Online Dynamic Dependence Analysis for Speculative Polyhedral Parallelization

Alexandra Jimborean[1], Philippe Clauss[2],
Juan Manuel Martinez[2], and Aravind Sukumaran-Rajam[2]

[1] UPMARC, University of Uppsala, Sweden
alexandra.jimborean@it.uu.se
[2] Team CAMUS, INRIA, ICube lab., University of Strasbourg, France
{philippe.clauss,juan-manuel.martinez-caamano,
aravind.sukumaran-rajam}@inria.fr

Abstract. We present a dynamic dependence analyzer whose goal is to compute dependences from instrumented execution samples of loop nests. The resulting information serves as a prediction of the execution behavior during the remaining iterations and can be used to select and apply a speculatively optimizing and parallelizing polyhedral transformation of the target sequential loop nest. Thus, a parallel lock-free version can be generated which should not induce any rollback if the prediction is correct. The dependence analyzer computes distance vectors and linear functions interpolating the memory addresses accessed by each memory instruction, and the values of some scalars. Phases showing a changing memory behavior are detected thanks to a dynamic adjustment of the instrumentation frequency.

The dependence analyzer takes part of a whole framework dedicated to speculative parallelization of loop nests which has been implemented with extensions of the LLVM compiler and an x86-64 runtime system.

Keywords: Dynamic online dependence analysis, polyhedral transformations, speculative, parallelization, optimization, runtime.

1 Introduction

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. 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. Most speculative systems dedicated to loop nest parallelization launch slices of the original sequential outermost loop in parallel threads, without handling any other code transformations. Thus, verification consists merely in monitoring concurrent updates of the same memory locations using a centralized data structure – which is an important performance bottleneck – and in validating the ones occurring at the earliest iteration according to the original loop indices. However, as soon as the

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 191–202, 2013.

outermost loop carries a dependence, this parallelization strategy fails in numerous rollbacks. Also, it does not consider the current execution context or other factors that impact performance, such as data locality. Hence, more advanced parallelizing and optimizing transformations are required, comparable to the ones applied at compile time when possible. But a new verification strategy is required, ensuring that not only memory writes, but also reads, have to be performed in a semantically correct order. This requires the computation of dependences between memory accesses and the verification of their constancy during the speculatively parallel execution of loop nests.

In this paper, we present a dynamic dependence analyzer of loop nests which incurs a minimal time overhead, such that its results can be used online, in the attempt of speculatively *optimizing* and *parallelizing* the code. It is based on a code instrumentation system specifically dedicated to loop nests, which is applied on small execution samples. This loop sampling mechanism relies on a multiversioning scheme, in which instrumented and non-instrumented versions of each target loop are generated at compile time. Additionally, it embeds a switching mechanism allowing to alternate the execution of instrumented and non-instrumented loop bodies. The instrumented bodies contain instructions devoted to collect the addresses that are accessed by the memory instructions and the values that are assigned to some specific scalars called *basic scalars*. From the collected information, the dependence analyzer computes distance vectors and linear functions interpolating the memory addresses and the values assigned to the basic scalars in order to allow their privatization when parallelizing.

The dependence analyzer is supported by a runtime system which alternates the execution of different versions of the target loop nest. Thus, phases showing a changing memory behavior are detected by launching instrumented versions at some execution points. The frequency in which they are launched can be either fixed or adjusted according to the constancy of the memory behavior. Since the dependence analysis is performed based on instrumenting execution samples, it serves as a prediction for the remaining iterations of the loop nest. Based on its results, the runtime system selects and applies a speculatively optimizing and parallelizing polyhedral transformation, generating a lock-free parallel code, which does not induce any rollback, if the prediction is correct. When parallelizing speculatively, the associated verification system consists in verifying the constancy of the linearly interpolating functions, instead of monitoring concurrent memory accesses, which is the classical approach in speculative systems. Hence, verification is completely distributed among the threads, and does not require any centralized data structure.

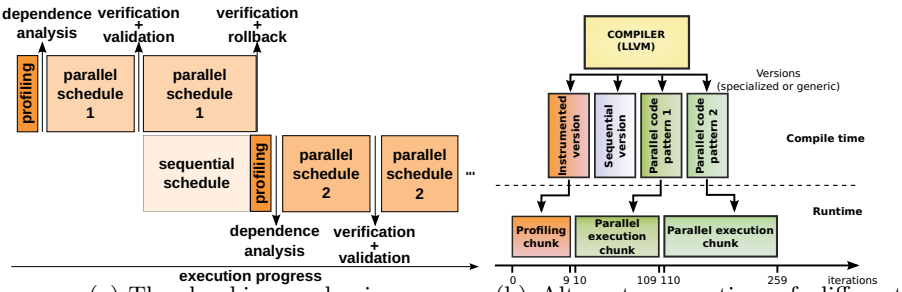
We show on a set of benchmarks that our analyzer is successful in identifying dependences across the iterations of a loop nest with a negligible runtime overhead. This property makes it insensitive to any variations of the input data or to program phases, since it can be applied repeatedly during one execution of the application, preceding the runtime optimizations.

2 Description of the Framework

This proposal focuses on the advanced dynamic dependence analysis we propose as part of the TLS framework [4], called VMAD, designed to apply polyhedral transformations at runtime, such as tiling, skewing, 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 [6]. The current proposal extends the dependence analyzer described previously [6], with the computation of the exact distance vectors, which are employed in validating more complex loop transformations, rather than straightforward parallelization.

A key aspect is instrumentation by sampling, in which the execution of instrumented and non-instrumented loop versions are alternated. The instrumented version is executed for a small number of consecutive iterations of each loop in the nest to collect sufficient information for performing the dynamic dependence analysis. Next, a non-instrumented version is launched, to limit the overhead. Instrumentation is re-launched with a varying frequency, in the view of detecting new phases characterized by a different pattern of the memory accesses. When there is a change of phases, the runtime system triggers a new instrumentation. This guarantees that the minimal amount of instrumentation is performed during the execution of the loop nest, but sufficient to characterize each new phase. Please note that in case the system detects frequent changes of phases (i.e. an instable behavior of the nest), it aborts the instrumentation and the attempt to speculatively parallelize the nest, since the speculations would most probably be invalidated. Hence, the system is able to self-control its overhead.

Using the chunking mechanism presented in Fig. 1(a), we slice the iteration space of the outermost loop into successive chunks, as detailed in our previous work [4]. Each chunk represents a subset of consecutive iterations of the outermost loop and can embed a different loop version (either instrumented, original or optimized). Note that chunking is performed at the level of the outermost loop only, nevertheless, during the execution of the profiling chunk, the instrumented and non-instrumented versions of the *innerloops* alternate, as presented in [6], to incur a minimal overhead. Following the results of the profiling, the dependence analysis validates a suitable polyhedral transformation for each loop phase. In this paper we focus on the process of computing the cross-iteration data dependences and validating polyhedral transformations, addressing the reader to our previous work [4] for more details regarding the TLS framework which applies the results of the dependence analyzer. During the speculative execution, the predictions are verified, initiating a rollback upon a misspeculation and resuming the execution with a sequential chunk. Misspeculations indicate a change of phase and they trigger a new instrumentation and analysis phase, after the faulty iterations are reexecuted sequentially. If validation succeeds, a new parallel chunk is launched. The process is depicted in Fig. 1(a). The implementation of VMAD consists of two parts: a *static* part, implemented in the LLVM compiler [10], designed to prepare the loops for instrumentation and parallelization,



(a) The chunking mechanism (b) Alternate execution of different versions during one loop nest's run

Fig. 1. Multiversioning

and a *dynamic* part, in the form of an x86-64 runtime system whose role is to build interpolating functions, to perform dynamic dependence analysis and transformation selection and to guide the execution.

Static Component. Our modified LLVM compiler generates customized versions of each loop nest of interest: original, instrumented and several parallel code patterns, together with a mechanism for switching between the versions. The patterns represent parameterized code versions, instantiated at runtime based on the results of the dependence analysis. To complete the loop's execution and adapt to the current phase, we automatically link at runtime the different versions of the original code. Each version is launched in a chunk to execute a subpart of the loop which is followed by the others, as in relay races. The support for chunking the outermost loop and linking distinct versions is illustrated in Fig. 1(b). The instrumented, original and two parallel code patterns are built at compile time. At runtime, one or another version is automatically selected to be executed for a number of iterations.

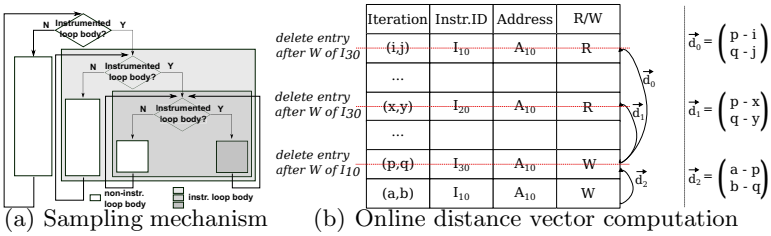
Dynamic Component. The runtime system collaborates tightly with the static component. During the instrumentation phase, it retrieves the accessed memory locations, the values assigned to the basic scalars, and computes interpolating linear functions of the enclosing loop indices. Instrumentation is performed on samples to limit the time overhead and is followed by the dependence analysis which evaluates whether a polyhedral transformation can be efficiently applied. If successful, this information can be useful in speculatively executing an optimized and parallelized version of the loop.

3 Dynamic Dependence Computation

A dedicated pragma allows the user to mark interesting loop nests in the source code. We have implemented dedicated extensions to the LLVM compiler that generate automatically, for our instrumentation purposes, two different versions of each target loop nest: instrumented and non-instrumented.

Instrumentation. The instrumented version associates to each memory instruction additional code which collects the target memory address and writes it in a buffer that will be read by the runtime system. Similarly, other instrumenting instructions are associated to monitor some specific scalars, called *basic scalars*. They have the interesting property of being at the origin of the computations of all other scalars used in the loop bodies, as for instance the target address computations. The basic scalars are identified at compile-time, being defined in the loop bodies as ϕ-nodes, since the intermediate representation of the LLVM compiler is in static single assignment form (SSA). They also carry dependences, since their values in a given iteration depend on the values they have been assigned in previous iterations. Hence, the opportunity for applying loop transformations and parallelizations depends on the possibility of privatizing them by predicting their values at each iteration. For this purpose, in the parallel code patterns, the basic scalars are initialized using the predicting linear functions (depending only on the indices of the transformed loops, and not on their value in the previous iterations). Straightforward examples of such basic scalars are the loop indices of for-loops, which are incremented at each iteration.

Since any kind of loops – for, while, do-while – are targeted, the instrumented and non-instrumented versions generated at compile-time contain one new iterator per loop, initialized with zero and incremented with a step of one. These iterators are injected by the compiler and used in the computation of the interpolating linear functions, as detailed in [5,6].



(a) Sampling mechanism for a 3-depth loop nest (b) Online distance vector computation

Fig. 2. Instrumentation by sampling and computation of distance vectors

Dedicated Sampling. Executions of the instrumented versions of the target loop nests are obviously more time-consuming than the original versions. However, these versions are run for small slices of the outermost loops of the target nests using the chunking mechanism presented in the previous section. Additionally, we implemented a dedicated sampling system allowing to instrumented only slices of each loop composing the nest. Thus, instrumentation activation does not depend only on the current loop, but also on the parent loops, making instrumented and non-instrumented bodies alternate, as illustrated in figure 2(a). Sizes of the instrumented slices can be either fixed or adjusted at runtime.

Polyhedral Transformations. The dynamic dependence analyzer is designed to compute distance vectors, and then verify if these distance vectors characterize

completely the memory behavior observed during the run of the instrumented version. This latter verification is achieved using an address value range analysis and a GCD test, as explained below. If the computed distance vectors conveniently characterize the target code, then they are used to select an optimizing parallelizing transformation of the loop nest, which results in the generation of a lock-free multithreaded version that should not induce any rollback if the memory behavior remains stable.Transformations that may be applied are polyhedral transformations [2] that change the order in which the iterations are scanned, such that at least one parallel loop is exhibited and the iteration schedule is optimized to address important issues, like data locality. Such a transformation is defined by a unimodular matrix T, applied to the space of the loop indices. T is valid *w.r.t.* any dependence distance vector d if the transformed vector $T \cdot d = d'$ is lexicographically positive, *i.e.*, if its first non-null component is positive. This component gives the depth of the loop which carries the associated dependence, therefore this loop cannot be parallelized. The outermost parallel loop is then the outermost loop which does not carry any dependence, considering all the transformed distance vectors.

Distance Vectors Computation. The runtime system reads the values communicated through the buffer and, when possible, builds linear interpolating functions for each memory instruction or basic scalar assignment, whose variables are the loop indices. It also computes linear functions to interpolate the loop bounds of the inner loops. Simultaneously, the collected memory addresses are used to compute online dependence distance vectors. The addresses are stored in a table whose entries also contain the access type (Read or Write), the loop index values at which the memory access occurred and the memory instruction identifier. Each time a new entry is created, a table look-up finds the previous accesses at the same address, computes the corresponding distance vectors and removes the entries that are becoming useless. The implemented algorithm is shown in the first part of table 1 and illustrated by figure 2(b).

Since only a sample of the execution tracks the memory accesses, the so-computed distance vectors may not entirely characterize the dependences that may occur during the whole execution of the target loop nest. If the instrumentation is performed on a loop slice of size S, a dependence whose distance is greater than S can obviously occur. We handle this issue by considering each couple of memory instructions, where at least one is a write, and for which no distance vector has been computed. Their associated interpolating linear functions are then used to verify if any dependence may occur between these instructions. First, a value range analysis is performed. For each linear function, their maximum and minimum reached values are computed using the interpolated loop bounds. If the respective ranges of touched addresses overlap, then a dependence may occur. In this case, a second analysis is performed through the GCD test, concluding if there may be a solution when considering the integer equation where both functions are equal. The algorithm is shown in the second part of table 1. These latter dependence tests are obviously less time-consuming than exact solving of integer equations, which would induce an overhead unacceptable for a dynamic parallelization

system. Moreover, an empty solution of these tests guarantees that the computed distance vectors entirely characterize the dependences, which allows one to validate the correctness of polyhedral transformations with a high probability, for a significant part of the execution.

Table 1. Dependence analysis algorithms

Distance vector computation algorithm.

create an entry in the table
if the current access is a write
  look for all reads at the same address, **until** finding a write
    for each found read, **compute** a distance vector which characterizes an *anti-dependence*
    if a write has been found, **compute** a distance vector which characterizes an *output dependence*
  remove all these entries excepting the current one
if the current access is a read
  look for a previous write at the same address
    if a write has been found, **compute** a distance vector which characterizes a *flow dependence*

Value range and GCD tests application algorithm.

build the couples of memory instructions not characterized by distance vectors, where at least one instruction is a write
for each couple
  compute their respective ranges of touched addresses by computing the extreme values reached by their associated linear functions
  if their ranges overlap
    perform the GCD test on the corresponding integer equation
    if the test fails (empty solution), *the couple does not carry any dependence*
    else, *the couple may carry a dependence*
  else, *the couple does not carry any dependence*

4 Experiments

Experiments were conducted on the Polybench benchmark suite [12]. For each program, we selected the most time-consuming loop nest. Although these codes can be analyzed statically to detect dependences, we stressed our system to extract dependences at runtime, in order to show its accuracy and ability in deducing speculative parallelizations. To test the ability of our system in detecting dependence phases, we modified the Polybench codes by introducing if-statements in the innermost loop body in order to alternate between three successive phases, each being characterized by slightly modified memory accesses that may introduce dependences. For instance, for a nest whose outermost loop ranges from 0 to N, we inserted if-statements that induce different memory behaviors for each subset of $N/3$ iterations. We modified the memory references for two of the three subsets by adding some integer constants to the original array references.

Our measurements are presented in table 2, where the kernel loop nest of each program is analyzed. The left part of the table shows the measurements performed on the original programs of the Polybench suite, while the right part shows the measurements performed on the modified programs exhibiting phases with different dependences. The second column shows the size of the instrumented chunks. For each program, we execute successively three instrumented runs with different instrumented chunk sizes (3, 10, 20) in order to compare the accuracy of the analyses, relatively to the number of instrumented iterations, as well as their respective overheads. The third column shows the percentage of time-overhead induced by instrumenting execution samples and determining dependences, computed as: *(instrumentationTime - originalTime)/originalTime*. The original codes were compiled using Clang-LLVM 3.0 with flag O3, on an AMD Opteron 6172, 2.1 Ghz, running Linux 3.2.0-27-generic x86_64.

For the experiments reported in this paper, the number of instrumented chunks launched by the runtime system depends on the number of iterations of the outermost loop, following the strategy: the first instrumented chunk is followed by a non-instrumented chunk of 100 iterations. Then again an instrumented chunk is launched. If the result of the dependency is equal to the result obtained from the previous instrumentation, then a non-instrumented chunk of $100 \times 2 = 200$ iterations is launched. Thus, the size of the non-instrumented chunk is doubled continuously as long as the dependences remain the same. If the dependences change, then the size of the non-instrumented chunk is reset to 100, as a new phase is detected. Please note that this strategy is devoted solely to the goal of performing online dependence analysis. In a complete speculatively parallelizing system, it is the speculation verification performed by the speculatively parallel code which would detect new phases and provoke the launching of an instrumented chunk. Additional experiments indicate that adjusting the frequency of the instrumentation based on phases detected at runtime by the TLS system, reduces the overhead of the analyzer to 15% at most, since it is performed only once for each phase, and the number of instrumented iterations is small, relative to the total number of iterations. However, this is out of the scope of the current article.

The fourth and fifth columns show the number of instrumented memory instructions and base scalar assignments, respectively; the sixth column shows the computed distance vectors for each phase and their types; finally, the seventh column suggests speculative parallelizations that could be applied considering the distance vectors and the results of the GCD tests.

The right part of the table illustrates the results on the modified programs exhibiting dependence phases. A new column shows the number of detected phases, and only one row per phase is presented for some instrumented chunk sizes, due to space constraints. For the same reason, although we successfully analyzed the dependences of all Polybench codes, it is impossible to show the results obtained for all of them. Only some representative ones are selected.

Table 2. Online dynamic dependence analysis on benchmark programs

Program	Instr. Chunk Sizes	DDA overhead	# Memory Accesses	# Basic Scalars	Dep. Dist. Vectors output - flow - anti	Speculative Par. Opportunities	DDA overhead	# Memory Accesses	# Basic Scalars	# Phases	Dep. Dist. Vectors output - flow - anti	Speculative Par. Opportunities
correlation/phase 1	**3**	2.96%	5	3	(0 0 1 : f)	outermost loop	3.25%	9	4	2	(0 0 1 : f)	outermost loop
phase 2											(0 0 1 : o - f - a 1 0 0 : o - a)	2^{nd} loop
phases 1,2	**10**	3.62%	5	3	same as above	same as above	5.63%	9	4	2	same as above	same as above
phases 1,2	**20**	67.74%	5	3	same as above	same as above	206.16%	9	4	2	same as above	same as above
cholesky/phase 1	**3**	224.55%	5	4	(1 −1 1 : a 1 0 1 : a 1 1 1 : a . . . …)	(1 1 1 0 1 0 0 0 1)	226.56%	7	4	3	(1 −1 1 : a 1 0 1 : a 1 1 1 : a . . . …)	(1 1 1 0 1 0 0 0 1)
phase 2											(1 7 1 : a 8 1 1 : a . . . …)	(1 1 1 0 1 0 0 0 1)
phase 3											(5 −3 1 : a 4 −3 1 : a . . . …)	(1 1 1 0 1 0 0 0 1)
phases 1,2,3	**10**	239.67%	5	4	(1 −1 1 : a 1 0 1 : a . . . …)	(1 1 1 0 1 0 0 0 1)	248.48%	7	4	3	larger set than above	(1 1 1 0 1 0 0 0 1)
phases 1,2,3	**20**	2408.03%	5	4	(1 −1 1 : a 1 0 1 : a . . . …)	(1 1 1 0 1 0 0 0 1)	4566.81%	7	4	3	larger set than above	(1 1 1 0 1 0 0 0 1)
adi/phase 1	**3**	3.19%	9	2	(0 1 : f - a)	outermost loop	1.47%	19	3	3	(0 1 : f - a)	outermost loop
phase 2											(0 1 : f - a 1 0 : a 1 1 : a)	(1 1 0 1)
phase 3											(0 1 : f - a 1 0 : f 1 −1 : f)	(2 1 1 0)
phases 1,2,3	**10**	4.7%	9	2	same as above	same as above	7.13%	19	3	3	same as above	same as above
phases 1,2,3	**20**	67.88%	9	2	same as above	same as above	131.52%	19	3	3	same as above	same as above
bigc/phase 1	**3**	31.08%	8	2	(1 0 : o - a 0 1 : f)	(1 1 0 1)	189.03%	10	2	3	(1 0 : o - a 0 1 : f)	(1 1 0 1)
phase 2											(1 0 : o - a 0 1 : o - f 1 −1 : a . . …)	none
phase 3											(1 0 : o - a 0 1 : o - f 1 −1 : a . . …)	none
phases 1,2,3	**10**	36.34%	8	2	same as above	same as above	206.93%	10	2	3	same as above	same as above
phases 1,2,3	**20**	178.26%	8	2	same as above	same as above	600.48%	10	2	3	same as above	same as above
gemm/phase 1	**3**	-14.12%	4	3	(0 0 1 : f)	outermost loop	-13.71%	10	3	2	(0 0 1 : f)	outermost loop
phase 2											(0 0 1 : f 1 0 0 : a)	2^{nd} loop
phases 1,2	**10**	-13.85%	4	3	same as above	same as above	-13.48%	10	3	2	same as above	same as above
phases 1,2	**20**	-3.73%	4	3	same as above	same as above	16.1%	10	3	2	same as above	same as above
3mm/phase 1	**3**	-22.49%	3	3	(0 0 1 : f)	outermost loop	-10.58%	4	3	2	(0 0 1 : f)	outermost loop
phase 2											(0 1 0 : a)	outermost loop
phases 1,2	**10**	8.38%	3	3	same as above	same as above	-10.38%	4	3	2	same as above	same as above
phases 1,2	**20**	19.41%	3	3	same as above	same as above	30.09%	4	3	2	same as above	same as above

Results in the table obviously show that larger instrumented chunks yield larger time overheads. In particular, one can observe a change of scale when considering chunks of size 20. However, since the accuracy of the dependency analysis is not significantly better with such large chunks, it argues to limit chunk sizes to at most 10. We observed that a high accuracy is often achieved with the smallest size of the instrumented chunk, 3. This is also due to the regularity of the handled benchmarks whose dependences are constant. In general, 10 iterations are sufficient to obtain a good accuracy, with a relatively low overhead. For the smallest instrumented chunk size, the overheads vary from -14% to 226%. Speed-ups can be explained by beneficial side-effects of chunking, or different optimizations triggered on our code versions compared to the ones generated by Clang on the original codes. When the time-overhead is the highest, it still remains acceptable: less than 3.5× with sizes 3 and 10, since parallelization should provide speed-ups that would substantially hide this overhead. With size 20, the overhead can become dangerously high in some cases, showing that instrumentation by sampling must remain under a relatively small threshold.

Our experiments also indicate that some codes require more advanced transformations than just parallelizing the original loops. It often occurs that every loop carries dependences. A standard TLS system would continuously rollback in such cases. On the other hand, a loop transformation, as the ones suggested by the transformation matrices in the table, would yield a semantically equivalent nest with at least one dependence-free loop that can be parallelized. This emphasizes the need of relatively accurate dependence analysis for TLS systems.

5 Related Work

Dependence analysis is an essential aspect in systems designed to perform automatic optimizations. However, previous research works focused on performing dynamic dependence analysis dedicated to an offline usage, thus, the overhead incurred by such profilers varies from 3×[13] to 70×[9]. Traditional TLS systems either rely on the results of such profilers or perform an optimistic, simple, straightforward parallelization of the outermost loop, for which no advanced dependence analysis is required. In contrast, we developed an ultra-fast dynamic dependence analyzer that can be used online and applied repeatedly during one execution, to adapt to different phases. This has been made possible thanks to a specific instrumentation system dedicated to loop nests, and able to switch between instrumented and non-instrumented code following a counter of the number of instrumented iterations. A similar approach to reduce the cost of instrumented code is presented in [1]. Our instrumentation system has two major differences. It is dedicated to loop nests and thus includes specific sampling management to coordinate loop levels as a whole. It also includes a different sampling mechanism provided by the chunking system to instrument only a small slice of the outermost loop. In the following, we review various state of the art techniques, however they are expected to be used offline, due to their large overhead.

Kim et al. [8] describe the fragility of static analysis, pleading for speculative parallelization, by speculating on some memory or control dependences.

Statically, a PDG (program dependence graph) is built. All dependences occurring less frequently than a certain threshold are speculatively removed from the PDG and the code is parallelized. Nevertheless, the dependence analysis is very simple and cannot be employed in validating aggressive code transformations, other than straightforward parallelization. The work of Praun et al. [14] identifies potential candidates for speculative parallelization by analyzing the density of runtime dependences in critical sections, *w.r.t.* the total number of executed instructions. Similarly to our proposal, the model can adapt to different program phases, and detect the ones suitable for speculative parallelization. No information regarding the profiler's overhead is presented, thus we conclude that the results of the analysis are used offline. The recent work of Vanka et al. [13] proposes a form of dependence analysis based on set operations using software signatures. In contrast to other works relying on sampling to achieve better performance, they group dependent operations into sets, and operate on relationships between sets, instead of considering pair-wise dependences. Additionally, they only profile queries relevant to the optimization being performed, rather than all possible queries. Thus, the profiler is highly accurate and well performing in comparison to previous works, introducing a 2.97× slowdown in average. Similarly, Oancea and Mycroft [11] propose a dynamic analysis for building dependence patterns. They map dependent iterations on the same thread, such that no dependence violations occur. Mapping is based on congruences of sets, computed from the dependence pattern. Still, the system incurs a considerable overhead.

Ketterlin and Clauss propose a system called Parwiz [7] that empirically builds a data dependence graph, after instrumenting samples or complete executions of an application with several representative inputs. Their goal is to identify potentially parallel regions of sequential programs and provide hints to the programmer. Similar tools analyze the data dependences across one execution and suggest parallelization strategies. Embla [3] performs an offline dynamic analysis and reports all occurring dependences, exhibiting parallelization opportunities. SD^3 [9] performs dynamic dependence analysis to provide suggestions to the developer on which modifications are desirable, such that the code becomes suitable for parallelization. SD^3 shows a 70× slowdown on average. Alchemist [15] is designed to identify dependences across loop iterations, loop boundaries and methods. It can be used offline by speculative systems, as it provides a very precise dependence analysis, analyzing complex data. Nevertheless, it induces a large overhead and it is not aimed for a runtime usage.

6 Conclusion

The increasing usage complexity of multi-core architectures require to shift advanced parallelization techniques from static to dynamic. Related to this purpose, we presented a dynamic dependence analyzer for loop nests dedicated to capture cross-iteration dependences, with a minimal time-overhead. Thus, it can be integrated in a TLS system for an online usage and even invoked repeatedly to characterize each new phase. The analyzer relies on instrumentation by sampling

and computes distance vectors, which can be employed in validating polyhedral transformations for each phase of the nest.

References

1. Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM.
2. U. Banerjee. *Loop Transformations for Restructuring Compilers - The Foundations*. Kluwer Academic Publishers, 1993.
3. K-F Faxén, K. Popov, S. Jansson, and L. Albertsson. Embla - data dependence profiling for parallel programming. In *Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS '08, pages 780–785, Washington, DC, USA, 2008. IEEE Computer Society.
4. A. Jimborean, Ph. Clauss, B. Pradelle, L. Mastrangelo, and V. Loechner. Adapting the polyhedral model as a framework for efficient speculative parallelization. In *PPoPP '12*, 2012.
5. A. Jimborean, M. Herrmann, V. Loechner, and Ph. Clauss. VMAD: A virtual machine for advanced dynamic analysis of programs. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 125–126, Washington, DC, USA, 2011. IEEE Computer Society.
6. A. Jimborean, L. Mastrangelo, V. Loechner, and Ph. Clauss. VMAD: An Advanced Dynamic Program Analysis and Instrumentation Framework. In Michael OBoyle, editor, *Compiler Construction*, volume 7210 of *Lecture Notes in Computer Science*, pages 220–239. Springer Berlin Heidelberg, 2012.
7. A. Ketterlin and Ph. Clauss. Profiling Data-Dependence to Assist Parallelization: Framework, Scope, and Optimization. In *MICRO-45, The 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Canada, 2012.
8. H. Kim, N.P. Johnson, J. W. Lee, S. A. Mahlke, and D. I. August. Automatic speculative doall for clusters. In *CGO '12*. ACM, 2012.
9. M. Kim, H. Kim, and C-K Luk. SD3: a scalable approach to dynamic data-dependence profiling. In *MICRO '43*. IEEE Computer Society, 2010.
10. LLVM compiler infrastructure. `http://llvm.org`.
11. C. E. Oancea and A. Mycroft. Set-congruence dynamic analysis for thread-level speculation (TLS). In *LCPC '08*. Springer-Verlag, 2008.
12. Polybenchs. http://www-rocq.inria.fr/pouchet/software/polybenchs.
13. R. Vanka and J. Tuck. Efficient and accurate data dependence profiling using software signatures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 186–195, NY, USA, 2012.
14. C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 185–196, New York, NY, USA, 2008. ACM.
15. X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *CGO '09*. IEEE Computer Society, 2009.