Adapting the Polyhedral Model as a Framework for Efficient Speculative Parallelization

Alexandra Jimborean Philippe Clauss Benoît Pradelle
Luis Mastrangelo Vincent Loechner

CAMUS group, INRIA & LSIIT & University of Strasbourg
Strasbourg
France
first_name.last_name@inria.fr

Abstract

In this paper, we present a Thread-Level Speculation (TLS) framework whose main feature is to be able to speculatively parallelize a sequential loop nest in various ways, by re-scheduling its iterations. The transformation to be applied is selected at runtime with the goal of minimizing the number of rollbacks and maximizing performance. We perform code transformations by applying the polyhedral model that we adapted for speculative and runtime code parallelization. For this purpose, we designed a parallel code pattern which is patched by our runtime system according to the profiling information collected on some execution samples. Adaptability is ensured by considering chunks of code of various sizes, that are launched successively, each of which being parallelized in a different manner, or run sequentially, depending on the currently observed behavior for accessing memory.

We show on several benchmarks that our framework yields good performance on codes which could not be handled efficiently by previously proposed TLS systems.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – *Run-time environments, Optimization*

General Terms Performance

Keywords Speculative parallelization, dynamic system, polyhedral model, dynamic code transformations

1. Overview

With the advent of multicore processors, automatically parallelizing sequential code became increasingly important. Particularly, it is a challenging task to parallelize code at runtime, if the information available at compile time is not sufficient. Runtime parallelization techniques are usually based on thread-level speculation (TLS) [2–4], where straightforward parallelization transformations are optimistically applied on the original sequential code. In most TLS proposals, modest performance gains were obtained, since parallelization is attempted on unmodified code generated by the compiler: when considering loop nests, the unique strategy usually applied is to cut the outermost loop into contiguous chunks and to run these chunks separately in simultaneous threads, which yields numerous rollbacks in case of a dependence carried by the outermost loop. To be more efficient, TLS systems must handle more complex code transformations that can be profitably selected at runtime, depending on the current execution context. To our knowledge, no attempt on applying various parallelizing transformations using the TLS systems have been reported yet.

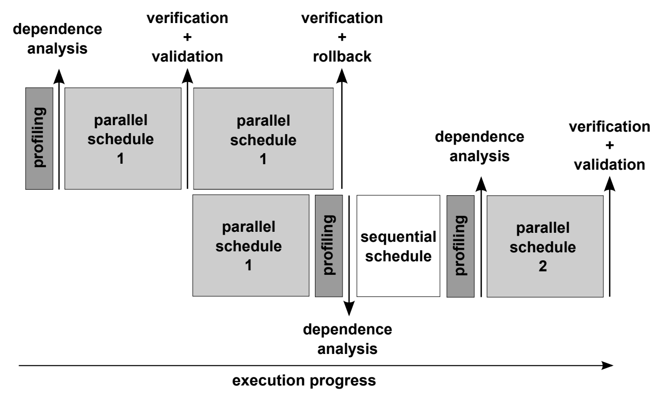

Figure 1. Illustration of the chunking mechanism

We propose performing advanced loop transformations provided by the polyhedral model [1] such as tiling, skewing, loop interchange, etc., by speculating on the linearity of the loop bounds and of the memory accesses. Not only that transforming code shows benefits in boosting performance, but also it exhibits parallelism in codes that cannot be otherwise parallelized in the original form. Additionally, our framework allows several parallel schedules, during one execution of the loop. We define a loop chunk as a set of consecutive iterations of the outermost loop and apply one parallel schedule per chunk.

Our goal is to parallelize the loop by chunks, by applying a suitable polyhedral transformation. We start by launching a profiling chunk dedicated to capturing the behaviour of the loop, based on which it can be decided whether a polyhedral transformation can be applied for parallelizing the loop. If the dependence analysis validates such a transformation, the corresponding code is generated and a new parallel chunk is launched. During the execution of the speculative parallel code, the speculation is verified and the chunk is validated, or a rollback is performed if a dependence violation occurs. In this case, the code is re-executed in a smaller chunk that completes its execution before the misprediction point, and another

Copyright is held by the author/owner(s).
PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

instrumentation chunk is launched which overcomes the rollback point and characterizes the new behavior of the loop. If the code cannot be parallelized, a sequential chunk is executed, followed again by an instrumented chunk. At some point, the loop might exhibit a new behavior allowing parallelism, provided that another polyhedral transformation is performed. On the other hand, if a parallel chunk is validated, execution continues with another parallel chunk of a larger size. This process is described in figure 1.

2. Implementation details

The framework consists of two parts: a *static* part, implemented in the LLVM compiler, designed to prepare the loops for instrumentation and parallelization, and a *dynamic* part, in the form of a runtime system whose role is to generate the parallel code and to guide the execution.

Static component: Loops are marked for speculative parallelization in the source code using a dedicated pragma. At compile time, these loops are automatically identified and three versions are generated: original, instrumented and a parallel code pattern, together with a mechanism for switching between the versions. Additionally, support for chunking the outermost loop is included.

The instrumented version contains instrumentation instructions which track the memory accesses performed inside the loop. The goal is to compute linear functions of the surrounding loop indices, used for performing the dependence analysis.

Instead of statically generating several parallel code versions, we build a generic code pattern from which these versions can be generated at runtime, by patching predefined code areas. The advantage is that the code size is significantly reduced and more parallel code versions can be build dynamically, guided by the results of the instrumentation. Also, patching a parallel code pattern is considerably faster than fully dynamic code generation. The limitation of the pattern is that it can only support a subset of the possible polyhedral transformations, which preserve the loop structure and do not reorder the statements.

In the parallel code pattern, the original loop is transformed into a *for*-loop with affine bounds and the original loop conditions are inserted as guarding code. Additionally, initialization code is inserted to assign the correct starting values for the variables at the beginning of each thread, using the linear functions obtained from instrumentation and a new polyhedral schedule. Correctness of the code is ensured by the verification code, which, for each memory access, compares the speculations against the actual values, at each iteration. The loop bounds, the initialization and the verification code, all use the linear functions computed during the profiling phase. Since they are not known at compile time, the coefficients are statically inserted as global variables, whose values will be assigned at runtime. Different values of these coefficients represent different schedules and will generate distinct parallel code versions.

A set of polyhedral transformations is proposed statically and is encoded in the binary file as matrices. Their computation follows a static dependence analysis, which ensures that the dependences which can be statically identified will not invalidate the schedules, minimizing the number of rollbacks.

Dynamic component: The runtime system collaborates tightly with the static component. During the instrumentation phase, it retrieves the memory locations being accessed and computes interpolating linear functions of the surrounding loop indices. Instrumentation is performed on loop samples, to limit the overhead, consequently the computed linear functions speculatively characterize the behavior of the loop. Instrumentation is followed by a dependence analysis which evaluates whether any of the proposed polyhedral transformations can be efficiently applied. If successful, the runtime system assigns values to the coefficients of the linear functions in the code pattern.

To limit the synchronization overhead, the validation system makes use of one flag per thread, which is set when a misprediction occurs. Each thread polls its own flag. As soon as a thread detects that a speculation is invalidated, it sets the flags of all threads. A misspeculation is followed by a rollback which restores the memory to a correct state. For this purpose, the runtime system creates a copy of the memory area that will be modified by the next parallel chunk, since it can be predicted using the interpolating linear functions. When a rollback is performed, the memory is overwritten with the content of the copy, and the rollbacked iterations are re-executed.

Results: We carried out experiments on synthetic benchmarks aimed to emphasize different characteristics of the framework. The "linked list" processes list elements allocated following either regular or irregular memory patterns. The "banded matrix" accesses elements of a matrix through indirect references whose linearity is discovered by our framework which then applies a parallelizing polyhedral transformation. The "cherry cake" performs different computations depending on properties of the elements of a linked list. It shows our system handling successively different parallel schedules. The "NO to overhead" evaluates the overhead of the instrumentation when no parallelization is possible.

Measurements were obtained by executing the benchmarks on 24 cores of two AMD Opteron 12 core-processors running Linux 2.6.35. Super-linear speed-up is obtained on the "banded matrix" thanks to the polyhdral transformation and loop tiling. Additionally, the memory accessing behavior remains unchanged and no rollbacks occur. In contrast, on the "linked list" and on the "cherry cake" examples, a number of rollbacks are performed, but their overhead is amorthized by polyhedral transformations and parallelization.

Benchmark	Sequential exec. time (s)	Spec. par. exec. time (s)	Speed-Up
linked list	26.65	3.78	7.04
banded matrix	219	8.4	26.07
cherry cake	516.23	57.15	9.03
NO to overhead	173.33	173.34	1

3. Conclusion

We propose a TLS system, using the polyhedral model at runtime, thus adapting to the current context, on chunks of the targeted loop nests, to perform speculative parallelization. The chunking strategy allows us to identify partial parallelism in loops and to apply the most suitable polyhedral transformation for each chunk.

References

[1] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08*, pages 101–113, 2008.

[2] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. In *Procs of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 158–167, New York, USA.

[3] E. Raman, N. Va hharajani, R. Rangan, and D. I. August. Spice: speculative parallel iteration chunk execution. In *Procs of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 175–184, New York, USA, 2008. ACM.

[4] L. Rauchwerger and D. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Procs of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 218–232, 1995.