Speculative Program Parallelization with Scalable and Decentralized Runtime Verification

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

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

Abstract. Thread Level Speculation (TLS) is a dynamic code parallelization technique proposed to keep the software in pace with the advances in hardware, in particular, to automatically parallelize programs to take advantage of the multi-core processors. Being speculative, frameworks of this type unavoidably rely on verification systems that are similar to software transactional memory, and that require voluminous inter-thread communications or centralized registering of the performed memory accesses. The high degree of communication is against the basic principles of high performance parallel computing, does not scale with an increasing number of processor cores, and yields weak performance. Moreover, TLS systems often apply one unique parallelization strategy consisting in slicing a loop into several parallel speculative threads. Such a strategy is also against the basic principles since loops in the original serial code are not necessarily parallel and also, it is well-known that the parallel schedule must promote data locality which is crucial in obtaining good performance. This situation appeals to scalable and decentralized verification systems and new strategies to dynamically generate efficient parallel code resulting from advanced optimizing parallelizing transformations. Such transformations require a more complex verification system that allows intra-thread iterations to be reordered. In this paper, we propose a verification system of this kind, based on a model built at runtime and predicting a linear memory behavior. This strategy is part of the Apollo speculative code parallelizer which is based on an adaptation for dynamic usage of the polyhedral model.

1 Introduction

Automatically parallelizing sequential code became increasingly important with the advent of multicore processors. However, static approaches applied at compile-time fail in handling codes which contain intractable control and memory instructions. For instance, while-loops, indirect array references or pointer accesses cannot generally be disambiguated at compile-time, thus preventing any automatic parallelization based exclusively on static dependence analysis. Such a situation appeals for the development of runtime parallelization systems, which are granted more power by the information discovered dynamically.

Runtime parallelization techniques of loop nests are usually based on thread-level speculation (TLS) [1–3] frameworks, which optimistically allow the 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, the register and memory state is rolled back to a previous valid state and sequential re-execution is initiated. Traditional TLS systems perform a simple, straightforward parallelization of loop nests by simply slicing the outermost loop into consecutive parallel threads [1, 2, 4]. Verifying the speculations consists in ensuring that the schedule of the accesses to shared memory locations in the parallel code matches the one of the original code. This general verification principle is made simple in the case of straightforward parallelization, since each parallel thread consists of a slice of successive iterations of the original serial loop nest, thus following internally the original sequential schedule. Modest performance improvements have been reported, due to an expensive verification system and poor parallelizing transformations. The verification system requires communication among the parallel threads to share which memory addresses are accessed, in order to detect conflicts and preserve memory coherency by rollbacking the delinquent threads. This strategy yields a *high communication traffic* that is significantly penalizing performance, and which is against the general optimization principles in parallel computing. Another important consequence is that a *centralized verification system* does not scale with the number of processor cores. This situation calls for a different strategy where each thread takes part independently in the verification of the global correctness of the speculative parallelization. Additionally, as soon as a dependence is carried by the outermost loop, it leads to numerous rollbacks, consequently, performance drops. Moreover, even if infrequent dependences occur, there is no guarantee that the resulting instruction schedule improves performance. Indeed, *poor data locality* and a high amount of data shared between the threads can yield a parallel execution slower than the original sequential one. To gain efficiency, TLS systems must handle more complex code optimizing transformations that can be selected at runtime, depending on the current execution context.

In this paper, we propose a verification strategy as an answer to these drawbacks. Our solution relies on a *prediction model* which is built by first observing a small sample of the target loop nest execution, and then it is used to verify the speculatively optimized and parallelized code, during execution. The parallel code is generated by applying advanced code transformations, thus, the iteration schedule in the parallel threads is no longer in accordance with the original serial schedule of the iterations. This is equivalent to saying that iterations are reordered not only across threads, but also within a thread. Yet, the parallel schedule is semantically correct as long as the prediction model holds. Each thread verifies independently that its execution is compliant with the prediction model, hence the verification is entirely de-centralized. The model adopted in this work to reason about the loop transformations is an adaptation of the *polyhedral model* to dynamic and speculative parallelization.

The polyhedral model [5], originally designed for compile-time loop optimization and parallelization, is known to show immense benefits for loops with linear iteration counts and array accesses. Such loops are characteristic to scientific codes or kernels designed for embedded systems. However, frequently, applying the polyhedral model statically is prohibited by factors such as: (i) bounds that cannot be statically predicted, (ii) complex control flows, or (iii) pointers accessing dynamically allocated structures, which leads to issues such as memory aliasing. Yet, such codes, although not statically analyzable, may exhibit a linear behaviour at runtime. Thus, they are amenable to precise polyhedral dependence analysis (based on information acquired by online profiling), in the view of performing complex parallelizing code transformations at runtime. This has important consequences: (a) runtime verification is required to validate the speculative code transformations; (b) an online recovery system, which will be triggered upon a misspeculation, must be designed; (c) the system should be lightweight enough to shadow the runtime overhead[1].

In this paper, we focus on the *verification system* of a polyhedral TLS framework called Apollo, for Automatic POLydedral Loop Optimizer. Apollo takes the best of the two worlds: as a TLS system, it targets non-statically analyzable loop nests and memory accesses (including while-loops with memory accesses to dynamic data structures via pointers which exhibit a linear runtime behavior); as a polyhedral optimizer, it applies polyhedral optimizations prior to parallelization, which makes Apollo novel and conceptually different than its TLS predecessors.

The paper is organized as follows: next section describes a classic program exhibiting parallel phases, depending on the input data. This kind of programs is a typical target for Apollo. In Section 3, the global functioning of our polyhedral TLS system is depicted, while its part dedicated to runtime verification of the speculative parallelizing and optimizing transformations is thoroughly detailed in Section 4. Related work addressing runtime verification of speculations in TLS systems is summarized in Section 5. Results of experiments showing the effectiveness of our approach are given in Section 6. Finally, Section 7 presents conclusions and perspectives.

2 Motivating Example

This section underlines an example code exhibiting polyhedral behavior in some execution contexts, which cannot be detected statically, thus preventing automatic parallelization at compilation time. Apollo is tailored to detect and dynamically optimize such codes. The example is the kernel loop nest of the *breadthFirstSearch* (BFS) algorithm from the Problem Based Benchmark Suite [6] shown in Listing 1.1.

The BFS method performs a breadth first search scan of a graph in the following way. The vertices of the input graph `GA` are identified as integer values ranging from 0 to `GA.n`. Thus, array `Visited` is used to mark each vertex which has already been visited, by storing respectively 0 or 1 at the vertex index value. Array `Frontier` is used to store the list of vertices whose neighbors have to be visited in some next iterations of the outer while-loop. As long as `top > bot`, there are still remaining vertices that have to be visited. Before entering the loop nest, the input starting vertex is identified by the variable `start`: it is stored in array `Frontier` as the first and still unique vertex whose neighbors must be visited, and the vertex itself is marked in array `Visited` as having been already visited. When entering the while-loop, the current vertex whose

[1] Stemming from online profiling, dynamic code transformations, support for a speculative execution and recovery from invalid speculations.

Listing 1.1. Main loop nest of the breadthFirstSearch benchmark code

```
pair<int,int> BFS(int start, graph<int> GA) {
 int numVertices = GA.n;
 int numEdges = GA.m;
 vertex<int> *G = GA.V;
 int* Frontier = newA(intT,numEdges);
 int* Visited = newA(intT,numVertices);
 for (intT i = 0; i < numVertices; i++) Visited[i] = 0;
 int bot = 0;
 int top = 1;
 Frontier[0] = start;
 Visited[start] = 1;
#pragma apollo_dcop { /* Dynamic Control OPtimization */
 while (top > bot) {
  int v = Frontier[bot++];
  int k = 0;
  for (int j=0; j < G[v].degree; j++) {
   int ngh = G[v].Neighbors[j];
   if (Visited[ngh] == 0) {
    Frontier[top++] = G[v].Neighbors[k++] = ngh;
    Visited[ngh] = 1; }
  }
  G[v].degree = k;
 } // end while
} // end pragma
 free(Frontier); free(Visited);
 return pair<int,int>(0,0);
}
```

neighbors are going to be visited is `Frontier[bot]`, assigned to variable `v`. The inner for-loop is used to scan all the neighbors of this current vertex `v`, their count being given by `G[v].degree`. For each neighbor, it is determined if it has already been visited by testing its corresponding element of array `Visited`. Otherwise, *i.e.* when `Visited[ngh]==0`, it is stored in array `Frontier` as a vertex whose neighbors have to be visited in the next iterations of the while-loop. The order in which vertices are stored and processed in array `Frontier` ensures the breadth first search order of the algorithm.

Compile-time automatic parallelization, as well as manual OpenMP parallelization, are prohibited by the presence of the while-loop. Additionally, the upper bound of the inner for-loop is sensitive to the input data. Dependences cannot be analyzed statically since some elements of array `Visited` may be updated several times depending on the value of `ngh = G[v].Neighbors[j]`, itself depending on the value of `v = Frontier[bot++]`. Even if experts in parallel programming would be able to handle its parallelization with considerable efforts, this loop nest is amenable to *automatic* parallelization only speculatively, at runtime. Some TLS systems would attempt to parallelize the outermost while-loop by slicing it into several speculative threads, with the assumption that values of `top` and `bot` can be predicted by the speculative system. Nevertheless, this would fail since reads of uninitialized array elements `Frontier[bot++]` at line 14 would be detected as faulty and not in compliance with the original serial order. In contrast, for particular input graphs, Apollo detects a Read-After-Write dependence between the update of `Frontier[top++]` in the inner loop and the read of `Frontier[bot++]` in the outer loop, from the initial run of a small instrumented slice of the outermost loop. Thus, Apollo would not attempt parallelization of the outer loop.

The unique possible loop parallelization is on the inner for-loop (for TLS systems also handling inner loops). Special care must be taken regarding accesses to array `G[v].Neighbors` which are carrying Write-After-Read dependences, as well as regarding read-write accesses to the variables `top` and `k` which are carrying Read-After-Write dependences. Without embedding a mechanism for privatizing in each parallel thread both latter variables and predict their values, a TLS system would fail. In contrast, thanks to instrumentation by sampling and linear interpolation, Apollo embeds their thread-privatization and the prediction of their values in the resulting parallel code. In consequence, their associated Read-After-Write dependences are eliminated.

Apollo is successful in parallelizing the inner loop for certain classes of input graphs. This example highlights a typical case where parallelization opportunities depend on the input data: for instance, if the input graph defines a regular grid, or a complete N-tree, then `G[v].degree` is constant, since by definition, each vertex has the same fixed number of neighbors. Also in this case, the conditional `Visited[ngh]==0` is evaluated as true for a large number of successive vertices which do not share the same neighbors. Thus, variable `k` is equal to variable `j` for large execution phases, which enables an accurate prediction of the memory accesses and the parallelization of large execution phases.

Let us consider a regular grid of n vertices and of degree d defined as follows: each vertex $i < n - d$ has d neighbors ranging from $(i + 1)\ mod\ n$ to $(i + d)\ mod\ n$, and each vertex $i \geq n - d$ has one neighbor which is vertex 1. Considering this grid as input to the breadthFirstSearch algorithm (List. 1.1), Apollo was successful in automatically parallelizing the inner loop on-the-fly. A first significant phase of n/d outer loop iterations was detected as amenable for parallelization. This phase corresponds to the continuous evaluation as true of the conditional `Visited[ngh]==0`. A rollback was initiated at the end of this phase, followed by the run of a small slice of instrumented iterations allowing Apollo to build a new prediction model and to parallelize a larger phase of $n - d - n/d$ outer loop iterations. This latter phase corresponds to the continuous evaluation as false of the conditional `Visited[ngh]==0`. While this phase was ended by rollbacking, a next instrumented slice of iterations was not able to build a linear prediction model. Thus, the execution was completed using the original serial code for the remaining outer loop iterations. A $9\times$ speed-up was obtained with $n = 10,000$ and $d = 1000$ on two AMD Opteron 6172 processors, of 12 cores each, running 32 parallel threads. Details on how Apollo handled this code, and particularly on how it ensured correctness of the speculative parallelization, are given in the next Section.

3 Dynamic and Speculative Polyhedral Parallelization with Apollo

The polytope model [7] has been proven to be a powerful mathematical and geometrical framework for analyzing and optimizing for-loop nests. The requirements are that (i) each loop iterates according to a unique index variable whose bounds are affine expressions of the enclosing loop indices, and (ii) the memory instructions are limited to accesses to simple scalar variables or to multi-dimensional array elements using affine expressions on the enclosing loop indices. Such loop nests are analyzed accurately with respect to data dependences that occur among the statements and across iterations. Thus, advanced optimizing transformations are proven to be semantically correct by preserving the dependences of the original program. The loop nest optimizations (e.g., skewing, interchange) are linear transformations of the iteration domains that are represented geometrically as polyhedra. Each tuple of loop indices values is associated with an integer point contained in the polyhedra. The order in which the iterations are executed translates to the lexicographic order of the tuples. Thus, transformations represent a reordering of the execution of iterations and are defined as scheduling matrices, which is equivalent to geometrically transforming a polyhedra into another equivalent form [7]. Representing loops nests as polyhedra enables one to reason about the valid transformations that can be performed.

Although very powerful, the polytope model is restrained to a small class of compute-intensive codes that can be analyzed accurately and transformed at compile-time. However, most legacy codes are not amenable to this model, due to dynamic data structures accessed through indirect references or pointers, which prevent a precise dependence analysis to be performed statically. On the other hand, applied entirely dynamically, the complex analyses and the polyhedral code transformations would entail significant overhead. As shown in Section 2, codes that do not exhibit characteristics suiting the polytope model may still be in compliance with the model, although this compliance can only be detected at runtime. Targeting such codes for automatic optimization and parallelization imposes to immerse the polytope model in the context of speculative and dynamic parallelization. In this context, runtime code analysis and transformation impose strategies which induce very low time-overheads that must be largely compensated by the gains provided by the polyhedral optimization and parallelization.

For loop nests that cannot be analyzed statically, our strategy for making the polyhedral model applicable at runtime relies on speculations, and thus, requires runtime verification. It consists of observing initially the original code during a very short sample of the whole run. If a polyhedral behavior has been observed on this sample, we speculate that the behavior will remain the same on the rest of the loop nest execution. Thus, we can abstract the loop to a polyhedral representation, reason about the inter-iteration dependences, and validate and apply a polyhedral optimizing and parallelizing transformation. As long as this prediction remains true, the generated parallel code is semantically correct by definition of the polyhedral model. In order to verify continuously the prediction, and thus verify the correctness of the parallel program, we implemented a decentralized runtime verification system embedded in the parallel code, as detailed in the next Section.

First, we recall the main steps of static polyhedral automatic parallelization and describe how these steps are handled in Apollo to turn this approach into its dynamic and speculative equivalent form. The framework description focuses on the two main goals: building the polyhedral prediction model and applying speculative parallelization including runtime verification of the prediction. Further details regarding dynamic code generation and other important parts of Apollo can be found in [8], where a former prototype version called VMAD is presented. Apollo consists of two main parts: a static part implemented as passes of the LLVM compiler [9], and a dynamic part implemented as a runtime system written in C++.

At compile-time, Apollo's static phase: (1) analyzes precisely memory instructions that can be disambiguated at compile-time; (2) generates an instrumented version to track memory accesses that *cannot* be disambiguated at compile-time. The instrumented version will run on a sample of the outermost loop iterations and the information acquired dynamically is used to build a prediction model of these statically non-analyzable memory accesses; (3) generates parallel code skeletons [8]. They are incomplete versions of the original loop nest and require runtime instantiation to generate the final code. Each instantiation represents a new optimization, therefore the code skeletons can be seen as highly generic templates that support a large set of optimizing and parallelizing transformations. Additionally, the skeletons embed support for speculations (e.g. verification and recovery code).

At runtime, Apollo's dynamic phase: (1) runs the instrumented version on a sample of consecutive outermost loop iterations; (2) builds a linear prediction model for the loop bounds and memory accesses; (3) computes dependences between the memory accesses; (4) instantiates a code skeleton and generates an optimized, parallel version of the original sequential code, semantically correct with respect to the prediction model; (5) during the execution of the multi-threaded code, each thread verifies independently if the prediction still holds. If not, a rollback is initiated and the system attempts to build a new prediction model. An optimization has been designed to limit the number of iterations required to rollback upon a misspeculation (see subsection 3.2).

3.1 Compliance with the Polyhedral Model

The programmer inserts a dedicated $pragma$ defining regions of code in which all loop nests will be considered for a speculative execution by Apollo. At compile-time, the target loop nests are analyzed and first the instrumented versions are generated. Additional counters named *virtual loop iterators* are systematically inserted to enable the framework in handling uniformly any kind of loops, e.g. for-loops or while-loops. They are also important in the speculative parallelization phase as it will be explained later. The static analysis consists in the following steps.

Every memory instructions is classified as *static* or *dynamic*[2]. For static memory accesses, the LLVM scalar evolution pass[3] is successful in expressing the sequence of accessed locations as an affine function of the enclosing loop iterators. This approach fails on dynamic memory accesses. For each couple of static memory instructions where at least one is a store, alias analysis is performed using a dedicated LLVM pass. The collected aliasing information will be used at runtime to save some

[2] i.e. which can be analyzed statically or requires dynamic instrumentation.

[3] `http://llvm.org/devmtg/2009-10/ScalarEvolutionAndLoopOptimization.pdf`

Table 1. Prediction model characteristics for the breadthFirstSearch code

#handled scalars	**predicting affine functions**
4	$0i + 100; 100i + 1$ $0i + 1j + 0; 100i + 1j + 1$
#memory instructions	**predicting affine functions**
9	$16i + 19282504; 16i + 19282496$ $400i + 4j + 19442512$ $400i + 4j + 27363348$ $400i + 4j + 19442512$ $400i + 4j + 23402932$ $400i + 4j + 27363348$ $16i + 0j + 19282504; 16i + 19282504$
#inner loop bounds	**predicting affine functions**
1	$0i + 100$

dependence types	**dependence equations** $\forall\ (i, j) \preceq (i', j')$
Write-After-Read	$\{\ i - i' = 0$
Write-After-Read	$\begin{cases} i - i' = 0 \\ j - j' = 0 \\ -j + j' \geq 0 \end{cases}$
Write-After-Read	$\begin{cases} i - i' = 0 \\ i - j' = 0 \end{cases}$
Write-After-Write	$\{\ i - i + 1' = 0$
Read-After-Write	$\begin{cases} i - i + 1' = 0 \\ j - j' = 0 \end{cases}$

dependence analysis time-overhead. Instrumentation instructions are inserted to collect the memory addresses touched by each dynamic memory instruction. Similarly, relying on the LLVM scalar evolution pass, Apollo attempts to build affine functions describing the loop bounds. If this attempt fails, instrumentation code is inserted to monitor the value of the loops bounds. Scalar variables required to maintain the control flow or to compute the memory addresses are also analyzed by scalar evolution or instrumented if the analysis fails. These scalar variables are detected at compile-time as being defined by phi-nodes in the LLVM Intermediate Representation (IR) which is in Static Single Assignment (SSA) form. Linearly dependent scalars are grouped to reduce instrumentation to one unique representative of the group to lower the instrumentation runtime cost. The linear functions computed by the scalar evolution pass are stored and will be transmitted to Apollo's runtime system to complete the information required for runtime dependence analysis.

The dynamic analysis consists of the following operations. When running, every instrumented instruction generates a stream of values (memory addresses or scalar values) that are interpolated as functions of the virtual loop iterators. If every stream of values, obtained from an execution sample, can be modeled as an *affine function* of the virtual loop iterators, then the target loop nest is speculatively predicted to be compliant with the polytope model. The so-built affine functions are finally used to complete the dynamic dependence analysis which is also performed using the streams of actual addresses that are collected from instrumentation.

In summary, the prediction model of each target loop nest is made of: (1) the dependence information which is used to select and validate a parallelizing code transformation; (2) the affine functions associated with the memory instructions and the scalar variables: these functions are essential for the runtime verification of the speculation and to predict the starting context of the parallel threads regarding the scalars. This part is fully detailed in the next Section. As an example, the prediction model computed by Apollo at runtime for the first phase of the breadthFirstSearch code (see Section 2) is detailed in Table 1, where $\preceq$ denotes the lexicographical order. Notice that even if some scalars and memory instructions can be intuitively related to the source code, it is generally difficult, since they are identified at compile-time on the LLVM-IR representation of the program, after some LLVM optimizations have been applied.

3.2 Speculative Parallelization and Runtime Verification

Speculative parallelization and runtime verification are performed using the prediction model as sketched in what follows. Runtime verification is specifically highlighted in the next Section.

Code skeletons: At compile-time, several variants of codes are generated from each loop nest that was marked in the source code by the user using the dedicated pragma: an instrumented version, as described in the previous subsection, but also a number of code skeletons, presented in detail in our previous work [8]. Skeletons can be seen as parametrized codes where the instantiation of their parameters results in the generation of a transformed optimized version of the target loop nest merging original computations and speculative parallelization management. They consist of three parts: the first part applies the transformation, which is populated at runtime; the second performs the original computation on the transformed iteration domain; and the third does the verification. Skeletons support classes of loop transformations as skewing, interchange, tiling, etc [10]. In the current implementation, Apollo's skeletons support skewing and interchange.

Parallelizing code transformation: As soon as the prediction model has been built, Apollo's runtime system performs a dependence analysis which determines if the target loop nest can be parallelized and optimized and what transformation has to be applied for this purpose. A polyhedral transformation merely refers to changing the order in which iterations are executed and is controlled by applying affine functions on the loop iterators. The transformation is encoded as a matrix, storing the coefficients of the affine functions which define the new schedule. Given a loop nest of depth two with iterators $\binom{i}{j}$ and a transformation matrix T, polyhedral loop transformations such as skewing, interchange or any affine transformation of the iteration domains [7] are obtained as: $T \times \binom{i}{j} = \binom{i'}{j'}$. This is achieved by invoking the polyhedral parallelizer Pluto [7] at runtime. More precisely, only the scheduler kernel of Pluto is used. It has been slightly customized to consume our dependence analysis output and to suggest a polyhedral transformation in return. Since Pluto aims simultaneously data locality optimization and parallelization, the generated schedule is expected to lead to a well-performing parallel code. Notice also that Pluto is initially a source-to-source code transformer used at compile-time, and that Apollo is the first known dynamic framework which is using it at runtime, with very low time-overhead.

Speculative code orchestration: The different code versions (instrumented, serial original, or instantiated skeleton) are launched in chunks of fixed sizes. These chunks are running a slice of successive iterations of the outermost original loop nest. Thus, optimizing parallelizing transformations are applied on such slices. At startup, Apollo launches a small chunk running the instrumented version in order to build the prediction model and perform the dependence analysis. The transformation suggested by Pluto from the dependence information is then used to instantiate the code skeleton devoted to the corresponding class of transformations. The resulting parallel code is then launched inside a larger chunk, after having previously backed-up the memory locations that are predicted to be updated. If the verification of the speculation detects a unpredicted behavior, memory is restored to cancel the execution of the current chunk. The execution of the chunk is re-initiated using the original serial version, in order to overcome the faulty execution point. Finally, an instrumented chunk is launched again to capture the changing behavior and build a new prediction model. If no miss-prediction was detected during the run of the parallel code, a next chunk using the same parallel code and running a next slice of the loop nest is launched.

4 Runtime Verification of Speculative Polyhedral Parallelization

The model handled currently by Apollo is the polyhedral model. Thus, the speculative prediction model claims (i) that every memory instruction targets a sequence of addresses represented by an affine function of the loop iterators, (ii) that every scalar variable, that is modified across iterations, either stores values also represented by such an affine function, or carries a dependence, and finally (iii) that every loop upper bound is also such an affine function (while the lower bound is 0)[4]. Each of these three characteristics must be verified while running the speculative parallel code which is semantically correct only if the prediction model holds. This is achieved thanks to dedicated code inserted at compile-time in the parallel code skeletons, and instantiated at runtime. This code triggers a rollback as soon as the verification fails.

The skeletons are generated automatically in the LLVM IR using our dedicated compilation pass. As depicted in the previous section, they are made of three types of instructions: (1) *instructions dedicated to apply the optimizing transformation, including parallelization:* these are the for-loops iterating over the introduced virtual iterators, which are transformed into new iterators through the linear transformation suggested by Pluto; and instructions in the header of each loop of the nest which are devoted to the initialization of the predicted scalar variables; (2) *instructions of the original code:* the original loop exit conditions serve as guards of the original loop bodies which are copied in the generated skeletons at compile-time; (3) *instructions devoted to the verification:* these instructions are inserted at several relevant points of the skeleton code to verify the adherence of each linear function constituting the prediction model with the original code behavior. They are related to memory accesses, scalar initializations and loop bounds verifications, and are detailed in the following subsections.

4.1 Target Memory Address Verification

Memory instructions executed speculatively are guarded by verification instructions, ensuring that no unsafe write operations are performed. Recall that the prediction model is based on representing the sequence of the addresses accessed by an instruction as affine functions of the (virtual) loop iterators. Based on this linearity of the memory accesses, a tightly coupled dependence analysis allows to apply an optimizing transformation of the target code which is semantically correct as long as the predicited dependences are still entirely characterizing the code. Thus, verifying completeness of the predicted dependences translates to verifying that all memory accesses follow their associated affine functions. This is ensured by comparing, for each memory instruction,

[4] These represent the bounds of the virtual loop iterators.

the actual target address against the value resulting from the evaluation of the predicting affine function. Notice that Apollo must verify the linear functions in the transformed space, not the linear functions which were obtained during instrumentation of the original, untransformed, sequential code. An example of the code verifying the update of array `G[v].Neighbors` in the breadthFirstSearch code of Figure 1.1 is shown in the below pseudo source code (instead of original LLVM-IR form). Variables `vi` and `vj` denote the virtual iterators of the two nested loops.

```
if (&G[v].Neighbors[k] != linear_eq(mem_instr_ID, vi, vj))
   rollback();
G[v].Neighbors[k++] = ngh;
```

4.2 Scalars Initialization and Verification

As depicted in previous Sections, scalar variables defined as phi-nodes in the LLVM intermediate representation are taking part of the prediction model. These scalars are also carrying dependences by being used and updated among loop iterations. As it is usually done manually when parallelizing serial codes, a common approach to remove such dependences is to privatize such scalars when possible. Privatization consists of replacing their incremental updates by the direct computation of their successive values using the current values of the loop iterators. For some scalars, the scalar evolution pass of the LLVM compiler may be successful in determining statically an affine expression to compute their values. Otherwise, Apollo's instrumentation by sampling provides to compute this affine function at runtime, as soon as it can be represented in this way. However, notice that privatization of such latter scalars is therefore speculative.

Since Apollo's code transformations may not follow the original iteration order, scalar variables must be initialized at their correct starting values in the header of each iteration. This is performed in the header of each loop of the target nest, as it is shown below in pseudo source code for the breadthFirstSearch code example and its `top` and `bot` scalar variables.

```
top = linear_eq(&top, vi) ;
bot = linear_eq(&bot, vi) ;
```

However, since these scalar initializations are speculative, they must verified. Generally, scalars used in loops are initiating an iteration while being assigned the very last value that has been assigned to them among the previous iterations. The same scheme is followed by Apollo's verification strategy: at the very end of each iteration, the prediction for the next iteration initial value is compared to the actual value of the scalar, *i.e.*, its very last value before the initiation of the next iteration. If the verification fails, a rollback is triggered, as it is shown below in pseudo source code form for the breadthFirstSearch code example and its `top` and `bot` scalar variables.

```
if (top != linear_eq(&top, vi+1) rollback() ;
if (bot != linear_eq(&bot, vi+1) rollback() ;
```

Notice that this verification strategy is verifying initial values for the next iteration according to the original sequential order. Since the current schedule may follow an entirely different order resulting from a parallelizing and optimizing transformation of the original code, some iterations may be run with scalar values that have not yet been verified. But since all iterations are run inside the same chunk (slice of the outermost loop), they have all been verified regarding their scalars as soon as their preceding iteration according to the serial order has been run. Globally, all iterations inside a chunk have been inevitably verified at the chunk completion. If any of these verifications fails during the execution of the chunk, rollback for the whole chunk is initiated and memory is restored.

4.3 Loop Bounds Verification

The polyhedral model imposes loop bounds to be affine functions of the enclosing loop iterators. These bounds can be either extracted at compile-time thanks to the scalar evolution pass, or must be built at runtime through interpolation and handled speculatively. For any target loop nest, bounds of the outermost loop and of the inner loops are handled in different ways.

When undefined at compile-time, the outermost loop bound can only be known after completion of the loop nest execution. Thus it cannot be used by Apollo when analyzing and transforming speculatively the target code. However, as it is addressed in subsection 3.2, the target loop nest is launched by chunks consisting of slices of the outermost loop. Therefore, outermost loop bounds are defined by the starting and finishing borders of the current chunk. When the original loop exit condition is met during the run of a chunk and before its completion, a rollback is initiated and the last chunk is run again in the serial original order. When the outermost loop bound is discovered statically, Apollo's runtime system is able to anticipate the final loop exit by launching the very last chunk of parallel code with the exact convenient size in order to avoid any final rollback. Once they cannot be obtained at compile-time, inner loop iteration counts are being interpolated by Apollo during instrumentation. This is mostly the case with while-loops whose exit conditions are made of values that are unknown at compile-time. For this kind of bounds, predicted values are verified by comparison with the current virtual iterators values. Moreover, the original exit condition must yield the same result. Otherwise, a rollback is initiated.

5 Related Work

TLS systems are a promising solution to automatic parallelization, but suffer from a high overhead, inherent to maintaining speculative states and version management. Attempts to alleviate synchronization in verifying dependences and speculations [11] lead to increased memory management data structures and rely on hardware support.

MiniTLS [12] makes use of a compact version management structure, which however, being centralized, requires thread synchronization. Lector [12], employs the inspector-executor technique, where a lightweight inspector runs ahead and verifies if dependence violations occur. Softspec [13] is a technique whose concepts represent preliminary ideas of our approach, as it is based on a profiling step and a prediction model. However, no code transformations are performed, except slicing. The runtime verification mechanism is similar to the one presented in this paper, as it does not require inter-core communication. However, since Apollo performs advanced code transformations, it must ensure that the last iteration of the original loop may execute before other iterations are executed within the same thread, which yields a more complex verification system. ParExC [14] targets automatic speculative parallelization of code that has been optimized at compile time, but it abounds in runtime checks designed to run in parallel. ParExC speculates on a failure free execution and aborts as soon as a misspeculation is encountered, relying on a transactional memory-based solution. Steffan *et al.* [15] propose a hardware-software co-design of a runtime verification based on the coherence protocol. Recent works of Kim *et al.* [16] describe automatic parallelization on clusters, by speculating on some memory or control dependences. The system executes a master process, non-speculative, and several speculative worker processes. Verification relies on transactional logs and is supported by rollback and recovery mechanisms.

Software transactional memory (STM) [17–19] was proposed to ensure the correctness of speculative code. STM enables a group of read and write operations to execute atomically, embedded in transactions. The reader is responsible for checking the correctness of execution and must ensure that no other thread has speculatively modified the reader's target location. If validation is successful, the transaction is committed, otherwise aborted, causing all of its prior changes to rollback. Despite increasing parallelism (speculatively), STM systems are notorious for the high overhead they introduce. The work of Adl-Tabatabai *et al.* [17] develops compiler and runtime optimizations for transactional memory constructs, using JIT. Static optimizations are employed to expose safe operations, such that redundant STM operations can be removed, while the STM library interface is tailored to handle JIT-compiled and optimized code. STM-lite [18] is a tool for light-weight software transactional memory, dedicated to automatic parallelization of loops, guided by a profiling step. Raman *et al.* [19] propose software multi-threaded transactions (SMTXs), which enable combining speculative work and pipeline transformations. SMTXs use memory versioning and separate the speculative and non-speculative states in different processes. While STMX has a centralized transaction commit manager, conflict detection is decoupled from the main execution.

6 Experiments

Our benchmarks were run on a platform embedding two AMD Opteron 6172 processors, of 12 cores each, at 2.1 Ghz, running Linux 3.11.0-17-generic x86_64. The set of benchmarks has been built from a collection of benchmark suites, such that the selected codes includes a main loop nest and highlights Apollo's capabilities: `backprop` and `needle` from the Rodinia benchmark suite [20], `mri-q`, `sgemm` and `stencil` from the Parboil benchmark suite [21], `maximalMatching` and `breadthFirstSearch` from the Problem Based benchmark suite [6], and finally `2mm` from the Polyhedral benchmark suite [22]. These codes cannot be statically analyzed and transformed for the following reasons: arrays are passed to functions using pointers, thus yielding aliasing issues, dynamic data structures, non-linear array references, conditionals inside loop bodies, while loops, and references to data structures through pointers. We compiled the original codes either using the `gcc` or `clang` compilers, with optimization flag `-O3`, and considered the shortest computation time among both executables, as the baseline

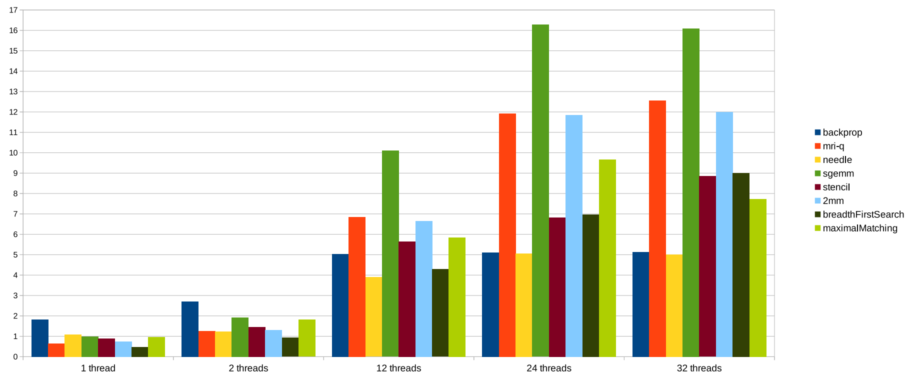

Fig. 1. Speed-ups obtained from codes speculatively parallelized with Apollo

for Apollo's speed-up (Figure 1). Apollo handled each code automatically and transparently. We measured the global resulting execution times of the target loop nests for 1, 2, 12, 24 and 32 threads and computed the resulting speed-up (Original computation time / Apollo's computation time). The execution times with Apollo from one run to another and with the same input were as stable as when running the original codes solely, since Apollo always selects the same transformation. Significant speed-ups were obtained for most of the codes, of up to $16.2\times$. Note that although some of the applications cannot be statically analyzed, they can be parallelized manually by an expert, as it is the case of the benchmarks extracted from Rodinia benchmark suite [20]. As expected, straightforward manual parallelization yields higher speed-ups, since there is no overhead incurred by instrumenting the application, generating code on-the-fly or providing support for a speculative execution. Nevertheless, the advantages of Apollo are emphasized by loops which only exhibit parallel phases (in contrast to OpenMP loops which are parallel for the entire execution), or codes which have a linear behavior and benefit from polyhedral transformations to enhance data locality or exhibit parallelism. Finally, as an automatic system, Apollo is entirely transparent and relieves the user from the parallelization effort, which is known to be an error-prone process.

An analysis of the time-overhead induced by the main processing steps of the runtime system of Apollo shows that the significant amounts of time are spent either in the memory backup (from 0.01% up to 24% of the whole execution time) or in the invocation of Pluto (up to 2%). Memory backup is costly, since it obviously involves many memory accesses. However, it has been optimized and parallelized with Apollo since each thread takes in charge the memory locations that it is supposed to touch in the next execution chunk. This approach also promotes a good data locality. Pluto is an external tool that may spend considerable times in handling some codes. Apollo could use another scheduler or define a time-out to avoid any excessive time spent by Pluto.

To exhibit the gain provided by the decentralized verification system of Apollo, we simulated the behavior of a centralized verification system regarding its additional required memory accesses. For this purpose, we annihilated our verification instructions that are associated to each memory instruction that is speculatively handled, and replaced them by memory writes to random addresses of a buffer which is common to all the parallel threads. Notice that this minimal simulation is still in favor of a centralized system, which would also require some additional processing. The execution time improvements provided by decentralized verification is shown in Figure 2. It shows the significant gain that is particularly obtained when the speed-up potential is high. For example `sgemm`, which is running with Apollo using 24 threads at a speed-up higher than 16×, is highly handicapped by a centralized verification system: in the Apollo parallel execution, data locality is promoted thanks to memory accesses occurring exclusively in separate memory areas, while a centralized system yields an important traffic in the memory hierarchy to ensure cache coherency, thus imposing much memory latency to the threads. Moreover, the gain improvement that can be observed for high speed-up potential codes when increasing the number of threads shows clearly that a centralized verification system does not scale.

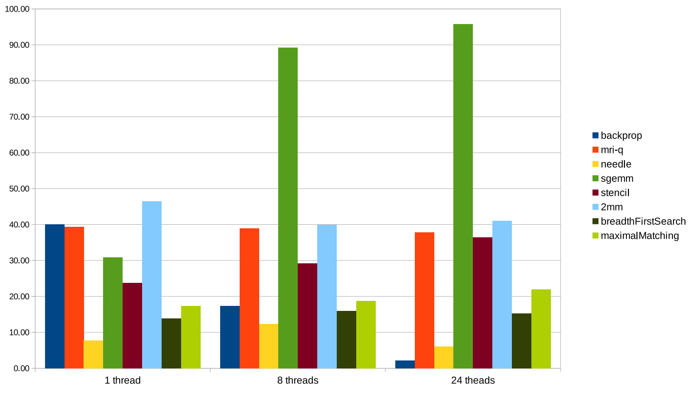

Fig. 2. Percentage of speedup attributable to decentralized verification

7 Conclusion

The software architecture of the Apollo framework is typical of TLS systems which do not require a centralized verification system and are able to apply advanced dynamic code optimizations. It encompasses two main collaborative phases combining static and dynamic analysis and transformation of the target loop nests, and is based on the lightweight construction of a prediction model at runtime. Although Apollo implements a speculative and dynamic adaptation of the polyhedral model, any model providing a sufficiently accurate characterization of the target program semantics could be used as soon as it allows to manage speculative and efficient parallel code. We currently investigate new models for handling codes that are not exhibiting a linear behavior. Alternatively, Apollo also highlights the fact that codes may exhibit interesting optimization opportunities depending on the processed input. This phenomenon opens to investigations related to new memory allocation and access strategies that may be better handled for code parallelization and optimization, either in software or hardware.

References

1. Rauchwerger, L., Padua, D.: The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In: PLDI 1995. ACM (1995)
2. Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., Torrellas, J.: POSH: a TLS compiler that exploits program structure. In: PPoPP 2006. ACM (2006)
3. Raman, E., Vachharajani, N., Rangan, R., August, D.I.: Spice: speculative parallel iteration chunk execution. In: CGO 2008. ACM (2008)
4. Johnson, T.A., Eigenmann, R., Vijaykumar, T.N.: Speculative thread decomposition through empirical optimization. In: PPoPP 2007. ACM (2007)
5. Feautrier, P., Lengauer, C.: Polyhedron model. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1581–1592. Springer, US (2011)
6. Shun, J., Blelloch, G.E., Fineman, J.T., Gibbons, P.B., Kyrola, A., Simhadri, H.V., Tangwongsan, K.: Brief announcement: the problem based benchmark suite. In: SPAA 2012. ACM (2012)
7. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: PLDI 2008. ACM (2008)
8. Jimborean, A., Clauss, P., Dollinger, J.F., Loechner, V., Juan Manuel, M.: Dynamic and Speculative Polyhedral Parallelization Using Compiler-Generated Skeletons. International Journal of Parallel Programming 42(4), 529–545 (2014)
9. LLVM: LLVM compiler infrastructure, `http://llvm.org`
10. Banerjee, U.: Loop Transformations for Restructuring Compilers - The Foundations. Kluwer Academic Publishers (1993)
11. Oancea, C.E., Mycroft, A., Harris, T.: A lightweight in-place implementation for software thread-level speculation. In: SPAA 2009. ACM (2009)
12. Yiapanis, P., Rosas-Ham, D., Brown, G., Luján, M.: Optimizing software runtime systems for speculative parallelization. ACM TACO 9(4), 39:1–39:27 (2013)
13. Bruening, D., Devabhaktuni, S., Amarasinghe, S.: Softspec: Software-based speculative parallelism. In: Workshop on Feedback-Directed and Dynamic Optimization 2000. ACM (2000)
14. Süßkraut, M., Weigert, S., Schiffel, U., Knauth, T., Nowack, M., de Brum, D.B., Fetzer, C.: Speculation for parallelizing runtime checks. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 698–710. Springer, Heidelberg (2009)
15. Steffan, J.G., Colohan, C.B., Zhai, A., Mowry, T.C.: A scalable approach to thread-level speculation. In: ISCA 2000. ACM (2000)
16. Kim, H., Johnson, N.P., Lee, J.W., Mahlke, S.A., August, D.I.: Automatic speculative doall for clusters. In: CGO 2012. ACM (2012)
17. Adl-Tabatabai, A.R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and runtime support for efficient software transactional memory. In: PLDI 2006 (2006)
18. Mehrara, M., Hao, J., Hsu, P.C., Mahlke, S.: Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. SIGPLAN Not. 44(6), 166–176 (2009)
19. Raman, A., Kim, H., Mason, T.R., Jablin, T.B., August, D.I.: Speculative parallelization using software multi-threaded transactions. In: ASPLOS 2010. ACM (2010)
20. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: IISWC 2009. IEEE (2009)
21. Stratton, J.A., Rodrigues, C., Sung, I.J., Obeid, N., Chang, L.W., Anssari, N., Liu, G.D.: mei W. Hwu, W.: The Parboil technical report. Technical report, IMPACT Technical Report, IMPACT-12-01, University of Illinois, at Urbana-Champaign (2012)
22. PolyBench, `http://sourceforge.net/projects/polybench`