Parallelize the Runtime Checks – Not the Application

Martin Süßkraut Stefan Weigert Martin Nowack Diogo Becker de Brum Christof Fetzer
TU Dresden
Computer Science Department
01062 Dresden, Germany
{suesskraut, stefan, martin, diogo, christof}@se.inf.tu-dresden.de

May 15, 2009

1 Introduction

Sequential and parallel applications are both prone to security and dependability bugs. Compilers can reduce the impact of these bugs by instrumenting runtime checks into the generated code. These runtime checks can have a dramatic negative impact on the performance of an application. For instance, our measurements show that compiler generated array-bounds checks can increase the application's runtime by 20x. To make compiler generated checks practically usable, the introduced application slowdowns must be decreased. Single thread performance is only expected to grow slowly. To compensate these single thread slowdowns, it is very desirable to parallelize checked applications and/or their runtime checks.

In this paper, we provide evidence that it is more promising to parallelize the runtime checks than the application: We therefore compare two frameworks using two different parallelization approaches: (1) parallelizing the application together with the runtime checks, and (2) parallelizing only the runtime checks.

We focus on two frameworks that were developed in our group:

Tanger The first implementation is the compiler extension Tanger [2] for software transactional memory (STM). Tanger puts every memory access within transactions under the control of the TinySTM++ library [3]. To apply this approach, an application must be parallelizable with transactional memory.

ParExC ParExC (*Par*allel *Ex*ecution *C*hecking) parallelizes the runtime checks but not the application itself. Therefore, it is also suited for applications that are difficult to parallelize. ParExC executes the application without any runtime checks in the so called *predictor* process. The predictor's execution is partitioned into epochs. Each epoch is replayed by an *executor* process, but this time with runtime checks. We scale by running the executors in parallel to each other and the predictor. Like Speck [5], ParExC allows a fast execution of the application by speculating on the success of the runtime checks. Speck parallelizes using dynamic binary instrumentation. Unlike Speck, ParExC parallelizes compiler generated checks at compile time.

Roadmap In Sections 2 and 3, we introduce the two approaches in more details. Our experiments in Section 4 show that the specialized ParExC approach scales better than applications instrumented by Tanger for a checker with heavy runtime overhead if at-most 8 cores are available.

2 Tanger

Transactional memory is an optimistic concurrency control mechanism for multi-threaded applications. It intends to simplify concurrent programming and eliminates several drawbacks of other synchronization mechanisms, in particular, locks.

A transaction is a set of statements that are executed atomically. It either commits or undoes its changes to the shared data if a conflict with another transaction occurs. To detect conflicts and perform roll-back, software trans-

actional memory systems require all shared data accesses within a transaction to be properly instrumented. This instrumentation incurs synchronization and book-keeping overhead, which is out-weighted by the ease of writing concurrent applications and the resulting performance improvements.

In our experiments, we used TinySTM++, a *C++* version of the TinySTM [3]. Instead of instrumenting the code by hand, we used Tanger [2]. Tanger is an extension for the LLVM [4] compiler framework that automatically transactifies applications. The programmer only has to mark the start and the end of the transactions. The instrumentation delegates all shared data accesses in these regions to an STM, in our case, TinySTM++.

Although an experienced programmer might exploit optimizations by manually transactifying the application, we expect that most existing software will not be manually transactified.

We have a two-step procedure to parallelize the application with runtime checks. First, we apply the same compiler transformation of the checker used in ParExC. Second, we apply the Tanger transformations on the code resulting from the first step.

3 ParExC

ParExC reduces the performance impact of expensive runtime checks by parallelizing them (see Fig. 1). At compile time, runtime checks are inserted into the original application. These checks (see Fig. 1 (b)) introduce, in general, substantial overhead (Fig. 1 (a)). ParExC executes the application without any runtime checks in the so called *predictor* process (Fig. 1 (c)). The predictor's execution is partitioned into *epochs*. The goal of the predictor is to predict the state of the application at the start of each epoch. An *executor* process re-executes the epoch – this time with the runtime checks. Executors run on additional cores – in parallel to each other and to the predictor. In contrast to them, the predictor's code does not contain runtime checks, i.e., the predictor does not even need to test if a certain runtime check needs to be executed.

We have designed and implemented the *ParExC* framework to support the development of checkers. ParExC is implemented on top of the open compiler framework LLVM [4]. Our approach for replaying the epochs is similar to flashback [6].

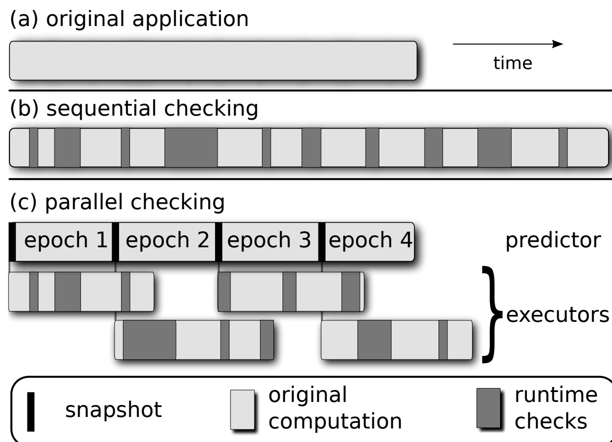

Figure 1: (a) Original application execution without checks. (b) Sequential execution with checks. (c) Parallelized execution with checks. Unchecked predictor for fast state prediction. Parallelized executors including slow checks.

Runtime checks often need to do book-keeping, e.g., an overflow checker needs to track the sizes of allocated arrays. The executors of one checker need to share this book-keeping state. We introduce *speculative variables* to decouple the executors as far as possible. For example, consider a bounds checker that checks that memory accesses do not overflow array bounds. Its book-keeping records which memory areas (base address and size) are allocated. Consider that executor E_i of epoch e_i needs to check a memory access to an array a that is allocated in the preceding epoch e_{i-1}. The book-keeping of E_i does not contain this array's size since E_i has not seen a's allocation. Instead of blocking E_i until its predecessor E_{i-1} has obtained this information, E_i continues speculatively - assuming that a is sufficiently large. After E_{i-1} has successfully finished checking e_{i-1}, executor E_i is able verify its assumption.

4 Experiments

We compared both approaches by applying them to the Genome and Vacation applications of the STAMP benchmark suite [1]. As checker, we implemented an out-of-

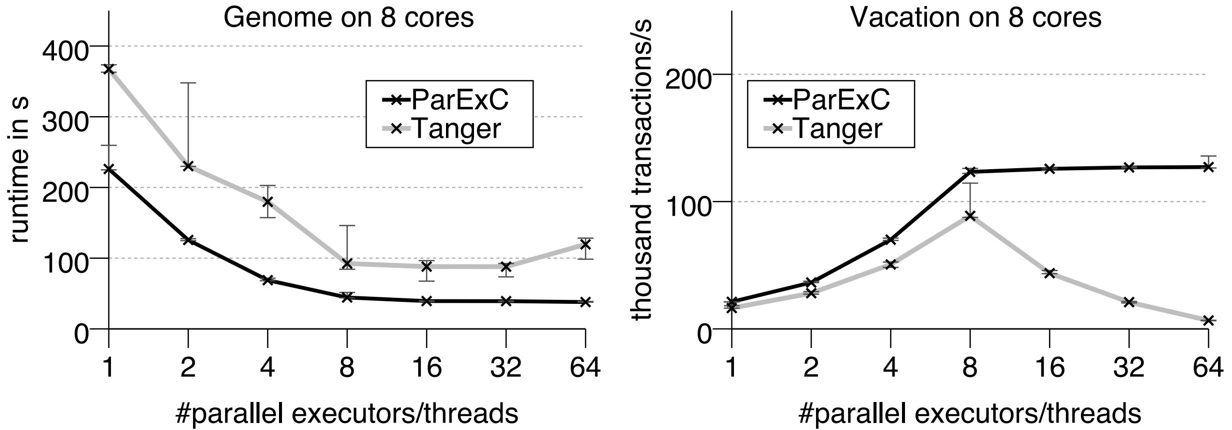

Figure 2: Comparison of ParExC and Tanger instrumented applications: the runtime of `Genome -t1 -g16384 -s64 -n16777216` (left) and throughput of `Vacation -n4 -q60 -u90 -r16384 -t10000000` (right).

bounds checker.

Out-Of-Bounds Our *out-of-bounds checker* (OOB) detects buffer overflow bugs. Buffer overflow bugs are a common security violation in—so called—unsafe languages like C/C++. The checker implements a fail-stop failure model, i.e., it aborts the application as soon as it detects a buffer overflow. The OOB checker instruments all memory allocation code for allocations on the heap. So, at runtime it can keep track of all currently allocated buffers. In LLVM, memory is only accessed with explicit load and stored instructions. Loads, stores, and address computations are also instrumented. Address computation is statically identifiable in LLVM since a special instruction is used. By instrumenting the address computations, the checker can identify the buffer accessed by a load or a store at runtime. For each load and store the used offset is checked against the allocated size of the identified buffer. If the offset is larger than the allocated size the program is aborted.

At runtime, the OOB checker keeps track of the array sizes. With ParExC, the checker synchronizes the access to this book-keeping state with speculative variables, as introduced in Section 3. In contrast, with Tanger/TinySTM++, every access to the book-keeping state is performed inside a transaction, therefore, incurring additional overhead.

Measurements We conducted the experiments, shown in Figure 2, using a Dell PowerEdge1950 with a dual Intel Xeon E5430 processor (8 cores) and 16GB of RAM. Every measured value represents the median over 5 runs. We used the *real* time, reported by the "time" command to measure the runtime. The error bars show the minimum and the maximum of our measurements, respectively.

The STAMP benchmarks are explicitly written for benchmarking STM libraries. Therefore, all shared memory accesses are explicitly marked. However, as we focus on as few manual changes as possible, Tanger ignores these markings (except the transaction boundaries) and puts all potentially shared memory accesses under the control of TinySTM++. If, instead, only TinySTM++ would be used, different results are expected, but this would additionally involve the manual transactification of the OOB checker, what we want to avoid.

We also measured the percentage of application runtime executed sequentially, in particular the startup and the cleanup phase of both benchmarks without any instrumentation. In Vacation, only 0.16% of the execution runs sequentially. In contrast, Genome's startup phase takes very long, i.e., 14.3%. While ParExC can parallelize the checks of this phase, Tanger cannot.

Discussion For both benchmarks, the sequential overhead is higher with the Tanger instrumentation than with

the ParExC instrumentation. In contrast to applications instrumented with ParExC, those instrumented with Tanger need to acquire and check locks for every memory access within transactions *and* in the OOB runtime library.

We can see in Figure 2 that both benchmarks scale better when instrumented with ParExC than when instrumented with Tanger. Vacation instrumented by Tanger actually scales worse if more threads are used than cores are available. One reason could be increased contention, what would lead to higher abort rates. Another reason could be the lack of transaction scheduling by TinySTM++. Both issues do not arise using the ParExC approach. First, there is no contention between executors. Second, executors are scheduled by the OS transparently. However, more measurements are necessary to clarify this issue.

Since only the checks are parallelized, the application parallelized using the ParExC framework cannot be faster than the original application. Therefore, if the workload is easily parallelizable, and enough cores are available, the Tanger approach could result in better scaling applications. On the other hand, the ParExC approach also works with applications or parts of applications that are difficult to parallelize, as long as heavy runtime checks have to be applied.

5 Conclusion

Our experiments suggest that, first, the overhead of runtime checks can be addressed by either parallelizing the application with runtime checks or by parallelizing the runtime checks exclusively. Second, using the specialized ParExC approach, applications scale better for a low number of available cores as found on current computers.

Our hypothesis is that for applications instrumented with heavy-weight runtime checks it is more promising to parallelize the runtime checks than the application itself. When the number of cores is getting larger than the slowdown introduced by the runtime checks, we expect that STMs will eventually do better for applications that scale well with the number of cores.

References

[1] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[2] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying applications using an open compiler framework. In *TRANSACT*, August 2007.

[3] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[4] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, California, 2004.

[5] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. *SIGARCH Comput. Archit. News*, 36(1):308–318, 2008.

[6] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, USA, 2004. USENIX Association.