Evaluation of two Formulations of the Conjugate Gradients Method with Transactional Memory

M. Schindewolf, B. Rocker, W. Karl, V. Heuveline

No. 2013-01

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)

$\frac{\partial u}{\partial t} - \nu\Delta u$
$+ (u \cdot \nabla) u + \frac{1}{\rho}\nabla p$
$\frac{\partial u}{\partial t}$

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)
ISSN 2191–0693
No. 2013-01

Impressum

Karlsruhe Institute of Technology (KIT)
Engineering Mathematics and Computing Lab (EMCL)

Fritz-Erler-Str. 23, building 01.86
76133 Karlsruhe
Germany

KIT – University of the State of Baden Wuerttemberg and
National Laboratory of the Helmholtz Association

Evaluation of two Formulations of the Conjugate Gradients Method with Transactional Memory

Martin Schindewolf[1], Björn Rocker[2], Wolfgang Karl[1], and Vincent Heuveline[3]

[1] Karlsruhe Institute of Technology (KIT), Chair for Computer Architecture and Parallel Processing, Haid-und-Neu-Straße 7, 76131 Karlsruhe, Germany
{schindewolf,karl}@kit.edu
[2] Robert Bosch GmbH, Corporate Sector Research and Advance Engineering, Robert-Bosch-Platz 1, 70839 Gerlingen-Schillerhöhe, Germany
bjoern.rocker@de.bosch.com
[3] Karlsruhe Institute of Technology (KIT), Engineering Mathematics and Computing Lab (EMCL), Fritz-Erler-Str. 23, 76133 Karlsruhe, Germany
vincent.heuveline@kit.edu

Abstract. Transactional Memory (TM) offers new possibilities for algorithmic design. This paper evaluates TM implementations of two algorithmic variations of the wide-spread conjugate gradients method (CG) regarding their performance on multi-core CPUs employing TM. Through applying tools for TM that visualize the TM application behavior, we show that the main bottleneck for both is the waiting times at barriers and illustrate the implementation of reduction operations with TM in a beneficial way. Performance monitoring through using the PAPI interface uncovers the quantity and type of instructions that each algorithms requires. This basic work is the key for environment-aware numerics as well as a hint for software developers who plan to use TM.

1 Motivation through previous Work

Transactional Memory (TM) has been proposed to facilitate the synchronization of multiple threads in a parallel shared memory program and promises performance gains through optimistic concurrency. In previous work [8], we investigated the possibility to apply Software Transactional Memory to the method of Conjugate Gradients (CG) formulated according to Saad's algorithm [10] without preconditioning. The method of Conjugate Gradients is a solver for linear systems of equations that is frequently used for problems in the area of structural mechanics and computational fluid dynamics. Due to its relevance, we investigate optimization opportunities through an in-depth analysis of the TM application's behavior and explore methods for its optimization in this paper.

Previous experiences with hardware TM systems show that transactions that include more shared memory updates show a better performance in case the contention between transactions is low [11]. Since contention has been low in previous experiments, we searched for a formulation of the CG algorithm in the literature that enables larger transaction sizes to transfer the optimization strategy from HTM to STM and found pipelined CG [14].

In this paper we demonstrate the implementation of the pipelined CG algorithm with TM and other OpenMP-based synchronization primitives to evaluate and compare these variants and illustrate the run time behavior in a post-processing step. In order to achieve this, we apply the components of a framework for the Visualization and Optimization of TM Applications (VisOTMA) to the resulting pipelined CG variant and its previous version in order to compare its performance, convergence behavior, and utilization of the microarchitecture. Compared with related work in tools for Transactional Memory applications, our approach targets the `C` programming language and complements TM events with readings of performance counters through the use of the PAPI interface [15]. Through this additional information, we reveal the cause that restricts performance with TM and the pipelined CG variant whereas a comparison with the previous version of CG shows the main differences in utilization of the microarchitecture.

2 Pipelined Conjugate Gradient Solver with OpenMP

Strzodka and Göddeke introduce the pipelined Conjugate Gradient solver to enable mixed precision and pipelined algorithms that accurately solve partial differential equations with low precision components on FPGAs [14]. From these collection of proposed algorithmic variants of the conjugate gradient method, we select the basic pipelined CG variant with three reduction operations that should be combined in one transaction. The idea behind the pipelined CG is that all computations on vector elements should be done in parallel. With this rearrangement, it becomes feasible to stream a vector instead of having to store all elements of the vector. First, Strzodka and Göddeke reorder all vector operations so that these can be performed in parallel [14]:

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \alpha_k \mathbf{p}_k, \tag{1}$$
$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{q}_k, \tag{2}$$
$$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k, \tag{3}$$
$$\mathbf{q}_{k+1} = \mathrm{A}\mathbf{p}_{k+1}. \tag{4}$$

The main contribution of this algorithm is to eliminate the requirement to compute all elements of the vector $\mathbf{r}_{k+1}$ in order to compute $\mathbf{p}_{k+1}$. Strzodka and Göddeke lift this restriction through introducing $\sigma_k = \rho_{k+1}$ that does not require knowledge of $\mathbf{r}_{k+1}$: $\sigma_k = \alpha_k(\alpha_k \mathbf{q}_k \cdot \mathbf{q}_k - \mathbf{p}_k \cdot \mathbf{q}_k)$. Then the scalar products are computed after the reordered vector operations shown in Equation 1:

$$\rho_{k+1} = \mathbf{r}_{k+1} \cdot \mathbf{r}_{k+1}, \tag{5}$$
$$\alpha_{k+1} = \frac{\rho_{k+1}}{\mathbf{p}_{k+1} \cdot \mathbf{q}_{k+1}}, \tag{6}$$
$$\sigma_{k+1} = \alpha_{k+1}(\alpha_{k+1}\mathbf{q}_{k+1} \cdot \mathbf{q}_{k+1} - \mathbf{p}_{k+1} \cdot \mathbf{q}_{k+1}), \tag{7}$$
$$\beta_{k+1} = \frac{\sigma_{k+1}}{\rho_{k+1}}. \tag{8}$$

This variant is useful in the case of a sparse matrix A that enables to compute one step of the algorithm in a fully pipelined fashion. The pipelined CG variant is suited in case the matrix A is sparse and does not require global communication. Therefore, the pipelined CG is e.g., applicable for solving the stationary heat equation without heat source.

This main loop of the pipelined CG with OpenMP iterates until $|\mathbf{r_{k+1}}| <= \epsilon$ being the convergence criteria for the algorithm. The algorithm converged to a solution when it found a solution that satisfies this condition. In the following, we will discuss the mapping of the algorithm from the Equation 1 and Equation 5 to this implementation. First, $\mathbf{u}_{k+1}$ and $\mathbf{r}_{k+1}$ are both computed according to Equation 1 and Equation 2 in an OpenMP `for` loop. Then, we compute p_{k+1} as described in Equation 3 and reset the vector $\mathbf{q}$. The sparse matrix multiplication, involving A and $\mathbf{p}$, takes place according to Equation 4. The implementation resets the scalar variables and performs three reductions to compute the scalar products $\mathbf{r_{k+1}} \cdot \mathbf{r_{k+1}}$, $\mathbf{p_{k+1}} \cdot \mathbf{q_{k+1}}$, $\mathbf{q_{k+1}} \cdot \mathbf{q_{k+1}}$. In comparison with the CG according to Saad [10], that demanded two separate reductions, the computation with pipelined CG requires three reductions. The advantage is that one enlarged critical section or transaction embraces all three of them. These three reductions implement the vector operations of Equation 5, 6, and 7. Please note that all of the above steps except resetting the scalar variables are performed in parallel. Computing Equation 8 again requires to serialize the execution and compute the values of $\alpha_{k+1}, \sigma_{k+1}$ and β_{k+1}. Finally we increment the number of iterations and compute the norm of r_{k+1}. The result is compared with ϵ in the `while` statement. In case the norm of r_{k+1} already satisfies the condition, the implementation will output the result and also compute the error. Otherwise, the algorithm performs another iteration of the loop until the solution satisfies the condition.

Our implementation uses OpenMP pragmas to mark parallel for loops and the single directive to implement the algorithm as described before. Note that our approach does not take advantage of the fact that pipelined CG supports the streaming of a vector. Instead our approach aims to implement the reduction, that can now be made three times larger than before, assuming a constant vector size. This different reduction pattern enables us to use larger transactions (or critical sections) and to implement them in two different ways. These different ways of implementing the reduction pattern are compared and analyzed in the following. The *Reduction* case uses the OpenMP reduction concatenating three reductions in one pragma: `#pragma omp for reduction(+:rho) reduction(+:qq) reduction(+:pq) schedule(static)`. The three reductions are implemented in three ways: *Fast*, *Slow Long* and *Slow Short*. *Fast* executes the accumulation with a thread-local variable over a thread private part of the vector. After finishing this calculation, each thread performs one update to add the thread-local variable (e.g., `pq_priv`) to the shared memory one (e.g., `pq`). Thus, contention between threads stems from a single update of the shared memory variable. In the following *th* represents the number of parallel threads and *dim* the dimension. Regardless of *dim* the *Fast* pattern leads to *th* updates of the shared variable which is a huge gain compared to the *Slow* which updates the shared variable directly with each computation. Of course, the complexity to implement the *Fast* pattern is slightly higher. The *Fast* version of pipelined CG updates all three shared memory variables in one transaction/critical section. This enlarges the size of the transaction because instead of one update with normal CG for each of the reduction, there are three updates in pipelined CG. *Slow Long* updates the three shared memory locations in one transaction or critical section and does not use a thread-local variable for storing intermediate results.

Slow Short also does not use thread-local variables to store intermediate results and performs each update of a shared memory location in a dedicated transaction or critical section. Thus, both *Slow* variants require the same amount of updates of the shared variable and differ only in the granularity of the applied synchronization mechanism. *Slow* updates the shared variable dim times. If the work is distributed evenly among th threads, each threads performs $\frac{dim}{th}$ updates. For $dim \gg th$ this pattern creates high contention on the shared variable because each thread accesses it multiple times. In a multi-core system this will result in coherency traffic that will invalidate the datum in the other caches, leading to performance loss. With TM, this leads to an increasing number of conflicts and, hence, rollbacks. For OpenMP atomic, the *Fast* version uses thread-local variables whereas the *Slow* version does not. If possible `omp atomic` maps to a native atomic instruction that updates one memory location without being interrupted by other processors. This atomicity is limited to one memory location and can not be extended. Thus, the *Atomic Fast* uses the thread-local variables to update a shared memory location and *Atomic Slow* updates a shared memory location for each new value. Since each value must be updated with a separate atomic instruction, there is no need to distinguish between long and short sections.

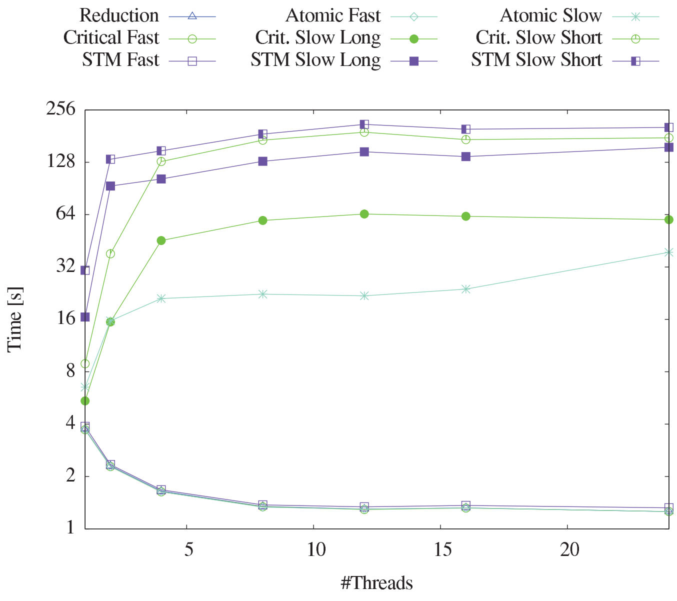


Fig. 1. Execution time of pipelined CG with 1 to 24 threads on W1. W1 is a two socket Westmere system with two Intel(R) Xeon(R) CPUs X5670 each with six cores, two way hyperthreading, and 2.93 GHz. Hence the total number of hardware threads is 24 and the upper limit for our exeriments.

The execution time of the pipelined CG method is depicted in Figure 1. The y-axis holds the average execution time in seconds over 17 runs and the x-axis the number of threads. Again, the *Fast* versions of *Atomic*, *Critical* and *Reduction* show a nearly identical behavior and are hard to distinguish. *Fast STM* also has a slightly higher execution time than e.g., *Reduction*. The ranking of the slow variants is as follows: *Atomic Slow Long*, *Critical Slow Long*, *STM Slow Long*, *Critical Slow Short*, *STM Slow Short*. Again, neither slow variant achieves the run time of the respective single thread. Thus, all slow variants show a slowdown for the execution with more than two threads.

The interesting insight is that neither of the short variants (STM or critical) performed as good as or better than a long variant. Thus, enlarging the granularity of critical sections under the given conditions results in a better performance, but the only way to achieve a speedup is the use of thread-local variables that avoid frequent updates of the shared memory and hereby reduce contention for shared locations.

2.1 Comparison of CG and Pipelined CG

In order to compare normal CG and pipelined CG we employ them to solve the one dimensional stationary heat equation without heat source which has been discretized by using finite differences with a 3-point stencil.

Dimension	Epsilon	Start vector	Solution
$5 * 10^6$	$1 * 10^{-13}$	$\mathbf{0}$	$\mathbf{1}$

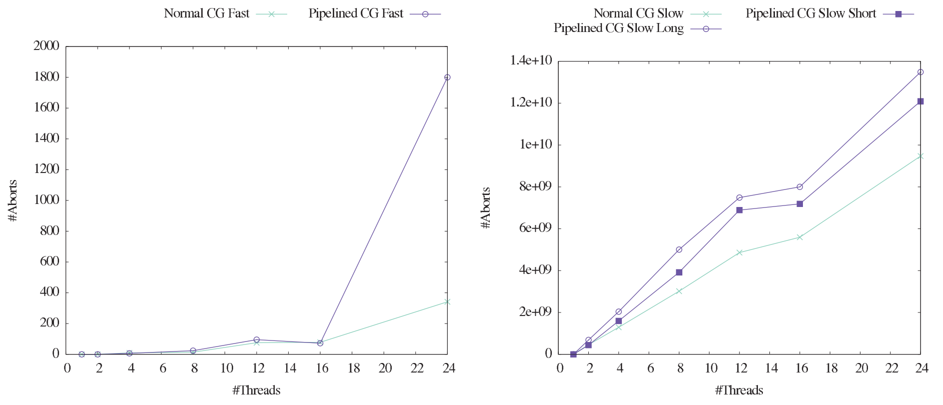
Table 1. Parameters for example problem solved with two implementations of the CG method.

The key parameters for the following experiments are shown in Table 1. Both CG variants execute a loop that iterates over the numerical algorithm. Each iteration refines the current solution and herewith reduces the error ($e_k = \|u_{sol} - u_k\|$). Usually, the error cannot be computed since the exact solution is unknown. In our experiments we have chosen the right hand side of the problem Au=b according to the formula $b_j = \sum_{i=1}^{n} a_{j,i}$ with $A \in \mathbb{R}^{n \times n}, b, u \in \mathbb{R}^n$ and $n \in \mathbb{N}$. As predicted by the theory, the error decreases monotonically. This guarantees that both formulations of the CG work correct. Due to the fact that the residual, unlike the error, does not decrease monotonically, it was not suitable for us to use this as criteria for the correctness of the implementation.

In order to compare a more realistic scenario, we have chosen an absolute stopping criteria for the residual in our experiments. The algorithm iterates as long as the residual $\|b - Au_k\|$ is greater than a given epsilon.

In practice the convergence may be perturbed through round-off errors that affect the numerical stability. Whether an algorithm is suited to find a solution to a given problem also depends on the algorithmic details as well as the implementation. Thus, a different formulation of the same algorithm may show a different convergence behavior. Moreover, even implementation details such as the order of elements when summing up a vector, may have an impact on the convergence behavior. Thus, the impact of new technologies, like TM, on the convergence behavior has to be researched thoroughly. We implement both CG variants in the programming language `C` and parallelize them, as described earlier, with OpenMP and the described synchronization mechanisms. GCC in version 4.6.1 generates both executables with the compiler options `-fopenmp -O3 -g3` that affect performance.

Software Transactional Memory Characteristics – The transactional characteristics of both CG variants are discussed first because these may dominate the utilization of architectural resources. For example a transaction that performs mainly integer operations and aborts frequently and, thus, repeats these



(a) Aborts with *Fast* versions of normal and pipelined CG

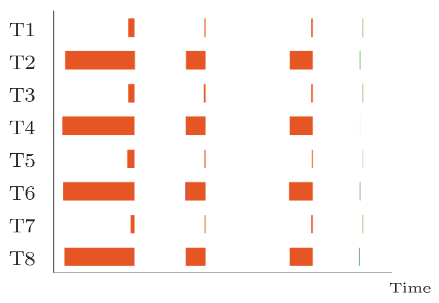
(b) Aborts in *Slow* versions of normal and pipelined CG

Fig. 2. Aborts with normal and pipelined CG with the number of threads ranging from 1 to 24 on W1.

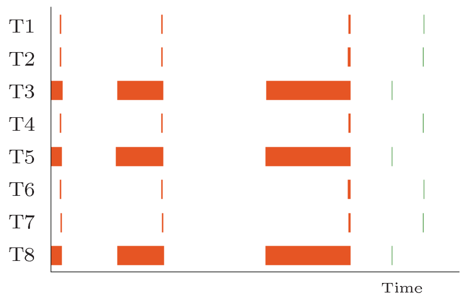
operations multiple times contributes a larger share of integer operations than a transaction that successfully commits. Hence, large abort rates may change the utilization of the functional units. Figure 2 depicts the absolute number of aborted transactions of all threads for the *Fast* (left hand side) and the *Slow* variants (right hand side) of normal CG and pipelined CG. All of the presented numbers are averages over 17 runs.

For the *Fast* version, illustrated in Figure 2(a), the number of aborts is below 100 for up to 16 threads and normal CG and pipelined CG. With 24 threads, it rises to ≈ 340 for normal CG and ≈ 1800 for pipelined CG. Here, this is the only configuration for the *Fast* versions where normal CG has significantly less aborts than pipelined CG. This is remarkable because pipelined CG executes three times the number of loads and stores per transaction and herewith should have a higher probability of conflict. The fact that all of these transactions access the same three variables in the same order leads to a scenario where a transaction will conflict with another transaction if they both run at the exact same time. As pipelined CG requires more time to execute the longer transactions, this increases the conflict probability because a longer time inside a transaction although means a longer time in which a second thread may start a transaction and conflict with the former. This effect is dominating only at 24 threads because prior to that, both versions of CG perform equal. The relative abort rate for *Fast* with 8 threads is $\approx 3.5\,\%$ for normal CG and $\approx 6\,\%$ for pipelined CG.

Figure 2(b) demonstrates the reason for the missing performance with the *Slow* variants. For only 2 threads, normal CG already has $\approx 460 * 10^6$ aborts. For pipelined CG, the aborts for 2 threads are $\approx 440 * 10^6$ for the short and $\approx 687 * 10^6$ for the long variant. The reason for these high aborts are the transactions that update a single variable (or in the best case three variables) for each iteration of the loop. Due to these high abort numbers, the threads will not make progress and, hence have long execution times and poor performance with the *Slow* variants.



(a) Visualization of normal CG



(b) Visualization of pipelined CG

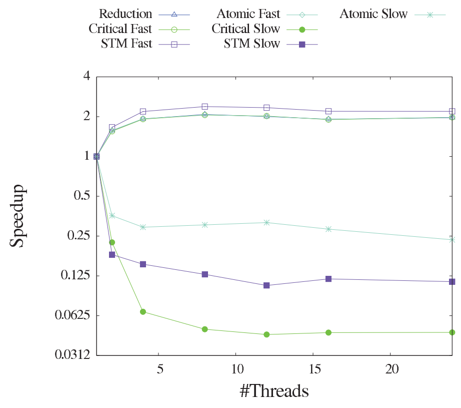
Fig. 3. Visualization of normal and pipelined CG with 8 threads on W1. Zoomed to relate transaction time (in green) with barrier wait time (in orange).

After studying the transactional characteristics in the previous paragraph, we would like to demonstrate the additional values and the flexibility of the VisOTMA framework by doing an in-depth analysis of the *Fast* versions of normal CG and pipelined CG. When first visualizing the TM application behavior with Paraver, we found many gaps between extremely small transactions. Thus, only a high zoom level would allow us to find the aborted transactions. After investigating these cases and finding that the overall TM performance for *Fast* is good (also cf. to previous paragraph), we decided to focus on the blank spots between the transactions. A code study reveals that, apart from computation, OpenMP constructs are most likely to consume the missing time in between the transactions. Both CG variants comprise 5 OpenMP `for` loops. By default these `for` loops come with an implicit barrier at the end of the execution. Thus, the fastest thread waits for the slowest one to complete its work and reach the barrier. OpenMP enables the programmer to specify the `nowait` clause to omit this barrier [4]. On the other hand, there is also the explicit `#pragma omp barrier` construct that produces a barrier. These manifold possibilities to generate barriers in OpenMP code and the importance for CG code convinced us to investigate the barrier wait times to relate these to the transaction execution times. We obtain the information about TM events through a tracing machinery [12] that features low intrusiveness and also access to hardware performance counters through PAPI [15]. The information is analyzed in a post-processing step and transformed into traces for the Paraver[4] tool for visualization. In our particular setup using the GCC compiler, `libgomp` is the OpenMP run time system and GCC does the expansion of OpenMP pragmas and the outlining of functions. In order to implement timed barriers, we needed to intercept the call to the original barrier function with one that would record the cycle counter of the processor on entry and exit of the barrier. These readings are then written to a thread-local trace file. Using the timed instead of the regular barriers is achieved through a simple replacement on assembly level. Through simply replacing the call to

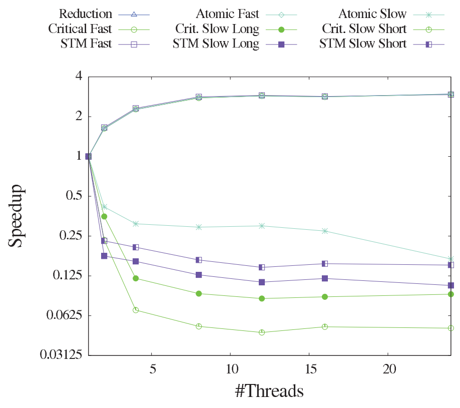
[4] Paraver Website http://www.bsc.es/paraver.

GOMP_barrier with a call to *ote_GOMP_barrier*, we achieved the desired functionality. Thus, *ote_GOMP_barrier* records the cycle counter before and after calling *GOMP_barrier*. Separate additional trace files for tracing these barriers are necessary because barriers are independent of executing transactions and the STM may not be initialized when calling a barrier. Thus, a post-processing step merges the barrier traces with the TM traces. Both trace files have the same time base and, hence, correlate in time. The visualization of the merged traces requires to register the new events at the various processing stages, but is straightforward. Figure 3 shows results of this effort for normal CG and pipelined. The picture is a timeline view of barriers and transactions executed by 8 threads. The threads, denoted with `T1` to `T8`, each occupy a slot on the y-axis. The x-axis shows the progress of time. The orange bars demonstrate the wait time of a particular thread at a barrier. These orange bars dominate Figure 3(a). Green bars illustrate how much time the execution of a transaction with a commit takes. These bars are present on the right hand side of Figure 3(a) and are extremely small. Figure 3(b) illustrates the run time behavior of the pipelined CG variant that exercises a similar execution pattern. Again, transaction times are hardly visible although these transactions have three times the amount of loads and stores of those transactions in normal CG. Additionally, we discover that pipelined CG shows a small perturbation that influences the start time of the transactions. In Figure 3(a) with normal CG all threads start their transaction at almost exactly the same point in time, whereas Figure 3(b) reveals that three threads start executing the transaction before all other threads. This behavior of pipelined CG is likely the cause for a better conflict rate than expected. Surprisingly, both figures highlight that the wait times at the barriers (colored in orange) exceed the execution time of a transaction (shown in green). These findings not only motivate research to avoid or omit these barriers, but also show that in a very regular setting, such as with the CG algorithm, TM can not show its strong side because the effects of optimistic concurrency, which enable some threads to proceed faster than others, are potentially turned into wait times at the barriers, waiting for the slowest thread that has been aborted to enable the progress of the fast threads.

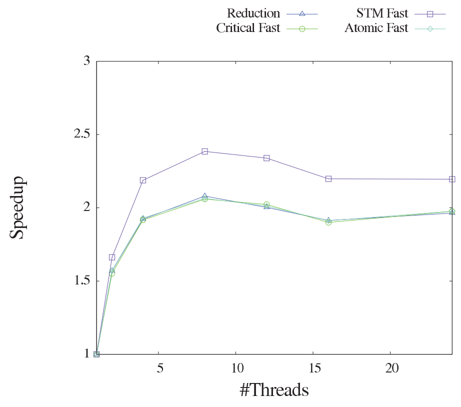
Speedup – We compare the achieved speedup of normal CG with that of pipelined CG. The speedup is computed according to $S(n) = \frac{T(1)}{T(n)}$, where $T(n)$ denotes the execution time with n threads and $T(1)$ is the respective single thread execution time (cf. to [7]). Often T_{seq} is used instead of $T(1)$ with T_{seq} being the serial reference implementation that does not incur the overheads of a threaded implementation. In these cases, often $T_{seq} < T(1)$ holds so that the resulting $S(n)$ would be smaller. Figure 4(a) depicts the speedup for normal CG whereas Figure 4(b) shows it for pipelined CG (both times on the y-axis). The x-axis holds the number of threads. Although the plots of the runtimes from previous sections contain the same information, this plot more evidently shows a slow down (speedup < 1) for the slow variants and a speedup for the fast variants. Setting the scale of the y-axis is a compromise to fit all variants on one plot. This makes identifying the maximum achieved speedup difficult because of the low resolution in this segment. Therefore, a second plot focuses on display-



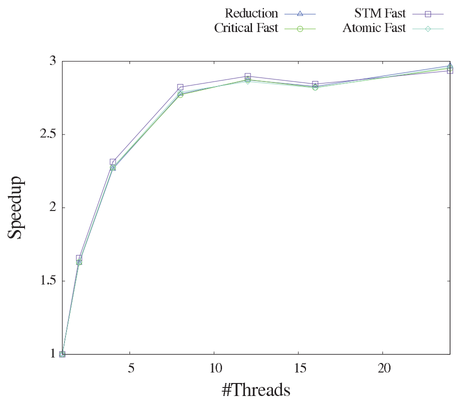
(a) Speedup of normal CG



(b) Speedup of pipelined CG



(c) Speedup of normal CG for fast versions only.

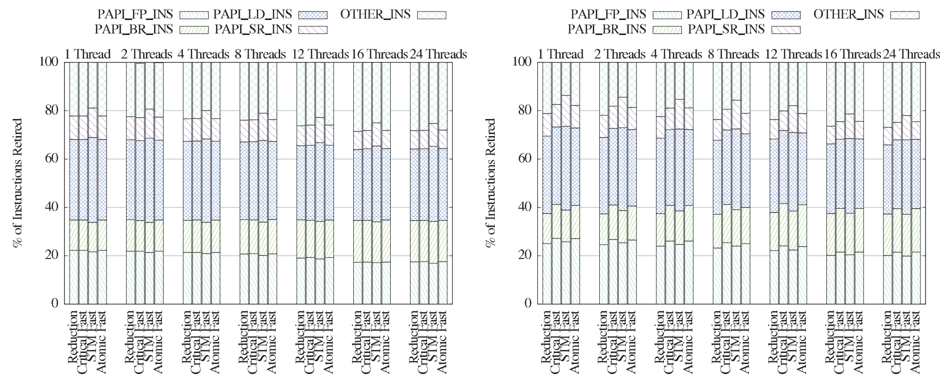


(d) Speedup of pipelined CG for fast versions only.

Fig. 4. Speedup with normal and pipelined CG with the number of threads ranging from 1 to 24 on W1.

ing the results of the fast variants only. Figure 4(c) and Figure 4(d) show the speedup with a linear scale on the y-axis. This plot illustrates that the achievable speedup over the respective single thread performance is higher with pipelined CG, achieving the highest speedup of 2.97 with the regular reduction and 24 threads. For normal CG, *STM Fast* with 8 threads achieves the best speedup of 2.38. The overhead of the single thread execution has a large influence on the reported speedups because a larger overhead (e.g., with STM) leads to a slower execution time. If the speedup is computed relative to this single thread execution time, this yields a higher speedup because the baseline is worse. This effect could be avoided by having a fixed serial execution time for all benchmarked variants. Here, this effect leads to the situation that *STM Fast* has a higher speedup for e.g., pipelined CG with 8 threads, but a higher execution time than e.g., *Reduction*.

Convergence Behavior – This paragraph presents the results of examining the convergence behavior of normal CG and pipelined CG applied to a problem that solves the stationary heat equation without heat source. The parameters setting is identical with the one that has been shown in Table 1. Both variants of CG show a consistent convergence behavior across all tested thread counts and synchronization mechanisms. Normal CG converges after 25 iterations to a solution that satisfies the criteria. Pipelined CG finds a solution to the problem that satisfies the convergence criteria after 26 iterations. Therefore, for the numerical problem solved in this experiment, the choice of the algorithmic variant has an impact on the convergence behavior. Pipelined CG needs to perform one additional iteration which is equal to a relative increase of computational complexity of 4 % for the considered problem.



(a) Normal CG (b) Pipelined CG

Fig. 5. Breaking down the total amount of instructions in CG on W1.

Which instruction type contributes the largest share? Figure 5 shows a breakdown of instructions retired into the measured type of instructions. The available types are: floating point instructions (denoted as PAPI_FP_INS), branch instructions (labeled PAPI_BR_INS), load and store instructions (PAPI_LD_INS and PAPI_SR_INS respectively) and remaining instruction with the label OTHER_-INS. Other instructions have not been measured, but computed as the difference from the measured ones with the remainder of the retired instructions (PAPI_TOT_INS). The Figure has a normalized y-axis that shows 100 % of the retired instructions. Each of the instruction types has a box that represents its share of the retired instructions. These bars are grouped according to the thread count and each group shows the used synchronization mechanisms (*Reduction*, *Critical Fast*, *STM Fast* and *Atomic Fast*). The number of threads for each group is also found below the legend. Figure 5(a) shows the breakdown for normal CG and Figure 5(b) shows pipelined CG. For both the following trends can be derived: the share of floating point instructions decreases as the number of threads increases although the actual number of floating point instruction is constant. This is due to the fact that the number of other instructions increases as the number of threads increases. These additional instructions stem from the spawning/coordinating more threads. For normal CG and *Reduction* the share of FP instructions decreases from 22 % for 1 thread down to 17 % for 16 threads.

Pipelined CG and *Reduction* yields similar numbers: the FP rate decreases from 25 % for 1 thread to 20 % for 16 threads. To summarize Figure 5, we conclude that loads and stores contribute the highest share to the retired instructions ($\approx 40\,\%$), floating point instructions contribute $\approx 20\,\%$ and branch instructions $\approx 15\,\%$ while other instructions contribute $\approx 25\,\%$ and become increasingly important with larger thread counts. The actual number of events varies depending on the synchronization variants, thread count and algorithmic choice, but the dominant instructions in our experiments have been loads and stores.

3 Related Work

Different possibilities to realize the concept of Transactional Memory have been proposed for Software, Hardware, or both [6]. The publications on tool support for TM are rare. For Software Transactional Memory profiling solutions have been shown for the programming languages `C#` [16], Haskell [13], `Java` [1], and `C` [9, 2] and for Hardware Transactional Memory for the TCC architecture [3].

In particular, none of these approaches attempted to optimize and evaluate a numerical algorithm through selecting, implementing, and evaluating a differently formulated variant of the algorithm that promised a higher TM performance. Hence, this work with its insights into the run time behavior and utilization of the microarchitecture through the two CG variants advances the state-of-the-art and may inspire other researchers that face the problem of optimizing a numerical algorithms that uses/with TM.

4 Findings with Variants of CG and Outlook

Our first finding is that the right way of organizing the reductions is key to performance. A reduction implemented with direct updates of the shared variable, as seen in the *Slow* synchronization variants, will not yield a speedup over execution with one thread regardless of the synchronization primitive. Instead thread-local variables that hold intermediate results, as demonstrated with *Fast* synchronization variants, are a requirement to achieve speedups. Moreover, the pipelined CG with larger transactions is a strong competitor for normal CG because the number of aborts is modest up to 16 threads. As a downside, pipelined CG required one more iteration to achieve convergence compared with normal CG for our example case. For both CG variants, the wait time at the barriers dominates the time for synchronization in the reduction operations of the *Fast* variants. This does not only limit the gains of parallel execution but also masks the effects of optimizing the TM performance. The regular problem structure of CG demands that barriers synchronize all threads after a step in the loop. Thus, a thread that executes a transaction and forces another thread to abort and execute again, simply waits longer at the next barrier for the remaining threads. This basic scenario still holds for longer transactions with pipelined CG and pipelined CG achieves a higher speedup than normal CG. As a result, the CG algorithm is not suited to demonstrate a performance gain with STM. On the other hand, the competitive execution time of pipelined CG with larger transactions and still moderate contention confirms the basic idea of optimizing the TM behavior through employing larger transactions. Moreover, the large difference in execution time for transactions and barriers suggests that future research should target more efficient barrier synchronization or techniques to elide barriers. Common to both CG variants, we found that higher thread counts lead to more L2 cache misses that hinder the scalability and that loads and stores contribute the largest amount to all kinds of instructions retired.

Future work should integrate the profiling of transactions with an existing profiler for OpenMP applications, e.g., `ompP` [5], in order to complement the time spent in transactions and barriers with other OpenMP constructs such as parallel sections and thread create or destroy. These measurements would complement the performance analysis and a programmer could relate the overheads associated with STM to those of OpenMP in general.

References

1. Ansari, M., Jarvis, K., Kotselidis, C., Luján, M., Kirkham, C., Watson, I.: Profiling Transactional Memory Applications. In: PDP '09, Washington, DC, USA, IEEE Computer Society (2009) 11–20
2. Castro, M., Georgiev, K., Marangozova-Martin, V., Mehaut, J.F., Fernandes, L.G., Santana, M.: Analysis and Tracing of Applications Based on Software Transactional Memory on Multicore Architectures. In: PDP '11, Washington, DC, USA, IEEE Computer Society (2011) 199–206
3. Chafi, H., Minh, C.C., McDonald, A., Carlstrom, B.D., Chung, J., Hammond, L., Kozyrakis, C., Olukotun, K.: TAPE: A Transactional Application Profiling Environment. In: ICS '05, New York, NY, USA, ACM (2005) 199–208
4. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering **05**(1) (1998) 46–55
5. Fuerlinger, K.: OpenMP Profiling with OmpP. In Padua, D., ed.: Encyclopedia of Parallel Computing, Springer US (2011) 1371–1379
6. Harris, T., Larus, J., Rajwar, R.: Transactional Memory. Volume 5. Morgan & Claypool Publishers (June 2010) 2nd edition, Synthesis Lectures on Computer Architecture.
7. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. 5th edn. Morgan Kaufmann Publ. Inc., San Franc., CA, USA (Oct. 2011).
8. Janko, S., Rocker, B., Schindewolf, M., Heuveline, V., Karl, W.: Transactional Memory, OpenMP and Pthreads applied to the CG Algorithm for solving the stationary Heat Equation. In: VECPAR. (July 2012) appears at Springer Verlag.
9. Lourenço, J., Dias, R., Luís, J., Rebelo, M., Pessanha, V.: Understanding the Behavior of Transactional Memory Applications. In: PADTAD '09, New York, NY, USA, ACM (2009) 3:1–3:9
10. Saad, Y.: Iterative Methods for Sparse Linear Systems. 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2003)
11. Schindewolf, M., Bihari, B., Gyllenhaal, J., Schulz, M., Wang, A., Karl, W.: What Scientific Applications Can Benefit from Hardware Transactional Memory? In: SC '12, Los Alamitos, CA, USA, IEEE Computer Society Press (2012) 90:1–90:11
12. Schindewolf, M., Karl, W.: Capturing Transactional Memory Application's Behavior – The Prerequisite for Performance Analysis. In: MSEPT 2012. Lecture Notes in Computer Science, Vol. 7303, Springer Verlag (May 31–June 1, 2012) 30–41
13. Sonmez, N., Cristal, A., Unsal, O., Harris, T., Valero, M.: Profiling Transactional Memory Applications on an atomic block basis: A Haskell case study. In: MULTIPROG 2009. (January 2009)
14. Strzodka, R., Göddeke, D.: Pipelined Mixed Precision Algorithms on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components. In: FCCM 2006, IEEE Computer Society Press (May 2006) 259–268 doi: 10.1109/FCCM.2006.57.
15. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting Performance Data with PAPI-C. In: Tools for High Performance Computing 2009, Springer Berlin Heidelberg (2010) 157–173
16. Zyulkyarov, F., Stipic, S., Harris, T., Unsal, O.S., Cristal, A., Hur, I., Valero, M.: Discovering and Understanding Performance Bottlenecks in Transactional Applications. In: PACT '10, New York, NY, USA, ACM (2010) 285–294