Hardware/Software Techniques for Assisted Execution Runtime Systems

Gokcen Kestor
Barcelona Supercomputing Center
gokcen.kestor@bsc.es

Roberto Gioiosa
Barcelona Supercomputing Center
roberto.gioiosa@bsc.es

Osman Unsal
Barcelona Supercomputing Center
osman.unsal@bsc.es

Adrian Cristal
IIIA - Artificial Intelligence Research Institute CSIC - Spanish National Research Council
adrian.cristal@bsc.es

Mateo Valero
Universitat Politecnica de Catalunya
mateo@ac.upc.edu

ABSTRACT

The increasing complexity of modern and future multi-core/multi-threaded processors rises the question of how to best utilize processor resources. On one side, Amdahl's Law limits the maximum theoretical speedup of parallel applications while, on the other side, the increasing complexity of runtime programming language may introduce implicit serialization points. Several studies demonstrated that it is often more convenient to use some of the hardware threads to assist execution than running supplementary application threads.

Assisted execution approaches, however, may lead to low processor utilization: in this paper, we explore fine-grained hardware resource allocation techniques to assign hardware resources to application and auxiliary threads at runtime, according to their actual computing power demand. As a test case, we apply fine-grained resource allocation to *STM*^{2}, the first parallel STM system that offload STM management operations to auxiliary threads. We implemented an integrated hardware/software solution in which each level performs well-defined tasks efficiently: 1) *STM*^{2} is enriched with a runtime mechanism to express computing power requirements of application and auxiliary threads; 2) the hardware enforces dynamic resource partitioning among running threads; 3) the operating system provides a simple and efficient interface between *STM*^{2} and the hardware resource allocation mechanism.

In this paper, we leverage the IBM POWER7 hardware thread prioritization mechanism to bias the allocation of hardware resources in favor of computing intensive application threads or overloaded auxiliary threads. We test fine-grained resource allocation solutions on a real IBM POWER7 system running a simple and malleable TM benchmark (Eigenbench) and applications from the STAMP benchmark suite. Results show that the proposed integrated solution achieves up to 65% and 11% of performance improvement over the standard *STM*^{2} design for Eigenbench and STAMP applications, respectively.

1. INTRODUCTION

In order to achieve high performance, modern chip multi-core processors (CMP) and chip multi-threaded (CMT) processors require programmers to parallelize their applications. The increased complexity of writing efficient parallel algorithms for CMP/CMT processors motivated the development of several novel programming models to help programmers expose thread level parallelism, by hiding low level synchronization details and leaving programmers free to focus on their algorithm implementations. Transactional memory (TM) [8, 11, 12] is one of such programming models. TM allows programmers to mark compound statements in parallel programs as atomic (in C++, transaction), with the expectation that the underlying run time implementation will execute such transactions concurrently whenever possible, generally by means of speculation – optimistic but checked execution, with rollback and retry when conflicts arise. The principal goal of TM is to simplify synchronization by raising the level of abstraction, breaking the connection between semantic atomicity and the means by which that atomicity is achieved. Secondarily, TM has the potential to improve performance, most notably when the practical alternative is coarse-grain locking.

Unfortunately, TM systems generally introduce run time overhead (especially for software transactional memory implementations, STM): read- and write-set management, read-set validation, and conflict detection may introduce implicit serialization points that may reduce applications speedups. The limited scalability of some TM applications, together with the fact that the theoretical speedup computed with the Amdahl's Law may not justify the use of all available cores/hardware threads, suggest that some of the computing elements (cores or hardware threads) could be used to support computation rather than being devoted to running additional application threads (assisted execution model). The intuition behind this 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 and reducing runtime overhead. Several examples of auxiliary threads are available in the literature, including exception handling [30], memory prefetching [16] and dynamic check in Java Script [21]. The drawback of this approach is the generally low processor utilization and the waste of resources, especially in cases when an application could use more cores.

This paper explores the use of fine-grained hardware resource allocation to increase overall processor utilization and application's performance when a static partition of hardware resource among application and auxiliary hardware threads leads to sub-optimal performance and processor underutilization. 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 regis-

ters, load/store queue entries, ROB slots, etc.) requires a collaboration between the software and the underlying hardware. Although fine-grained resource allocation is more complicated to implement, it has the potential to provide higher performance and better adapt to frequent changes in the application's behavior.

We propose to apply fine-grained hardware resource partitioning to *STM*^{2} (*Software Transactional Memory for Simultaneous Multi-threading* processors) [15], the first parallel software transactional memory system that uses additional auxiliary threads to offload TM operations. 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, read- and write-set management, conflict detection, etc.) 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 order to increase processor utilization and overall performance through fine-grained resource allocation, we used an integrated hardware/software approach. We divide system functionalities among three different components: 1) *STM*^{2} is enriched with a runtime mechanism that expresses the expected computing demand of application and auxiliary threads; 2) the hardware provides actuators to dynamically partition hardware resources at run time; 3) the operating system (Linux in our case) provides an interface between *STM*^{2} and the dynamic hardware resource allocation mechanism.

A wide range of mechanisms to control hardware resources allocated to a particular core or hardware thread have been proposed in the literature [6, 7, 17, 18]. Some of these proposals have also been implemented in real IBM [10, 26, 27] or Intel [14] processors, thus fine-grained resource allocation solution can be also implemented on real systems. We leverage the IBM POWER7 (a 8-core, 4-way SMT processor design) *hardware thread prioritization* [26, 29] mechanism to allocate hardware resources (e.g., renaming registers or load/store queue entries) to application or auxiliary threads at run time. Each hardware thread is associated with a *hardware thread priority* and receives an amount of hardware resources proportional to this value: the higher the hardware thread priority, the larger the amount of hardware resources assigned [2, 5]. By modifying the hardware thread priorities of application and auxiliary threads, we are able to efficiently and dynamically partition processor resources and improve overall performance. In particular we reduce the priority of auxiliary threads when their corresponding application threads are not performing any transaction (auxiliary threads are idle), which increase application threads' performance. Similarly, hardware resources can be dynamically distributed between application and auxiliary threads according to the degree of computing power required when execution transactions.

Our first contribution in this study is the impact of the POWER7 hardware priority mechanism on *STM*^{2}'s performance: Although the mechanism has been extensively studied for IBM POWER5 [2] processors, this work is, to the best of our knowledge, the first that uses the hardware thread prioritization mechanism for POWER7 processors which, to the contrary of previous IBM multithreading processors, have four hardware threads per core, rather than two. We show, with simple scenarios, what is the extent of performance improvements that this mechanism allows us to achieve for *STM*^{2}. The second contribution of this work is the exploration of all possible opportunity to apply fine-grained resource allocation for *STM*^{2} and the performance implication. We propose a set of techniques that can be applied when configuring *STM*^{2} to partition hardware resources between application and auxiliary threads.[1]

We test our proposals on a real IBM POWER7 system using two sets of benchmarks: first, we explain the potentialities of fine-grained resource allocation using Eigenbench [13] (a malleable TM micro-benchmark developed to explore TM systems' corner cases) and then we apply our solutions to STAMP applications [23]. Our results show performance improvement for both Eigenbench and STAMP applications up to 65% and 11%, respectively. Our work spans the full hardware/software stack, for example we use a special version of Linux 2.6.33 patched to enable the full range of hardware priorities from within user level programs. More specifically, we use the `htm_set()` system call to set the hardware priority from within a language runtime system dynamically.

The rest of this paper is organized as follows: Section 2 provides a short background on IBM POWER7 hardware thread priority mechanism. Section 3 details our techniques and the impact of POWER7 hardware thread prioritization on *STM*^{2}. Section 4 provides our experimental results for Eigenbench and for applications from the STAMP benchmark suite. Section 5 details the related work. Finally, Section 6 concludes this work.

2. HARDWARE THREAD PRIORITIZATION

IBM POWER7 [1, 29] processors are out-of-order, 8-core design with each core having up to 4 SMT threads (32 hardware threads in total). Each core region (or "chiplet") contains a 32KB 4-way set associative L1 I-cache and a 32KB 8-way set associative L1 D-cache, a private per-core 256KB L2 cache and a 4MB portion of the shared 32MB L3 cache. The L2 is fully inclusive of both the local D/I L1 caches.

Table 1: Hardware thread priority levels in the IBM POWER7 processor

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`

Besides multi-core and multithreading capabilities, IBM POWER7 processors provide a mechanism to bias hardware resource partitioning in favor of some hardware threads in the same core [26]. Each hardware thread in a core has a *hardware thread priority*, an integer value in the range of 0 (hardware thread off) to 7 (the core is running in single thread mode), as illustrated in Table 1. The amount of hardware resources assigned to a hardware thread (and therefore its performance) is proportional to the difference between the thread's priority and the priorities of the other hardware threads. In general, the higher the priority of a hardware thread, the higher the amount of hardware resources assigned to that thread (if the other hardware thread priorities are constant). POWER7 cores implicitly assign hardware resources to each hardware thread by fetching and decoding more instructions from one hardware thread than from the others [26].

[1]No application's source modification is required in order to apply these techniques. Since *STM*^{2} is transparent to applications (which are not aware of auxiliary threads), using these techniques from within the application would result in a very complicated task and requires compiler assistance.

The priority value of a hardware thread in POWER7 processors 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: issuing an `or`-nop instruction or using the *Thread Status Register* (`TSR`). The current thread priority of a hardware thread can be read from the local `TSR` register using a `mfspr` instruction. 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, we use a special Linux 2.6.33 kernel patched with the Hardware Managed Threads priority (HMT) patch [2, 3, 4]. 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 our purpose than the `sysfs` interface because it allows us to dynamically change the application and auxiliary thread priority with lower overhead.

3. INTEGRATED FINE-GRAINED RESOURCE PARTITIONING

STM^2 is mainly designed to run applications that spend a considerable amount of time performing transactions and for which the overhead of TM operations limits scalability. For these cases, *STM*^{2} provides higher speedups over canonical STM systems. However, real applications are complex and may not present the same structure throughout the whole execution. For example, an application could spend most of its execution time in an embarrassingly parallel section (in which the application does not need to use transactions to protect concurrent accesses to shared memory locations) or alternate phases in which it heavily accesses shared locations through transactions with accesses to private variables.

In this section we describe fine-grained resource allocation techniques that can be applied at compile time to improve processor utilization and efficiency when auxiliary threads are mostly idle or overloaded. Applying these techniques, however, requires an understanding of the IBM POWER7 hardware thread priority mechanism. Previous studies [2, 19, 22] focus on IBM POWER5 processor generations which, to the contrary of POWER7, feature only two hardware threads per core rather than four. In order to better explain the performance implication of POWER7 hardware thread prioritization mechanism and the effects of fine-grained resource partitioning, we follow a step-by-step approach and use Eigenbench [13], a simple TM micro-benchmark that allows programmers to tune orthogonal TM characteristics (e.g., the number of local accesses outside or inside transactions, the conflict level, 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 described in the previous section 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. Positive values of Δ_p denote that an application thread has higher priority than its paired auxiliary thread, whereas negative values of Δ_p indicate that the auxiliary thread has higher priority than its corresponding application thread.

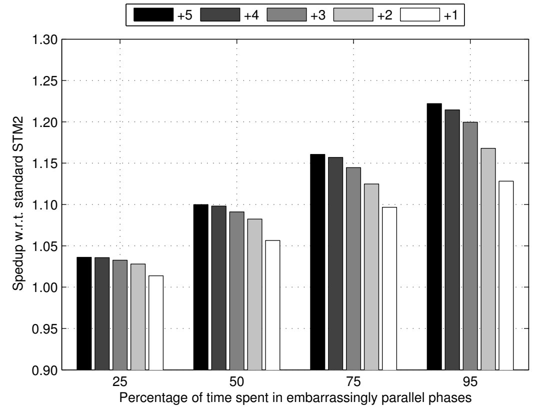


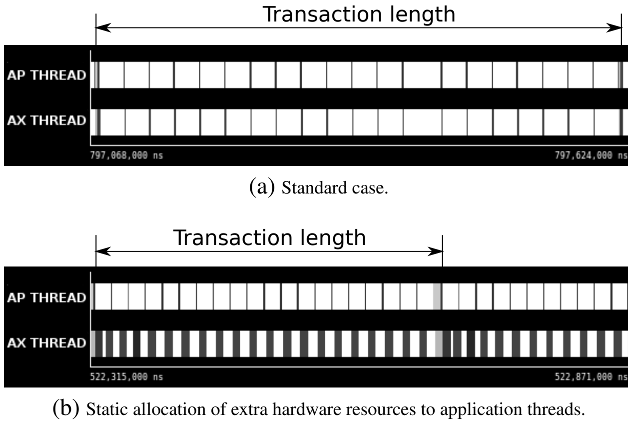
Figure 1: Performance impact of reducing the priority of auxiliary threads for varying the percentage of time spent performing embarrassingly parallel (EP) computation and for varying values of Δ_p. The graph shows that the performance improvement obtained by reducing the priority of the idle auxiliary threads is proportional to the percentage of time the application spends in embarrassingly parallel phases and to Δ_p.

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 that could be used by application threads. Although spinning usually guarantees higher responsiveness, a waiting auxiliary thread could spin at a lower speed, i.e., consuming fewer hardware resources, without suffering performance degradation. We thus reduce the hardware priority of waiting auxiliary threads (AxT_p) and restore it to its initial value as soon as their corresponding application threads start a new transaction.

The impact of reducing auxiliary threads priority during embarrassingly parallel computation phases on the performance of the whole application depends on 1) the percentage of time the application spends performing embarrassingly parallel computation, and 2) the amount of extra hardware resources assigned to application threads (Δ_p). Figure 1 shows the performance improvement of Eigenbench 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 computation phases from 25% to 95% and the value of the hardware thread priority of the auxiliary threads (AxT_p) while keeping $AT_p = 6$. In this graph we only focus on the performance improvement obtained from reducing the hardware thread priority of auxiliary threads during embarrassing parallel phases and we maintain $AT_p = AxT_p = 6$ inside transactions. As expected, Figure 1 shows that reducing AxT_p during embarrassingly parallel computation phases provides performance improvements proportional to the percentage of time the application spends in embarrassingly parallel computation. The graph also shows that the best

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



(a) Standard case.

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

Figure 2: Frequently idle auxiliary threads. (a) In this scenario, application threads issue transactional operations at a low rate, thus, auxiliary threads are frequently idle. In this, trace white denotes local computation within a transaction while gray bars denote transactional read or write. (b) Application threads receive more hardware resources ($AxT_p = 1$ and $AT_p = 6$) but auxiliary threads are still able to complete all TM operations before the corresponding application threads reach the commit phase, hence, the transaction's total execution time is reduced. The elapsed time in both traces is the same.

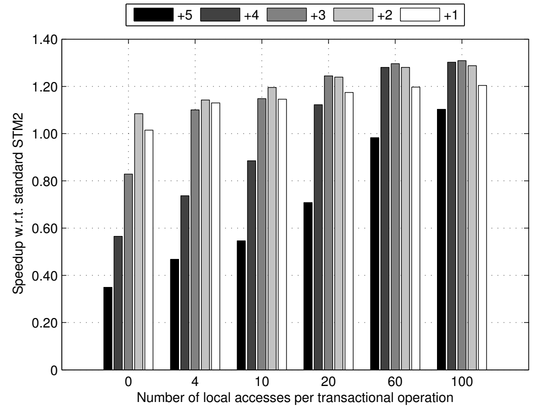


Figure 3: Performance impact of reducing the priority of auxiliary threads in presence of load imbalance (overloaded application threads). In this graph, we vary the number of local accesses per transactional operation (N_{local}) and the value of Δ_p. The results show that the best value of Δ_p is not always the same and that aggressive values of Δ_p provide performance improvement only when N_{local} is large.

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 $AxT_p = 2$), which would provide a performance improvement of 16.8% (in the 95% case) instead of 22.3% obtained with $\Delta_p = 5$. Finally, 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.

3.2 Load imbalance inside transactions

Load imbalance [3, 4, 25, 28] is a well-known problem for parallel applications that need to synchronize at certain determined points, such as at barriers or fork/join constructs. Load imbalance happens when one or more tasks in a parallel application have more work to perform than the others with the result that the whole application proceeds at the speed of the slowest tasks, which may severely limit overall performance.

In STM^2, each application/auxiliary thread pair needs to synchronize at the end of each transaction (`commit()`) before moving to the next phase. Load imbalance may occur because: 1) the application thread issues TM operations at a low rate, thus the 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.

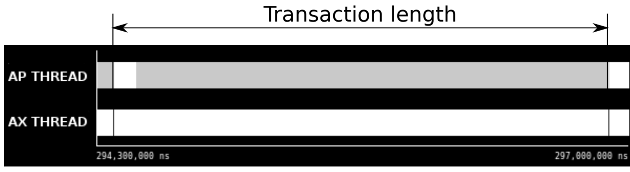
3.2.1 Overloaded application threads

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. In order to obtain these execution traces, we instrumented STM^2 and produced traces that can be visualized with Paraver [24], a performance analysis tool commonly used to study parallel applications. We tuned Eigenbench 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 $N_{shared} = 20$ (thus, 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.

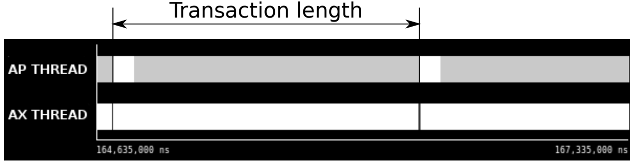
Figure 2a shows the standard STM^2 case: the auxiliary thread is frequently idle but consumes hardware resources by spinning on the communication channel for incoming messages (`MSG_READ` or `MSG_WRITE`). The application thread, on the other hand, can only use a partial amount of the shared hardware resources, with the results that its speed is limited. In this simple scenario the programmer could configure STM^2 to reduce the priority of the auxiliary thread (AxT_p), therefore assigning more hardware resources to the application thread. Figure 2b shows the effect of setting $AT_p = 6$ and $AxT_p = 1$ ($\Delta_p = 5$): by reducing AxT_p and assigning extra hardware resources to the application thread, performance improves considerably. In fact, although the auxiliary thread proceeds at slower speed than the one in Figure 2a (the trace shows that each STM operation now takes longer), the application thread does not have to wait and can proceed with its computation. On the other hand, 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 if we reduce 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.

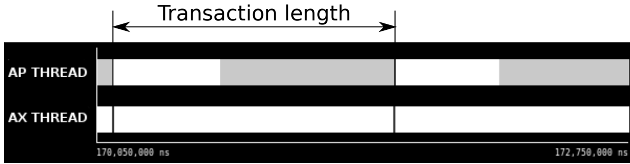
In order to quantify the effect of fine-grained resource allocation on applications with imbalanced transactions, we performed a complete design space exploration, varying the number of accesses to local variables (N_{local}) per TM operation (N_{shared}) within a transaction and the priority values of the auxiliary threads (AxT_p); $AT_p = 6$ in all cases. Figure 3 analyzes the performance improve-



(a) Standard case.



(b) Spin-only: Static allocation of extra resources to auxiliary threads at commit phase.



(c) Entire transaction: Static allocation of extra resources to auxiliary threads throughout the whole 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. In this scenario auxiliary threads are overloaded and cannot complete all TM operations before their corresponding application threads reach the commit phase. Increasing the amount of hardware resources assigned to auxiliary threads improves overall performance. The elapsed time is the same for all the traces.

ment (or degradation) on the whole application. As we can see from the graph, 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., $\Delta_p = 5$ or 4). For $N_{local} = 0$, reducing the hardware thread priority of auxiliary threads may degrade 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 $N_{local} = 100$, we can be aggressive and achieve an overall performance improvement of 44%.

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, 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 affects how often read-set validation has to be performed during a transaction.

STM^2 is an eager-conflict detection STM, thus read-set validation is performed when a potential conflict arises. Note that not all read-set validations, although required, 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 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).

Eigenbench does not allow us to control the number of read-set validations per transaction,[3] thus, in order to create the scenario in Figure 4a, we introduced extra (although not always necessary) read-set validations that allow us to simulate potential conflicts induced by large read-sets and large numbers of concurrent threads. In scenarios such as the one depicted in Figure 4a, we prioritize auxiliary threads with respect to application threads ($\Delta_p < 0$). 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 allows auxiliary threads to speed up the execution of TM management operations and achieve an overall performance improvement. This solution, similarly to the case described in Section 3.1, is straightforward and does not introduce any performance degradation because the application thread is not performing useful work while waiting. 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. As for the case in Section 3.1 (see Figure 1), overall performance improves[4] with the decrease of AT_p, thus the best performance is achieved with $\Delta_p = -5$. Although the performance of application threads reduces with their priorities, there is a net gain, as long as we do not reverse the imbalance.

Entire transaction: Figure 4c shows a solution that decreases the priority of the application thread 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: by prioritizing auxiliary threads during the transaction computation phase, the performance of application threads considerably reduces. 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, with the more aggressive approach, the performance improves with respect to both the baseline (65%) and the safe, spin-only approach (7%). However, statically reducing the priority of application threads also has the side effect of reducing the rate at which application threads inject messages into the communication channel. Consequently, auxiliary threads might spend time waiting for the next incoming message, which would reduce the net benefit. This situation arises especially for large negative values of Δ_p.

As we discussed above, the number of read-set validations per TM operation depends on application characteristics, such as the

[3] The only way to induce read-set validations is to increase the conflict rate. Unfortunately, these are real conflicts that will cause the transaction to abort, which makes it difficult to deterministically reproduce experiments.

[4] In fact, there may be a slight performance degradation caused by the lower frequency with which the application thread checks the receive of the `SIG_READYTOCOMMIT` signal but we have not noticed any measurable slowdown in our experiments.

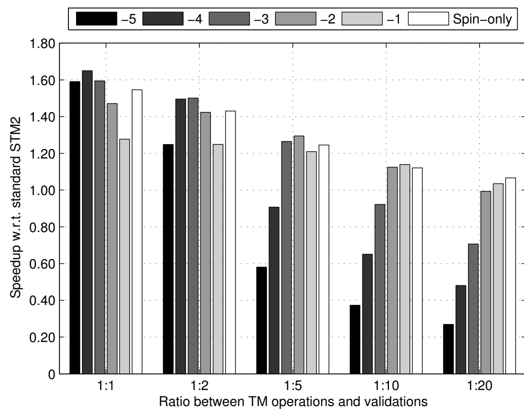


Figure 5: Performance impact of increasing the priority of overloaded auxiliary threads when varying the number of read-set validations per transactional operation and Δ_p. The graph shows that the right value of Δ_p is not always the same and that aggressive settings of Δ_p are only suitable when the ratio between read-set validations and transactional operation is high.

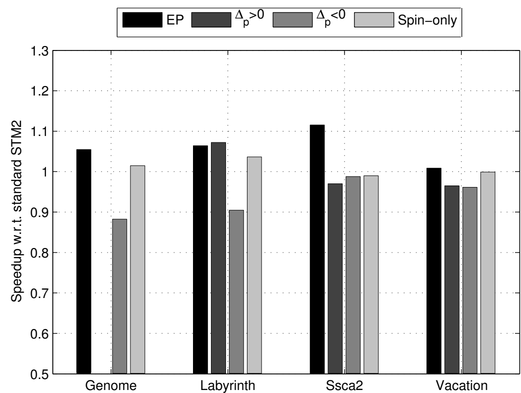


Figure 6: Performance impact of static (best values among all combinations) resource partitioning for STAMP applications. EP = Embarrassingly Parallel.

number of concurrent writers. 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). The figure reports the performance improvement over the standard STM^2 when performing read-set validation every N transactional operations (1:N) and varying the value of Δ_p. Our experiments show that increasing AxT_p for transactions that require a high number of validations generally provides higher performance improvements than by just reducing AT_p at commit phase. For example, when performing one validation for every transactional operation (1:1), increasing AxT_p from the beginning of the transaction provides a performance improvement of 65% over the standard STM^2 while reducing AT_p to one at commit phase provides 55% 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. We can, therefore, apply this technique and use it as a fall-back mechanism in case we cannot perfectly balance application and auxiliary threads by increasing AxT_p at the beginning of a transaction.

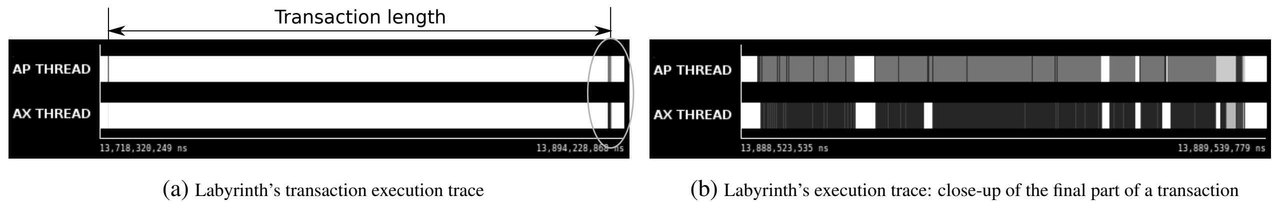
Note that the best value of Δ_p is not always the same for all ratios and that aggressive settings are only possible when the ratio between the number of read-set validations and transactional operations is high. Figure 5 shows, in fact, that incorrect setting of AT_p and AxT_p when prioritizing auxiliary 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. This work explores opportunities for performance improvement through fine-grained resource allocation, using STM^2 as a test case. We leave the design and development of a system that automatically detects load imbalance within transactions and the right settings of hardware thread priorities as future work.

4. STAMP APPLICATIONS

In the previous section we used Eigenbench to show challenging cases in which the design of STM^2 may not prove efficient. In this section we show that those scenarios are indeed common to more complex applications. We use some of the applications from the STAMP benchmark suite, a set of applications widely used to test transactional memory systems, as test cases. We selected applications from the STAMP benchmark suite that expose some of the problems mentioned in Section 3, namely *Labyrinth*, *Genome*, *SSCA2* and *Vacation*. Experiments are performed on a IBM POWER7 system (8 cores, 4 hardware threads per core) equipped with 64 GB of RAM. We compiled all applications with GCC 4.3.4 with optimization level `O3` and report the average of 25 runs for each application. In order to use all hardware priority levels, we use a version of the Linux 2.6.33 kernel patched with the HMT patch [3]. In all the experiments, the STAMP applications use all the available hardware threads (32 in the tested configuration): 16 application threads and 16 auxiliary threads. Each pair application/auxiliary thread is mapped on two hardware threads in the same core. Finally, we use the reference (large) input sets [23].

Figure 6 shows the performance of (separately) applying our fine-grained resource allocation techniques to STAMP applications. For each technique, we report the best values of the pair (AT_p, AxT_p) among all possible configurations. We should remark that several configurations, especially when decreasing AxT_p, do not work, as auxiliary threads become too slow and application threads completely fill up the communication channel (which aborts the execution of the application). As we can see, several applications from the STAMP benchmark suite show benefits when applying fine-grained resource allocation, although the improvements may be related to different reasons. This first observation proves that, indeed, there are cases in which partitioning hardware resources provides performance improvements even for complex applications.

All benchmarks benefit from reducing AxT_p when applications are involved in embarrassingly parallel computation (up to 11%). Some of the benchmarks only spend a marginal fraction of their execution time in these phases, hence, the impact on these applications is limited. The overhead of unnecessary changing the values of AT_p and AxT_p also affects the performance of fine-grained resource allocation: if the time between two transactions is short, for example, the overhead of invoking a system call may outweigh the performance improvement obtained throughout embarrassingly



(a) Labyrinth's transaction execution trace

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

Figure 7: Labyrinth's transactions alternate a large local computation phase (white in the figure) with a burst of transactional operations (gray bars) at the end.

parallel sections. This situation does not often arise with Eigenbench, where we have the complete control of the application's structure, but it appears in some of the STAMP benchmarks (e.g., *SSCA2* or *Vacation*). Inside transactions, on the other hand, it is more difficult to select the best values of AT_p and AxT_p when configuring STM^2. For well-balanced applications, such as *Vacation*, varying the values of AT_p and AxT_p provide performance degradation or have almost no effect (performance variation within 1%).

Among the tested STAMP applications, *Labyrinth* and *SSCA2* are the most interesting cases for our study. *Labyrinth* presents very large, back-to-back transactions with a large number of accesses to local memory locations followed by a burst of shared memory accesses (TM operations). Figure 7a shows 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 reducing AxT_p provides good performance improvements (Figure 6). Figure 7b shows a close-up of the final part of the transaction, the burst of TM operations.[5] Notice that, 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 (which explains the small performance improvement for the spin-only case in Figure 6), though the auxiliary thread is mainly idle during the transaction.

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 reducing the priority of the auxiliary threads in the first phase, we achieve 11.3% of performance improvement over the standard STM^2 design. This case is similar to the examples shown in Section 3.1. In the second part of the application, *SSCA2* performs very short and balanced transactions with a low conflict rate and several concurrent writers. With short transactions, the overhead of applying fine-grained resource partitioning slightly reduces performance. The net result is a performance improvement of 11% over STM^2.

For *Genome*, reducing AxT_p causes application threads to saturate the communication channel, even for $\Delta_p = 1$. In fact, in the first application phase, *Genome* quickly issues a burst of TM operations at the beginning of a transaction and then reduces the rate at which the application issues transactional operations, giving auxiliary threads time to complete its work before the application threads reach the commit phase. On the other hand, statically increasing AxT_p does not increase performance either (in fact, it introduces a performance degradation of 11%) because, in the second part of the application, transactions with auxiliary threads mainly idle are dominant.

Finally *Vacation* shows well-balanced transactions with marginal time spent in embarrassingly parallel sections. None of our techniques provide performance improvement over the original STM^2, for main computation and STM management support operations are already evenly distributed between application and auxiliary threads.

[5] Since the TM operations are very dense in this figure, it is difficult to identify single TM operations.

5. RELATED WORK

Hardware thread prioritization [9, 20] has been introduced by IBM in the POWER5 processor family. Hardware thread prioritization allows users to dynamically bias the amount of hardware resources assigned to hardware threads in the same core. AIX [9] 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 [22] 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. [19] proposed a holistic approach that aims at reducing Operating System (OS) jitter by utilizing the additional threads or cores in a system. The authors tried to handle jitter through different approaches, one of the approaches is setting the hardware priority of the primary SMT thread to priority 6 and that of the secondary SMT thread to priority 1 in order to reduce jitter caused by SMT interference.

6. CONCLUSIONS AND FUTURE WORK

In this paper we propose to use fine-grained resource allocation to improve the efficiency and the performance of assisted-execution systems. We used an integrated hardware/software approach to implement fine-grained resource allocation and divide tasks between hardware, OS and runtime system. As test case, we applied fine-grained hardware resource allocation to STM^2, a parallel software transactional memory system that offloads STM time-consuming operations to auxiliary threads, and leverage the IBM POWER7 hardware thread prioritization mechanism to dynamically partition hardware resources at runtime. We improve performance and resource utilization for application that spend a

considerable amount of time performing embarrassingly parallel computation or that show load imbalance between application and auxiliary threads within a transaction, both of which prove to be challenging scenarios for STM^2.

Our current solutions can be applied when configuring STM^2: Results obtained on a state-of-the-art, IBM POWER7 system with 32 hardware threads (8 cores, 4 hardware threads per core) show that these techniques provide performance improvement up to 65% over the standard STM^2 design for Eigenbench, a simple and malleable TM benchmark, and up to 11% for more complex applications from the STAMP benchmark suite. Our experience with the IBM POWER7 hardware prioritization mechanism suggests that integrated hardware/software solution are interesting and can be employed to efficiently solving problems that may be difficult to solve completely at one level. Moreover, we notice that large values of Δ_p can only be used for extreme cases and are unlikely to be useful for complex applications. A more fine-grained hardware prioritization mechanism that provides more intermediate values rather than extreme values, such as the current IBM POWER7 mechanism, would further help fine tuning integrated hardware/software solutions.

As the experiments in Sections 3 and 4 suggest, however, setting the correct values of AT_p and AxT_p for complex applications requires a deep understanding of the application characteristics. As future work, we plan to extend STM^2 to automatically detect load imbalance within transactions, transparently to the programmer, and the apply the correct settings of hardware thread priorities.

Finally, we remark that, although we applied fine-grained hardware resource allocation to STM^2, this approach can be used for other auxiliary-based systems, such as dynamic check in Java Script [21] or OS exception handlers [30].

7. 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] S. Choi and D. Yeung. Learning-based SMT processor resource distribution via hill-climbing. *SIGARCH Computer Architecture News*, (2), 2006.

[8] U. Drepper. Parallel programming with transactional memory. *ACM Queue*, pages 38–45, 2008.

[9] B. Gibbs, B. Atyam, F. Berres, B. Blanchard, L. Castillo, P. Coelho, N. Guerin, L. Liu, C. D. Maciel, and C. Thirumalai. *Advanced POWER Virtualization on IBM eServer p5 Servers: Architecture and Performance Considerations*. IBM Redbook, 2005.

[10] 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.

[11] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2nd edition, 2010.

[12] 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.

[13] 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.

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

[15] 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 *To appear in the Proc. of the 20th IEEE Int. Conference on Parallel Architectures and Compilation Techniques*, 2011. Available from https://www.bscmsrc.eu/research/papers/Gokcen Kestor/all.

[16] S. S. Liao, P. H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *Proc. of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 117–128, 2002.

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

[18] 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.

[19] 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.

[20] H. M. Mathis, A. E. Mericas, J. D. McCalpin, R. J. Eickemeyer, and S. R. Kunkel. Characterization of simultaneous multithreading (SMT) efficiency in POWER5. *IBM J. Res. Dev.*, (4/5):555–564, 2005.

[21] 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.

[22] 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.

[23] 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.

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

[25] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. Technical report, University of Minnesota, Minneapolis, 1999.

[26] 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.

[27] 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.

[28] C. Walshaw and M. Cross. Dynamic mesh partitioning and load-balancing for parallel computational mechanics codes. *Computational Mechanics Using High Performance Computing, Saxe-Coburg Publications, Stirling*, 2002.

[29] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. C. Rubio, F. Rawson, and J. B. Carter. Architecting for power management: The IBM POWER7 approach. In *Proc. of the 16th Intl. Symp. on High Performance Computer Architecture*, 2010.

[30] 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.