Enhancing the Performance of Assisted Execution Runtime Systems through Hardware/Software Techniques

Gokcen Kestor
Barcelona Supercomputing Center
C/ Jordi Girona 34
08034 Barcelona, Spain
gokcen.kestor@bsc.es

Roberto Gioiosa[*]
Pacific Northwest National Laboratory
902 Battelle Boulevard
Richland, WA 99352
roberto.gioiosa@pnnl.gov

Osman Unsal
Barcelona Supercomputing Center
C/ Jordi Girona 34
08034 Barcelona, Spain
osman.unsal@bsc.es

Adrian Cristal
IIIA - CSIC - Spanish National Research Council
C/ Jordi Girona 34
08034 Barcelona, Spain
adrian.cristal@bsc.es

Mateo Valero
Universitat Politecnica de Catalunya
C/ Jordi Girona 34
08034 Barcelona, Spain
mateo@ac.upc.edu

ABSTRACT

To meet the expected performance, future exascale systems will require programmers to increase the level of parallelism of their applications. Novel programming models simplify parallel programming at the cost of increasing runtime overheard. Assisted execution models have the potential of reducing this overhead but they generally also reduce processor utilization.

We propose an integrated hardware/software solution that automatically partition hardware resources between application and auxiliary threads. Each system level performs well-defined tasks efficiently: 1) the runtime system is enriched with a mechanism that automatically detects computing power requirements of running threads and drives the hardware actuators; 2) the hardware enforces dynamic resource partitioning; 3) the operating system provides an efficient interface between the runtime system and the hardware resource allocation mechanism. As a test case, we apply this adaptive approach to *STM*^{2}, an software transactional memory system that implements the assisted execution model.

We evaluate the proposed adaptive solution on an IBM POWER7 system using Eigenbench and STAMP benchmark suite. Results show that our approach performs equal or better than the original *STM*^{2} and achieves up to 65% and 86% performance improvement for Eigenbench and STAMP applications, respectively.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

Keywords

Exascale, Assisted execution, Transactional Memory, Performance

[*]Dr. Gioiosa was at BSC during this work.

1. INTRODUCTION

In order to achieve 10^{18} FLOPS with a limited power budget (25MW) [18], exascale systems will push thread level parallelism (TLP) to the extreme. Exascale systems are expected to feature a total of 10^8 - 10^9 high-efficient, high-density, low-power cores per systems, with 1-10 thousands cores just within a single node. The complexity of such high level of concurrency poses considerable challenges to programmers: classical message-passing or lock-based programming models, such as MPI or Pthreads, do not seem to have the capability of expressing the desired TLP. In particular, at least within a single node, it seems reasonable to leverage the use of shared memory to reduce synchronization overhead. Shared memory programming models, such as OpenMP, PGAS [25] (e.g., UPC and GA), and Transactional Memory [14], that have the potentiality to simplify parallel programming and to enable users to extract higher level of parallelism are seeing wider use in HPC. While simplifying parallel programming, however, these richer runtime systems introduce runtime overhead that may reduce applications speedups. Moreover, even for systems with limited runtime overhead, the theoretical speedup computed with the Amdahl's Law may not justify the use of all available cores/hardware threads. Both programming model runtime overhead and the Amdahl's Law limit on the theoretical speedup suggest the use of assisted execution models [11], in which some of the computing elements (cores or hardware threads) are used to support computation rather than being devoted to running additional application threads [22, 31]. The intuition behind assisted execution models is that some of the computing elements can accelerate sequential part of the application and/or relieve application threads from handling runtime functionalities, therefore pushing further the theoretical Amdahl's Law's speedup. The main drawback of assisted execution models is the generally low processor utilization and the waste of resources, especially in phases when applications could use all available hardware threads. Waste of hardware resources cannot be tolerated for exascale systems that need to achieve an efficiency of 40 GFLOPS/Watt. A tighter interaction between hardware and software is essential to reach this level of system efficiency.

This paper explores the use of fine-grained hardware resource allocation to increase overall processor utilization and application's performance for assisted execution runtime systems. As opposed to coarse-grained resource allocation (adding or removing cores/hardware threads to a particular task) that can be implemented at software level, fine-grained resource allocation (partitioning renaming registers, load/store queue entries, ROB slots, etc.) requires a collaboration between the software and the underlying hardware. Although this fine-grained resource allocation requires a deep understanding of all the layers involved, from the hardware to the applications, it has the potential to provide higher performance and better adapt to frequent changes in the application's behavior.

Transactional memory (TM) has recently received considerable interest, which eventually motivated the development of processors with hardware support for TM, such as IBM BG/Q [13]. Software-only solutions (STMs) are especially compelling because they remove some of the hardware transactional memory limitations [7, 9] and provide high portability. STM systems, however, usually suffer from high overhead, which makes them good candidates for assisted execution models. As a test case, we apply fine-grained hardware resource partitioning to *STM*^{2} (*Software Transactional Memory for Simultaneous Multithreading* processors), an assisted execution STM system [17]. With *STM*^{2}, transactional operations are divided between *application* and *auxiliary* threads: application threads optimistically perform computation, while time-consuming TM management operations, such as read-set validation, are handled by auxiliary threads. Application and auxiliary threads run on separate but paired hardware threads, thus computation and TM management operations are effectively performed in parallel.

In this work, we propose an integrated hardware/software approach where system functionalities are divided among three different components: 1) *STM*^{2} is enriched with a mechanism that automatically detects computing demand of application and auxiliary threads and drives the underlying hardware actuators; 2) the hardware enforces resource partitioning among the running threads; 3) the OS provides an interface between *STM*^{2} and the hardware. We leverage the IBM POWER7 *hardware thread prioritization* [2, 5, 28] to dynamically partition hardware resource (e.g., renaming registers or load/store queue entries) between the running threads.

We begin by proposing a set of static techniques that can be applied when configuring *STM*^{2} to partition hardware resources between application/auxiliary thread pairs. We show that static techniques work for simple applications but might not work for applications with irregular transaction structures. Hence, we propose an adaptive solution that automatically partitions hardware resources between application and auxiliary threads at run time, transparently to the programmer and with no need of manual reconfiguration. Our adaptive solution monitors the computing power demand of application and auxiliary threads and adapts to 1) phases within an application, 2) different behaviors of each application/auxiliary threads pair within the same application, and 3) the structure of the particular transaction executed by a thread at a given moment.

We test our proposals on a IBM POWER7 system using two sets of benchmarks: first, we explain the potentialities of fine-grained resource allocation using Eigenbench [15] and then we apply our solutions to STAMP applications [24]. Experimental results show that static approaches are only effective for simple scenarios while more realistic and complex applications require the use of adaptive solutions. The adaptive solution matches static approaches for simple cases, and outperforms the original *STM*^{2} for complex scenarios, up to 65% and 85% for Eigenbench and STAMP applications.

This work is organized as follows: Section 2 describes IBM POWER7 hardware thread priority mechanism. Section 3 and Section 4 introduce static and adaptive solutions for assisted execution systems, respectively. Section 5 provides experimental results. Section 6 details the related work and Section 7 concludes.

Table 1: Hardware thread priority levels for IBM POWER7.

Priority	Priority level	Privilege level	or-nop inst.
0	Thread shut off	Hypervisor	-
1	Very low	Supervisor	`or 31,31,31`
2	Low	User	`or 1,1,1`
3	Medium-Low	User	`or 6,6,6`
4	Medium	User	`or 2,2,2`
5	Medium-high	Supervisor	`or 5,5,5`
6	High	Supervisor	`or 3,3,3`
7	Very high	Hypervisor	`or 7,7,7`

2. HARDWARE RESOURCE PARTITIONING

Fine-grained hardware resource allocation generally requires hardware support to dynamically partition resources at run time with acceptable latency. A wide range of mechanisms to control hardware resources allocated to a particular core or hardware thread have been proposed in the literature [6, 19, 20]. Some of these proposals have been implemented in real IBM [12, 28, 29] or Intel [16] processors, which allows system developers to implement fine-grained resource allocation solutions on real systems. In this work fine-grained hardware resource allocation for *STM*^{2} is implemented upon IBM POWER7 processors, using the hardware thread prioritization mechanism to dynamically assign processor resources to the running threads at run time.

IBM POWER7 [1] processors are out-of-order, 8-core design with each core having up to 4 SMT threads. IBM POWER7 provides a mechanism to partition hardware resource within a core by fetching and decoding more instructions from one hardware thread than from the others [28]. Each hardware thread in a core has a *hardware thread priority*, an integer value in the range of 0 to 7, as illustrated in Table 1. The amount of hardware resources assigned to a hardware thread is proportional to the difference between the thread's priority and the priorities of the other hardware threads in the same core. The higher the priority of a hardware thread, the higher the amount of hardware resources assigned to that thread.

The priority value of a hardware thread in IBM POWER7 can be controlled by software and dynamically modified during the execution of an application. IBM POWER7 processors provide two different interfaces to change the priority of a thread: `or`-nop instructions or *Thread Status Register* (`TSR`). As Table 1 shows, not all hardware thread priority values can be set by applications: user software can only set priority levels 2, 3, 4; the operating system (OS) can set 6 out of 8 levels, from 1 to 6; the Hypervisor can span the whole range of priorities. In order to use all possible levels of priorities, a special Linux 2.6.33 kernel patched with the Hardware Managed Threads priority (HMT) patch [2, 3, 4] is required. This custom kernel provides two interfaces (a `sysfs` and a system call) through which the users can set the current hardware thread priority, including the ones that require OS or Hypervisor privilege (the OS issues a special Hypervisor call to set priority 0 and 7). The system call interface (`hmt_set()`) is more suitable for the purpose of this work because it introduces lower overhead.

3. STATIC FINE-GRAINED RESOURCE PARTITIONING

Runtime systems benefit from the assisted execution model if application threads often require support to perform time-consuming operations [22, 31]. Under these hypothesis, devoting hardware threads to perform runtime operations rather than main computation may provide higher performance than using all available hardware resources to run application threads. *STM*^{2}, in particular, provides significant speedups over canonical STM systems (between

1.8x and 5.2x on average) [17] for applications that spend a considerable amount of time performing transactions. However, similarly to other assisted execution systems, *STM*^{2} may not fully utilize the processor's resources for some cases.

This section describes fine-grained resource allocation techniques that can be applied to *STM*^{2} at configuration time to improve processor utilization and efficiency when auxiliary threads are mostly idle or overloaded.[1] Applying these techniques requires an comprehensive understanding of the IBM POWER7 hardware thread priority mechanism. In order to better explain the effects of fine-grained resource partitioning, this section follows a step-by-step approach. Experiments are performed using Eigenbench [15], a simple TM micro-benchmark that allows programmers to tune orthogonal TM characteristics, such as the number of local accesses outside or inside transactions, the conflict level, or the number of transactional operations per transaction. Eigenbench performs N consecutive iterations of a computation block, where each block consists of an embarrassingly parallel computation part and a transaction. We properly tune Eigenbench to create challenging scenarios for *STM*^{2}. We then leverage the IBM POWER7 hardware thread priority mechanism to improve *STM*^{2}'s performance in these challenging scenarios. In the following sections, AT_p denotes the priority of an application thread, AxT_p the priority of an auxiliary thread, and $\Delta_p = AT_p - AxT_p$ the difference between the priority of an application thread and its corresponding auxiliary thread.

3.1 Embarrassingly parallel phases

During embarrassingly parallel phases, threads perform computation on private data and do not need to protect accesses to memory locations. In *STM*^{2}, auxiliary threads paired with application threads not performing transactions at a given time sit idle, waiting for a new transaction to start. Since each thread runs on a dedicated hardware thread and the system is not over-provisioned, this design may lead to overall processor under-utilization. In fact, waiting auxiliary threads do not perform any useful work but consume hardware resources. A naïve solution to this problem consists of suspending waiting auxiliary threads and resuming their execution as soon as their paired application threads enter a transaction. This approach usually reduces responsiveness, which may limit overall performance, especially if the application frequently alternates short transactions and embarrassingly parallel phases. Moreover, suspending idle auxiliary threads, in general, does not increase processor utilization: the hardware thread that was running the auxiliary thread is released back to the operating system (OS) which may decide to either run another task or leave it idle. If there is no other runnable task available to the idle hardware thread, the OS may reduce the priority of the idle hardware thread, implicitly increasing the performance of the other hardware thread. This decision is not under the control of *STM*^{2} and depends on the system status at the time of suspending an auxiliary thread. On the other hand, spinning usually guarantees higher responsiveness at the cost of unnecessarily consuming hardware resources without making any progress. To increase processor utilization while maintaining high responsiveness, we reduce the hardware priority of spinning auxiliary threads (AxT_p) and restore it to its initial value as soon as the corresponding application threads start a new transaction.

The impact of reducing AxT_p during embarrassingly parallel phases on the performance of the whole application depends on the percentage of time the application spends performing embarrassingly parallel computation and the amount of extra hardware resources assigned to application threads (Δ_p). Figure 1 shows the performance improvement of Eigenbench over *STM*^{2} when running 1000 iterations per thread, with one transaction per iteration. In this experiment, the number of transactional operations per iteration is fixed to 20.[2] We then vary the percentage of time spent by Eigenbench in embarrassingly parallel phases from 25% to 95% and the value of the AxT_p while keeping $AT_p = 6$. This experiment only focuses on the performance improvement obtained from reducing the AxT_p during embarrassing parallel phases, hence $AT_p = AxT_p = 6$ inside transactions. As expected, reducing AxT_p during embarrassingly parallel computation phases provides performance improvements proportional to the percentage of time the application spends in embarrassingly parallel computation. Figure 1 also shows that the best performance values are obtained with $\Delta_p = 5$ (i.e., $AT_p = 6$ and $AxT_p = 1$). This is an important design point because this value of Δ_p can only be achieved through the HMT Linux patch. Had we limited the use of priority to the user-available levels, the maximum Δ_p would have been 2 ($AT_p = 4$ and $AxT_p = 2$), meaning that the performance improvement would have been 16.8% instead of 22.3%. This performance improvement comes essentially free of any drawbacks, as reducing hardware resources does not have any impact on the performance of waiting auxiliary threads.

The graph also reports the performance improvement obtained suspending idle auxiliary threads (*naïve-EP*) and resuming them as new transactions begin. Suspending waiting auxiliary threads provides some performance improvement, mainly because the system only runs one application and, thus, there is a high probability that the OS will reduce the hardware thread priority of the hardware thread previously running the waiting auxiliary thread. However, the performance improvement achieved with this approach does not match the one obtained with $\Delta_p = 5$. This is due to two main reasons: first, the OS is free to schedule any other process or kernel daemon on idle hardware threads previously occupied by the waiting auxiliary threads. This external process may introduce even larger slowdown on the applications threads (data cache lines eviction, TLB entries eviction, resource contention). Second, the overhead of resuming auxiliary threads may reduce the overall benefit.

3.2 Load imbalance inside transactions

In *STM*^{2}, each application/auxiliary thread pair needs to synchronize at the end of each transaction (`commit()`) before moving to the next phase. For each application/auxiliary thread pair, load imbalance may occur because: 1) the application thread issues TM

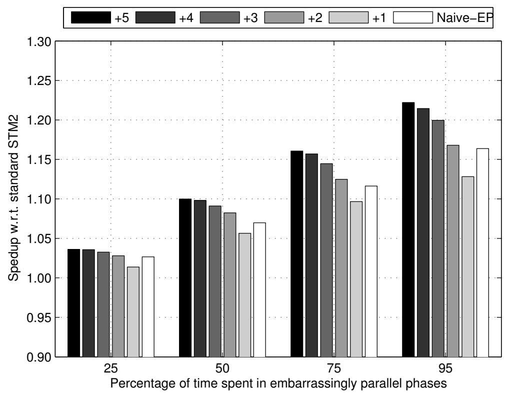
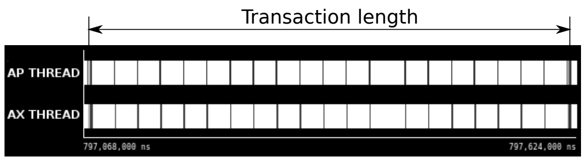


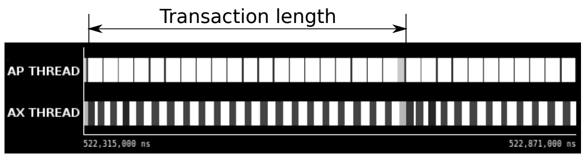
Figure 1: Performance impact of reducing AxT_p when varying the percentage of time spent performing embarrassingly parallel computation and the value of Δ_p.

[1]No application's source modification is required in order to apply these techniques.

[2]Here, and in the rest of the paper, Eigenbench is configured to perform 10% of transactional writes.



(a) Standard Case: Application threads issue TM operations at a low rate.



(b) Static allocation of extra hardware resources to application threads.

Figure 2: Frequently idle auxiliary threads within a transaction. In this trace white denotes local computation while gray bars denote transactional reads or writes. The elapsed time in both traces is the same.

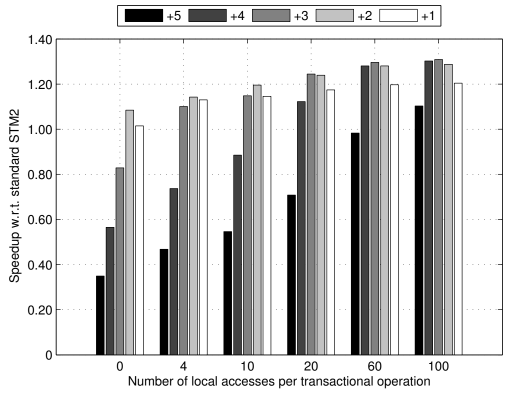


Figure 3: Performance impact of reducing AxT_p in presence of load imbalance (overloaded application threads) when varying the number of local accesses per transactional operation and Δ_p.

operations at a low rate, thus its corresponding auxiliary thread is frequently idle (Section 3.2.1), and 2) the auxiliary thread has not completed all TM management operations when the corresponding application thread reaches the commit phase (Section 3.2.2). The next sub-sections explain these scenarios with details.

3.2.1 *Overloaded application threads*

Figure 2 shows a partial execution trace (one transaction) of a scenario in which application threads perform TM operations at a low rate. In this figure, local computation (operations on private variables) is depicted in white and TM operations are drawn as gray bars. To obtain these execution traces, we instrumented STM^2 and produced traces that can be visualized with Paraver [26], a performance analysis tool used to study parallel applications. In this experiment Eigenbench is configured in such a way that application threads perform N_{local} local operations for every shared access (N_{shared}). In the example shown in Figure 2, $N_{local} = 300$ and $N_{shared} = 20$ (the total number of operations is $N_{local} \times N_{shared} + N_{shared} = 6,020$), which results in the auxiliary thread being idle for 95% of the time during the execution of a transaction.

Figure 2a shows the standard STM^2 case: the auxiliary thread is frequently idle but consumes hardware resources by spinning on the communication channel for incoming read/write messages. The application thread, on the other hand, can only use a partial amount of the shared hardware resources, with the results that its speed is limited. In this simple scenario the programmer could configure STM^2 to reduce the AxT_p, therefore assigning more hardware resources to the application thread. Figure 2b shows the effect of setting $AT_p = 6$ and $AxT_p = 1$ ($\Delta_p = 5$): Although the auxiliary thread proceeds at slower speed than the one in Figure 2a (the trace shows that each TM operation now takes longer), the application thread does not have to wait and can proceed with its computation. This happens because the auxiliary thread has still enough time to complete all TM operations before its corresponding application thread reaches the commit phase, thus the application thread does not wait to complete the transaction.

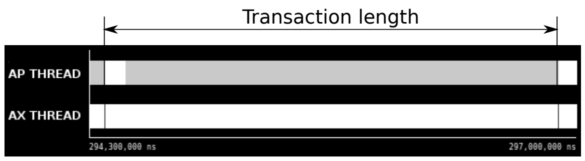
Unfortunately, setting the correct values of AT_p and AxT_p is not always straightforward: since the internal design of the IBM POWER7 hardware thread priority mechanism is not symmetric [2], the performance degradation of the lower priority thread is usually higher than the performance improvement of the higher priority thread. This design does not lead to performance degradation when reducing the priority of auxiliary threads that are actually not doing any progress, like in embarrassingly parallel computation phases. However, applying the wrong set of priorities when both threads are performing useful work may reverse the imbalance, with the final effect of worsening the overall performance.

In order to quantify the effect of fine-grained resource allocation on applications with imbalanced transactions, we performed a complete design space exploration, varying the number of accesses to local variables (N_{local}) per TM operation (N_{shared}) within a transaction and the priority values of the auxiliary threads (AxT_p); $AT_p = 6$ in all cases. The result of this design space exploration is reported in Figure 3. When the number of local accesses is limited or null, excessively reducing AxT_p reverses the imbalance: auxiliary threads become the bottleneck and application threads have to wait at commit phase for their auxiliary threads to complete their work. This often leads to performance degradation, especially when the priority difference is large (e.g., $\Delta_p \geq 4$). For $N_{local} = 0$, reducing the hardware thread priority of auxiliary threads degrades performance up to 63% ($AT_p = 6$ and $AxT_p = 1$, $\Delta_p = 5$). As the number of local accesses per TM operation increases, auxiliary threads are able to complete their work even with fewer hardware resources: for $N_{local} = 100$, aggressive settings achieve overall performance improvement of 30%.

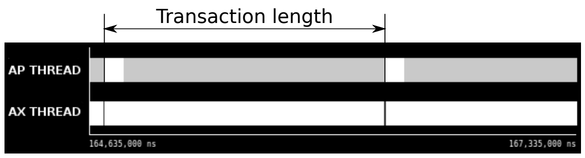
3.2.2 *Overloaded auxiliary threads*

Some TM management operations, such as read-set validation or conflict detection, require a variable amount of time to be completed. For example, the read-set validation overhead depends on the number of individual shared memory locations read during a transaction and the number of concurrent writers. The former determines the size of the read-set while the latter determines the frequency with which read-set validation is performed.

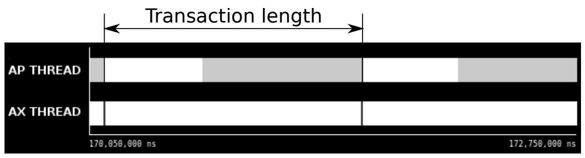
STM^2 is an eager-conflict detection STM, thus read-set validation is performed when a potential conflict arises. Note that, although required, not all read-set validations result in aborting the transaction. If an application triggers several read-set validations, auxiliary threads may not be able to complete all their TM operations before the corresponding application threads reach the commit phase. If such a situation arises, application threads are forced to wait at commit phase. Figure 4a illustrates this case: the auxiliary thread is not able to complete all TM management operations before its corresponding application thread reaches the commit phase, thus the application thread is forced to wait, effectively serializing part of computation and TM management operations. In



(a) Standard case.



(b) Spin-only.



(c) Entire transaction.

Figure 4: Overloaded auxiliary threads. In the traces, white denotes transaction computation (both local and shared accesses) while light gray denotes application threads' waiting time at commit phase. The elapsed time is the same for all the traces.

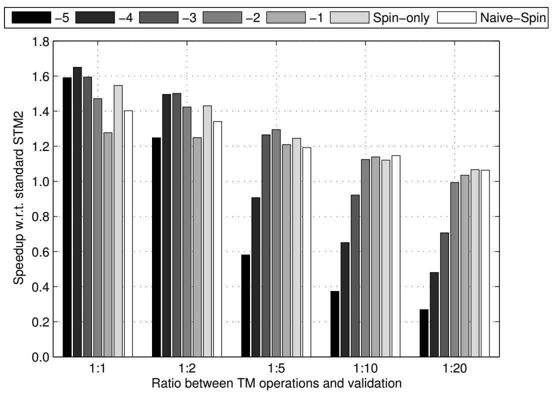


Figure 5: Performance impact of increasing the AxT_p when varying the number of read-set validations per transactional operation and Δ_p.

the trace, application thread's waiting time at commit phase is denoted with light gray while white depicts the execution of a transaction (both local computation and transactional operations). As shown in Figure 4a, prioritizing auxiliary threads ($\Delta_p < 0$) may provide performance benefits. This technique can be applied just at commit phase (Spin-only) or throughout the whole transaction (Entire Transaction).

Spin-only: Figures 4b shows how reducing AT_p while an application thread is waiting at commit phase speeds up the execution of TM management operations and improve overall performance. This solution, similarly to the case described in Section 3.1, is straightforward and does not introduce any performance degradation because application threads do not perform useful work while waiting at commit phase. In particular, the figure shows the case in which $AT_p = 1$ and $AxT_p = 6$ ($\Delta_p = -5$). Comparing Figures 4a and 4b, there is no performance degradation for the application thread computing phase (white in the traces), while the spinning time (light gray) is considerably reduced. Performance improvement, in this case, is proportional to the spinning time reduction.

Entire transaction: Figure 4c shows a solution that decreases AT_P at the beginning of the transaction and maintains $\Delta_p < 0$ for the entire transaction execution. This approach is more aggressive than the previous spin-only solution: the performance of application threads considerably reduces by prioritizing auxiliary threads during the transaction computation phase. This can be observed by comparing Figures 4a and 4c: the application thread computing phase (white) takes considerably longer than in the standard case. On the other hand, the auxiliary thread does not accumulate too many pending TM operations, hence its corresponding application thread has to spin for less time at commit phase. The net result is that performance improves with respect to both the baseline (65%) and the safe, spin-only approach (7%).

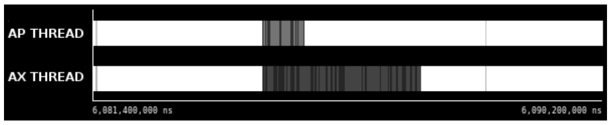
Figure 5 shows the impact of statically increasing AxT_p ($\Delta_p < 0$) at the beginning of the transaction when the number of read-set validations per TM operation decreases. The graph also shows the performance of reducing AT_p to one ($\Delta_p = -5$) when application threads wait at commit phase (i.e., the case depicted in Figure 4b). Finally, the graph reports the effect of suspending waiting applications threads and resuming them once the transaction is ready to commit (*naïve-spin*). The figure reports the performance improvement over the standard STM^2 when performing one read-set validation every N transactional operations (1:N) and varying the value of Δ_p. These experiments show that increasing AxT_p for transactions that require a high number of validations generally provides higher performance improvements than just reducing AT_p at commit phase or suspending/resuming waiting applications. For example, when performing one validation for every transactional operation (1:1), increasing AxT_p from the beginning of the transaction provides performance improvement of 65% over the standard STM^2 while reducing AT_p to one at commit phase provides 55% performance improvement and suspending/resuming waiting application threads provides 40% performance improvement. On the other hand, reducing AT_p when an application thread is spinning at commit phase is a safe operation that does not introduce any measurable performance degradation. This technique can, therefore, be applied as fall-back mechanism in case a perfect balance between application and auxiliary threads cannot be achieved by increasing AxT_p at the beginning of a transaction.

Finally, the best value of Δ_p is not always the same for all ratios and aggressive settings are only possible when the number of read-set validations per transactional operations is high. Figure 5 shows, in fact, that incorrect settings of AT_p and AxT_p when prioritizing auxiliary threads may lead to considerable performance degradation (up to 70%), especially if the number of read-set validations per transaction is low. As for the case of reducing AxT_p for frequently idle auxiliary threads (Section 3.2.1), manually setting AT_p and AxT_p is a complicated task, even for simple micro-benchmarks.

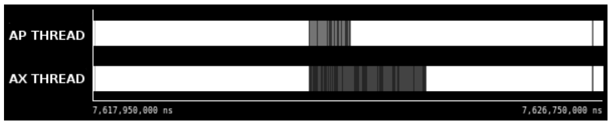
4. ADAPTIVE FINE-GRAINED RESOURCE PARTITIONING

Section 3 shows that, for simple scenarios, setting the best values of AT_p and AxT_p can be set at configuration time by an expert programmer. For complex scenarios, instead, setting the right value of Δ_p is not trivial and depends on the actual work performed by application and auxiliary threads.

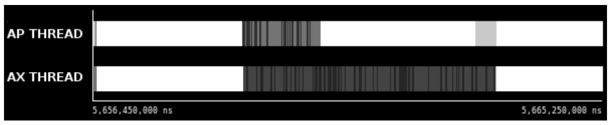
Figure 6a shows an execution trace of a Eigenbench transaction with a burst of accesses to shared memory locations roughly starting in the middle of the transaction: the application thread performs



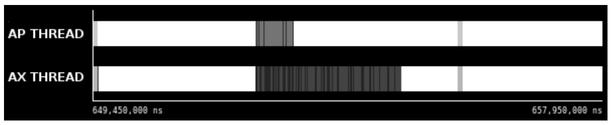
(a) Standard case



(b) Static allocation of extra resources to auxiliary threads



(c) Static allocation of extra resources to application threads



(d) Adaptive resource partitioning

Figure 6: Irregular transactions with bursts of transactional operations in the middle. White denotes local computation within transaction, gray bars denote transactional reads and writes, the light gray at the end of the transaction denotes application threads waiting at commit phase.

some local computation (white in the trace) followed by a burst of shared memory accesses (the gray bars in the trace denote transactional read and write operations), and then performs computation on local data. This simple example shows that a large part of the execution of TM operations overlaps with the application computation. However, the same trace shows that the auxiliary thread is mainly idle during the local computation phases and considerably overloaded when the sudden burst of shared memory accesses starts. As with many static approaches, the main problem in this example is that it is not trivial to configure, at compile time, the values of AT_p and AxT_p that provide the best performance for both local computation and bursts of shared memory accesses.

Noticing that the burst of shared accesses results in a considerable amount of work for the auxiliary thread, one possible approach is to increase AxT_p ($\Delta_p < 0$). Figure 6b shows the effect of setting $AT_p = 5$ and $AxT_p = 6$ ($\Delta_p = -1$): although the performance of the auxiliary thread improves considerably during the execution of the TM operations, the performance degradation suffered in the local computation phases outweighs the improvement, which results in an overall 27% slowdown with respect to the standard STM^2 design. The opposite approach consists of reducing the priority of the auxiliary thread to benefit the application thread. Figure 6c shows that the performance of the application thread during the local computation phase improves by 12%, while, obviously, the performance of the auxiliary thread when performing TM operations decreases. In fact, the trace shows that the auxiliary thread is still performing TM operations when the application thread reaches the end of the transaction, thus, the application thread has to wait at commit phase. Although local computation phases are predominant in this transaction structure and part of the TM operations overlaps with the second local computation phase, there is still a slight performance degradation. Neither solution is optimal, as both of them gain and suffer in opposite phases during the execution of the transaction. The common problem to both solutions is that static approaches seldom adapt to non-uniform structures, such as the one presented in Figure 6a.

This section introduces an automatic solution that adapts, at run time, to the transaction's structure and automatically sets the best values of AT_p and AxT_p. This adaptive solution is based on heuristics and on the lessons learned when applying the static solutions described in Section 3. The proposed heuristics are designed according to the following key principles:

P1 Reducing the hardware thread priority of either application or auxiliary threads introduces an asymmetric performance degradation [2]. Thus, we need to be careful when decreasing the priority of a thread, especially for large values of $|\Delta_p|$.

P2 Reducing the priority of a waiting auxiliary thread considerably improves performance (up to 30%) without any performance degradation. The heuristics should decrease AxT_p whenever the application enters a (large) embarranssingly parallel section.

P3 If application threads issue bursts of TM operations, their corresponding auxiliary threads may not be able to complete all TM operations before the application threads reach commit phase. The heuristics should consider prioritizing auxiliary threads ($\Delta_p < 0$) in such scenarios. However, auxiliary thread prioritization reduces application threads performance and must be done judiciously.

P4 Large values of Δ_p provide higher performance improvements. However, this required the use of the system call `hmt_set()`, which introduces some overhead. We tend to modify the priority of waiting threads (mainly the auxiliary threads), as this directly affects the value of $\Delta_p = AT_p - AxT_p$.

Besides these basic key principles, the adaptive solution employs the static mechanisms that do not depend on the actual application and auxiliary thread structure, such as reducing AxT_p during embarrassingly parallel phases or reducing AT_p when spinning at commit phase. For short embarrassingly parallel sections or for cases in which all TM operations are completed when an application thread reaches the commit phase, the overhead of invoking a system call may outweigh the performance improvement. To avoid such situations, the adaptive solution snoozes for a short time before actually issuing the system call. At the beginning of a transaction the initial values are $AT_p = AT_p(0) = 5$ and $AxT_p = AxT_p(0) = 5$. Since AT_p remains constant during the execution of a transaction ($P4$), these settings allow the enriched STM^2 to reach large positive values of Δ_p ($P2$) but also to be able to prioritize auxiliary threads ($P3$), depending on the structure of the transaction.

Detecting load imbalance: The first problem towards the development of an adaptive solution is how to determine if there is load imbalance and which thread is the bottleneck. To this extent, we monitor the number of messages queued in the communication channel between application and auxiliary threads, which introduces negligible overhead. In fact, if the application thread issues transactional operations at a low rate, the queue would be frequently empty. Conversely, if the application thread issues TM operations at a rate higher than the rate at which the auxiliary thread completes its work, the queue would gradually fill up.

Reducing the priority of auxiliary threads: In case it is determined that an auxiliary thread is often idle (i.e., the queue is empty most of the time), the heuristic aims at reducing AxT_p so that the paired application thread receives more hardware resources. In order not to reverse the load imbalance, the heuristic decreases AxT_p only after a certain amount of consecutive attempts to dequeue a message that reveal that the queue is empty. This quantity is denoted as *snooze_time*. A constant *snooze_time* would decrease the priority regardless of the current value of AxT_p, however, according to principle $P1$, the priority of a thread should be decreased judiciously, especially if the priority difference Δ_p is already large. On the other hand, Figure 3 shows that, even for small values of N_{local}, decreasing AxT_p provides benefits. The solution adopted here is to make *snooze_time* variable: every time the auxiliary thread decreases its priority, the heuristic computes a new *snooze_time* value as a function of Δ_p. The general idea is that the larger the value of Δ_p, the larger the value of *snooze_time*, i.e., the value of AxT_p is reduced less often if the Δ_p is large. The heuristics used in this work to compute *snooze_time* is the following:

$$snooze\_time = a + b * (\Delta_p + 1)^c$$

where $a = 40$, $b = 10$ and $c = 3$ have been computed empirically.

Increasing the priority of auxiliary threads: In this adaptive system, the value of AxT_p at time t may be lower than its initial value $AxT_p(0)$. This may happen because the auxiliary thread was frequently idle before t and the heuristic decreased its hardware thread priority. If a burst of TM operations such as the one depicted in Figure 6a arises, the heuristic should increase the value of AxT_p or else the auxiliary thread will not be able to complete its work before its corresponding application thread reaches the commit phase (see Figure 6c). Raising the value of AxT_p should not be too impulsive as, if the burst is particularly short, it might be worth keeping $AxT_p < AxT_p(0)$ (Figure 3). The heuristic increases the value of AxT_p until it reaches $AxT_p(0)$ if there are more than `QS_THRESHOLD` elements in the queue (in the current implementation this value is 20), which denotes that the auxiliary thread is potentially accumulating work. Additionally, auxiliary threads are allowed to increase their priority up to $AxT_p = 6$ (i.e., $\Delta_p = -1$) if the number of pending messages in the queue is larger than `QS_THRESHOLD_CRITICAL` (128 in our implementation). Notice that higher values of $|\Delta_p|$ (i.e., $\Delta_p = -2, -3, ..$) are also possible but cumbersome. Instead, in case the load cannot be balanced with $\Delta_p = -1$, the heuristic reduces the priority of the application thread while spinning at commit phase (see Figure 4b).

Figure 6d shows that the dynamic solution is able to adapt to the irregular structure of the transaction and provides higher performance than both static approaches. First, the adaptive solution reduces the priority of the auxiliary thread during the initial local computation phase, therefore improving the performance of the application thread. Next, when the sudden burst of accesses to shared memory locations starts, the adaptive solution gradually increases AxT_p and, if necessary, reaches values higher than AT_p ($\Delta_p < 0$). Finally, once the auxiliary thread has completed all its TM operations, the adaptive mechanism reduces AxT_p again, improving the performance of the application thread in the last part of the transaction. The overall result is performance improvement of 15%, where static approaches result in performance degradation when decreasing or increasing AxT_p, respectively (based on the best values among all possible settings of AT_p and AxT_p).

The adaptive solution's heuristics can be evaluated with two different metrics: 1) the convergence to the best value of Δ_p, and 2) the speed at which the heuristics reach that value. Figure 7 shows the value of AxT_p (AT_p=$AT_p(0)$ throughout the execution) as function of the elapsed time since the beginning of the transaction, for

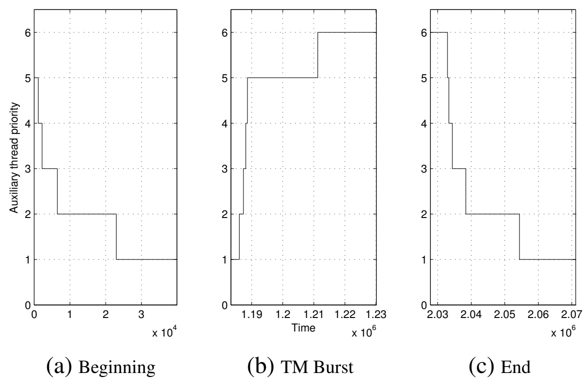


Figure 7: The adaptive solution automatically changes the value of AxT_p according to the structure of the transaction and the computing demand of application and auxiliary threads. These graphs show the values of AxT_p during one transaction for the case discussed in Figure 6d. The x-axis reports time since the beginning of the transaction.

a transaction with a burst of TM operations in the middle (the same case reported in Figure 6d). As Figure 7a shows, the adaptive solution successfully converges to $AxT_p = 1$ (stable state), first quickly and then, as Δ_p increases, more slowly (the steps get larger as Δ_p increases). When the sudden burst of TM operations starts (Figure 7b), the heuristic quickly adapts to the new scenario and converges to $AxT_p = 5$. Since increasing AxT_p does not depend on Δ_p, the steps are much smaller than in Figure 7a. In the meanwhile, the auxiliary thread has accumulated a considerable amount of work, thus, the heuristic further increases $AxT_p = 6$ ($\Delta_p = -1$), giving more hardware resources to the auxiliary thread. Finally, once all the accumulated work has been processed, the heuristic automatically reduces the auxiliary thread priority until it reaches the value $AxT_p = 1$ (Figure 7c), similarly to what happens in the first part.

5. EXPERIMENTAL RESULTS

This section presents the evaluation of the static and adaptive solutions on an IBM POWER7 system (8 cores, 4 hardware threads/-core) equipped with 64 GB of RAM. *STM*^{2} and all the applications are compiled with GCC 4.3.4 with optimization level `O3`; the results reported for each application are the average of 25 runs. In order to use all hardware priority levels, all tests are performed on a custom version of the Linux 2.6.33 kernel patched with the HMT patch [3]. In all the experiments, Eigenbench and STAMP applications use all the 32 available hardware threads: 16 application threads and 16 auxiliary threads.

5.1 Eigenbench

Figure 6 shows the effect of statically increasing (Figure 6b) and decreasing (Figure 6c) AxT_p for transactions with irregular structures. For this particular example, neither statically decreasing nor increasing AxT_p provides performance improvement over the standard *STM*^{2}, while the adaptive solution effectively allocates hardware resources on demand and improves performance by 15%.

For transactions with burst of TM operations, the performance improvement depends on both the size and the position of the burst. The size of the burst clearly affects whether prioritizing application/auxiliary threads provides higher performance. In the extreme cases (bursts of 0% or 100%), the examples degenerate to the cases presented in Section 3. If a short burst of TM operations occurs at the beginning of a transaction, the auxiliary thread has enough time to complete all TM management operations before the ap-

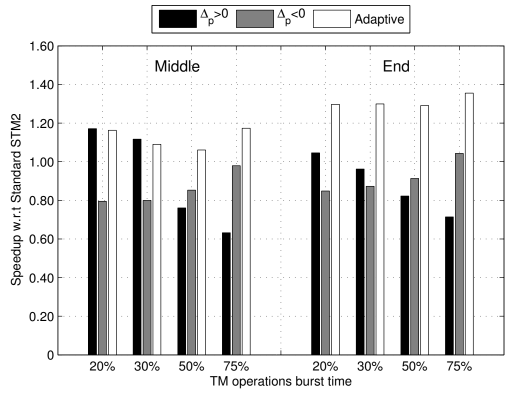


Figure 8: Performance of static (best values among all combinations) and adaptive solutions for application with irregular transaction structures and varying size/position of burst of shared accesses.

plication thread reaches the commit phase, even without applying fine-grained resource allocation. As the burst of TM operations moves towards the end of the transaction, the auxiliary thread may not be able to complete all the TM operations before its corresponding application thread reaches the commit phase. If that happens, the application thread will spin at commit phase, which leads to sub-optimal performance. Obviously, the worst case occurs when the burst of TM operations appears at the end of the transaction: in this case application and auxiliary threads essentially run sequentially, invalidating most of the advantages of assisted execution.

Figure 8 shows the performance of the best combination for the proposed static approaches and the adaptive solution over the standard *STM*^{2} design, when varying the size of the burst (from 20% to 75% of the transaction execution time) and the position of the burst (middle and end of the transaction). The experiments in the graph show that statically partitioning hardware resources provides performance improvement only for extreme cases (20% and 75%), either because there is a large part of the transaction in which the auxiliary thread is mainly idle, or because the burst is large enough to cause the application thread to spin at commit phase for a considerable amount of time. In the other scenarios (30% and 50%), *STM*^{2} effectively runs TM operations and computation in parallel. The dynamic approach, on the other hand, 1) provides performance improvement over both static approaches and 2) more importantly, always outperforms the standard *STM*^{2} for both the "Middle" and the "End" cases. This experiment shows that the automatic solution is able to adapt to the structure of the transaction, properly increasing or decreasing the value of AxT_p on demand. In a nutshell, the adaptive solution provides performance improvements between 6% and 38% over the standard *STM*^{2}and outperforms the best performance provided by both static techniques.

5.2 STAMP applications

In this section we apply static and adaptive fine-grained resource allocation to STAMP applications [24]. Figure 9 reports the performance impact of (separately) applying the static approaches and the adaptive solution. The graph reports, for each static technique, the best values of the pair (AT_p, AxT_p) among all possible configurations. For several applications, applying fine-grained resource allocation results in performance improvement (*Bayes*, *Genome*, *Yada*, *Labyrinth*, and *SSCA2*). Others applications (*Vacation*, *Intruder*, and *Kmeans*), instead, show limited or no performance improvement. These applications are well balanced and the original *STM*^{2}

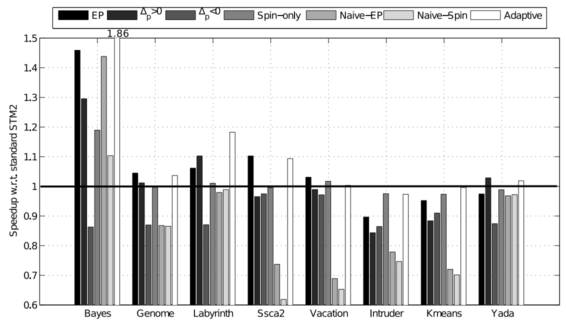


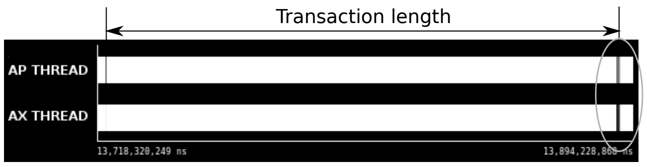
Figure 9: Performance impact of static (best values among all combinations) and adaptive solutions for STAMP applications.

design already provides optimal performance. In these cases, the adaptive solution aims at not worsening performance.

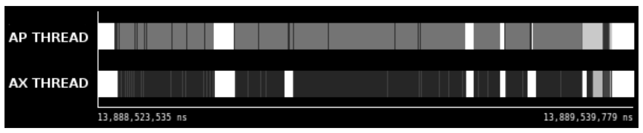
Statically decreasing AxT_p during embarrassingly parallel phases generally improves performance, up to 46% for *Bayes*. Even in this simple case, however, static solutions may suffer from the overhead of unnecessary changing the values of AT_p, e.g., if the application present back-to-back transactions with a short interval time between transactions, which may lead to performance degradation up to 10% (*Intruder*). The same scenario arises when application threads are waiting at commit phase: if the auxiliary thread has already performed all the TM operations when the application thread reaches the commit phase, the system call overhead may induce performance degradation. Suspending idle auxiliary threads (*naïve-EP*) or spinning application threads (*naïve-Spin*) introduces an even larger overhead: in the worst cases (large number of small transactions) the performance slowdown can be up to 38% (*SSCA2*). In general, *naïve-EP* and *naïve-Spin* perform worst or equal than their hardware thread priority counterparts (*EP* and *Spin-only*, respectively). In order to avoid these situations, the adaptive solution snoozes for a short time before reducing the priority of waiting application/auxiliary threads. In particular, the adaptive solution reduces AT_p only if there are at least `SPIN_THRESHOLD` messages (20 in the current implementation) in the communication channel when an application thread reaches the commit phase. Similarly, the adaptive solution reduces AxT_p in embarrassingly parallel phases only after `EP_THRESHOLD` cycles.

Within transactions, static techniques do not usually provide performance improvement: Prioritizing auxiliary threads ($\Delta_p < 0$) always reduces application's performance while prioritizing application threads ($\Delta_p > 0$) provides speedups for *Bayes*, *Labyrinth* and *Yada*. Spinning at commit phase provides measurable performance improvement only for *Bayes*, which suggests that auxiliary threads are not usually overloaded and AxT_p should not generally be greater than AT_p. For well-balanced applications, such as*Vacation*, static approaches provide performance degradation or have almost no effect. In these cases, the adaptive solution does not detect load imbalance and, therefore, does not priority change.

Among the STAMP applications, *Labyrinth*, *SSCA2* and *Bayes* are the most interesting cases for this study. *Labyrinth* presents very large, back-to-back transactions with a large number of local accesses followed by a burst of transactional accesses. Figure 10a depicts the execution trace of one of *Labyrinth*'s transaction: the picture clearly shows that the local computation phase (white in the trace) is predominant, which explains why statically reducing AxT_p provides some performance improvements (Figure 9). Figure 10b shows a close-up of the final part of the transaction, the burst of



(a) Labyrinth's transaction execution trace



(b) Labyrinth's execution trace: close-up of the final part of a transaction

Figure 10: Labyrinth's transactions alternate a large local computation phase (white in the figure) with a burst of transactional operations (gray bars) at the end.

TM operations. Since the burst is at the end of the transaction, the application thread has to wait at commit phase for the auxiliary thread to complete all TM management operations, though the auxiliary thread is mainly idle during the transaction (which explains the small performance improvement for the spin-only case in Figure 9). While static solutions fail to capture the application's characteristics due to the irregular structure of the transaction, the adaptive solution is able to lower AxT_p in the first part of the transaction, reaching $AxT_p = 1$, and increase AxT_p when the application thread issues the burst of TM operations. Overall the adaptive solution outperforms the standard STM^2 by 19%.

SSCA2 presents two separate execution phases: in the first phase, the application generates the graph that will be solved in the second phase. Both phases are parallel but, while the second phase uses transactions to protect shared memory locations, the first phase is embarrassingly parallel, as each thread works on its local portion of the graph. The original STM^2 assigns half the available hardware threads to run auxiliary threads even in the embarrassingly parallel phase: by statically reducing the priority of the auxiliary threads in the first phase, static solutions achieve 10.3% of performance improvement over the standard STM^2 design. In the second part of the application, *SSCA2* performs very short and balanced transactions with a low conflict rate and several concurrent writers. In this phase, the application is well balanced and applying static solutions decreases performance. The adaptive solution also lower AxT_p in the first phase but does not detect load imbalance in the second phase, hence it does not react. The net result is a performance improvement of 9.8% over STM^2. For this application, suspending waiting application or auxiliary threads has a dramatic impact on performance caused by the large number of short transactions.

Bayes is the application that shows the largest performance improvement: Even static approaches achieve improvement in the order of 30-45%. *Bayes* implements an algorithm for learning the structure of Bayesian networks from observed data through a hill-climbing strategy. Similarly to *SSCA2*, the applications performs two parallel parts: the first is mainly embarrassingly parallel and devoting more hardware resources to application threads considerably increases performance (up to 45%). In the second part, instead, the applications uses a few large transactions. However, similarly to *Labyrinth*, auxiliary threads are frequently idle, thus decreasing AxT_p provides benefits. For *Bayes* the adaptive solution precisely captures the application's structure and combine the positive effects observed for *Labyrinth* and *SSCA2*, providing a final performance improvement of 85% over the standard STM^2.

The examples shown in this section demonstrate that there are cases in which fine-grained hardware resource partitioning can be used to improve the performance of assisted execution systems, such as STM^2. For not well-balanced applications, like *Labyrinth* and *Bayes*, and for applications with large (sequential or parallel) independent computing phases, like *SSCA2* and *Bayes*, the adaptive solution successfully employs dynamic fine-grained hardware

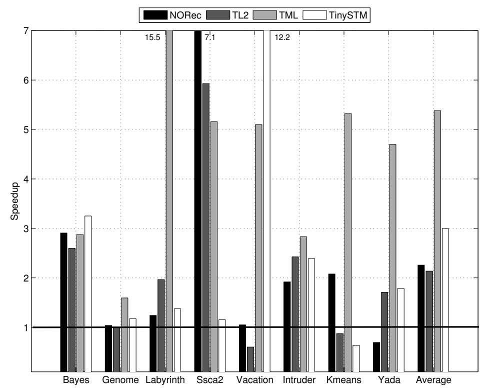


Figure 11: Speedup of adaptive fine-grained hardware resource allocation with respect to other state-of-the-art STMs.

resource partitioning and provides considerable performance improvements without any effort from the programmer, modification of the applications or library re-linking. Only *Intruder* among all the STAMP applications shows minimal performance degradation. Finally, Figure 11 shows the speedup of adaptive hardware resource allocation with respect to other state-of-the-art STMs. The adaptive solution outperforms, on average, TinySTM [27], NORec [8], TL2 [10], and TML [30], by 3x, 2.3x, 2.1x, and 5.4x, respectively.

6. RELATED WORK

Hardware thread prioritization [2, 29] has been introduced by IBM in the POWER5 processor family. AIX provides the users with an interface to modify hardware thread priorities. Linux kernels use hardware prioritization when 1) a thread is spinning on a lock, 2) a thread is waiting for another thread to complete a required operation (`smp_call_function()`), or 3) a thread is idle. Linux resets the priority of a thread after receiving an interrupt or an exception and does not keep a per-process priority status. Moreover, Linux does not consider the priority of the paired thread and, since the prioritization mechanism works with the priority difference, arbitrarily modifying the priority of one hardware thread may invalidate the decision taken on the other. Boneti et al. [2] characterized the use of hardware thread prioritization for POWER5 processors running micro-benchmarks and SPEC benchmarks. Other researchers [23] have also investigated the effect of hardware thread priorities on the execution time of co-scheduled application pairs on a trace-driven simulator of the POWER5 processor. Moreover, in a follow-up work, Boneti et al. used hardware prioritization to transparently balance high performance computing applications [3, 4], achieving up to 18% performance improvement.

Mann et al. [21] proposed a holistic approach that aims at reducing OS jitter by utilizing the additional threads or cores in a system. The authors reduce jitter through different approaches, one of which sets the hardware priorities of the primary and secondary hardware thread to 6 and 1, respectively, in order to reduce jitter caused by SMT interference.

7. CONCLUSIONS

Assisted execution systems aim at improving overall performance by relieving application threads from the overhead of performing system functionalities. These approaches are promising especially for runtime systems of novel programming models that help programmers expose a higher level of parallelism but whose runtime systems may introduce considerable overhead.

In this paper we examine the use adaptive fine-grained resource allocation to improve the efficiency and the performance of assisted execution systems. In particular, we propose an integrated hardware/software approach to dynamically partition hardware resources at run time among the running threads. We implement static and adaptive solutions for *STM*^{2}, a parallel STM system that offloads time-consuming TM operations to auxiliary threads.

While static solutions provide performance improvements only for simple cases, the proposed adaptive solution improves performance and resource utilization for applications that prove to be challenging for the original *STM*^{2}. Results obtained on a state-of-the-art IBM POWER7 system with 32 hardware threads show that adaptive fine-grained resource allocation provides performance improvement up to 65% and 86% over the standard *STM*^{2} design for Eigenbench and STAMP applications, respectively.

Finally, we remark that, although we applied fine-grained hardware resource allocation to *STM*^{2}, this approach can be used for other assisted execution systems, such as OS exception handlers [31] or dynamic check in Java Script [22].

Acknowledgment

We would like to thank Dr. Robert W. Wisniewski from IBM, Nehir Sonmez and Vasileios Karakostas from BSC for their helpful comments. This work is supported by the cooperation agreement between the BSC and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIN2007-60625, JCI-2008-3688, 2009501052, and TIN2008-02055-E, by the HiPEAC European Network of Excellence (IST-004408) and by the European Commission FP7 project VELOX (216852). Gokcen Kestor is also supported by a scholarship from the Government of Catalonia.

8. REFERENCES

[1] J. Abeles, L. Brochard, L. Capps, D. DeSota, J. Edwards, B. Elkin, J. Lewars, E. Michel, R. Panda, R. Ravindran, J. Robichaux, S. Kandadai, and S. Vemuganti. Performance guide for HPC applications on IBM power 755 system, 2010.

[2] C. Boneti, F. Cazorla, R. Gioiosa, C.-Y. Cher, A. Buyuktosunoglu, and M. Valero. Software-Controlled Priority Characterization of POWER5 Processor. In *Proc. of the 35th IEEE Intl. Symp. on Computer Architecture*, pages 415–426, Beijing, China, June 2008.

[3] C. Boneti, R. Gioiosa, F. Cazorla, J. Corbalan, J. Labarta, and M. Valero. Balancing HPC applications through smart allocation of resources in MT processors. In *Proc. of the 22nd IEEE Intl. Parallel and Distributed Processing Symp.*, Miami, FL, 2008.

[4] C. Boneti, R. Gioiosa, F. Cazorla, and M. Valero. A dynamic scheduler for balancing HPC applications. In *Proc. of the 2008 ACM/IEEE Conf. on Supercomputing*, 2008.

[5] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded PowerPC processor for commercial servers. *IBM Journal of Research and Development*, (6):885–898, 2000.

[6] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. QoS for high-performance SMT processors in embedded systems. *IEEE Micro*, (4), 2004.

[7] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proc. of the 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, USA, Mar. 2011.

[8] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Jan. 2010.

[9] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[11] M. Dubois and H. Song. Assisted execution. Technical Report GENG-98-25, University of Southern California, Los Angeles, 1998.

[12] H. Q. Le, W. J Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, (6), 2007.

[13] R. Haring. The Blue Gene/Q compute chip. In *The 23rd Symposium on High Performance Chips (Hot Chips)*, 2011.

[14] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th IEEE Annual Intl. Symp. on Computer Architecture*, 1993.

[15] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In *Proc. of the IEEE Int. Symp. on Workload Characterization*, 2010.

[16] Intel Corporation. Intel AtomTM processor n450, d410 and d510 for embedded applications, 2010. Document Number: 323439-001 EN, revision 1.0.

[17] G. Kestor, R. Gioiosa, T. Harris, A. Crystal, O. Unsal, I. Hur, and M. Valero. STM2: A parallel STM for high performance simultaneous multithreading systems. In *Proc. of the 20th IEEE Int. Conference on Parallel Architectures and Compilation Techniques*, 2011.

[18] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. A. Yelick. Exascale computing study: Technology challenges in achieving exascale systems. Technical Report DARPA-2008-13, DARPA IPTO, September 2008.

[19] K. Luo, M. Franklin, S. S. Mukherjee, and A. Seznec. Boosting SMT performance by speculation control. In *Proc. of the 15th IEEE Int. Parallel & Distributed Processing Symposium*, pages 2–, 2001.

[20] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. MLP-Aware Dynamic Cache Partitioning. *Int. Conference on High Performance Embedded Architectures and Compilers*, 2008.

[21] P. D. V. Mann and U. Mittaly. Handling OS jitter on multicore multithreaded systems. In *Proc. of the 2009 IEEE Inter. Symp. on Parallel and Distributed Processing*, pages 1–12, 2009.

[22] M. Mehrara and S. A. Mahlke. Dynamically accelerating client-side web applications through decoupled execution. In *Proc. of the 9th IEEE Int. Symp. on Code Generation and Optimization*, pages 74–84, 2011.

[23] M. R. Meswani and P. J. Teller. Evaluating the performance impact of hardware thread priorities in simultaneous multithreaded processors using SPEC CPU2000. In *Workshop on Operating System Interference in High Performance Applications*, 2006.

[24] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. of the IEEE Intl. Symp. on Workload Characterization*, 2008.

[25] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20:203–231, 2006.

[26] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualize and analyze parallel code. Technical report, In WoTUG-18, 1995.

[27] T. Riegel, C. Fetzer, and P. Felber. Time-based Transactional Memory with scalable time bases. In *19th ACM Symp. on Parallelism in Algorithms and Architectures*, 2007.

[28] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. IBM POWER7 multicore server processor. *IBM J. Res. Dev.*, pages 191–219, May 2011.

[29] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, (4/5):505–521, 2005.

[30] M. F. Spear, A. Shriraman, L. Dalessandro, and M. L. Scott. Transactional mutex locks. In *Proc. of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, USA, Feb. 2009.

[31] C. B. Zilles, J. S. Emer, and G. S. Sohi. The use of multithreading for exception handling. In *Proc. of the 32nd annual ACM/IEEE Intl. Symp. on Microarchitecture*, 1999.