Support for Thread-level Speculation into OpenMP

Sergio Aldea, Diego R. Llanos, and Arturo González-Escribano

Departamento de Informática, Universidad de Valladolid,
Campus Miguel Delibes, 47011 Valladolid, Spain
{sergio,diego,arturo}@infor.uva.es

Abstract. Software-based, thread-level speculation (TLS) systems allow the parallel execution of loops that can not be analyzed at compile time. TLS systems optimistically assume that the loop is parallelizable, and augment the original code with functions that check the consistency of the parallel execution. If a dependence violation is detected, offending threads are restarted to consume correct values. Although many TLS implementations have been developed so far, robustness issues and changes required to existent compiler technology prevent them to reach the mainstream. In this paper we propose a different approach: To add TLS support to OpenMP. A new OpenMP *speculative* clause would allow to execute in parallel loops whose dependence analysis can not be done at compile time.

Keywords: TLS systems, speculative parallelization, OpenMP

1 Introduction

Manual development of parallel versions of existent, sequential applications requires an in-depth knowledge of the problem, the architecture, and the parallel programming model. On the other hand, using automatic parallelization mechanisms we can only extract parallelism from a small fraction of loops, decided at compile time.

The most promising runtime technique to extract parallelism from fragments of code that can not be analyzed at compile time is called software-based Speculative Parallelization (SP). This technique, also called Thread-Level Speculation (TLS) [2, 4, 5] or even Optimistic Parallelization [6, 7] aims to automatically extract loop- and task-level parallelism when a compile-time dependence analysis can not guarantee that a given sequential code is safely parallelizable. SP optimistically assumes that the code can be executed in parallel, relying on a runtime monitor to ensure correctness. The original code is augmented with function calls that distribute iterations among processors, monitor the use of all variables that may lead to a dependence violation, and perform in-order commits to store the results obtained by successful iterations. If a dependence violation appears at runtime, these library functions stop the offending threads and re-starts them in order to use the updated values, thus preserving sequential semantics.

The purpose of this paper is to discuss how to add SP support into OpenMP. Parallel applications written with OpenMP should explicitly declare parallel regions of code. In the case of parallel loops, the programmer should classifies all variable used inside the loop, according to their use, in "private", or "shared". This task is extremely difficult when the parallel loop consists of more than a few dozen lines.

To help the programmer in the development of a parallel version of a sequential loop, our proposal is to offer a new "speculative" clause. This clause would allow the programmer to handle variables whose use can potentially lead to a dependence violation, and therefore should be monitored at runtime in order to obtain correct results. Note that the use of such a category effectively frees the programmer from the task of deciding whether a particular variable is private or shared. To the best of our knowledge, no production-state parallel programming model incorporates support for thread-level speculation.

Our research group has worked for a decade in the field of software-based speculative parallelization. The research carried out so far have led to both a production-level SP runtime library [3] and a prototype of a SP compiler framework [1]. We believe that adding support for speculative parallelization in OpenMP will help to reduce the intrinsic difficulties of manual parallelization of existent code. If successful, parallel code will be much easier to write and maintain.

2 Our proposal

We have developed a software-only TLS system [2] that has proven its usefulness in the parallel execution of loops that can not be analyzed at compile time, both with and without dependence violations [3].

Our TLS system is implemented using OpenMP for thread management. The loop to be parallelized is transformed in a loop with as many iterations as available threads. At the beginning of the loop body, a scheduling method assigns to the current thread the block of iterations to be executed. Read and write operations to the speculative structure are replaced at compile time with `specload()` and `specstore()` function calls. `specload()` obtains the most up-to-date value of the element being accessed. `specstore()` writes the datum in the version copy of the current processor, and ensures that no thread executing a subsequent iteration has already consumed an outdated value for this structure element, a situation called "dependence violation". If such a violation is detected, the offending thread and its successors are stopped and restarted. Finally, a `commit_or_discard()` function is called once the thread has finished the execution of the chunk assigned. If the execution was successful, the version copy of the data is committed to the main copy; otherwise, version data is discarded.

From the programmer point of view, the structure of a loop being speculatively parallelized is not so different from a loop parallelized with OpenMP directives. Current OpenMP parallel constructs force the programmer to explicitly declare the variables used into the parallel region according to their use,

which can be an extremely hard and error-prone task if the loop has more than a few dozen lines.

The problem of adding speculative parallelization support to OpenMP can be handled from two points of view. One requires the addition of a new directive, for example `pragma omp speculative for`. However, this option demands more effort, because there are many OpenMP related components that should be modified. We believe that it is preferable to use a different approach to add a new clause to current parallel constructs that allows the programmer to enumerate which variables should be updated speculatively. The syntax of this clause would be

speculative(variable[, var_list])

In this way, if the programmer is unsure about the use of a certain structure, he could simply label it as speculative. In this case, the OpenMP library would replace all definitions and uses of this structure with the corresponding `specload()` and `specstore()` function calls. An additional `commit_or_discard()` function should be automatically invoked once each thread has finished its chunk of iterations, to either commit the results, or restart the execution if the thread has been squashed.

In order to integrate our TLS system, already written using OpenMP, into an experimental OpenMP implementation that also supports speculative parallelization, the particularities of our TLS system should be taken into account. For example, our TLS system needs to set its own control variables as *private* and *shared*. This implies that, if a *speculative* clause is found by the compiler, declaring that there are variables that should be handled speculatively, the use of our TLS system to guide the speculative execution needs to add several *private* and *shared* variables to the current lists. Fortunately, OpenMP allows the repetition of clauses, so the implementation of this new *speculative* clause may add additional *private* and *shared* clauses that will later be expanded by the compiler.

There are two additional issues to be considered. First, the current scheduling methods implemented by OpenMP are not enough to handle speculative parallelization. These methods assume that the task will never fail, and therefore they do not take into account the possibility of restarting an iteration that has failed due to a dependence violation. Therefore, it is necessary to use an speculative scheduling method. This method assigns to each free thread the following chunk of iterations to be executed. If a thread has successfully finished a chunk, it will receive a brand new chunk not been executed yet. Otherwise, the scheduling method may assign to that thread the same chunk whose execution had failed, in order to improve locality and cache reutilization.

We have already developed both Fortran and C versions of our TLS system. Since implementation of OpenMP for C, C++ and Fortran only differs in their respective front ends, adding TLS support for a different language should not require to modify the middle or the back end.

3 Conclusions

Adding speculative support to OpenMP would greatly increase the number of loops that could be parallelized with this programming model. The programmer may label some of the variables involved as *private* or *shared*, using *speculative* for the rest. With this approach, in the first parallel version of a given sequential loop, the programmer might simply label all variables as *speculative*. Of course, the execution of such a loop would lead to an enormous performance penalty, since all definitions and uses of all variables would have been transformed into `specload()` and `specstore()` function calls, that will not perform any useful task if the variables are indeed private or read-only shared. Note that our proposal would let to transform *any* loop into a parallel loop, although the parallel performance will depend of the number of dependence violations being triggered at runtime. The approach described here is being currently implemented.

Acknowledgments. This work has been partially supported by MICINN (Spain) and the European Union FEDER (CENIT OCEANLIDER, CAPAP-H3 network, TIN2010-12011-E, TIN2011-25639), and the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures Initiative. Sergio Aldea is supported by a research grant of Junta de Castilla y León, Spain.

References

1. Sergio Aldea, Diego R. Llanos, and Arturo Gonzalez-Escribano. Towards a compiler framework for thread-level speculation. In *PDP '11*, pages 267–271. IEEE, February 2011.
2. Marcelo Cintra and Diego R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *ACM PPoPP '03*, June 2003.
3. Marcelo Cintra and Diego R. Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE TPDS '05*, 16(6):562–576, 2005.
4. Francis Dang, Hoo Yu, and Lawrence Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *IEEE IPDPS '02*, April 2002.
5. Manish Gupta and Rahul Nim. Techniques for speculative run-time parallelization of loops. In *Proc. of the 1998 ACM/IEEE Conference on Supercomputing*, pages 1–12, 1998.
6. Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Optimistic parallelism benefits from data partitioning. In *ACM ASPLOS '08*, pages 233–243, Seattle, WA, USA, 2008. ACM.
7. Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *ACM PLDI '07*, pages 211–222, San Diego, California, USA, 2007. ACM.