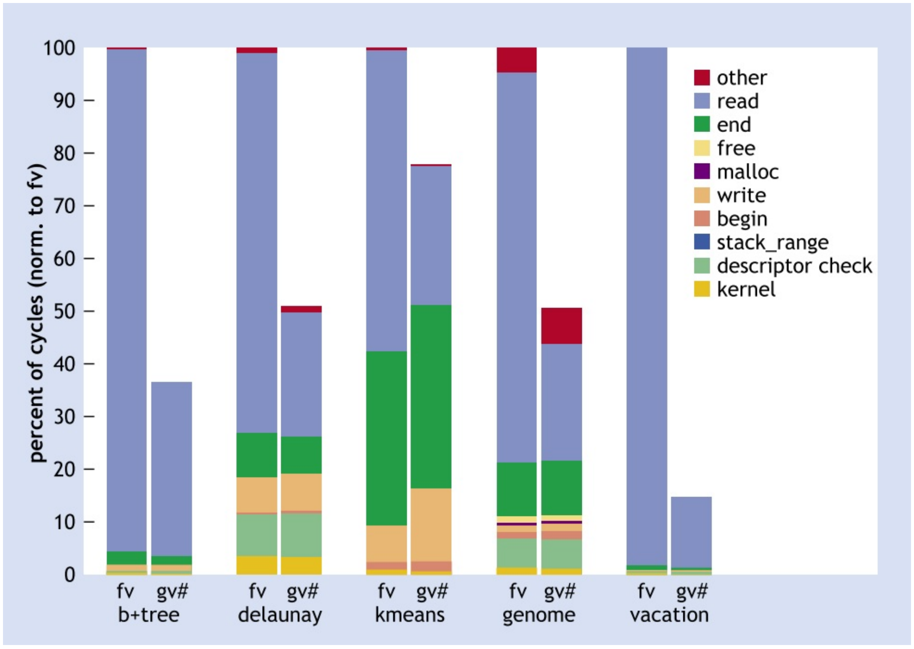


Figure 3: Percentage of Time Spent in Different STM Operations from [34]

sequential execution. The tests are performed on a Intel Xeon quad core CPU with two-way hyper-threading. *Kmeans*, *vacation* and *genome* are part of the STAMP benchmark suite, while *delaunay* is a transactional implementation of the Delaunay Mesh Refinement algorithm external to STAMP. All the STM implementations require 3 to 4 concurrent threads to reach the performance of the sequential implementation in all the considered benchmarks. K-means shows limited scalability for IBM and Sun STM's due to high abort rate. Single thread executions of the transactional implementations display on average an overhead close to 100% compared to the sequential implementations.

Figure 3 shows the percentage of time spent in the execution of different STM operations. It compares the results of two different STM implementation: one that fully validates (fv) its read set at each transactional read and the other that is based on a global version number to avoid validating at each read (gv). In all the tests the read operations dominate the transactional cost due to their higher occurrence. The global version implementation trades faster read operations with more costly end operations which seems to be a beneficial trade-off for the tested workloads.

In applications with limited shared data conflicts, ”transactions have an advantage over locks in terms of performance as well as energy, due to fewer accesses to shared memory” [37]. That’s the case due to the high amount of reads of the lock value by waiting threads.

1.9 Hardware Transactional Memory

In the last 20 years there has been increasing interest in the idea of hardware implemented transactional memory(HTM). In 1993 Herlihy and Moss [38] proposed a multiprocessor architecture that provides hardware support to lock-free data structures allowing the programmers to “define customized read-modify-write operations that apply to multiple, independently-chosen words of memory”.

HTM could provide many advantages compared to software-only implementations:

- significantly lower overhead in the execution of transactions;
- more efficient power and energy profiles;
- less invasive on the existing execution environment;
- can provide isolation without requiring any change to non-transactional code.

In 2013 Intel launched a new family of microprocessors which provide hardware transactional memory support through an extension of the X86 instruction set architecture called *Transactional Synchronization Extensions* (TSX). Inspired by the Herlihy and Moss proposal, TSX exploits the first level cache and the cache coherence protocol to achieve the properties required for a transactional execution. The conflicts are detected at the granularity of a cache line. During the execution of a transaction writes are performed on the first level cache without being visible to other cores.

Tentative changes are made visible atomically when a transaction successfully commits. TSX provides two different software interfaces to be used in different scenarios: *Hardware Lock Elision* (HLE) and *Restricted Transactional Memory* (RTM).

HLE is a legacy compatible instruction set extension that can be used as a way of increasing the parallel performance of programs using the conventional lock-based synchronization. Software written using the HLE interface can also be executed on hardware not supporting TSX in which case the related instructions would simply be ignored and the synchronization would be achieved through ordinary locks. In a critical section delimited by HLE instructions, updates to shared data structures are performed speculatively using transactional memory to detect conflicts. If a conflict is detected the critical section is re-executed acquiring regular locks.

RTM is a software interface that allows developers to define transactions in a similar way to how they are defined in STM systems. Unlike HLE, it cannot be used on hardware that doesn't support the extended TSX instruction set as it would generate an undefined instruction exception. The RTM interface gives the programmer the possibility of defining a fallback code path to be executed if a transaction aborts more than a defined number of times. Unfortunately this alternative path is necessary due to the limitations of RTM. A transaction aborts if any other core, even if not inside a transaction, reads a location in the transaction read-set or writes to a location that is either in the transaction read-set or write-set.

Unfortunately, in the Intel hardware implementation the motivations of a transaction abort are not always linked to shared data contention. As a matter of fact in general workloads up to 99% of aborts are not related to conflicts [39]. Transactions could also abort due to context switches, interrupts, page faults, capacity problem of the L1 cache and updates of the PTE accessed and dirty bits.

Considering this restrictions, Intel HTM capabilities should only be used in workloads dominated by very short transactions. Moreover, by requiring the definition of a lock-based path of execution the software engineering problems related to this concurrent programming paradigm are still relevant. For this reason in the last few years there has been growing interest in the idea of developing an hybrid transactional memory which is based on the general structure and interface of regular STM implementations but uses the capabilities of HTM to improve the performance when applicable [40].

This thesis focuses studying and exploiting the relation of performance and energy efficiency of applications using STM systems due to its broader applicability. However, some of the conclusions and methodologies should also extend to current and future iterations of HTM systems.

1.10 Non-Uniform Memory Access

A symmetric multiprocessor system (SMP) is a system where multiple processors are connected to a single shared main memory and are controlled by the same operating systems. However, generally only one processor can access memory at any time which leads to a significant performance degradation that scales poorly with an increasing number of processors in the system as well as the an higher amount of cores per single CPU.

In 2003 AMD launched a new generation of Opteron processors targeted at the server market which supported Non-Uniform Memory Access(NUMA). NUMA systems consists of several nodes each containing a subset of CPU cores and a portion of main memory. A core accessing memory from within the node is called a local access while accessing a different node is called a remote access. In modern NUMA systems a remote access takes on average 30% longer than a local one, while on older hardware it could take up to seven times longer [41]. Remote accesses have an higher latency because they must traverse one or more interconnect links, i.e. north-bridge buses that connect different NUMA nodes. Efficient software running on NUMA systems must consider how nodes are connected, where the program's memory is placed and how it is accessed by different nodes to avoid bottlenecks produced by remote accesses.

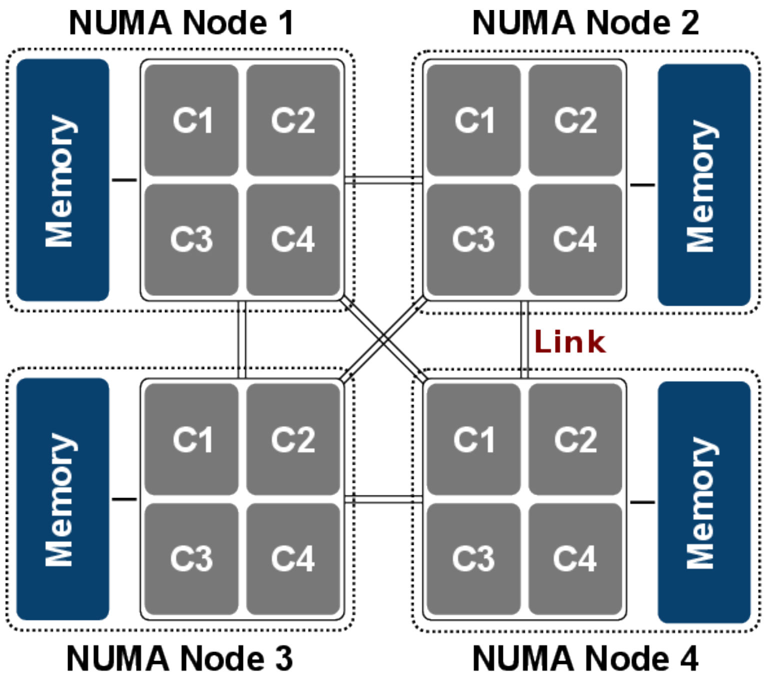


Figure 4: Non-Uniform Memory Access Architecture

1.11 CPU power consumption

The CPU is one of the components that has the highest impact on the energy consumption of a whole system. Unfortunately a big portion of this power is converted to heat due to the impedance of the electronic circuits. The main reason behind the switch from single core to multi core processor was the existence of the so-called power-wall, namely, it was no longer possible to dissipate with commodity cooling solutions the heat generated by cores of higher frequencies. There are three main components that contribute to the CPU power consumption: dynamic power consumption, short-circuit power consumption and power loss due to transistor leakage currents.

$$P_{cpu} = P_{dyn} + P_{sc} + P_{leak}$$

The dynamic power consumption is related to the switching of logic gates through the charge and discharge of capacitors. It is proportional to the CPU frequency and to the square of the CPU voltage:

$$P = CV^2 f$$

where C is the capacitance, V is the voltage and f is the frequency. The nonlinear relation between the power consumption and the CPU voltage creates interesting possibilities and was one of the main inspirations for this study. Unfortunately reducing the voltage often requires a reduced processor frequency: when running at lower voltages takes more time to recognize the correct state represented by a voltage transition.

The short-circuit power is dissipated due to a temporary direct path between the source and the ground of a transistor during simultaneous conduction of both p- and n- block of CMOS cell [42]. The power consumed by transistor leakage currents are related to small amounts of currents flowing between the differently doped parts of the transistor. They are considered as a form of parasitic power because they are unrelated to transitions.

A decade ago the dynamic power consumption was the dominant factor in the chip total power consumption. However, the reduction in size of transistors inflated the relevance of leakage currents making it the most relevant. Last generation processors attempt to reduce this effect exploiting power gating during idle, high-k metal gates and voltage scaling.

1.12 System power consumption and energy proportional computing

In 2007, Google engineers Luiz André Barroso and Urs Hölzle published an article that supported "The case for Energy-Proportional Computing" [43]. A typical server has an average utilization between 10% and 50% as over provisioning is required in order to deal with load spikes. Unfortunately this utilization level region is also the least energy-efficient due to the high static energy consumption of components. In the article the authors suggest that computer architects should focus on developing

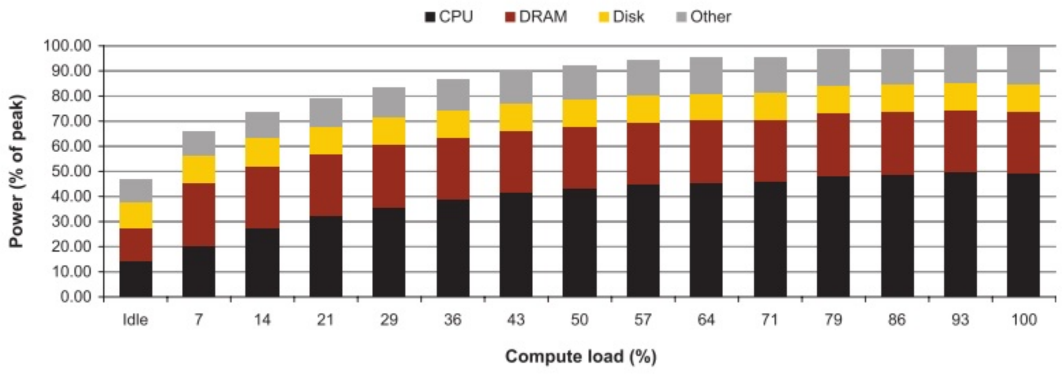


Figure 5: Percentage of the Total System Power Consumption from [44]

hardware that exhibits an energy consumption proportional to its utilization level. In their analysis "energy efficiency in the 20 to 30 percent utilization range—the point at which servers spend most of their time—has dropped to less than half the energy efficiency at peak performance".

Figure 5, taken from another study in 2009 of the same Google researchers on energy efficiency of data centers [44], shows the percentage of the total system power consumption consumed by the most relevant subsystems. Disk and DRAM consume more power than the CPU with low load. Both subsystems show a very limited dynamic range, i.e, the increase in power consumption of the subsystem from idle to max load. The study from Figure 1 reports the dynamic range of CPU to be approximately 3.5X, 2.0X for memory, 1.3X for disks and less than 1.2X for networking switches. A portion of the energy in input to the system power supply is wasted in heat. The amount of wasted energy is expressed by the efficiency of the power supply which varies at different power output levels. This can introduce an additional element of power dis-proportionality considering that the peak efficiency values are usually closer to the system full load than to the 20 percent utilization range.

In the last 10 years researchers focused most of their energy in the energy efficiency of the CPU. At the moment of writing the newest generations of processors show a much higher dynamic range which can be considered energy proportional.

Moreover, in Chapter 3 we show the possibility of remarkably higher efficiency below max CPU utilization by exploiting dynamic voltage and frequency scaling. Reduced voltages also reduce the effect of leaking currents making low voltage operational CPU states attractive.

However, the improvements in CPU power efficiency reduce the significance of the subsystem in the total system power consumption. Unfortunately memory, hard disk drives and network adapters still do not show a proportional energy consumption. Nonetheless, new and improved technologies can allow a reduction in the power consumption of this subsystems making the more scalable CPU portion more relevant on the overall system wide power utilization. Low voltage and SO-DIMM used in mobile devices can reduce the power consumption with a minor decrease in memory bandwidth. Additionally, last generation DDR4 memories consume up to 30% less than DDR3 memories at the same frequency [45].

The energy consumption of hard disk drives is dominated by the mechanical energy necessary to rapidly rotate the disk which limits the possibilities of energy efficiency improvements due to mechanical limitations. Solid state drives are an interesting new technology that can drastically improve performance and energy efficiency while also being power proportional [46].

1.13 ACPI interface

Advanced Configuration and Power Interface (ACPI) is a specification that provides an open standard for operating systems to discovery and configure hardware components, monitor the system status and perform power management. ACPI has three main components at the firmware level:

- *ACPI Tables*: define the interfaces to the hardware through configuration tables and executable functions defined in ACPI Machine Language (AML) bytecode. This code is parsed by the kernel and executed using an embedded minimal virtual machine;

- *ACPI BIOS*: boots up the machine and provides basic power management operations such as putting the system to sleep and wake it up;
- *ACPI Registers*: a set of hardware management registers defined by the ACPI specification.

An example of an *ACPI Table* is the *System Resource Affinity Table* which is evaluated at boot time by the Operating System. It associates physical memory ranges to processors giving to the system the information of which memory portion has the minimal distance to a given set of cores in a NUMA architecture.

ACPI provides power management control to the Operating Systems. It allows a more flexible development of energy efficient systems compared to old systems which were limited to platform-specific power management in firmware. The ACPI specification defines the power management interface through a set of platform-independent states.

Global states model system wide power status: state G0 specifies a running system state with the CPU executing instructions, G1 defines sleeping mode which is in itself divided in 4 different sleeping states from S1 to S4; G2 defines the state where the computer is shut down but the power supply still supplies power and its possible to restart the system pressing the on button; G3 represents the state where the computer is mechanically off, the power supply unit doesn't provide any power and the power cord can be safely removed. Similarly, device states ranging from D0 to D3 express the operating state of devices.

More relevant to the work of this thesis are the processor states (C-state) and the performance states (P-state). In the set of processors states, only the states from C0 to C3 are defined by the ACPI specifications: further states can be defined by manufacturers to provide operating systems a more fine-grained control over the CPU energy consumption. C-states are also known as idle states because, excluded C0 which is operational, in all the other states the processor is idle. A processor transitioning to an higher C-state reduces the CPU power consumption but also increase the delay required to return to an operational state. The ACPI specified C-states are:

- C0: processor is executing instructions;
- C1: known as *Halt*. The processor is not executing instructions but can return to C0 almost instantaneously;
- C2: known as *Stop-Clock*. Returning to C0 requires a delay. Processor responds to cache coherence traffic;
- C3: known as *Sleep*. Processor does not need to keep cache coherent. However, the state of processor registers is preserved.

Only C0 and C1 are mandatory in ACPI compliant systems. C1E is a common additional state defined by manufacturers: it is similar to C1 but also reduces the frequency and voltage of the processor further reducing its power consumption. Current generation Intel CPU's also provide package C-states which allow a fine-tuning of energy-states independently for each package of a multiprocessor system [47].

While a processor operates at C0 it can be in one of the several performance states. P0 defines the CPU state with highest frequency and voltage which entails respectively maximum performance and maximum energy consumption. Higher states must have an increasingly reduced frequency and voltage. The value of voltage and frequency of C-states is completely processor dependent. The only requirement is that they are ordered according to the decrease in power consumption and performance. The number of available C-states and respective values can be discovered at run-time by the Operating System. The predetermined ordering and run-time discoverability provide an effective interface for the definition of algorithms that optimize the CPU energy efficiency independently of the processor specifications. The technique of dynamically decreasing frequency and voltage of the processor to reduce power consumption, also known as Dynamic Voltage and Frequency Scaling (DVFS), has been broadly studied in the context of mobile devices but it is still a relatively new approach for desktop and server systems.

An experimental analysis on different families of Intel processors shows that the delay for a P-state transition can be estimated in the range from 20 to 70 microseconds [48]. The study also detects a ”variable cost of a frequency increase compared to the nearly fixed cost of a frequency decrease”. Current generation processors are provided with technologies, such as *Intel Turbo Boost* and *AMD Power Boos*t, that allow them to run at even higher frequencies than those defined in their specifications as long as the processor is running below power and temperature limits. This frequency increments are controlled directly by hardware and are enabled only when the processor is running in state P0. This technologies can optimize performance and energy in parallel executions with unbalanced load between cores. They allow to speed-up the execution of the application's critical path while saving energy in the execution of the non-critical portions.

1.14 Power Management on Linux

As anticipated in the previous paragraph, ACPI gives to the Operating System the responsibility of managing the CPU power state. *Cpufreq* is a subsystem implemented in the Linux kernel that provides an interface for setting the processor frequency and voltage through the use of P-states. CPU-specific drivers must be loaded on the system to able to perform efficient frequency and voltage changes. The cpufreq infrastructure allows to select between multiple governors which are different policies of dynamic p-state selection based on different criteria. The cpufreq governors available by default in most kernel images are: performance, powersave, ondemand, conservative and userspace [49]. The latter one is particularly interesting as it allows user space control of the processor P-state through the interaction with pseudo files exposed by the *sysfs* file system. The userspace governor allows to set different P-states to different cores of a multicore CPU. However, the actual hardware results of this lack of homogeneity can vary depending on different processors and *cpufreq* drivers.

Unfortunately, the dynamic control of C-states presents more complications than the management of performance states. In many systems, the most conservative C-states are disabled by default and should be enabled from the BIOS interface. Idle states are entered when the processor is not executing any instruction thus cannot be explicitly set from a running user space program. It's possible to limit from software the number of C-states that can be used by writing the maximum allowable latency for returning to C0 to a pseudo file in `/dev/cpu_dma_latency` [50]. However, this limitation only stands until the pseudo file is kept open. *Cpuidle* is a generic processor idle management framework that tries to emulate the architecture of drivers and governors used by cpufreq for performance states and applies it to the context of processor power states [51]. There is a tradeoff between the reduced power consumption and the time required to enter and exit an idle state. Higher C-states discard portions of the processor state which must be restored when returning to C0. However, this process of restoration requires power: entering a lower power state only for a short duration might be energetically inefficient. The wake-up duration from C1 on Intel Sandy Bridge and Westmere processors are below 1 microseconds while are close to 10 microseconds in AMD Bulldozer CPU's [47]. Respectively, C3 results are approximately 30, 15 and 50 microseconds. For all the considered platforms the wake-up time is reduced when the processor operates in a faster P-state.

1.15 Real-time software energy monitoring

Real-time decisions based on power consumption, such as the dynamic overclocking performed by *Intel Turbo Boost*, used to rely on power consumption models which are by definition conservatives. In 2011, Intel introduced in its new Sandy Bridge processors onboard power meter capabilities which allow better dynamic power management decisions from the hardware. In addition, real-time power related informations are exposed to the software through a set of *Machine Specific Registers*(MSRs) and PCIe config space which define an interface known as *Running Average Power*

Limit (RAPL) [52]. The software availability of this data creates new possibilities in the development of energy aware applications that can use the power consumption as an input of energy optimization algorithms. RAPL also provides the possibility of setting power limits on processors packages and memory which can be really useful in data centers scenarios where there is a requirement on power and cooling budgets.

The interface provides fine-grained information on the power consumption and power limits of different packages. Within a package it also discriminates between the energy consumed by the core, i.e. the components involved in executing instructions, and the uncore which is constituted by auxiliary systems such as QPI controllers and L3 cache controllers.

Powercap is a Linux kernel framework that provides a convenient wrapper of the RAPL interface by defining a hierarchical structure of folders and pseudo files in the *sysfs* file system. Each power zone is associated with a folder that contains energy monitoring attributes and constraint controls. The pseudo file named energy_uj displays as an increasing counter the energy consumption, expressed in μJoule, of the respective power zone.

2 State of the Art

In this section we briefly introduce the present State of the Art in the research fields of STM performance and energy optimization, DVFS optimized applications, speculative execution and energy efficiency, and finally the studies that try to combine this different research areas for further optimization of both performance and energy consumption in STM systems.

2.1 STM performance optimization

The speculative processing of transactions in STM systems creates a limit on the degree of parallelism that can be exploited without incurring in performance penalties in a transactional application. This phenomenon, known as *trashing*, is produced by an unacceptably high percentage of aborting transactions. The concurrency degree limit is strictly related to the application access pattern to shared data at a given point in time. Aborted transactions not only have a negative effect on performance but also increase the amount of useless work performed by the system which results in wasted energy. Therefore a STM system that attempts to optimize the performance by reducing the abort rate can also have a positive effect on the system energy efficiency.

In literature we can find different orthogonal approaches used to reduce the abort rate of transactional applications:

- early conflict detection and contention managers [53, 54] ;
- transaction scheduling which ”is based on delaying the execution of a transaction depending on the current system state and some scheduler-embedded policy” [55]. To decide which transactions should be delayed the system must estimate the abort probability which could be either dependent on the overall system state or reliant on transaction specific features;

- thread scheduling methodologies that determine the optimal number of parallel threads that should be used in a transactional application running on top a STM system.

The architecture proposed in Chapter 4 combines thread scheduling and DVFS. Accordingly, in this section we will mostly cover different iterations of this optimization approach.

We can distinguish two main classes of scheduling approaches valid both for thread scheduling and transaction scheduling:

- scheduling approaches based on performance predictions models such as analytical or machine learning models which require an a-priori profiling of the system;
- those based on heuristic methods which may require users to define scheduler parameters.

Among these, we can furthermore distinguish between solutions that cope with static or dynamic application execution profiles. Methodologies targeted at static execution profiles cannot predict the optimal degree of concurrency in situations where the contention profile changes, e.g. when the number of accessed objects per transaction increases over the execution time of the application.

The work in [56] proposes a self-regulating dynamic tuning of the concurrency degree of transactional applications relying ”on a parametric analytical performance model aimed at predicting the scalability of the STM application as a function of the actual workload profile”. The goal of this work is to combine the benefits of machine learning approaches and analytical model approaches. The parameters of the model are instantiated through regression analysis on a very limited number of profiling samples of the application execution at different concurrency levels. It is a much faster process compared to machine learning approaches which require an extensive training that should cover all the possible parameter values. This approach doesn’t rely on general assumptions which are usually required in the definition of pure analytical models while still providing extrapolation capabilities of the behavior on concurrency levels distant from those used for the model initialization.

Another interesting hybrid approach takes advantage of analytical techniques to reduce the training time of machine learning approaches while keeping the higher accuracy and reliability of this techniques compared to other solutions [57]. The core idea of this combination is the introduction of a new type of training set referred as Virtual Training Set which is constructed by a combination of real samples and interpolated samples obtained from the analytical model via regression. Experimental results show that this solution provides the same accuracy of pure machine learning techniques while requiring around half the time for instantiating the application specific performance model.

In [58] the authors present a heuristic based scheme that dynamically tunes the number of concurrent threads running in the application. This solution performs an iterative exploration of the space composed of different numbers of parallel threads where the search direction is defined by the hill-climbing optimization technique. The iterations do not stop until the end of the execution in order to adapt to dynamically changing workloads. Each iteration is constituted by three phases:

- measurement phase: the application runs with a fixed number of threads and measures performance metrics such as the commit rate, which gives an indication of the application's throughput, and the abort rate which quantifies the contention;

- decision phase: the algorithm decides if it should increase or decrease the number of parallel threads. If the last measurement phase displays an improvement of the commit rate compared to previous one, continue exploring in the same direction; else invert it.

- transition phase: an external controller either adds or removes thread based on the outcome of the decision phase.

It should be noted that this approach only converges to a local maximum. However the authors couldn't find workloads with multiple maxima in any of the tested benchmarks. Experimental results on the *intruder* benchmark (STAMP), which includes large variations in its workload, display an increased average throughput compared to all the static executions, i.e. those with a fixed number of parallel threads.

2.2 Energy optimization through DVFS

The work in [59] presents an algorithm that "computes task slowdown factors based on the contribution of the processor leakage and standby energy consumption of the resources in the system". Despite being often ignored, the standby power consumption of devices like memory banks, network adapters and disks has a great relevance on the overall system energy efficiency, especially in the context of distributed computing. In the last 15 years leakage currents have increased with the advances of CMOS technology. Clock gating allows to negate leakage currents when the processor is idle, therefore longer execution time results in more leakage energy consumed. At the same time, a running processor with a decreased voltage, and consequently lower frequency, has a reduced amount of leakage currents. The proposed solution attempts to take in consideration all this elements to compute for each task a slowdown factor, normalized to the maximum processor speed, at which is more efficient to execute a set of task while operating within applications specific constraints. Each task in the system is represented by a tuple that defines the task period, the relative deadline and the worst case execution time of the task running at the maximum processor speed. The tasks are scheduled following an Earliest Deadline First scheduling policy that considers the tasks deadlines to be equal to their period. The problem of computing the optimal slowdown factor for all the tasks has an unknown complexity. The authors defined an heuristic algorithm that computes the critical speed of each task and then increases the task slowdown factor if the resulting set of tasks is not feasible. Compared to traditional dynamic voltage scaling, which ignores the system wide standby power consumption and assigns to each task the minimum task slowdown, this approach can save on average 10% energy when considering a set of randomly generated tasks. This work is targeted to single core processors which have many differences in power management and overall power consumption compared to current generation multicore CPU. However, the idea of slowing down the execution in order to achieve reduced system power consumption compared to default DVFS governors is interesting and can be considered one of the foundations of this thesis.

The work in [60] proposes a run-time system called ”Adagio” for High Performance Computing applications that can achieve significant energy savings while only incurring in negligible delays. The target programming model is defined by a set of distributed memory systems using the *Message Passing Interface* (MPI) for any communication within processes. Each system has a single core processor with DVFS support. The general idea of this solutions is to dynamically detect the application critical path and slow down other portions of the application without ever increasing the overall application delay. The run-time is based on an online scheduler that identifies individual MPI calls, with the respective parameters, by hashing the stack trace. Each of this calls is associated with performance counters initialized after the first execution and exploited to define the operating frequency of subsequent executions. If a task incurs significant slowdown due to communication or other forms of blocking, ”e.g. ” MPI Barrier, it will be executed at a slower frequency in the following execution. This approach assumes that the behavior of each task should be identical over different executions, which is often the case for scientific applications. Experimental results show up to 18% decreased energy consumption with an increased application delay of less than 1%.

Child and Wilsey explore the possibility of improving both performance and energy efficiency of Time Warp simulation using the userspace DVFS governor [61]. Time Warp is a form of Parallel Discrete Event Simulation which relies on the concept of Virtual Time as a synchronization paradigm [62]. Events are characterized by their send and receive time and are used for the communication between objects. A set of these objects, which model physical processes relevant for the simulation, are grouped in *Logical Processes* (LP) which are mapped to *parallel execution units* (PEs). The time ordering of events imposes limitations on the scalability of pessimist parallel simulations as each object at a given point in the simulated time could be influenced by messages sent by another objects at a previous simulated time: only events with the same time-stamp can be safely processed in parallel. Time Warp proposes an optimistic approach where each execution unit processes events following its own *Local Virtual Time* (LVT) which can deviate from the LVT of other execution units. Whenever a LP receives a message with a time-stamp lower than its LVT it must rollback to a previous state that can process the event according to its causal order. In order to preserve the system wide causal order of messages a LP performing a rollback should not only undone processed events but also send anti-messages to inform other LP that previously sent messages are not causally ordered. The speculative execution of Time Warp results in all the PEs having an equally high CPU utilization. However not all the LP provide the same amount of useful work as a result of rollbacks. Default DVFS governors and boost technologies in this scenario are sub-optimal because they cannot deduce the critical path of the problem solely from CPU utilization metrics. This shortcomings are shared with other forms of speculative execution such as STM systems. Child and Wilsey propose three different metrics for estimating the amount of "useful work" performed by different execution units:

- effective utilization: defined "as the fraction of work on a given node that will not be rolled back";
- number of rollbacks: which are simply the number of rollbacks that occur during a measuring cycle;
- efficiency: which is an estimation of the percentage of committed event executions.

This metrics are used to manually increase the speed of the LP involved in the critical section, potentially using boost technologies, while also slowing down the non-critical processes. The results are particularly interesting as they show both increased performance and energy savings. Unfortunately this approach cannot be directly converted to STM systems where transactions do not present a causal order. Moreover the execution of transactions is by definition logically independent which complicates in the transactional context the definition of critical path.

2.3 DVFS and STM: performance and energy efficiency

In this subsection we present different solutions that combine DVFS and STM to achieve improved performance and energy efficiency in transactional applications. The solution introduced in [63] characterizes the energy consumption of STM systems using a cycle accurate simulation platform based on ARMv7 processors. The results of this simulation provide the evidence that energy efficiency and speedup are dissociated. In addition, the authors present a DVFS-based strategy applicable to any contention manager that causes transactions to wait. This pause of the thread execution is usually implemented by busy waiting; asynchronous wait primitives are too unresponsive for fine-grained back-offs and introduce overhead. Before entering backoff-mode the related core reduces its frequency and voltage which are then restored to the previous high performance values when the backoff period completes. This approach can reduce the energy wasted by the processor while not performing any useful work. It is particularly effective in workloads with high contention where threads are often waiting due to conflicts. Scaling frequency and voltage introduces a delay that increases the backoff duration. This delay produces two different results depending on the amount of conflicts:

- in high contention workloads the resulting longer backoff time can avoid "a premature re-execution which was doomed to fail" resulting in a reduced number of aborts and consequently increased performance;

- in low contention workloads the increased backoff time introduces an unnecessary delay that increases the application execution time.

The presented strategy has the weakness of only being effective when used on systems that allow different cores of the same multicore CPU to run at a different voltage and frequency at the same time.

Green-CM is an interesting contention manager designed to optimize both performance and energy efficiency [2]. It is based on three key techniques:

1. hybrid back-off primitive that alternates between busy waiting and timer interrupts;
2. an *Asymmetric Contention Manager* policy that promotes "the exploitation of DVFS capabilities via the usage of asymmetric back-off policies";
3. lightweight reinforcement learning techniques used to dynamically adapt the parameters that define in different scenarios which backoff primitive should be preferred or what degree of asymmetry is more appropriate.

This solution does not use software based DVFS controls since they rely on system calls that introduce a relevant cost at the granularity of contention managers. Differently, it attempts to approximate this manual frequency and voltage control "by using a lightweight design that aims at favoring the spontaneous activation of hardware-controlled DVFS mechanisms". This approach can also promote the exploitation of hardware DVFS technologies like *Turbo Boost* which does not provide any software control. The *Asymmetric Contention Manager* defines two different set of threads characterized by different back-off policies:

- aggressive policy with linearly increasing back-off duration;
- conservative policy with exponentially increasing back-off duration.

In medium to high contention scenarios, threads managed with the aggressive policy are less likely to block and more likely to be executed by a core spontaneously boosted to higher frequencies. On the contrary threads, threads managed with the conservative policy are often blocked and are executed on cores pushed towards lower operating frequencies. The optimal number of threads in each set is highly system and workload dependent and is dynamically estimated using reinforce learning techniques.

The work in [1] applies the concept of low frequency busy waiting to the orthogonal problem of thread scheduling. As anticipated at the start of this chapter, thread scheduling solutions determine the number of active threads that provides the best performance for the current execution and block all the threads in surplus. As usual for STM systems, we assume that the maximum number of threads in a transactional application is equal to the number of parallel cores in the system and each thread is associated with a predefined core. Predetermined time blocking primitives are not feasible because it is unknown a priori how long a thread should be blocked: the contention profile of an application can change over its execution which requires the dynamic tuning of the number of active threads to not lose performance and waste energy. The most common approaches used by thread scheduling solutions are semaphores and busy waiting which present a trade-off between performance and energy consumption. On the one hand, busy waiting does not introduce any overhead and is highly energy consuming. On the other hand, semaphores introduce overheads but have reduced energy consumption as operating systems DVFS governors can spontaneously reduce the power state of the blocked core. Low frequency busy waiting combines both the benefits of this approaches without sharing their drawbacks. Similarly to [63], frequency and voltage are reduced whenever a thread enters the blocking phase and put back to the previous values when unblocked by the thread scheduler. However, differently from the previously discussed contention manager strategy, the delay introduced by frequency and performance has an negligible relevance in the overall system performance as thread scheduling is a more coarse-grained approach. Experimental results display relevant energy savings compared to both standard busy waiting and semaphores.

3 Static analysis of STM and DVFS

In this section we investigate the performance and energy consumption of transactional applications running with a variable number of threads and frequencies on two different current generation multicore CPU's. This initial study is essential for understanding the relationship between energy and performance for this new class of applications and is used as the foundation for the definition of optimization heuristics in the following chapter. The STAMP benchmark suite is used to simulate applications with a diverse transactional profile.

The STM system utilized for this evalution is a slightly modified version of the open source *TinySTM* framework. *TinySTM* is a word-based STM implementantion using encounter time locking for write operations and timestamp-based read validation [53]. Updates are buffered and flushed to main memory at commit time. The contention manager policy used for the tests is referred as *delay*: it aborts the transaction that detects a conflict but only restarts it after the contended lock that caused the abort has been released. This approach can increase the probability that transactions can succeed with no interruption upon retry. We introduced in the framework the logic required for monitoring the energy consumption and power consumption of executed transactional applications. This data is obtained by the *powercap* framework and distinguishes between multiple power zones defined by the RAPL interface for the respective physical system.

The experimental study is performed on two different systems belonging to different classes:

- Desktop class system: Intel i5 6600 with 4 physical cores and no HyperThreading, 16 GB of DDR4 memory, Ubuntu Server 16.04 with kernel release 4.4.0-53-generic ;
- Server class system: NUMA architecture with dual Intel Xeon E5 2630v4 with 10 physical cores and 20 virtual cores each, 256 GB of REG ECC DDR4 memory,

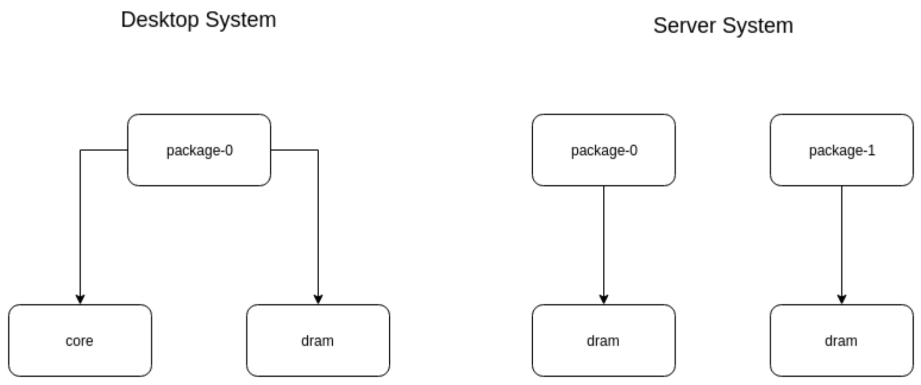


Figure 6: RAPL defined Power Zones for the Systems

Debian 8 Jessie with kernel release 4.7.0-0.bpo.1-amd64.

Figure 6 shows the hierarchical structure of the different power zones for both systems as defined by the RAPL interface. The desktop system provides a fine-grained real-time energy monitoring distinguishing between core and DRAM within the single package. Differently, the server system contains two distinct packages with only the respective DRAM as sub-zones.

Since SandyBridge Intel CPU family, Intel systems on Linux use by default the *intel_pstate* driver for DVFS which provides only two governors: *powersave* and *performance*. This driver can improve the energy efficiency compared to the more general *acpifreq_drivers* but unfortunately does not provide an userspace governor that allows users to manually set the CPU P-state. Consequently, we disabled the "intel_pstate" driver at kernel start-up for all the experimental evaluations. The desktop system provides 16 different p-states ranging from 3.301 GHz to 0.8 GHz while the server system is limited to 12 p-states ranging from 2.201 GHz to 1.2 GHz. Table 1 and 2 show the related frequencies for each P-state respectively for the desktop system and the server system. When running in state P0 (3.301 GHz and 2.201 GHz respectively), the *Intel Turbo Boost* technology is enabled which allows the CPU to run at even higher frequencies depending on the dynamic thermal and power limitations. It is controlled by hardware and there is no software indication of the actual boosted

P-state	Frequency(MHz)
P0	3301
P1	3300
P2	3100
P3	2900
P4	2800
P5	2600
P6	2400
P7	2200
P8	2000
P9	1900
P10	1700
P11	1500
P12	1300
P13	1200
P14	1000
P15	800

Table 1: Desktop system P-states

P-state	Frequency(MHz)
P0	2201
P1	2200
P2	2100
P3	2000
P4	1900
P5	1800
P6	1700
P7	1600
P8	1500
P9	1400
P10	1300
P11	1200

Table 2: Server system P-states

frequency used at any given time during the execution. We don't consider the results with Hyper-Threading as it is rarely used in High Performance Computing. In the first portion of this section we mostly focus on the desktop system as its results are influenced by a reduced number of variables and offers a broader power dynamic range. In the last portion of this section we consider the elements introduced by the NUMA architecture and compare the remarkably different results obtained by the two systems.

3.1 Power consumption of different workloads

An interesting initial study is the analysis of the average power consumption of the different power zones when running applications with different characteristics. All the considered applications are executed on the desktop system with four threads and the CPU cores set to the state P1. This performance state is the fastest that is not affected by the non-determinism introduced by Turbo Boost. We consider different classes of workloads:

- *bayes*, *genome*, *intruder*, *ssca2*, *vacation*: transactional applications from the STAMP benchmark suite;

- *prime*, *oltp*: parallel applications from the the *Sysbench* benchmark suite. *Prime* simulates an highly CPU bound workload that requires very limited synchronization between threads. *Oltp* reproduces an online transaction processing system that executes a fixed number of read operations on a predefined MySQL database;

- *atomic*: simple applications where multiple threads attempt to increase the value of a shared memory variable using the Read-Modify-Write operation fetch-and-add;

- *idle*: idle power consumption of CPU and memory in C-state C2 for the package and C7 for the cores. It is not a actual workload but it is used as a reference to

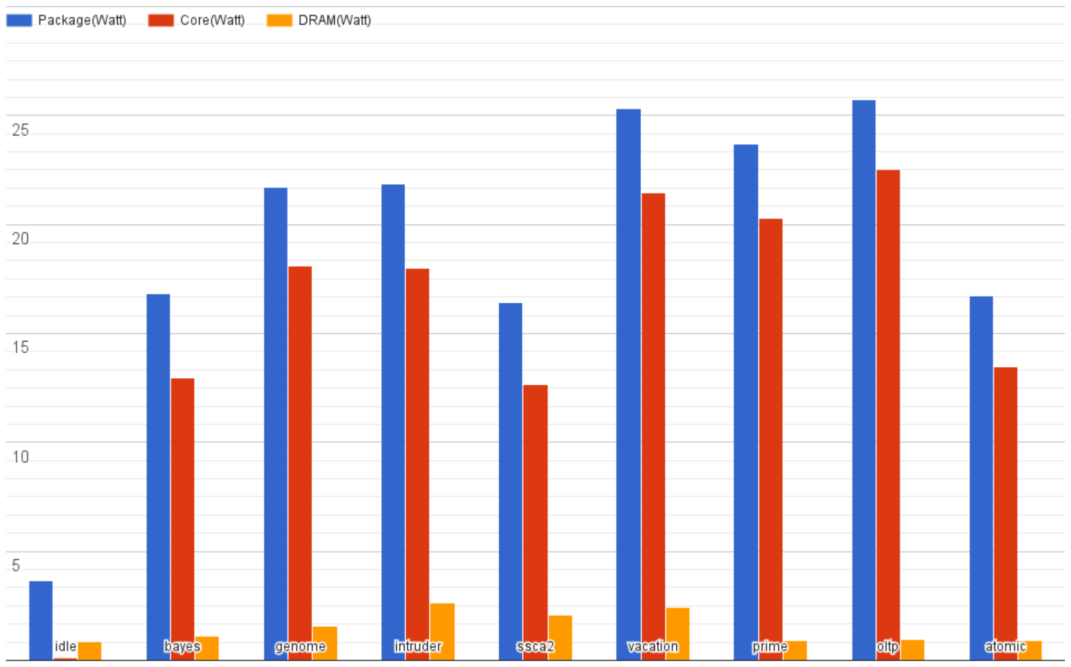


Figure 7: Package, Core and DRAM Power Consumption expressed in Watt

understand the components of the power consumption during the deepest idle state.

As previously mentioned, the STAMP applications run on top of our modified version of *TinySTM* which provides the power consumption in output at the end of the execution. For this class of applications, we performed ten runs and computed the average. For the evaluation of *prime*, *oltp* and *atomic* we manually read the power consumption during peak load in multiple runs and computed the average. The *Atomic* test doesn't simulate the workload of a realistic application as the application makes very limited processing: the four parallel threads continuously attempt to write on the same memory address and most of the time fail. However it is an interesting analysis to understand the effect of hardware contention and cache coherence protocols on the power consumption of both CPU and memory. Figure 7 summarizes the results. The difference in the package power consumption when considering different workloads is significant. The core predictably consumes the highest portion of package power during the execution with values between 77.22%

(*bayes*) and 87.53% (*oltp*) while consumes close to no energy in idle (0.12 W). The peak percentage value of *oltp* can be attributed to the low memory power consumption which is only marginally higher than on idle (+11.49%). *Intruder* has the highest memory power consumption at 2.65 Watt which is 3.04 times higher than for the non-operational state. *Vacation* has a varied workload which involves both main memory operations and different forms of computations which results in the highest power consumption for both the overall package (25.30 W) and the core (21.44 W) among the transactional applications. *Oltp* shows the overall higher package and core power consumption with 25.75 Watt and 22.54 Watt respectively. Other transactional applications show a reduced power consumption which can be attributed to different reasons. *Bayes* and *ssca2* exhibit a sequential initial phase which takes the majority of the execution time and show power values close to the respective single thread execution. *Genome* and *intruder* fully utilize all the CPU cores but multiple factors like an increased abort rate, which results in a brief waiting phase, or less power demanding operations such as memory operations could reduce the overall average power consumption for this applications. Hardware level contention, as shown by the results of *atomic*, reduces the power consumption of the CPU core compared to general computations despite the higher frequency of cache coherence protocol messages. A realistic hypothesis that justifies this behavior could be that for most of the time the cores must wait the response of cache coherence messages thus reducing the dynamic power resulted from the switching of the CPU logical gates. The power consumption of the uncore, computed as the difference between the package power consumption and the core power consumption, presents very limited variation across all the workloads and even the idle state with values ranging from 3.26 Watt (*atomic*) to 3.85 Watt (*intruder*).

3.2 Performance and power consumption at different P-states

In this subsection we investigate the relationship between power consumption and application runtime at different P-states with a fixed number of threads. The Intel i5-6600 CPU doesn't support configurations with different cores at different P-states. It is possible to set different frequencies from software but all the cores will effectively run at the frequency and voltage defined by the core with the lowest P-state. Heterogeneous combinations of performance states will be considered in a following subsection using the server system. We don't consider the results of *bayes* because its runs do not present a fixed number of commits which introduces a unacceptable variance to the results.

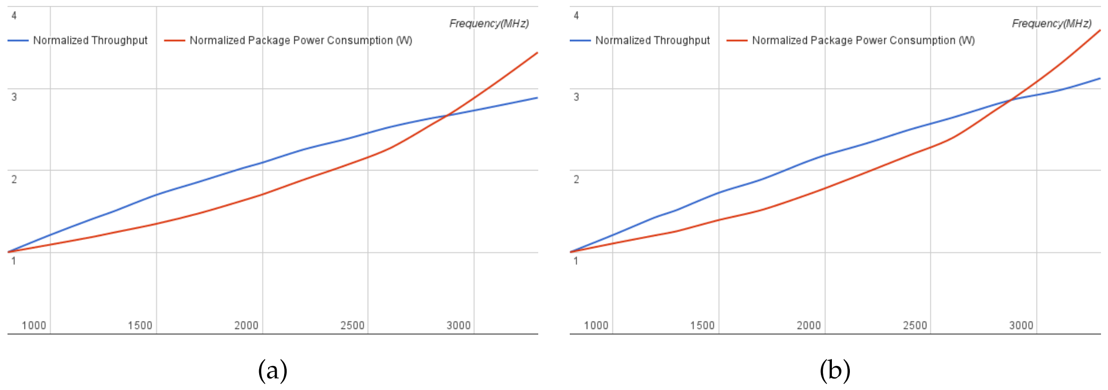
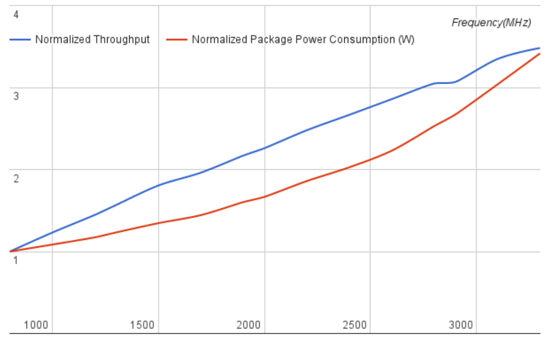
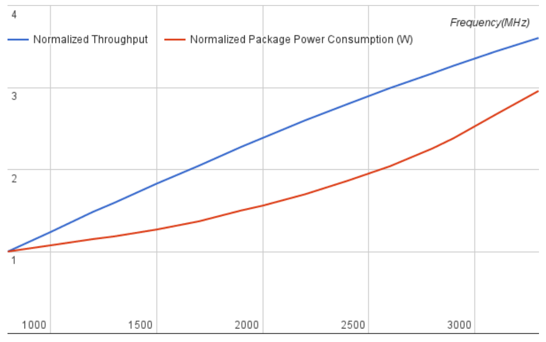


Figure 8: Normalized throughput and package power consumption for intruder (a) and vacation (b)

Figure 8 displays the scaling of throughput and package power consumption at increased frequencies for intruder (a) and vacation (b) normalized to the respective results at 800 MHz. Both the benchmarks show very similar characteristics with a mostly linear increase in throughput and an exponential increase in the package power consumption. As already mentioned, an higher P-state results in both an higher CPU frequency and an higher CPU voltage thus increasing the power consumption cubically. At the same time, an higher voltage also increases the amount of passive currents. The normalized package power consumption exceeds the normal-



(a)



(b)

Figure 9: Normalized throughput and package power consumption for genome (a) and ssca (b)

ized runtime at around 2900 MHz. The results on the performance state P0 are not represented in the graphs due to their high variability. They show on average a very relevant increase in power consumption with only a limited execution speedup. The slope of the throughput function, which is mostly linear until roughly 2600 MHz, shows a slight decrease which results in a scenario of diminishing returns at higher frequencies.

Differently, the results for *genome* (a) and *ssca2* (b), presented in Figure 9, display some distinct characteristics. In both these tests the normalized throughput is always higher then the normalized power consumption in spite of the overall shape of the functions being fairly similar. This is the case due to an higher increase in throughput for *genome* and *ssca2* compared to *intruder* and *vacation*. The initial sequential phase of *ssca2* produces a less steep package power consumption function.

Figure 10 compares the throughput and package power consumption at state P1 for different applications, both transactional and non-transactional. Once again the results are normalized to state P15 and the number of threads is fixed at 4 for all the executions. Transactional applications show a reduced speedup compared to *oltp* and *prime*. *Intruder* has the lowest benefit from an increased frequency with a speedup of 2.88 with a frequency 4.125 times higher. *Ssca2* has the best scaling results for the transactional applications with a speedup of 3.6. *Prime*, whose workload is

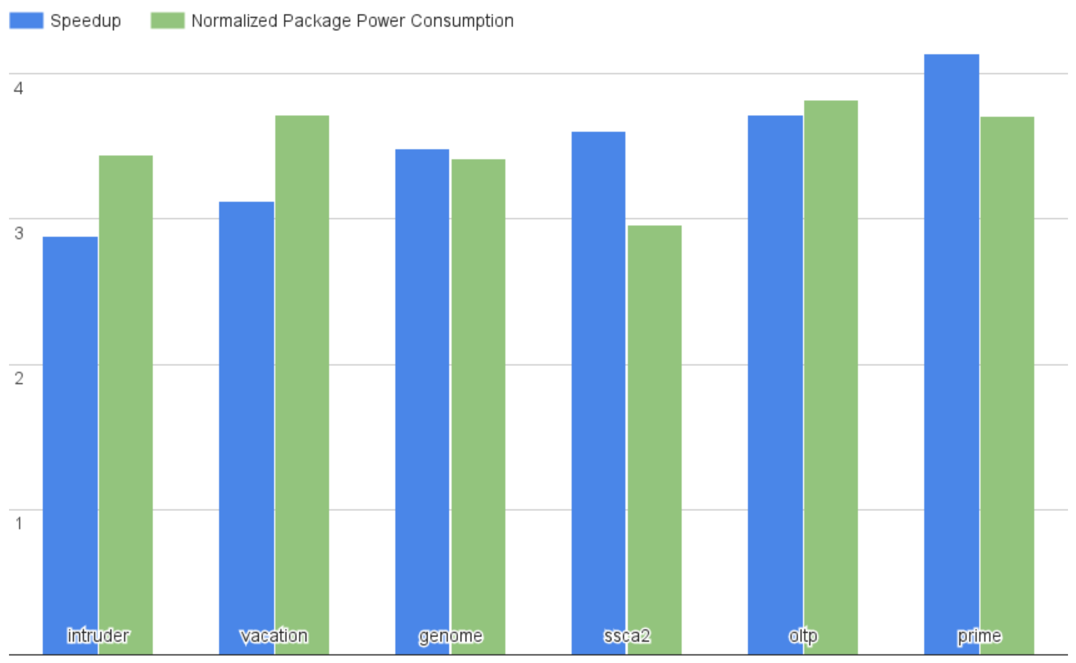


Figure 10: Normalized speedup and package power consumption for different workloads at state P1

entirely constituted by CPU computations, displays exactly the same speedup value (4.12) as the normalized frequency increase.

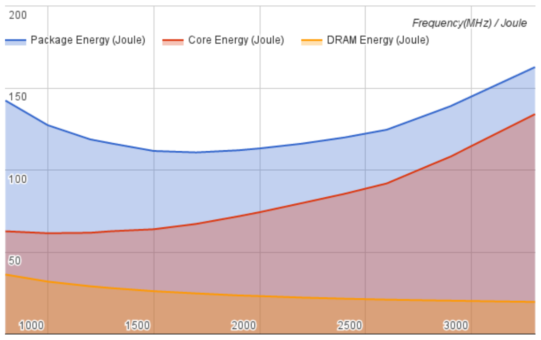
At the same time, an application constituted by different classes of operations has an higher package power consumption compared to computationally-bound applications such as *prime*. The combination of high power consumption and limited speedup for transactional application makes the prospect of energy savings exploiting lower CPU frequency executions particularly attractive for transactional applications.

The motivations that can lead to a sub-optimal performance scaling could be multiple and not all directly linked to the STM paradigm as *oltp* only shows a 3.71 speedup. The time required for operations on main memory and I/O devices is not increased when the processor runs at an higher frequency which represents a constant component independent of the different performance states. This component is increased by STM systems that must introduce an overhead in the execution to ensure correctness which is mostly constituted by main memory operations.

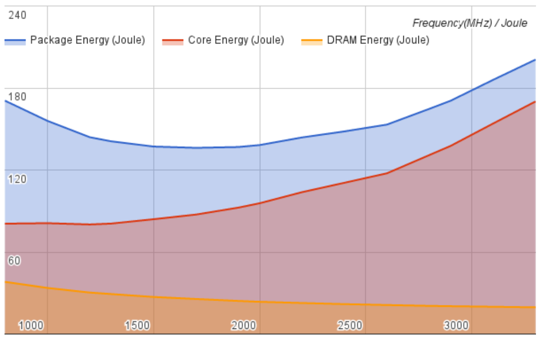
A peculiarity of STM systems is the possibility of aborting transactions. We investigated the relationship between the number of aborts and the CPU frequencies by performing at each P-state 50 runs of these benchmarks from the STAMP suite at a fixed number of threads and computed the average number of aborts. We considered for each execution 4 threads for the Desktop system and the highest number of threads that would not lead to *trashing* for the server system. The results do not show a statistically significant variation as the number of aborts for all the available frequencies can be considered uniform.

3.3 Energy consumption at different P-states

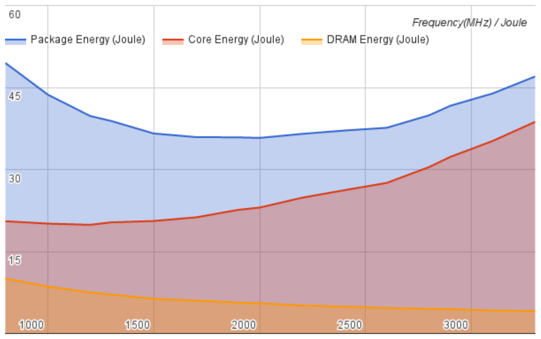
In the previous subsections we considered independently the power consumption and the run-time at different P-states. In this subsection we combine these two metrics and analyze the overall energy cost of executing a transactional applications at different CPU frequencies. The results are presented as amount of energy consumed per application execution but can be easily converted with a simple division to the amount of energy used per committed transaction considering that all the tested applications complete after a fixed number of commits. As displayed in Figure 11, all the tested applications show a convex energy consumption curve with a minimum in the range between 1.7 GHz and 2.2 GHz. These results are a direct consequence of the exponential scaling of the power consumption and the linear scaling of the throughput at increased frequency. The package energy behavior is similar to the one presented by DeVogeleer et al. for mobile devices that defines a *Convexity Rule* for the device energy consumption based on both theoretical and practical evidence [64]. The low throughput scaling at high frequency for *intruder* and *vacation* produces a lower execution energy consumption at the state P15 compared to P1; the minimum energy cost for the execution of both application is at 1.7 GHz. The results are the opposite for *genome* and *ssca2* that shown really high energy consumption at



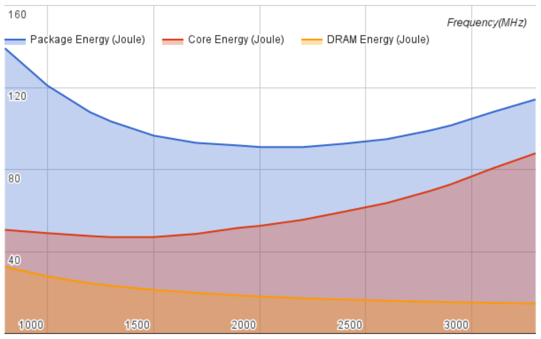
(a) Intruder



(b) Vacation



(c) Genome



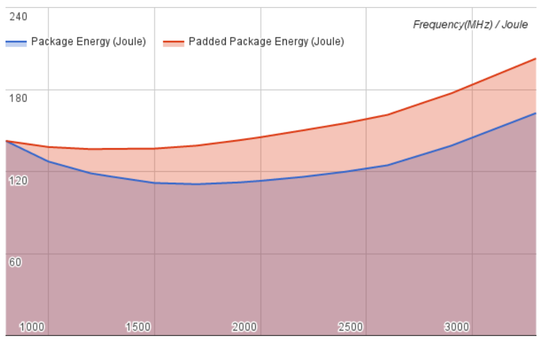
(d) Ssca2

Figure 11: Energy consumption of Package, Core and DRAM at different frequency for execution transactional applications

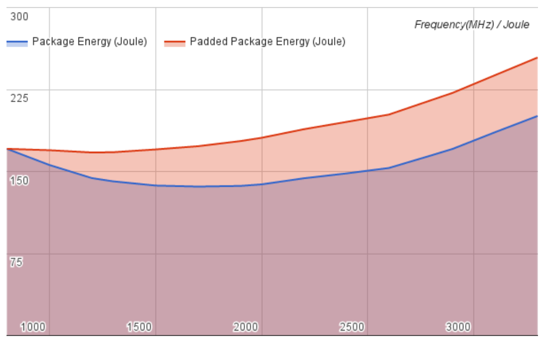
the lowest operational frequency, with the minimum energy cost at 2 GHz and 2.2 GHz respectively.

The core energy consumption is monotonically increasing for all the applications but *ssca2* where it has a very slight decrease at low frequencies. Therefore, if the mostly static energy consumption of the uncore was not considered, the tradeoff of higher run-time at lower frequency would be beneficial for saving energy in the execution of this class of applications. Differently, memory energy consumption is monotonically decreasing as the DRAM power consumption increases slower than the throughput when lowering the CPU P-state.

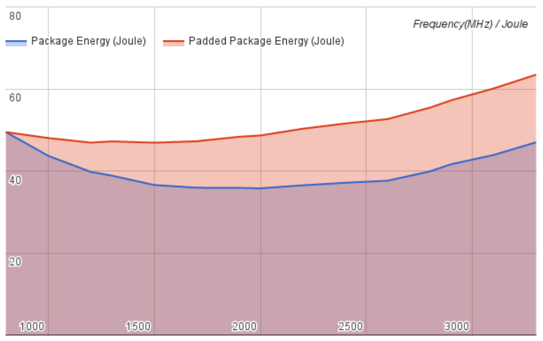
All these results consider the energy consumed by the respective power zones between the start and the end of an application's execution without factoring in the amount of energy spent in idle. An application executed at a faster P-state will complete faster and enter idle state earlier thus resulting in a longer idle time compared to a slower execution that requires more time to complete. Despite its lower value, a system in idle still consumes energy which is particularly relevant in a server scenario where systems are always kept powered on and the average system load is often less than 30% percent. Figure 12 shows the padded package energy consumption



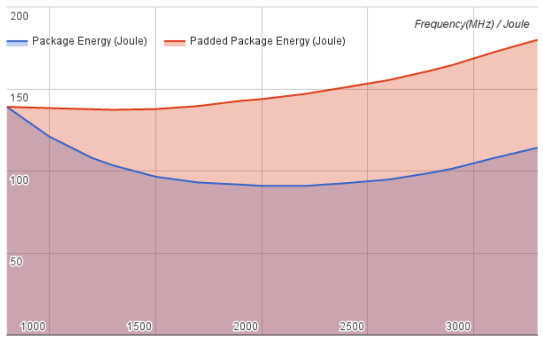
(a) Intruder



(b) Vacation



(c) Genome



(d) Ssca2

Figure 12: Comparison of padded and non-padded package energy consumption

where we added to the previous package power results, for each P-state, the amount of energy consumed during idle in the time interval between the end of the application execution at the considered P-state and the end of the execution at 800 MHz. As expected, the shape of the obtained functions is very similar to the respective core energy consumption function. Interestingly, *genome* and *ssca2* which had the worst results at low frequency have the bigger difference with respect to the non-padded

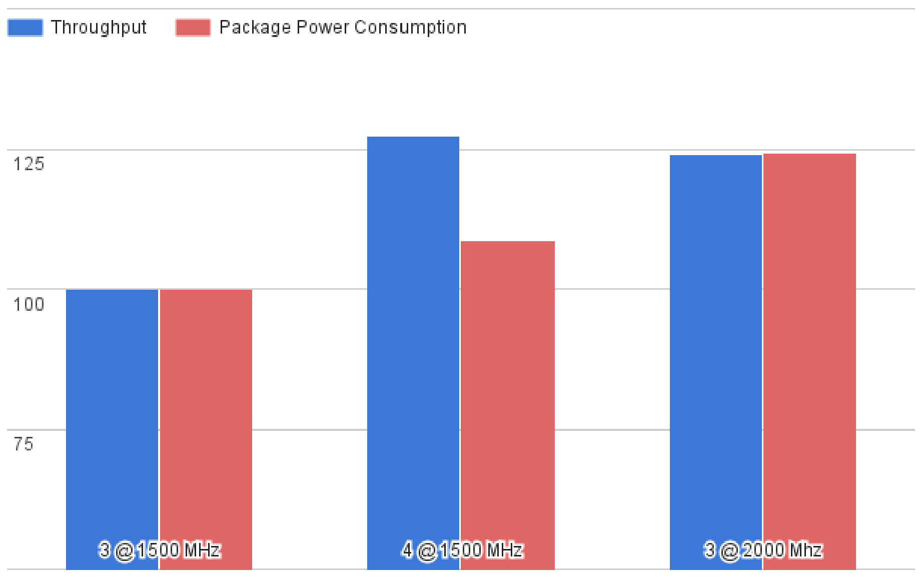


Figure 13: Percentage increase of throughput and package power consumption in *intruder* with a 33.3% increase in the number of threads and frequency with low abort rate

values: an high throughput scaling makes the difference in execution time larger and consequently increases the relevance of idle energy consumption. Differently from the non-padded results, all the tested applications have a energy consumption minimum at 1.3 GHz. Lower frequencies (P14 and P15) are sub-optimal as they introduce a delay in the execution without reducing the amount of energy used by the system.

3.4 Threads versus frequency

In this subsection we compare the performance and energy consumption obtained by increasing either the CPU frequency or the number of threads. Figure 13 shows the results of 3 different executions of the *intruder* benchmark:

- 3 threads at 1500 MHz: used as baseline;
- 4 threads at 1500 MHz: 33.3% increase over baseline in number of threads;
- 3 threads at 2000 MHz with a 33.3% increase over baseline in CPU frequency.

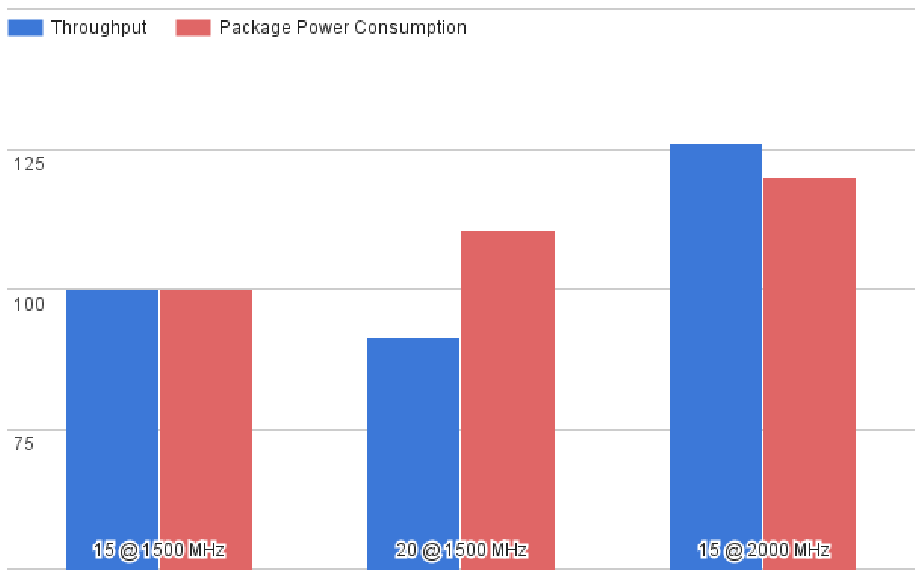


Figure 14: Percentage increase of throughput and package power consumption in *intruder* with a 33.3% increase in the number of threads and frequency with high abort rate

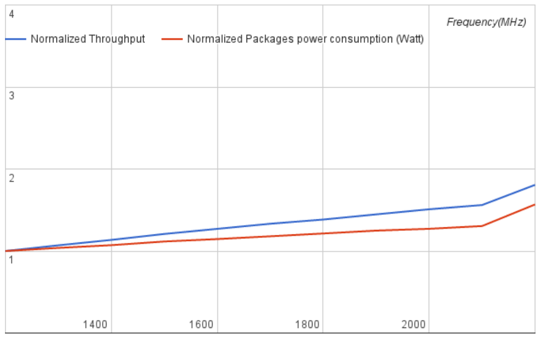
In this situation, increasing the number of threads is preferable over increasing the frequency. Increasing the number of threads from 3 to 4 only increase the power consumption by 8% compared to the 24% increase for the higher frequency while also providing better performance. The significant static power component of modern processors has a uniform cost that is independent of the number of cores actually used.

However, increasing the number of threads on transactional applications also increases the contention on shared data and consequently the abort rate. Figure 14 displays a similar test with the same 33.3% percentage increase of the number of threads and frequency but considering 15 and 20 threads respectively. The test was performed on the the server system in order to allow each thread to run on a separate core. The results are much different with a decreased throughput at 20 threads compared to 15 threads at the same frequency. This dissimilar behavior can be attributed to the *trashing* phenomenon with a much higher abort rate at 20 threads (78,62%) compared to 4 threads (7,83%). Differently, the relative increase in power consumption is equivalent to the previous analysis. In this scenario, increasing the frequency is the only viable choice for increasing the throughput. This is an extreme scenario but we can generally expect a trend of diminishing returns for the throughput at an increased number of threads for applications with an high contention profile.

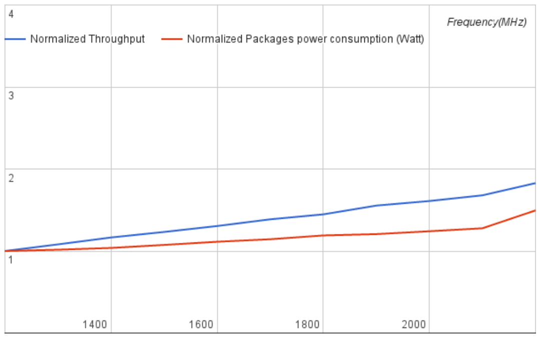
For low contention applications, a many-core system using on average a limited number of cores at high frequency is not using its processing power efficiently. This inefficient scenario is unfortunately often induced by the combination of DVFS governors and schedulers provided by default in modern operating systems that attempt to complete the execution of processes as fast as possible.

3.5 Server system power and energy consumption

In this subsection we briefly analyze the power consumption and energy consumption of the server system and compare it to the previous results obtained on the desktop system. We consider *intruder* and *ssca2* since they have a very distinct transactional profile. For *ssca2* we used 20 threads, one per physical core of the system, while for *intruder* we only used 6 parallel threads due to the high abort rate. As usual, we consider P1 as the fastest P-state. The power and energy results of the individual packages are aggregated summing the respective values. An independent analysis of the consumption of the two packages is presented in the following subsection. For these evaluations, we used the default NUMA policy. The increase in both throughput and power consumption is very reduced compared to the tests performed on the desktop system. The P-states for the server processors only support a limited range of frequencies, from 1.2 GHz to 2.2 GHz, compared to the much wider range on the desktop system from 0.8 GHz to 3.3 GHz. Figure 15 shows the scaling of throughput and packages power consumption normalized to the result at the slowest P-state. In this limited frequency range the exponential increase of the power consumption can be generally considered linear, with a steeper increase starting around 2.0 GHz. Moreover, the normalized increase of the throughput is always



(a) *intruder*



(b) *ssca2*

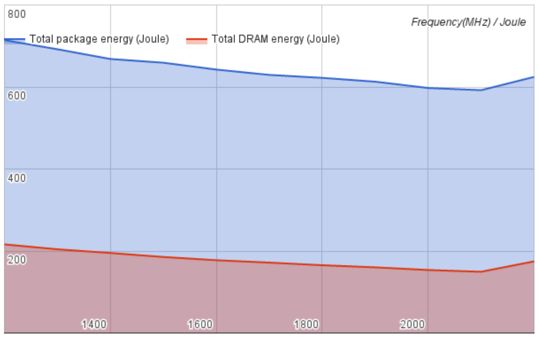
Figure 15: Normalized throughput and packages energy consumption of intruder and ssca2 on server system at different frequencies

higher than the normalized increase of the packages power consumption.

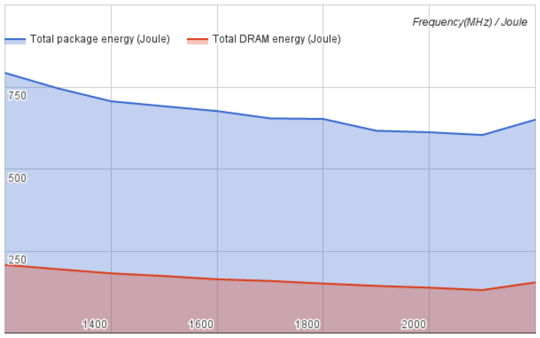
Figure 16 displays the energy consumption at different frequencies. For both the benchmarks, the summed energy consumption of the two packages decreases until its minimum at 2.1 GHz with a relevant increase at 2.2 GHz. This is very different from the convex behavior obtained on the desktop system. Differently, the DRAM energy consumption of the two packages has a slower decrease with a trend inversion at 2.2 GHz. This is the consequence of the much higher DRAM power consumption scaling that reaches 16.75 Watt at state P1 during the execution of *intruder* while only having an idle value of 3,1 Watt. The padded packages energy consumption is mostly linear in the frequency range between 1.2 GHz and 2,1 GHz with once again an higher value at 2.2 GHz.

Intel clearly limited the available frequencies and voltages for the Xeon 2630v4 in order to exclude, directly from hardware, the least energy efficient operational regions. As shown for the desktop results, the range between 1,2 GHz and 2,2 GHz contains all the minimums, even when considering padded energy consumption. This higher average energy efficiency was required to allow each single CPU to contain 10 physical cores with a TDP of only 85 Watt.

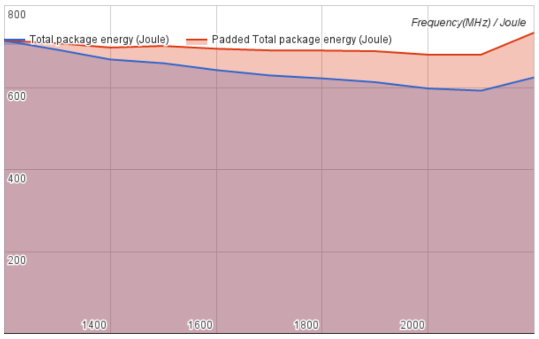
However, we should consider that we excluded the performance state P0 that



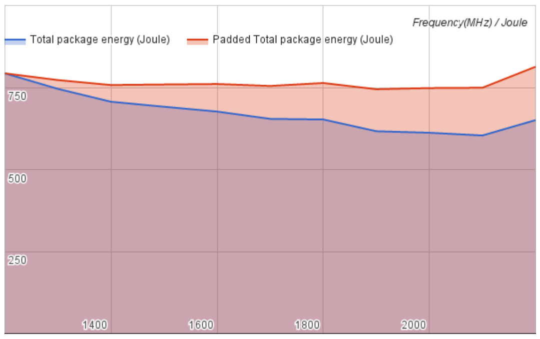
(a) *intruder*



(b) *ssca2*



(c) *intruder* padded comparison



(d) *ssca2* padded comparison

Figure 16: Packages and memory energy consumption on top (a) (b), comparison between padded and non-padded packages energy consumption on bottom (c) (d)

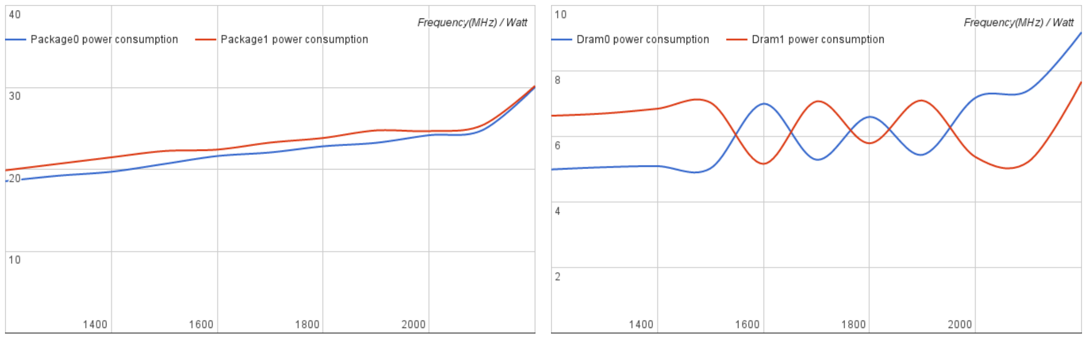
reaches up to 3.1 GHz exploiting the Intel Turbo Boost technology. This high frequency can be reached when only few cores have an high load , which as we said earlier, is a common scenario favored by the operating system. In this boosted frequency region we can expect similar results to the desktop system at the same frequencies with a decrease in the overall energy efficiency.

3.6 NUMA: performance power consumption

In this subsection we study the effects of NUMA on the overall system efficiency using the *intruder* benchmark since it presents high contention even with a limited number of threads. Initially, we consider the results obtained using the default scheduler policy:

- the scheduler attempts to balance the work between the two processors, executing most of the time with 3 threads each;
- first touch memory allocation policy where each thread allocates memory on its own NUMA node.

Most application, included *intruder*, have an initial allocation phase performed on a single core. Consequently, most of the shared memory items will be located on the NUMA node of the core executing this initialization. Figure 17 shows on the left



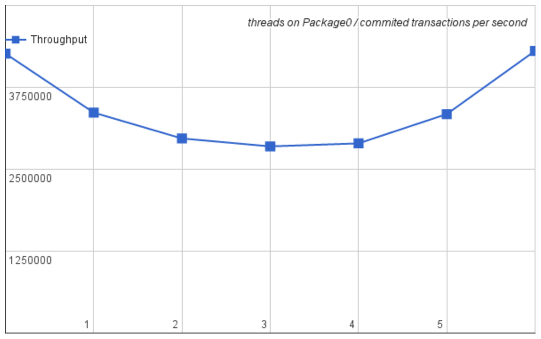
(a) Package0 and Package1 power

(b) DRAM0 and DRAM1 power

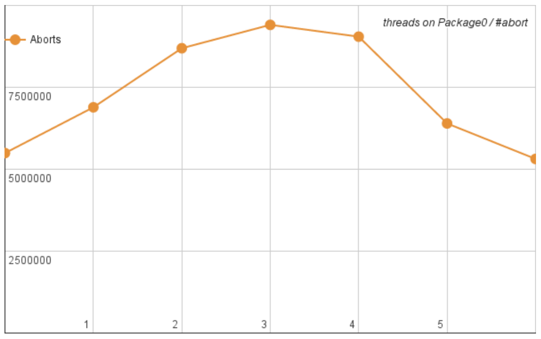
Figure 17: Power Consumption for Package0 and Package1 (a), and DRAM0 and DRAM1 (b) with default NUMA policy

(a) the power consumption of the two packages and on the right (b) the power consumption of the respective memories. The power consumption is close for the two packages with a slightly higher value for package1. Differently, the DRAM power consumption has a significant variation that is directly related to the different location of the shared data which is allocated in local memory by the first thread of the application. Considering a single run in order to avoid the reduced variation induced by the average, the highest power consumption difference is 9.5 Watt and 7.14 Watt for memory and 31.47 Watt and 29.18 Watt for the packages. For both, there is an increased consumption for the node that initially allocates the shared memory. Considering that each package power zone contains the respective memory power zone, and that the amount of the variation of the power consumption between the packages is equal to the differences between the two memory zones, we can expect that the difference in the package power consumption should be mostly attributed to the increased memory power consumption.

An interesting evaluation is understanding the behavior of transactional applications, regarding both performance and energy efficiency, we considering a different balance in the number of threads scheduled on each CPU. Once again, we used the *intruder* benchmark with suggested parameters and 6 threads with both processors in performance state P1. We used the command line utility *numactl* to bind threads to physical cores and executed 10 runs for each combination of thread scheduling between the two CPUs. Figure 18 displays the throughput (a) and the number of



(a) Throughput



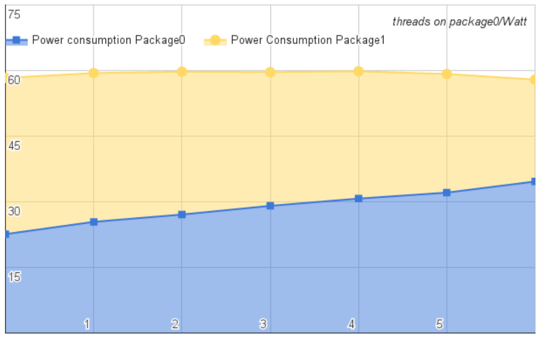
(b) Aborts

Figure 18: Throughput (a) and number of aborts (b) with different number of threads scheduled on package0

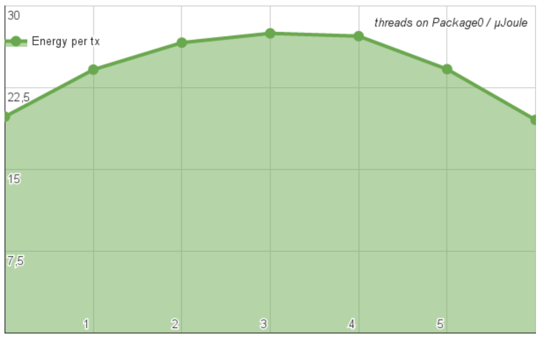
aborts (b) when varying the number of threads bound to package0 from 6 to 0. The throughput is defined by a convex curve that reaches its maxima when all threads run on a single package, either package0 or package1, with respectively 51% and 49% better performance then the configuration with 3 threads each. It should be noted that the slower configuration is also the one used by the default scheduling policy. There are different reasons behind this remarkably different results, some of which are peculiar to transactional applications.

The increased latency for accessing memory located in different NUMA nodes slows down the overall execution, reducing the throughput. This is particularly relevant for transactional applications that introduce an overhead constituted mostly by main memory operations to ensure correctness. At the same time, the number of aborts when running with 3 threads per package is 76% higher then when all threads run on a single package. The different average memory access latency for the two packages creates two different sets of threads that can process transactions at two distinct paces. Considering an high level prospective of a general STM system, and in particular of *TinySTM* that uses commit-time locking, a transaction aborts if any item in its read-set is visibly written by another thread in the time period between its initial read and the transaction commit. Consequently, a transaction that takes longer than average to complete is more likely to abort than a transaction with average length as its vulnerability window, namely the interval during which other threads could write an item in its read-set, is larger. Therefore, we can expect that the transactions executed by the slower set of threads are more likely to abort and consequently the overall number of aborts in the execution is higher. This should be only considered as a speculation: a more in-depth analysis would be required to find a proper explanation for this phenomenon which is unfortunately outside the scope of this thesis. Another positive side effect of the reduced abort rate is the possibility of running applications with an increased number of parallel threads, if still within a single package, without incurring in the *trashing* phenomenon. *Intruder* with default parameters and the default NUMA scheduling policy has the highest throughput at 6 threads while, when running only on a single package, it scales up to 10 threads showing a 74.48% increase in throughput compared to the balanced thread configuration.

Figure 19 shows the packages power consumption (a) and the energy per commited transactions (b) for the same test of *intruder* with a varying number of threads scheduled on package0. The sum of the packages power consumption presents limited variation for all the configurations. However, the executions on a single pack-



(a) Packages power consumption



(b) Energy per committed transaction

Figure 19: Packages power consumption (a) and energy used per committed transaction (b) with different number of threads scheduled on Package0

age have a slightly lower value with 58 Watt compared to 59,5 Watt for the other configurations. Interestingly, even when no threads run on a package its power consumption (23.1 Watt) is much higher than when the overall system is in idle (8.7). As a result of the reduced throughput and slightly higher consumption when running on multiple packages, the energy consumed per commited transaction is 40% higher compared to the execution with all the threads running on package0. Once again the worst result is obtained with the balanced configuration proposed by the default NUMA scheduling policy.

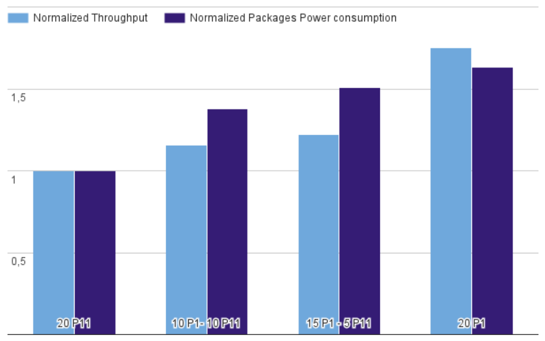
3.7 Heterogeneous P-state configurations

In this subsection we briefly study the results of transactional executions when running with different P-states between different cores and different packages. These tests are performed only on the server system as the desktop system doesn't support heterogeneous P-state configurations with the userspace governor. We executed multiple runs of *vacation* with 20 threads considering 4 different configurations:

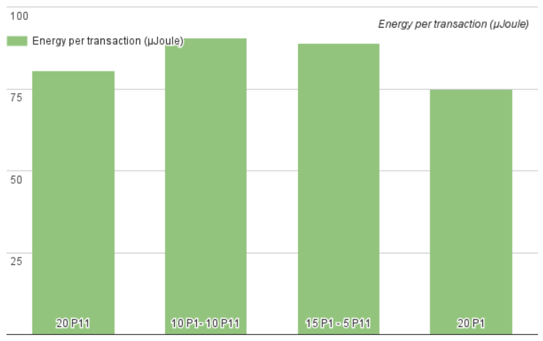
1. all cores in state P11;
2. package0 cores in state P1 and package1 cores in state P11;

3. package0 cores in state P1, 5 cores of package1 in P1 and 5 in P11;

4. all cores in state P1.



(a) Normalized throughput and package power consumption



(b) Energy per committed transaction

Figure 20: Normalized throughput and normalized packages power consumption on left (a) and energy per committed transaction on right(b) considering different heterogeneous P-state configurations

Figure 20a shows the throughput and the sum of the packages power consumption of the different configurations normalized with respect to the results of configuration 1. Unlike configuration 4, configurations 2 and 3 present an higher increase in the power consumption compared to the increase in throughput. Consequently, the energy consumed per commited transaction, as displayed in figure 20b, is higher on this two configurations than on both the configurations using an uniform frequency.

Following the idea speculated on the increased number of aborts when running on multiple packages, we expected an increase in the number of aborts also when executing at different frequencies. However, the experimental results show a constant number of aborts across all configurations on multiple applications.

In conclusion, the execution of transactional applications on processors with an heterogeneous P-state configuration doesn't seem an attractive solution regarding energy efficiency. In addition, even if minor optimization could be achieved exploiting the finer-grained frequency control on different workloads, the number of possible configuration would increase exponentially to the product of the number of total cores in the system with the number of CPU P-states resulting in a domain of possible combinations too wide to be efficient explored at run-time.

4 Architecture for efficient Software Transactional Memory

In this section we present an architecture, built on top of *TinySTM*, that optimizes the performance and energy efficiency of transactional applications using a combination of thread scheduling and DVFS based on the results obtained in Chapter 3. The exponential scaling of the CPU power consumption when running at a faster P-state and the less than proportional increase in throughput compared to the increase in frequency makes the execution of transactional applications at high CPU frequency energy inefficient. At the same time, increasing the number of parallel threads only has a minor impact on the CPU power consumption but provides different results based on the level of contention on shared data. In this scenario, a minor reduction in throughput could allow significant energy savings.

We developed different heuristics that attempt to provide the highest throughput in the execution of transactional applications while running within user defined constraints on either the maximum package power consumption or on the average amount of energy used per committed transaction. These constraints should be considered knobs that can be used to tune the power consumption and energy efficiency based on the performance requirements of applications at different points in time .

All the proposed heuristics are based on a run-time exploration of the bi-dimensional domain defined by the combination of all the CPU P-states and the number of physical cores available in the system.

Figure 21 shows the different components of the proposed architecture. The *Statistics Collector* gathers real-time data from two different sources:

- metrics on the processing of transactions such as throughput and abort rate obtained by wrapping the lifetime functions of transactions defined in *TinySTM*;
- energy related statistics obtained from the *powercap* framework.

The *Statistics Collector* aggregates these data in order to be used by the Heuristic

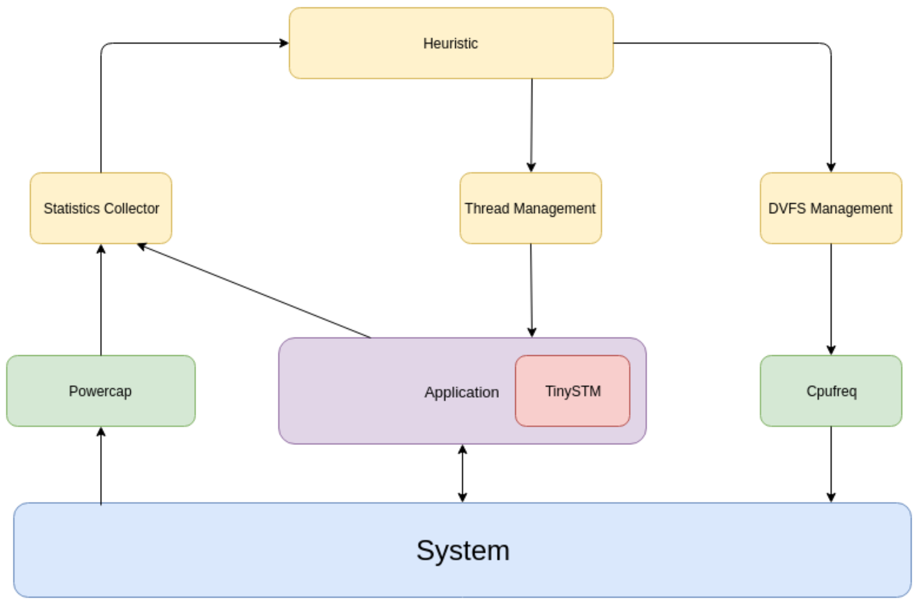


Figure 21: Architecture overview

component to define the next step in the exploration of the domain of possible configurations. The *Thread Management* component is used by the heuristic function to tune the number of active threads used by the transactional application. Differently, the *DVFS Management* component provides an interface for dynamically changing the CPU P-state by exploiting the pseudo-files exposed by the *cpufreq* framework.

The functions and data structures of the architecture are partially implemented within the source files of *TinySTM* and partially in external files referenced by *TinySTM* functions. All the code is written in the C programming language. The architecture is designed with the primary goal of introducing the lowest possible overhead during the execution.

This Chapter is logically divided in three main portions: initially we provide a description of the main components of the architecture; in the second portion we define the exploration problem and propose different optimization heuristics; in the last portion we present the experimental results.

4.1 Thread Management

Thread Management is one of the main components of the architecture that provides an interface for tuning the number of active threads during the execution. It is initialized during the start-up of the *TinySTM* framework in the *stm_init()* function. It is based on two main data structures:

- *running_array*: array of integers of the size of the maximum number of active threads, considered equal to the number of physical cores in the system, that contains in each cell either 1 or 0 that discriminates if the respective thread should be either running or stopped;

- *pthread_ids*: array of pointers to *pthread_t* data structures of the size of the maximum number of active threads that contains the respective *pthread* identifier for each thread. Each cell of the array is written by the respective thread when started.

Each thread is also internally identified by an integer in the range between 0 and $(TotalThreads - 1)$ that is used as an index to access the respective items in the two arrays.

Before starting a new transaction, each thread reads its *running_array* cell: if it contains 1 it can start the new transaction; differently, if it contains 0 the thread calls the system call *pause()* that suspends the process until it receives a signal. Threads are resumed by sending the *SIGUSR1* signal using as address the identifier obtained from the *pthread_ids* data structure. In the initialization of the *Thread Management* module, an empty function is registered as the handler of *SIGUSR1* to prevent the process termination forced by the operating system when the user signals do not have a registered handler. Whenever resumed, threads check in the *running_array* if they should be running: if the respective cell is still 0 they once again call the *pause()* system call. This mechanism prevents the wake up of paused threads when receiving signals different than *SIGUSR1*.

The combination of *pause()* and signals is particularly fitting for an architecture that focuses not only on performance but also energy efficiency. Busy-waiting introduces a lower overhead and has a faster wake-up time but is in energy inefficient for configurations that only use a limited number of parallel threads. At the same time, the overhead introduced by system calls, such as *pause()* and *kill()*, is not particularly relevant in a thread scheduling scenario where the number of threads is tuned by the *Heuristic* function at a much slower pace compared to the execution time of transactions.

4.2 DVFS Management

The DVFS Management module provides an interface for changing the CPU performance state at run-time. Since configurations with heterogeneous P-states exhibit reduced energy efficiency and a much higher exploration complexity, the module only considers configurations with uniform frequencies for all the CPU cores. The *Cpufreq* framework offers different controls for each core expressed as different folders in the *sys* pseudo file system.

At start-up, the module retrieves the number of CPU cores with the *sysconf()* function and sets each core to the *userspace* governor. Successively it reads the CPU available frequencies defined in a pseudo file in each core's folder and allocates an array that associates to each P-state the respective frequency in KHz. The index of the slower P-state is saved in a global variable called *max_pstate*. This mechanism provides an effective abstraction for tuning the CPU performance state. In order to change P-state, the *cpufreq* framework requires the respective CPU frequency in KHz to be written in the *scaling_setspeed* file for each core, which can be read from the array using the P-state number as index. At the same time, the ordering of the performance states in the range between 0 and *max_pstate* allows the definition of algorithms that tune the system performance and power consumption independently of the number of system supported P-states.

As anticipated, changing the CPU performance state requires writing on a different pseudo-file for each CPU core, thus, the performance cost of this operations increases linearly with the number of cores in the system.

4.3 Statistics Collector

The *Statistics Collector* component aggregates at runt-time the information required by the *Heuristic* module to accurately tune both performance and power consumption. Unlike the previously described components, statistics are gathered at the granularity of single transactions. Consequently, one of the main design goals of this component is to introduce the least possible overhead.

The heuristic function defines the next step of the exploration at the end of each round of statistics collection whose duration is defined by a number of commits that is user defined. Each round is also evenly divided in multiple slices, defined by a variable amount of commits, where each thread provides its own statistics in a round-robin fashion. A global variable points to an array of pointers to a data structure, one per thread, that stores the partial results within a round. This data structure is allocated directly by the respective thread to make sure it is located in the nearest NUMA node when executing with the default first-touch policy. Moreover, the size of the L1 data cache is read at start-up in order to allocate each of these data structures aligned to the size of the CPU cache line. The data structure contains the following fields:

- flag that indicates if the thread is in the collector phase;
- number of commits that should be processed in the current statistics round for the thread;
- number of commits in the current round;
- number of aborts in the current round;
- number of transactions started in the current round;

- start and end value package energy consumption obtained respectively at the start and at the end of the collection phase;
- start and end value of time obtained respectively at the start and at the end of the collection phase.

The package energy consumption is obtained by reading the pseudo-files exposed by the *powercap* framework. For the start time and end time we used a monotonic clock defined in *time.h* that provides accurate values independently of the CPU frequency.

Each thread in *TinySTM* is associated with a data structure that is loaded at the start of each function regarding that thread. In order to reduce the number of memory accesses, exploiting cache locality, we added to this structure a direct pointer to the respective statistics data structure and replicated the value of the collector flag. At any point in time only one thread collects statistics. Whenever a thread completes its slice of committed transactions it sets to 0 its own collector flag and sets to 1 the flag of the following thread. When the round completes, thread 0 collects the results obtained from all threads and calls the *Heuristic* module passing as parameter the aggregated statistics of last round: throughput, abort rate, average package power consumption and average amount of energy used per committed transaction.

The round based design of the Statistic Collectors provides two main benefits. Firstly, collecting statistics one thread at a time reduces the collection overhead and makes it increasingly less relevant as the number of cores in the system increases. Secondly, the even division of commits between all the active threads provides fair aggregated results in NUMA systems where threads may have a remarkably different throughput and energy consumption.

4.4 Heuristics module

The *Heuristic* module defines different approaches for tuning the performance and energy consumption of transactional applications at runtime exploiting different exploration policies. The exploration heuristics are based on two different constraints:

average package power consumption or average amount of energy consumed by the package per committed transaction. We can define the following function that characterizes the relevant relations for the optimization problem:

$$f : [S \times T] \mapsto [TP \times P_{package} \times E_{tx}]$$

where S is the CPU P-state, T is the number of active threads, TP is the throughput, $P_{package}$ is the average package power consumption and E_{tx} is the average energy used per committed transactions. The goal of the search heuristics is to find the input configuration that provides the maximum throughput while obtaining an average package power consumption or average amount of energy spent per committed transaction lower than the value defined by the respective constraint.

All the proposed exploration heuristics are based on a extended version of the hill-climbing optimization technique. Each execution starts with an user defined number of starting threads at the slowest P-state which allows to explore the performance states only on the direction of increased frequency. As observed in Chapter 3, the slowest P-state is generally more energy efficient than the fastest, making it a preferable starting point. Generally, the heuristics implement a mono-dimensional hill-climbing technique at each P-state exploiting the results obtained in the previous performance state to reduce the search time. The constant number of aborts at different frequencies suggests that the optimal number of threads at each P-state should be similar. The different constraints introduce some peculiarities that will be presented when defining each exploration heuristic independently.

In this scenario, there could be a relevant distance in the 2D domain space among the points that provide the highest throughput. This is particularly relevant for applications with high contention and relatively low average power or energy limit. Consequently, dynamically adapting the optimal configuration during the execution is considerably more complex than on mono-dimensional searching techniques. Restarting the whole exploration process whenever a change in the execution profile is detected is sub-optimal considering the longer convergence time compared to mono-dimensional explorations. Consequently, in this initial work we consider the execution profile at the time of the exploration to be the static execution profile of the application. The idea of periodically testing the better performing configurations, even if distant in the 2D space, in order to swiftly adapt to changing execution profiles without requiring a long exploration phase could be an interesting idea that we might explore in future works.

The *Heuristic* module can be configured with a specific file that defines the following parameters:

- exploration policy defined by an integer from 0 to 4;
- number of threads used in the first step of the exploration;
- total number of commits per round;
- maximum average power consumption for the package;
- maximum average energy consumed by the package per committed transactions;
- parameter used by heuristic 2 to express the expected proximity of the package power consumption of the optimal configuration to the user defined power limit.

In NUMA architectures we consider the package power consumption and package energy as the sum of the respective results for each package in the system. The parameter that defines the total number of commits per round should be tuned to the specific application based on its average transaction length. A too low value could result in an inaccurate convergence to a sub-optimal configuration while a too high value would increase the duration of the exploration phase and consequently reduce the portion of time spent in the optimal configuration. The number of commits per round is fixed for each configuration, thus, rounds last longer for slow configuration.

This is unfortunately necessary to provide real-time statistics with the same level of accuracy across all configurations.

We refer to our exploration policies as heuristics because we cannot formally prove that they converge to the optimal configuration at each execution. They are based on the results of experimental evaluations, which are only performed on a limited number of system and workloads. Modern system are incredibly complex: we cannot formally prove that for each execution of every transactional application, increasing the CPU frequency with a fixed number of active threads results in an increased throughput; we can only consider it an intuition that can be used to reduce the convergence time compared to an exploration policy that simply tests all configurations.

4.5 Exploration policies

We defined 4 distinct exploration heuristics based on different sets of assumptions on performance,power and energy consumption of the system with different configurations of active threads and P-states. As previously explained, these assumptions are required to reduce the number of exploration steps. The basic policy of testing all configurations, feasible in a thread scheduling approach that only optimizes the number of active threads, is not efficient in this bi-dimensional scenario where the number of steps would be equal to the product of the total number of cores in the system with the number of available P-states. This number is equal to 64 steps for the desktop system and 240 steps for the server system.

The exploration heuristics are divided in two groups defined by two diverse constraints:

- *heuristic 0*, *heuristic 1* and *heuristic 2* are constrained by a limit on the package power consumption;
- *heuristic 3* is constrained by a limit on the amount of energy used per committed transaction.

All the heuristics try to converge to the configuration that provides the best performance without overcoming the defined constraints. The different nature of the constraints allows capturing different requirements.

Limiting the package power consumption reduces the amount of heat produced by the package which can be particularly useful in data centers where heat dissipation is a major concern both technically and economically. It can also provide an increased control over the peak power consumption of data centers allowing a more aggressive oversubscription of the power available in the facility. It is a common technique used in data centers exploiting the consideration that is very unlikely that all the single servers are at full load at the same time [44].

Differently, the constraint on the energy used per committed transactions can be exploited to optimize the overall energy efficiency of the execution. In many situations, the configuration that provides the lowest energy cost per committed transaction doesn't provide the highest throughput. Therefore this constraint presents a trade-off between energy efficiency and performance.

Heuristic 1 and *heuristic 2* use the output of a specifically developed profiler, executed offline, to reduce the number of steps required to converge to the optimal configuration. It is based on the parallel computation of prime numbers in a fixed range which shows similar power consumption results to the *vacation* and *intruder* benchmark. The profiler performs the computation for each combination of active parallel threads and available P-states and saves the respective average package power consumption in a file. As shown in Chapter 3, the package power consumption has a significant variation across different workloads. Therefore, the values obtained by the profiler are only used as a baseline and are not considered completely accurate by the heuristics. Moreover, the heuristics only use the proportion between the values provided by the profiler to interpolate the results of unexplored configurations starting from the configurations already explored.

Unfortunately, the definition of a general offline profiler that approximates the average energy used per committed transaction at different states is much more complex than for the power consumption as the amount of energy used is directly related to the throughput which is specific to each application.

For all the defined heuristics the exploration is divided in multiple phases where each phase only considers the configurations with a fixed CPU P-state. The CPU frequency and voltage increases monotonically at each phase of the exploration, starting from the slowest P-state.

Even when the assumptions are true for the given workload and system, we cannot prove that the exploration heuristics always converge to the optimal configuration. Firstly, the information obtained by the *Statistic Collector* are only statistical samples which are influenced by variance. Secondly, most applications do not have a perfectly static execution profile, which could change during the exploration, and consequently affect the run-time evaluation of the optimal configuration.

4.5.1 *Heuristic 0*

The goal of this heuristic is to find the configuration with the highest throughput that has an average package power consumption lower than an user defined limit. In this heuristic, whenever the exploration moves to the subsequent phase, the exploration starts with the number of active threads equal to the amount used by the configuration that provided the best result at the previous P-state, i.e. the configuration that displayed the highest throughput while operating within the power limit. The heuristic relies on the following assumptions in order to reduce the number of steps required to converge:

1. the function $T_P = f(t, s)$ where T_P is the throughput, t is the number of active threads and s is the CPU P-state, has a single global maximum and no local minima for each fixed value of s;

2. increasing the number of parallel threads with a fixed CPU P-state increases the package power consumption;

3. increasing the CPU voltage and frequency (lower P-state) with a fixed number of parallel threads increases the package power consumption.

The first assumption is necessary for all the exploration heuristics as it allows to only explore following the positive trend of the throughput when varying the number of active threads at a fixed P-state. Assumption 2 allows to not explore configurations with an higher number of active threads if the previous configuration has a package power consumption higher than the limit. Similarly, assumption 3 makes possible to avoid configurations with a faster CPU P-state when the already explored configuration with a slower P-state and the same number of active threads has already an higher power consumption than the limit. The combination of the second and the third assumption provide the possibility of stopping the exploration before reaching the lowest P-state whenever a configuration with only 1 active thread shows a too high power consumption. The first assumptions seems to be always true for all the applications in the STAMP benchmark suite [58]. The second and third assumptions, while we cannot consider proven, are definitely reasonable considering the static and dynamic components of the CPU power consumption, and resulted true for all the tested workloads and systems.

4.5.2 *Heuristic 1*

Similarly to *heuristic 0*, *heuristic 1* searches for the configuration that provides the highest throughput while using less than the limit power consumption. Unlike the previous heuristic, it exploits the data provided by the offline execution of the power consumption profiler. The heuristic relies on the following assumptions to reduce the number of exploration steps:

1. the function $T_P = f(t, s)$ where T_P is the throughput, t is the number of active threads and s is the CPU P-state, has a single global maximum and no local minima for each fixed value of s;

2. increasing the number of parallel threads with a fixed CPU P-state increases the package power consumption;

3. increasing the CPU voltage and frequency (lower P-state) with a fixed number of parallel threads increases the package power consumption.

4. the proportion between the package power consumption of any two configurations provided by the profiler is similar to the proportion of the package power consumption of the same two configurations in the actual application execution;

The first three assumptions are the same of *heuristic 0* and can provide similar optimization for the exploration process. The forth assumption provides a qualitative guarantee on the values obtained by the profiler. We cannot define a quantitative definition of the similarity between the profiler and the actual execution as it would require an extensive experimental analysis which is outside the scope of this work. We simply want to investigate if using a power profiler could provide some benefits. Exploiting the proportions defined by the profiler, the exploration follows the increase in power consumption when changing P-state. Unlike the previous heuristic that when entering a new phase directly used the number of threads that resulted optimal in the previous phase, *heuristic 1* always attempts to first explore configurations that are within the power limit, possibly avoiding those that are likely to be invalid. The profiler data is only used to define the first configuration in each phase; the exploration of the optimal number of active threads at each P-state follows the same policy of *heuristic 0*. Therefore, a mismatch of the values provided by the power profiler with the effective power consumption during the execution can only result in an increased number of exploration steps; the configuration chosen as optimal would still be the same that would have been selected by *heuristic 0*. The benefits of this different exploration policy are more tangible when the difference in power consumption between consecutive P-states is higher. An interesting benefit of heuristic 1 is that by following the increase in the power consumption, it can reduce the average package power consumption during the overall exploration phase compared to the previous heuristic.

4.5.3 *Heuristic 2*

Heuristic 2 has the same goal of the previous exploration heuristic but uses the data provided by the power profiler to potentially skip the exploration of configurations that are expected to show a package power consumption much lower than the limit. It is based on the following assumptions:

1. the function $T_P = f(t, s)$ where T_P is the throughput, t is the number of active threads and s is the CPU P-state, has a single global maximum and no local minima for each fixed value of s;

2. increasing the number of parallel threads with a fixed CPU P-state increases the package power consumption;

3. increasing the CPU voltage and frequency (lower P-state) with a fixed number of parallel threads increases the package power consumption.

4. the proportion between the package power consumption of any two configurations provided by the profiler is similar to the proportion of the package power consumption of the same two configurations in the actual application execution;

5. the optimal configuration has a package power consumption within an user defined percentage from the power limit;

The first four assumptions are the same used by *heuristic 1*. The exploration starts at the slowest P-state where the heuristic seeks for the number of active threads that provides the highest throughput. Successively, it uses the data provided by the power profiler to predict which should be the configuration with the same number of threads and the highest P-state such that it has a package power consumption in the range defined by $[limit_{power} * (1 - percentage), limit_{power}]$ where limit is the package power consumption constraint, and percentage defines the size of the range, starting from the limit, that the user expects to be worth exploring in order to find the optimal configuration. As an example, if the parameter is set to 15%, the exploration heuristic will skip all the configurations that are expected to have a package power consumption lower than 85% of the package power consumption limit. A lower value of this parameter can reduce the exploration time. However, a too low value could exclude configurations that could be optimal, particularly in applications with a very high shared data contention. This heuristic is particularly effective when the power limit is set to a relatively high value compared to the consumption at the slower P-state.

4.5.4 *Heuristic 3*

Heuristic 3 has the goal of finding the configuration that provides the highest throughput while consuming per committed transaction, on average, less energy than the user defined limit. It only relies on the following basic assumption shared by all heuristics:

1. the function $T_P = f(t, s)$ where T_P is the throughput, t is the number of active threads and s is the CPU P-state, has a single global maximum and no local minima for each fixed value of s;

Similarly to *heuristic 0*, in each phase, excluded the first, the exploration starts with the same number of active threads as those used by the optimal configuration of the previous phase. We only consider one basic assumption as the amount of consumed energy is not only related to the package power consumption but also to the throughput which is specific to each application. As shown in Chapter 3, configurations with only few active threads are often inefficient. At the same time, an excessive amount of parallel threads in high contention workloads could lead to thrashing. Therefore, this heuristic generally requires an higher number of steps to find the optimal configuration than those based on a power constraint as it is always required, at each phase, to explore configurations with both an higher and lower amount of active threads. Since the heuristic doesn't rely on any assumption on the behavior of the energy spent per committed transaction in different configurations it is not possible to stop the exploration before reaching the highest P-state.

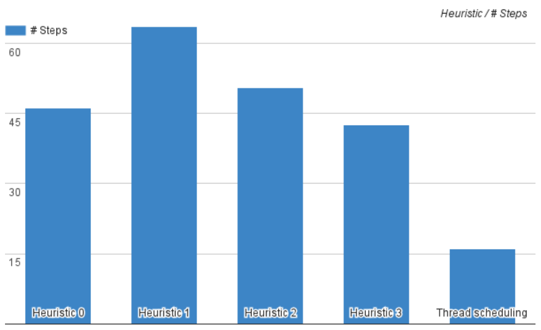
4.6 Comparison of the exploration heuristics

In this subsection we compare the number of steps required by the different heuristics for converging to the optimal configuration. We analyze the results on the server system as it has a wider domain of possible configurations (240). We consider 4 distinct workloads:

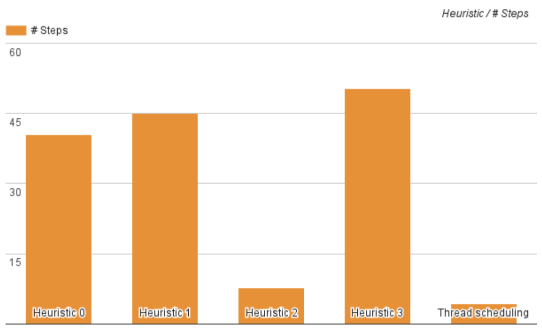
- *vacation* benchmark with 80 Watt power limit, limit of 60 μJoule per committed transaction and 10000 commits per round;
- *intruder* benchmark with 80 Watt power limit, 20 μJoule energy limit per transaction and 2000 commits per round;
- *intruder* benchmark with the same parameters as the previous but executed with all threads bound to package 0 using the *numactl* command line tool;
- *yada* benchmark with 80 Watt power limit, 75 μJoule energy per transaction limit and 10000 commits per round.

All the executions start with 5 parallel threads and the *percentage* parameter set to 15%. For comparison, we also show the number of steps required to converge for a standard heuristic-based thread scheduling approach that searches for the number of active threads that provides the highest throughput only considering configurations with the highest P-state for all the CPU cores. The package power limit is set for all applications to 80 Watt which is lower than the maximum observed value of 106 Watt. The energy per committed transaction constraint is clearly different among the tested applications due to the different transaction length and contention level. Similarly, the number of commits that defines each round is tuned to each specific application. A too low value could result in an inaccurate sampling where random variations in the workload could lead to wrong decisions in the exploration phase. At the same time, most applications do not show a static execution profile which is usually influenced by the number of transactions committed since the start of the execution. Therefore, a too high number of commits per round could lead to situations where the execution profile changes significantly during the exploration phase which could result in a sub-optimal chosen configuration. *Vacation* and *yada* only show a limited execution profile variation that allows a relatively high round size. Differently, *intruder* has a significant increase in the abort rate during the execution which made us select a lower number of commits per round compared to the other applications. All the presented results are computed as the average of 20 runs.

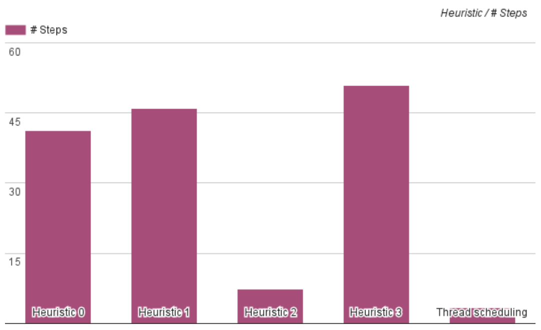
Figure 22 shows the number of steps required on average by each heuristic to converge to the chosen configuration. We recall that *heuristic 0*, *heuristic 1* and *heuristic 2* are based on a package power consumption limit while heuristic 3 is constrained by a limit on the energy used by the package per committed transaction. Interestingly, despite the significantly higher throughput and energy efficiency when running with all the threads bound to package 0, the exploration heuristics based on the power constraint converge to the same configurations for both the tests of *intruders*. The limit of 80 Watt doesn't allow an increase in the number of active threads which could be possibly beneficial to the throughput. *Heuristic* 0 and *heuristic* 1 always converge to the same configuration in each application. Differently *heuristic 2*, which can skip exploration phases depending on the value of the *percentage* parameter, converges to the same configuration in *vacation* but has a slightly different result in *yada* and both the *intruder* tests. In *intruder* it favors the configuration with 7 threads at P-state 0 over the configuration with 6 threads at P-state 0 chosen by the other two heuristics. The difference should be attributed to the non-static execution profile of *intruder* and the reduced number of steps required by *heuristic 2*



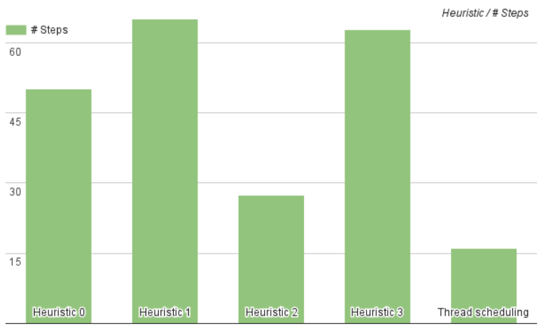
(a) *Vacation*



(b) *Intruder*



(c) *Intruder* bound to package 0

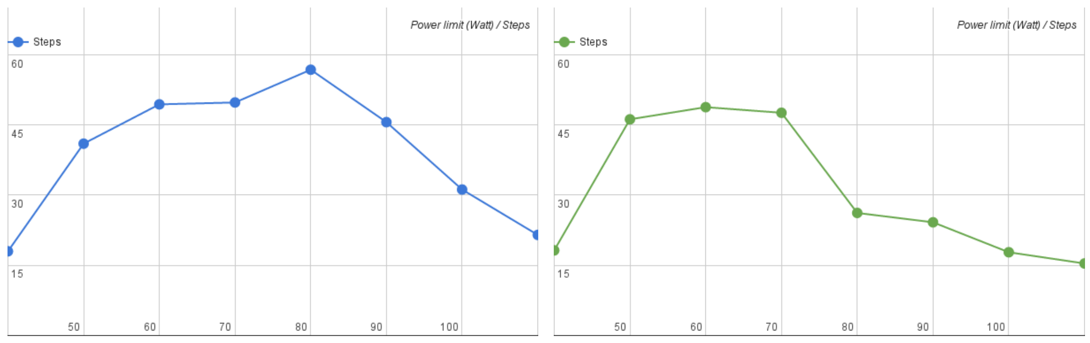


(d) *Yada*

Figure 22: Number of steps required by the heuristics to converge to the optimal configuration in different transactional applications

which evaluates the two configuration at an earlier point in the execution time. In *yada*, *heuristic 2* converges on average to 15.8 threads with P-state 1.5 compared to 19 threads with performance state 2.2 of the other two heuristics. Both the configurations show a very similar throughput but the configuration with more threads at a lower frequency is more energy efficient. The different exploration outcome should be attributed to the too value of the *percentage* parameter used by *heuristic 2* that excludes the most optimal configuration. We don't compare the convergence results of *heuristic 3* as it is the only exploration heuristic based on a constraint on the amount of energy used per committed which is set to a different value for each application.

In all tested applications, *heuristic 1* requires more steps to converge than *heuristic 0* despite the similarity of the power consumption values provided by the profiler to the actual run-time values. The direction defined by the increase in power consumption can lead to the exploration of configurations with few threads at high frequency that are generally inefficient in applications with limited contention like *yada* and *vacation*. Differently, *heuristic 2* can provide very good results on both the *intruder* executions and *yada*, requiring on average only between the 19% and 54% of the steps required by heuristic 0. In *vacation* it cannot provide the same results which could be attributed to a sub-optimal similarity of the effective run-time power consumption with the values obtained by the profiler for configurations close to 80 Watt. As expected, *Heuristic 3* generally needs more steps than the other heuristic to converge as it relies on weaker assumptions. However, even with basic assumptions, it can reduce the number of steps for all the tested applications to less than 65 for all the considered executions, which is a significantly lower number than the 240 steps required to test all configurations.



(a) Vacation (b) Yada

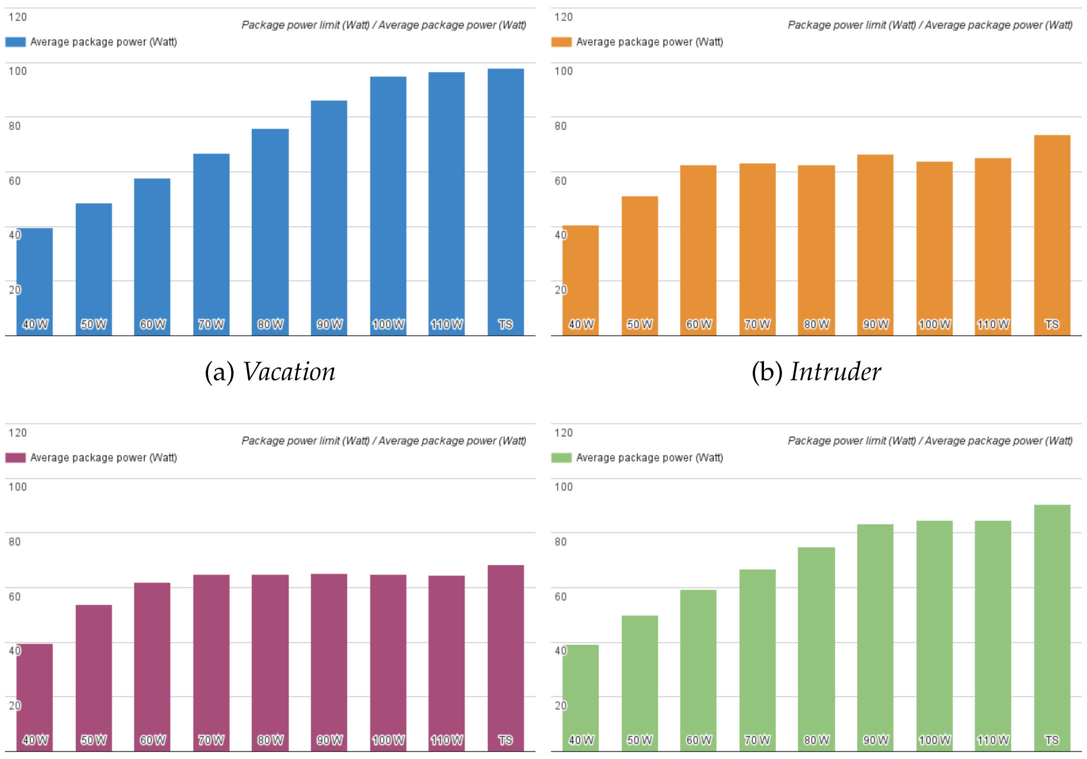
Figure 23: Number of steps required to converge for *heuristic 2* with different values of package power limit

Figure 23 shows the number of steps required by heuristic 2 to converge considering different values of package power limit defined as 10 Watt increments in the range between 40 Watt and 110 Watt. The *percentage* parameter was fixed to 20% for all executions. The number of required steps is the highest in the range between 60 Watt and 80 Watt with the lowest value at the two extremes (40 Watt and 110 Watt).

All the power constrained heuristics stop the exploration whenever a configuration with only 1 active thread at a given P-state exceeds the limit which can reduce the required number of steps when the power limit is set to low values for the respective system. Differently, considering that *heuristics 2* skips configurations outside the range defined by the *percentage* parameter, the exploration can skip an higher portion of the configurations when the power limit is set to higher values. We don't display the results of the other heuristics as they are less interesting. The number of steps steadily increases when increasing the package power limit for *heuristic 0* and *heuristic 1*. *Heuristic 3* presents a constant number of steps for all executions since it must always explore all P-states.

4.7 Experimental results with different power consumption constraints

In this subsection we present the performance and average package power consumption of transactional applications considering different values for the power limit. As usual, we consider the package power consumption as the sum of the power consumption of package 0 and package 1 for the server system. We only consider executions with *heuristic 0* as it does provide more consistent results. *Heuristic 2* could provide a lower exploration time and consequently a lower overhead but it would require a fine tuning of the *percentage* parameter to always converge to the optimal configuration. In any case, the exploration phase is only executed at the start of the applications: the longer is the application execution time and the less relevant becomes the exploration overhead to the overall execution metrics. Unlike the analysis on Chapter 3, we also allow configurations with performance state P0 as it is can provide increased throughput when the power limit is set to sufficiently high value. We executed the same workloads presented on the previous subsection using the same parameters with the exception of the power limit which is set to the appropriate value for each execution. All results are obtained by computing the average of 10 runs.



(a) *Vacation* (b) *Intruder*

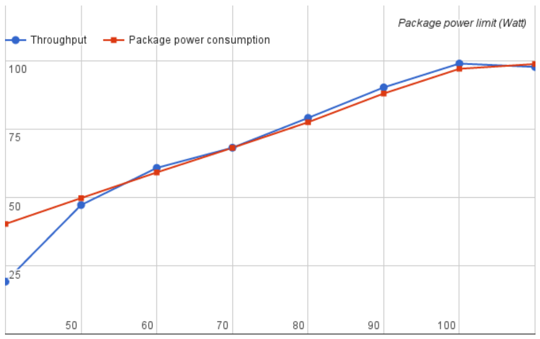
(c) *Intruder* bound to package 0 (d) *Yada*

Figure 24: Average package power consumption during the execution of transactional applications considering different values of package power limit and basic thread scheduling

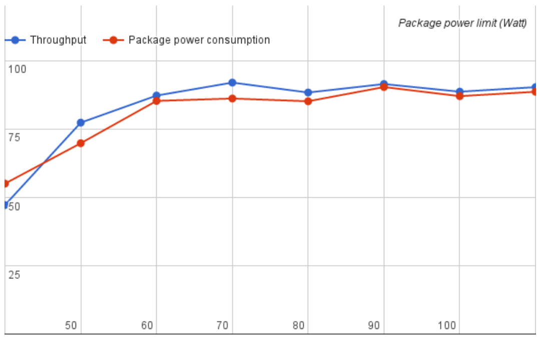
Figure 24 shows the average package power consumption of the execution of transactional applications with different values for the package power limit. We also consider the results of a basic thread scheduling approach that only considers configurations with different number of active threads in state P0. The power limits are defined as 10 Watt increments in the range from 40 Watt to 110 Watt. We couldn't find any application that exceeds 110 W during its execution with any configuration on the system. Therefore, the exploration with a 110 Watt limit should always converge to the configuration with the highest throughput despite the power consumption. Differently, some applications can run with a lower power consumption than 40 Watt when using only a limited number of threads. We don't consider a lower limit value because the resulting configurations would be usually inefficient and 40 Watt is already very close to the lowest value.

The heuristic can successfully provide executions within the defined limit for all the considered applications. Generally, the power consumption during the exploration phase is on average lower than the power consumption of the chosen configuration. Consequently average power consumption of the overall execution should be slightly lower than the power consumption of the configuration selected by the heuristic. We should note that the exploration heuristic seeks the configuration that provides the highest throughput while operating within the power limit. Therefore, it could be possible that even with an higher value for the limit the execution power consumption stays constant because the configuration with the highest throughput doesn't require an higher amount of power. An example of this phenomenon is the *intruder* benchmark where the best performing configuration requires less than 70 Watt, both with default scheduling and with threads bound to package 0. Differently, in *vacation* and *yada* the average power consumption increases linearly until 100 Watt. These applications have a reduced abort rate compared to *intruder* which allows them to scale up to 19 to 20 threads in state P0. All the executions based only on thread scheduling show an higher package power consumption due to the shorter exploration phase.

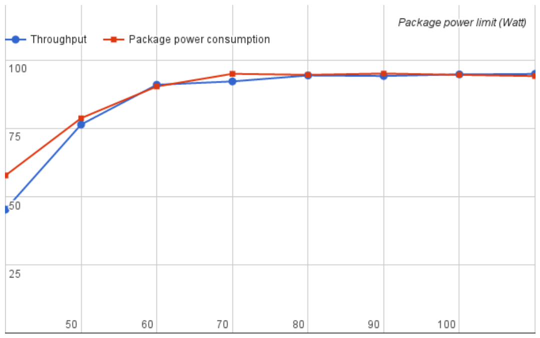
Figure 25 shows the throughput and average package power consumption of transactional applications with different values of power limit normalized to the results of the execution with the basic thread scheduling policy. The normalized throughput is generally higher than the normalized power consumption when the power limit is higher than 60 Watt. Configurations with consumption lower than 40 Watt are particularly inefficient as they only use a limited number of parallel threads. This is expected considering that, as analyzed in Chapter 3, increasing the number of threads is considerably more power efficient than increasing the frequency with low shared data contention. *Yada* has very interesting results with 99.8% of the



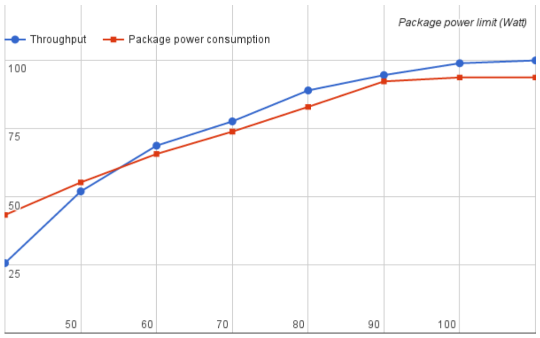
(a) *Vacation*



(b) *Intruder*



(c) *Intruder* bound to package 0



(d) *Yada*

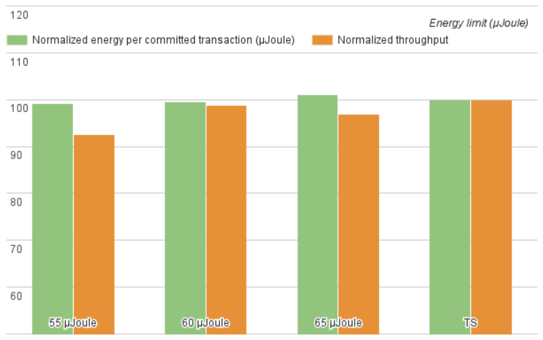
Figure 25: Throghput and package power consumption with different power limits normalized to the thread scheduling results

throughput obtained by thread scheduling execution with only 93% average power consumption. The uncapped (110 Watt) executions of *intruder* only shows 88% and 94% of the throughput obtained by the thread scheduling policy. That is the case due to an higher convergence variability, as a consequence of the dynamic execution profile, and the lower overall execution time that makes the overhead of the exploration phase more prominent.

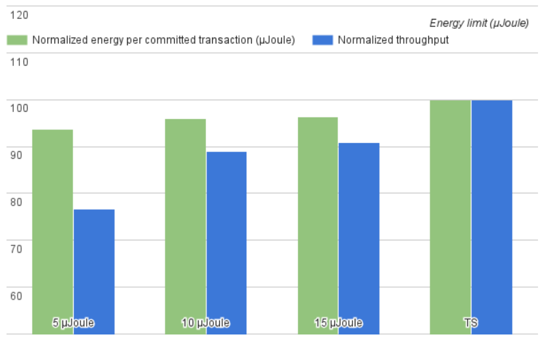
4.8 Experimental results with different energy constraints

In this subsection we study the performance and energy efficiency of transactional applications with different constraints on the average amount of energy used per committed transaction. Differently from the heuristics with a power consumption constraint, *heuristic 3* cannot provide an average energy per committed transaction for the overall execution within the defined limits. The exploration phase is performed only at the start of the execution. However, applications often rely on data structures with a non-constant access time that increases over time during the execution. Moreover, for transactional application, this could also increase the average duration of transactions which may increase the abort rate. Therefore, a decreasing throughput with a constant package power consumption results in an ever increasing amount of energy spent per committed transaction. In this scenario, no static exploration heuristic could provide guarantees on the amount of energy spent per committed transaction for applications with a dynamic execution profile. However, even if only based on the profile of the application at the start of its execution, the limit on the average energy per committed transaction can define a trade-off between energy efficiency and performance with the assumption that the relative efficiency of each configuration is mostly constant during the execution. We should note that differently from the analysis on power consumption where the goal was to obtain an higher scaling of the throughput compared to the increase in power consumption, the energy spent per committed transaction already takes into account the increase in run-time, thus, it could be legitimately considered a measure of energy efficiency. Therefore, a configuration with lower average amount of energy spent per committed transaction should be considered more energy efficient even if it offers a considerably reduced throughput.

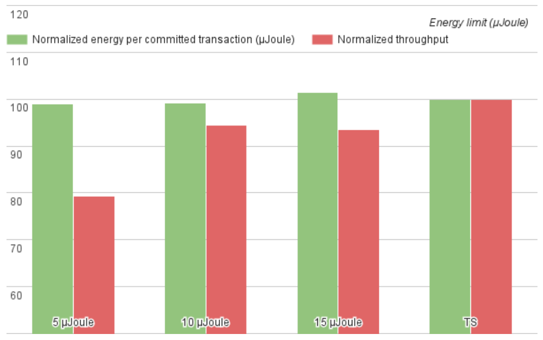
Figure 26 displays the energy per committed transaction and throughput with different energy limits for the same transactional applications considered in the previous subsection. The results are normalized with respect to the thread scheduling execution that optimizes the performance without considering the energy efficiency. All executions are performed on the server system. As investigated in Chapter 3, the energy efficiency of the server system is mostly uniform for all P-states since it only supports a limited range of CPU frequencies. Therefore, we could only defined small variations in the energy per committed transaction limit. In *intruder* the



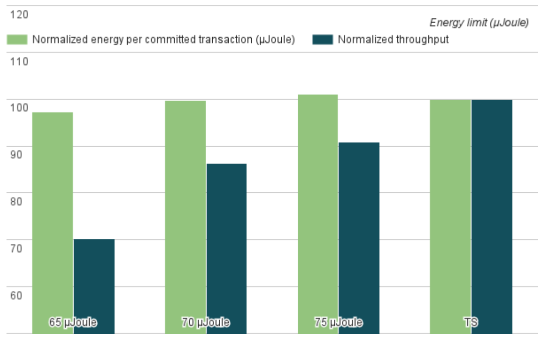
(a) *Vacation*



(b) *Intruder*



(c) *Intruder* bound to package 0

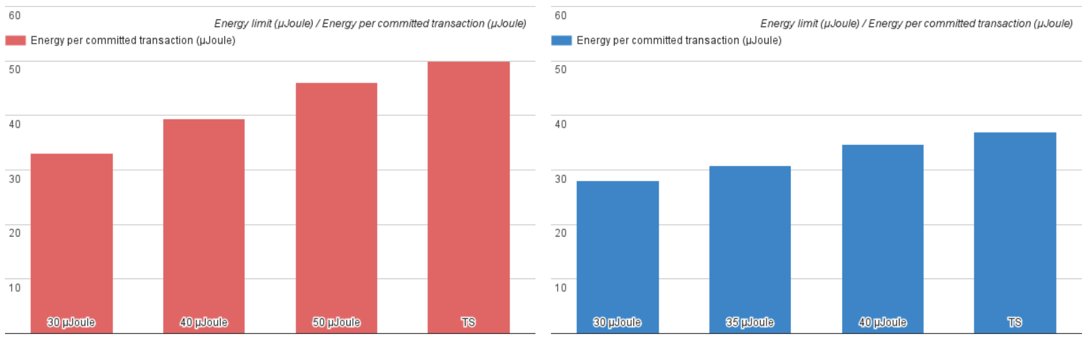


(d) *Yada*

Figure 26: Throghput and energy per committed transaction with different values of energy limits normalized to the results of the thread scheduling execution.

heuristic with 5 μJoule energy limit can provide a 6.31% lower energy per committed transaction with a 23.35% decrease in throughput with respect to the thread scheduling execution. In *vacation*, *intruder* with threads bound to package 0 and *yada* it can reduce the energy consumption by 1%, 1.1% and 2.67% respectively. We should note that these results don't consider the increased effect of the idle power consumption on the energy consumption of applications that execute for a longer time. However, considering the results obtained in Chapter 3 on the padded energy consumption at different frequencies and the relatively small differences in the throughput we can expect only marginal improvements. Even if the energy savings are only minor, they could be interesting in a data center scenario in time intervals with a low expected load during which the trade-off of reduced throughput for a lower energy consumption could be beneficial.

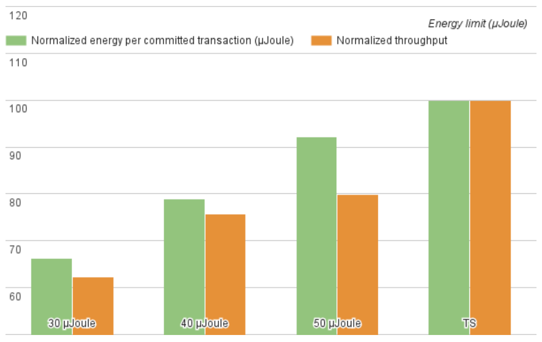
We also performed a similar test with different limit values of energy per committed transaction on the desktop system. We recall that this system offers a much wider CPU frequency range with a convex energy consumption curve at different performance states. We scaled the length of the benchmarks proportionally to the reduced throughput compared to the executions on the server system with the goal of obtaining the same proportions on the duration of the exploration phase compared to the overall run-time. The energy limits are set to different values as both the package power consumption and the throughput are system dependent. We only consider *yada* and *vacation* since they provide a less dynamic execution profile. Moreover, the higher data contention of *intruder* would be irrelevant with only 4 maximum threads. The number of starting threads is set to 2 for all executions.



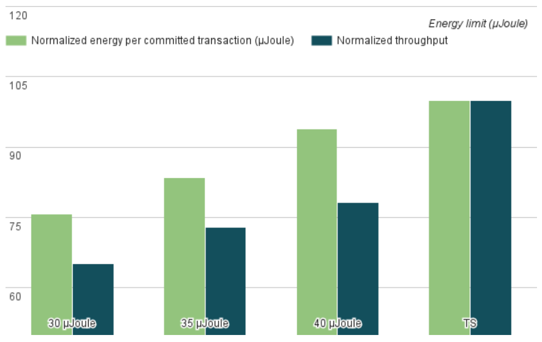
(a) Vacation (b) Yada

Figure 27: Average energy per committed transaction with different energy limits in *vacation* (a) and *yada* (b) executed on the desktop system

Interestingly, as shown in Figure 27, the heuristic can successfully provide for all executions values of energy per committed transaction within the respective limits. There are two main elements that concur to this different result compared to the executions on the server system. Firstly, the throughput is considerably lower which results in a slower evolution of the execution profile. Secondly, the energy consumption of the exploration phase should be lower as intermediate P-states are generally more energy efficient than performance states with higher frequency.



(a) Vacation



(b) Yada

Figure 28: Throghput and package power consumption with different power limits normalized to the thread scheduling results executed on the desktop system

Figure 28 displays the throughput and the amount of energy spent per committed transaction with different values of energy limit in the execution of *vacation* and *yada*. The results are normalized with respect to the thread scheduling execution that converges to 4 threads at P-state P0 in both the applications. Independently of the energy limit, all the executions converge to configurations with 4 threads with different P-states. The heuristic can provide compelling trade-offs that can lead to significant energy savings. In *vacation*, with the limit set to 30 μJoule, the heuristic can provide a 33.35 % lower energy consumption per committed transaction with a decrease in performance of 37.75%. If an higher level of performance is required, the execution with the energy limit set to 40 μJoule is also very compelling as it can provide a 21.5% energy reduction with a 24.36% reduction in the application throughput. In *yada* the heuristic can reduce the energy per committed transaction up to 24.24% with a slowdown of 34.86%. The energy savings could be even more relevant on workloads with a longer execution time or if taking into account the higher relevance of the idle power consumption on slowed down executions.

5 Conclusions

In this work we investigated the relation between performance and energy efficiency of parallel applications that rely on a software transactional memory framework to manage the concurrent access to shared data. Initially we analyzed the power consumption, energy consumption and throughput of multiple transactional application running with different static configurations of parallel threads and CPU performance states. Each performance state is associated with an hardware-defined frequency and voltage, which can be set via software exploiting the *cpufreq* framework available in modern Linux kernel releases. We considered the results on two distinct current generation systems: desktop class system with 4 physical cores and a server class system constituted by two processors with 20 physical cores each. The desktop system showed an higher energy efficiency at mid to low frequencies as the power consumption increases exponentially at faster performance states while the throughput only increases linearly. The server system provides a limited range of available frequency which results in a mostly uniform energy efficiency for all the available performance states. The results show that, when the abort rate is reasonable, increasing the number of active threads is considerably more energy efficiency than increasing the CPU frequency and voltage. The abort rate appears to be constant when considering configuration with different performance state with the same level of parallelism. Differently, we found that the not uniform memory access time in NUMA architectures can increase the number of aborts by up to 76% compared to executions with all threads scheduled to a single package. We used the results of the investigation to develop an architecture, built on top of the *TinySTM* framework, that provides compelling performance and energy trade-offs in the execution of transactional applications. The architecture relies on an exploration-based approach that searches for the configuration of parallel threads and P-state that provides the highest throughput while operating withing user defined constraint on power consumption and energy consumption. These constraints can be used to tune the power consumption and energy efficiency of applications based on the performance requirements of applications at different points in time. We developed 4 distinct exploration heuristics that require on average the exploration of less than one forth of the domain to converge to the optimal configuration. The executions constrained on the average power consumption show a reduction in throughput higher than the reduction in power consumption. The energy per committed transaction limit defines a trade-off between performance and energy efficiency. In the server system the architecture can obtain up to a 6.31% decrease in the average energy cost of committed transaction with a 23.35% decrease in throughput. The energy savings are considerably higher on the desktop system as it offers a wider range of possible frequencies. In this system, the architecture can provide up to a 33.5% lower energy consumption with a 37.75% execution slowdown. In this thesis, we worked on transactional memories as they can provide a run-time indication on the amount of shared data contention which can be a limiting factor when increasing the number of parallel threads. However, we expect that the energy savings obtainable by running at lower performance states with an high level of parallelism should be applicable to general parallel applications running on current generation hardware.

References

[1] P. D. Sanzo and B. Ciciani, "CPU-core Frequency Scaling for Efficient Thread Scheduling in Transactional Memories,"

[2] S. Issa, P. Romano, and M. Brorsson, "Green-CM: Energy efficient contention management for transactional memory," *Proceedings of the International Conference on Parallel Processing*, vol. 2015-Decem, pp. 550–559, 2015.

[3] "Introduction and overview of the multics system." `https://courses.cs.washington.edu/courses/cse451/16wi/readings/lecture_readings/MulticsDesign.pdf`. (Accessed on 12/04/2016).

[4] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, pp. 569–, Sept. 1965.

[5] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff.," *IEEE Solid-State Circuits Society Newsletter*, vol. 11, pp. 33–35, Sept 2006.

[6] "Concurrency is not parallelism." `https://talks.golang.org/2012/waza.slide#1`. (Accessed on 12/04/2016).

[7] Y. Denneulin, J. Geib, and J. M. Ehaut, "A multithreaded-based methodology to solve irregular problems *,"

[8] M. D. Hill and M. R. Marty, "Amdahl 's Law in the Multicore Era," *Computer*, vol. 41, no. July, pp. 33–38, 2008.

[9] M. Gillespie, "Amdahl's Law, Gustafson's Trend, and the Performance Limits of Parallel Applications,"

[10] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[11] X.-H. Sun and L. M. Ni, "Another Veiw on Parallel Speedup*,"

[12] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: X86-TSO," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5674 LNCS, pp. 391–407, 2009.

[13] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs," *ISORC 2010 - 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, vol. 1, pp. 185–192, 2010.

[14] B. K. Rosen, "Correctness of parallel programs: The Church-Rosser approach," *Theoretical Computer Science*, vol. 2, no. 2, pp. 183–207, 1976.

[15] "Linearizability versus serializability — peter bailis." `http://www.bailis.org/blog/linearizability-versus-serializability/`. (Accessed on 12/04/2016).

[16] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[17] "What is a transaction? (windows)." `https://msdn.microsoft.com/en-us/library/aa366402(VS.85).aspx`. (Accessed on 12/04/2016).

[18] T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *Computing Surveys*, vol. 15, no. 4, pp. 287–317, 1983.

[19] C. H. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM*, vol. 26, no. 4, pp. 631–653, 1979.

[20] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice*

of Parallel Programming, PPoPP '08, (New York, NY, USA), pp. 175–184, ACM, 2008.

[21] M. Herlihy, "Wait-free Synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, 1991.

[22] D. B. Lomet, "Process structuring, synchronization, and recovery using atomic actions," *ACM SIGSOFT Software Engineering Notes*, vol. 2, no. 2, pp. 128–137, 1977.

[23] T. Knight, "An architecture for mostly functional languages," *Proceedings of the 1986 ACM conference on LISP and functional programming - LFP '86*, pp. 105–112, 1986.

[24] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, (New York, NY, USA), pp. 204–213, ACM, 1995.

[25] "Transactional synchronization in haswell — intel® software." `https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`. (Accessed on 12/04/2016).

[26] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero, "Atomic quake," *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '09*, vol. 44, no. 4, p. 25, 2008.

[27] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," *Distributed Computing*, vol. 4167, pp. 194–208, 2006.

[28] R. Guerraoui and M. Kapalka, "The semantics of progress in lock-based transactional memory," *SIGPLAN Not.*, vol. 44, pp. 404–415, Jan. 2009.

[29] W. Scherer, "Synchronization and Concurrency in User-level Software Systems," *Computer*, 2006.

[30] C. M. Chí, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," *2008 IEEE International Symposium on Workload Characterization, IISWC'08*, pp. 35–46, 2008.

[31] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 346–365, Sept 1961.

[32] D. A. Bader and K. Madduri, *Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors*, pp. 465–476. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[33] J. Ruppert, "A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation," *Journal of Algorithms*, vol. 18, pp. 548–585, 1994.

[34] R. Barrier, "Software Transactional Memory," no. September, pp. 1–14, 2012.

[35] "8263-intel_c_2b_2b_stm_compiler_prototype_edition_-_language_extensions_and_user_5c_27s_guide_v3_0.pdf." `https://software.intel.com/sites/default/files/m/8/5/4/f/1/8263-Intel_C_2B_2B_STM_Compiler_Prototype_Edition_-_Language_Extensions_and_User_5C_27s_Guide_V3_0.pdf`. (Accessed on 12/04/2016).

[36] "docview.wss." `http://www-01.ibm.com/support/docview.wss?uid=swg27041783&aid=1`. (Accessed on 12/04/2016).

[37] T. Moreshet, R. I. Bahar, and M. Herlihy, "Energy Reduction in Multiprocessor Systems Using Transactional Memory *,"

[38] M. Herlihy, J. Eliot, and B. Moss, "Transactional Memory: Architectural Support For Lock-free Data Structures," *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 289–300, 1993.

[39] D. Rughetti, P. Romano, F. Quaglia, and B. Ciciani, "Automatic tuning of the parallelism degree in hardware transactional memory," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8632 LNCS, pp. 475–486, 2014.

[40] P. Damron, A. Fedorova, and Y. Lev, "Hybrid transactional memory," *ACM SIGARCH Computer Architecture News*, vol. 34, p. 336, 2006.

[41] F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Quéma, R. Lachaize, and M. Roth, "Challenges of memory management on modern NUMA systems," *Communications of the ACM*, vol. 58, no. 12, pp. 59–66, 2015.

[42] A. Morgenshtein, "Short-Circuit Power Reduction by Using High-Threshold Transistors," *Journal of Low Power Electronics and Applications*, vol. 2, pp. 69–78, 2012.

[43] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, pp. 33–37, Dec 2007.

[44] L. A. Barroso and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines," *Synthesis Lectures on Computer Architecture*, vol. 4, no. 1, pp. 1–108, 2009.

[45] "Quantifying the power savings by upgrading to ddr4 memory on lenovo servers." `https://lenovopress.com/lp0083.pdf`. (Accessed on 12/04/2016).

[46] P. Llopis, J. G. Blas, F. Isaila, and J. Carretero, "Survey of energy-efficient and power-proportional storage systems," *Computer Journal*, vol. 57, no. 7, pp. 1017–1032, 2014.

[47] R. Schöne, D. Molka, and M. Werner, "Wake-up latencies for processor idle states on current x86 processors," *Computer Science - Research and Development*, vol. 30, no. 2, pp. 219–227, 2015.

[48] A. Mazouz, A. Laurent, B. Pradelle, and W. Jalby, "Evaluation of CPU frequency transition latency," *Computer Science - Research and Development*, vol. 29, no. 3-4, pp. 187–195, 2014.

[49] A. Pallipadi and A. Starikovskiy, "The ondemand governor: past, present and future," *Proceedings of the Linux Symposium*, pp. 215–230, 2006.

[50] S. Hayes, "Controlling Processor C-State Usage in Linux," *Report*, p. 8, 2013.

[51] V. Pallipadi, S. Li, I. Open, S. Technology, and A. Belay, "cpuidle—Do nothing, efficiently. . .,"

[52] "Running average power limit – rapl — 01.org." `https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl`. (Accessed on 12/04/2016).

[53] P. Felber, P. Felber, C. Fetzer, C. Fetzer, T. Riegel, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 237–246, 2008.

[54] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott, "A comprehensive strategy for contention management in software transactional memory," *ACM SIGPLAN Notices*, vol. 44, no. 4, p. 141, 2009.

[55] P. D. Sanzo, M. Sannicandro, B. Ciciani, and F. Quaglia, "Markov Chain-Based Adaptive Scheduling in Software Transactional Memory," *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016*, pp. 373–382, 2016.

[56] P. D. Sanzo, F. D. Re, D. Rughetti, B. Ciciani, and F. Quaglia, "Regulating concurrency in Software transactional memory: An effective model-based approach," *International Conference on Self-Adaptive and Self-Organizing Systems, SASO*, pp. 31–40, 2013.

[57] D. Rughetti, P. D. Sanzo, B. Ciciani, and F. Quaglia, "Analytical/ML mixed approach for concurrency regulation in software transactional memory," *Proceedings - 14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2014*, pp. 81–91, 2014.

[58] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker, "Identifying the Optimal Level of Parallelism in Transactional Memory Applications,"

[59] R. Jejurikar and R. Gupta, "Dynamic Voltage Scaling for Systemwi Minimization in Real-Time Embedded Systems,"

[60] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch, "Adagio," *Proceedings of the 23rd international conference on Conference on Supercomputing - ICS '09*, p. 460, 2009.

[61] R. Child and P. A. Wilsey, "Using DVFS to Optimize Time Warp Simulations," 2012.

[62] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 404–425, July 1985.

[63] A. Baldassin, F. Klein, G. Araujo, R. Azevedo, and P. Centoducatte, "Characterizing the Energy Consumption of Software Transactional Memory," *Ieee Computer Architecture Letters*, vol. 8, no. 2, pp. 56–59, 2009.

[64] K. De Vogeleer, G. Memmi, P. Jouvelot, and F. Coelho, "The energy/frequency convexity rule: Modeling and experimental validation on mobile devices," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intel-*

ligence and Lecture Notes in Bioinformatics), vol. 8384 LNCS, no. PART 1, pp. 793–803, 2014.

Acknowledgments

First of all, I would like to thank my advisor Prof. Francesco Quaglia for the many passionate and inspiring lessons and for always believing in my work.

I would like to thank Pierangelo Di Sanzo for introducing me to the concept of transactional memory and for the many hours spent working together to solve problems faced during this work.

A word of gratitude goes to all the friends in the HPDCS research group for the many inspiring discussions, constant support and mostly importantly for making me feel like one of them since the first day I started working on this thesis.

To my friends, both new and old, for making every day more joyful.

I would like to thank my family for always believing in me and supporting me.

To Martina for all the happiness, motivation and love she brings in my life.