Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 2575 – 2578

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Dynamic and Speculative Polyhedral Parallelization of Loop Nests Using Binary Code Patterns

Alexandra Jimborean[a], Philippe Clauss[b], Jean-François Dollinger[b], Vincent Loechner[b], Juan Manuel Martinez Caamaño[b]

[a] *University of Uppsala, UPMARC, Uppsala, Sweden*
[b] *University of Strasbourg, LSIIT, INRIA, CNRS, Strasbourg, France*

Abstract

Speculative parallelization is a classic strategy for automatically parallelizing codes that cannot be handled at compile-time due to the use of dynamic data and control structures. Another motivation of being speculative is to adapt the code to the current execution context, by selecting at run-time an efficient parallel schedule. However, since this parallelization scheme requires on-the-fly semantics verification, it is in general difficult to perform advanced transformations for optimization and parallelism extraction. We propose a framework dedicated to speculative parallelization of scientific nested loop kernels, able to transform the code at runtime by re-scheduling the iterations to exhibit parallelism and data locality. The run-time process includes a transformation selection guided by profiling phases on short samples, using an instrumented version of the code. During this phase, the accessed memory addresses are interpolated to build a predictor of the forthcoming accesses. The collected addresses are also used to compute on-the-fly dependence distance vectors by tracking accesses to common addresses. Interpolating functions and distance vectors are then employed in dynamic dependence analysis and in selecting a parallelizing transformation that, if the prediction is correct, does not induce any rollback during execution. In order to ensure that the rollback time overhead stays low, the code is executed in successive slices of the outermost original loop of the nest. Each slice can be either a parallelized version, a sequential original version, or an instrumented version. Moreover, such slicing of the execution provides the opportunity of transforming differently the code to adapt to the observed execution phases. Parallel code generation is achieved almost at no cost by using binary code patterns that are generated at compile-time and that are simply patched at run-time to result in the transformed code.

The framework has been implemented with extensions of the LLVM compiler and an x86-64 runtime system. Significant speed-ups are shown on a set of benchmarks that could not have been handled efficiently by a compiler.

Keywords: Dynamic optimizations, speculative parallelization, binary code patterns, templates, polyhedral model, runtime

1. Introduction

Automatically parallelizing sequential code became increasingly important with the advent of multicore processors. Particularly, the polyhedral model [1], originally designed for compile-time loop optimizations and parallelization, is known to show immense benefits for kernels written as loops with affine control of their iteration counts and array accesses (from hereafter, denoted as a *linear behavior*). However, frequently, even scientific

E-mail address: philippe.clauss@inria.fr.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.443

codes embed loop nests with bounds that cannot be statically predicted, having complex control flows or containing pointers leading to issues such as memory aliasing. As static analysis is too fragile, one relies on run-time speculations to achieve a stable performance . Additionally, to generate highly performing code, it is crucial to perform optimizations prior to parallelization. It is, nevertheless, a challenging task to parallelize code at runtime, due to the time overhead generated by the required analysis and transformation phases.

Runtime parallelization techniques are usually based on thread-level speculation (TLS) [4] frameworks, which optimistically allow a parallel execution of code regions before all dependences are known. Hardware or software mechanisms track register and memory accesses to determine if any dependence violation occur. In such cases, register and memory state are rolled back to a previous correct state and sequential re-execution is initiated. To gain efficiency, TLS systems must handle more complex code optimizations that can be profitably selected at runtime, depending on the current execution context.

We propose a lightweight static-dynamic system called VMAD – for *Virtual Machine for Advanced Dynamic analysis and transformation* –, which is a subclass of TLS systems, dedicated to codes that exhibit linear behavior phases at runtime: not only the loop bounds and the memory accesses can be represented as affine functions of the loop indices, but also the values assigned to some particular variables, called *basic scalars*. VMAD can aggressively optimize such codes by applying polyhedral optimizations dynamically and speculatively. However, performing aggressive code transformations at runtime might be prohibitively expensive. To overcome this limit, we advocate the use of binary code patterns, for fast dynamic code generation.

Overview of our system. This proposal extends our work on designing a TLS framework able to apply polyhedral transformations at runtime [3], such as tiling, skewing, loop interchange, etc., by speculating on the linearity of the loop bounds, of the memory accesses and of the values taken by specific variables, the basic scalars. Speculations are guided by online profiling phases. The instrumentation and analysis processes are thoroughly described in our previous work [2]. The system is entirely automatic and attempts the parallelization of the loop nests without any intervention of the user. Moreover, the programmer can, but is not required to, specify the loop nests of interest.

The implementation of VMAD 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 an x86-64 runtime system whose role is to build interpolating functions, perform dynamic dependence analysis and transformation selection, generate the parallel code and guide the execution.

For a light-weight dynamic generation of parallel code, we build generic code patterns from which several parallel code versions can be generated at runtime, by patching predefined code areas. The patterns are embedded in the compiler-generated binary file. The advantages are that the code size is significantly reduced and that many different parallel code versions can be build dynamically, guided by the results of the dynamic analysis.

2. Code patterns

The code patterns are prepared statically and instantiated at runtime to generate distinct code versions. To be able to handle all types of loops in the same manner, being them for-, while- or do-while loops, we introduce the notion of *virtual iterators*. They are canonical iterators inserted in the loops, starting from 0 and incremented with a step of 1. They allow us to handle loops that originally did not have any iterators in the code and to apply polyhedral transformations. As an example, consider the loop nest in Table 1, column 1 and its equivalent form in column 2 with virtual iterators. To preserve the correct semantics of the original code, the patterns must contain *guarding*, *initialization* and *verification code*, as shown in column 3 and detailed below. Additionally, patterns include explicit calls to the GOMP/OpenMP library to spawn parallel threads.

2.1. Guarding code

Since any target loop nest is first transformed as a for-loop nest, the computation of the new loop bounds of the parallelized loop nest has to be done automatically at runtime, for any loop nest depth. This is classically done using the Fourier-Motzkin elimination algorithm. Note that the conditions of the original while loops are preserved in the code by copying the original loop bodies in the pattern, thus ensuring that we do not execute mispredicted iterations. Similarly, we check that all iterations have been executed, by verifying that the exiting iteration, with respect to the sequential order, executes when predicted.

```
while (p!=NULL){
   q = q0;
   while(q!=NULL){
      p->val += q->val;
      q = q->next;}
   p=p->next;}
```

```
for i = low_chunk, upp_chunk
   for j = 0, u*i+v
      if (p!=NULL)
         ...
         if (q!=NULL)
            ....
```

```
do x = lb_x , ub_x {
   do y = lb_y , ub_y {
      /**initialization code**/
      p = a*x+b*y+c
      if (!(p!=NULL)) rollback();
      else {
       q = q0;
       if (guarding code){
        /**initialization code**/
        q = d*x+e*y+f
        if (!(q!=NULL)) {
          if (j ≠ α·x+β·y+γ)
             rollback();
          else continue;}
        else {
           p->val += q->val;
           q = q->next; }
        if (guarding code)
           p=p->next;
   } } } }
```

Table 1. Simplified code patterns

2.2. Initialization code

We use the linear functions obtained from the profiling phase, to initialize the basic scalars at runtime. The initialization code is equivalent to *privatization*, since all values that depend on other iterations are re-declared locally in each thread and initialized using the predicting linear functions. Thus, the new shape of the loop nest complies with the polyhedral model and the loops can be further transformed as in the case of statically analyzable code, by applying an affine, unimodular transformation.

2.3. Verification code

Since all code transformations, including parallelization, rely on speculations, one must periodically check the predictions in order to validate/invalidate the execution. Not only we maintain the correct memory state, but we also transform the control flow of the loop, using new loop iterators and bounds. The model we propose is based on the linear description of the memory accessing behavior. Hence, validating a transformation is equivalent to verifying the correctness of the interpolating linear functions. Under these circumstances, it suffices to compare the actual addresses being accessed, by the original instructions, to our predictions, given by the linear functions. Recall that the code inside the body of the loops in the code patterns is a copy of the original code. *Thus, the memory accesses are performed by a copy of the original memory instructions, whose target addresses are computed directly or indirectly from the basic scalars, which are initialized at each iteration.* Similarly the system verifies the values reached by the basic scalars and the loop bounds.

3. Experimental results

We evaluated our approach on an a platform embedding AMD Opteron 6172, of 12 cores each, and on an Intel Xeon X5650 at 2.67GHz, with 12 cores hyper-threaded. Our measurements are given in Table 2. We compare the speed-up of our system for the target loop nests relatively to manual parallelization using OpenMP, when possible, on both architectures, with 12 and 24 threads. Among the interesting examples, we underline `backprop`, which is parallel in the original form, however, VMAD detects a polyhedral transformation that further improves the performance; `cholesky` is not parallel originally, thus traditional TLS systems cannot parallelize it, however VMAD applies a transformation which allows a correct parallel execution; `floyd` illustrates the capability of our system to adapt dynamically to the behavior of the code and to exploit partial parallelism, since it is only parallel for a subset of iterations, due to conditionals.

Program	# threads	AMD Opteron 6172		Intel Xeon X5650		Transformation
		Speed-up VMAD	Speed-up OpenMP	Speed-up VMAD	Speed-up OpenMP	
adi	12	1.78	13.49	1.80	5.21	I_2
	24	1.82	13.34	4.09	4.75	identity, first loop parallel
backprop	12	12.53	11.24	1.23	1.83	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
	24	15.62	17.86	1.86	2.05	interchange, first loop parallel
cholesky	12	1.93	N/A	1.67	N/A	$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
	24	1.81	N/A	1.47	N/A	polyh. tr., second loop parallel
floyd	12	0.77	N/A	0.73	N/A	I_2
	24	1.43	N/A	0.73	N/A	identity, second loop parallel
fir2dim	12	2.74	N/A	3.29	N/A	I_2
	24	2.61	N/A	2.93	N/A	identity, first loop parallel
covariance	12	4.30	6.19	4.03	5.86	I_2
	24	7.45	12.07	4.55	8.92	identity, first loop parallel
correlation	12	4.29	6.26	3.88	5.73	I_2
	24	7.47	12.18	4.64	8.55	identity, first loop parallel
qr_decomp	12	2.87	12.02	2.11	11.03	I_2
	24	4.69	20.02	2.48	12.22	identity, first loop parallel
grayscale	12	1.81	8.73	1.13	3.59	I_2
	24	2.03	6.61	1.25	2.28	identity, first loop parallel

Table 2. Code speculatively parallelized with VMAD, compared to OpenMP

4. Conclusions

All in all, VMAD provides important contributions and advancements to the state of the art and is successful in optimizing and parallelizing scientific kernels, that are not accessible to traditional TLS systems or to static analysis: (1) Our system is able to automatically parallelize codes which do not exhibit parallelism in their original form. (2) Our system can discover optimization opportunities in codes that can already be parallelized in the original form. By applying such optimizing transformations prior to parallelization, the performance of the generated code can be significantly boosted. (3) The overhead of our system can be masked by the performance improvements provided both by parallelization and by the optimizing polyhedral transformations. We aim to describe more general models, to transform also non-linear codes.

References

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

[2] A. Jimborean, L. Mastrangelo, V. Loechner, and P. Clauss. VMAD: An advanced dynamic program analysis and instrumentation framework. In *CC '12*.

[3] A. Jimborean, P. Clauss, B. Pradelle, L. Mastrangelo, and V. Loechner. Adapting the polyhedral model as a framework for efficient speculative parallelization. In *PPoPP '12*, 2012.

[4] E. Raman, N. Vachharajani, R. Rangan, and D. I. August. Spice: speculative parallel iteration chunk execution. In *CGO '08*. ACM, 2008.