Int J Parallel Prog (2014) 42:529–545
DOI 10.1007/s10766-013-0259-4

Dynamic and Speculative Polyhedral Parallelization Using Compiler-Generated Skeletons

Alexandra Jimborean · Philippe Clauss · Jean-François Dollinger · Vincent Loechner · Juan Manuel Martinez Caamaño

Received: 28 February 2013 / Accepted: 27 July 2013 / Published online: 9 August 2013

Abstract We propose a framework based on an original generation and use of algorithmic skeletons, and dedicated to speculative parallelization of scientific nested loop kernels, able to apply at run-time polyhedral transformations to the target code in order to exhibit parallelism and data locality. Parallel code generation is achieved almost at no cost by using binary algorithmic skeletons that are generated at compile-time, and that embed the original code and operations devoted to instantiate a polyhedral parallelizing transformation and to verify the speculations on dependences. The skeletons are patched at run-time to generate the executable code. The run-time process includes a transformation selection guided by online profiling phases on short samples, using an instrumented version of the code. During this phase, the accessed memory addresses are used to compute on-the-fly dependence distance vectors, and are also interpolated to build a predictor of the forthcoming accesses. Interpolating functions and distance vectors are then employed for dependence analysis to select 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

A. Jimborean
UPMARC, University of Uppsala, Uppsala, Sweden
e-mail: alexandra.jimborean@it.uu.se

P. Clauss (✉) · J.-F. Dollinger · V. Loechner · J. M. Martinez Caamaño
ICube, INRIA, CNRS, University of Strasbourg, Strasbourg, France
e-mail: philippe.clauss@inria.fr

J.-F. Dollinger
e-mail: jean-francois.dollinger@inria.fr

V. Loechner
e-mail: vincent.loechner@inria.fr

J. M. Martinez Caamaño
e-mail: juan-manuel.martinez-caamano@inria.fr

code is executed in successive slices of the outermost original loop of the nest. Each slice can be either a parallel version which instantiates a skeleton, 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, by patching differently one of the pre-built skeletons. 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 Algorithmic skeletons · Polytope model · Automatic parallelization · Speculative parallelization · Dynamic parallelization · Loop nests · Compilation

1 Introduction

Automatically parallelizing sequential code became increasingly important with the advent of multicore processors. Particularly, the polyhedral model [2], 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. However, frequently, even scientific 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 [12]. 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 run-time, due to the time overhead generated by the required analysis and transformation phases.

Runtime parallelization techniques are usually based on thread-level speculation (TLS) [15,23,24] 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, the register and memory state is rolled back to a previous correct 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 speculative parallel threads [10,15,24]. As soon as a dependence is carried by the outermost loop, this approach leads to numerous rollbacks and performance drops. Moreover, even if infrequent dependences occur, nothing ensures that the resulting instruction schedule improves performance. Indeed, poor data locality and a high amount of shared data between threads can yield a parallel execution slower than the original sequential one. To gain efficiency, TLS systems must handle more complex code optimizations that can be profitably selected at runtime, depending on the current execution context.

At a higher level of parallel programming, a growing interest has been paid to the use of algorithmic skeletons [4,14], since many parallel algorithms can be characterized and classified by their adherence to one or more generic patterns of computation and interaction. In [4], Cole highlights that the use of skeletons offers scope for static and

dynamic optimization by explicitly documenting information on algorithmic structure which would often be impossible to extract from equivalent unstructured programs. On the other hand, Li et al. propose in [14] a lower level implementation of data-parallel skeletons in the form of a library, as a good compromise between simplicity and expressiveness.

According to these features, we extend the concept of algorithmic skeletons in the following ways. Our skeletons:

- are generated automatically at compile-time and embedded in the final executable file;
- are specialized to the loop nests that are initially marked in the source code using a dedicated pragma;
- implement a given combination of polyhedral loop transformations as loop interchange, skewing or tiling [2];
- have a fixed algorithmic structure with instructions of three categories: (1) original program instructions, (2) polyhedral transformation instantiation and (3) speculation management, where categories (2) and (3) contain parameterized code, i.e., with unknown values of some variables;
- are rendered executable at runtime by assigning values to the parameters of instruction categories (2) and (3).

Such skeletons, embedded in the executable file of the target program, significantly improve dynamic and speculative parallelization opportunities by allowing very fast code generation and advanced automatic parallelizing transformations of loop nests. Unlike previous works [11,17,27] which report on simple patterns (templates) with "holes", replacing branches, that are filled dynamically, we design skeletons which dynamically instantiate a polyhedral loop transformation and embed instructions to manage speculative parallelization. Starting from the skeletons, different, optimized code versions can be generated, by assigning values to some parameters.

To support the generation and the use of these skeletons, 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, devoted to loop nests that exhibit linear behavior phases at runtime. We define linear behavior phases as being characterized by outermost loop slices where:

- All accessed memory addresses can be represented as affine functions of the loop indices;
- All loop bounds, except the outermost, can be represented as affine functions of the enclosing loop indices;
- The values assigned to some particular variables, called *basic scalars*, can also be represented as affine functions of the loop indices. These scalars are detected at compile-time as being variables defined by ϕ-nodes in the LLVM Static Single Assignment (SSA) intermediate representation. They have the interesting property of being at the origin of the computations of all other scalars used in the loop bodies.

Our contributions can be summarized as:

- Perform dynamic and advanced loop transformations and generate efficiently the resulting parallel code by using compiler-generated algorithmic skeletons at runtime.

- Exhibit parallelism in codes that cannot be parallelized in the original form.
- Adapt dynamically to the current behavior of the code and apply a suitable skeletal code transformation.
- Exploit partial parallelism, i.e., parallelism that can only be exploited on some slices of the outermost loop.
- Apply the polyhedral model on *for*, *while* and *do-while* loops that exhibit linear behavior phases;
- Do not require any hardware support dedicated to speculative parallelization.

Preliminary ideas of our speculative system are presented in our previous work [7], while the proposal presented in [8] details the instrumentation and analysis phase, required for building an abstract model of the loop nest. The current paper focuses on using the abstract representation of the loops in order to automatically perform speculative optimizations and parallelization at runtime. All aspects and details of the whole framework can be found in [9].

We present in Sect. 2 an overview of the related work. Sect. 3 provides an overall description of our framework, while Sect. 4 details the algorithmic skeletons and the mechanisms for prediction, initialization and verification; Finally, the experiments we conducted to evaluate VMAD are presented in Sect. 5 and the conclusions and perspectives in Sect. 6.

2 Related Work

Recent works of Kim et al. [12] describe a speculative DOALL system, targeting automatic parallelization on clusters, by speculating on some memory or control dependences. SPEC DOALL shows how sensitive static analysis is to the implementation style, emphasizing the urge for speculative parallelization.

Similar to speculative parallelization, the purpose of dynamic optimizations is to perform transformations adapted to the current execution behavior. Hence it relies on a phase of dynamic profiling, followed by code transformations applied on-the-fly. The most common approach for performing dynamic optimizations is to use JIT compilation [1,13]. While dynamic transformations offer a solution to codes which cannot be statically optimized, applying software dynamic translation can be prohibitively costly, especially on codes with a changing behavior. Until now, limited progress has been reported on complex, dynamic optimization of loop nests, due to heavy processes that must be performed at runtime, such as data dependence analysis and code generation. Nevertheless, it is our goal to address these problems.

The Polytope model, known also as the *polyhedral model*, provides powerful mathematical theory to represent loop nests and to reason about the correctness of complex loop transformations. The polyhedral model is dedicated to loop nests in which the memory access functions and the loop bounds are affine functions of the enclosing loop iterators, typically matrix computations. Particularly, it is applicable on statically analyzable loop nests (static control structures, affine bounds and memory references), as it requires a precise characterization of data dependences. A detailed description on how the model is applied in a research compiler is given in [2]. Empirical tests show that polyhedral optimizations only, without parallelization, can lead to

speed-ups of up to 14× [20], solely by improving data locality. However, applying the full traditional methodology is extremely time-consuming. To benefit from such a powerful optimization model, we propose a lightweight system, able to fill-in at runtime the missing information, to perform polyhedral transformations dynamically and to automatically parallelize loops, thanks to compiler-generated skeletons used at run-time. To our knowledge, VMAD is the very first attempt to apply speculatively and dynamically the polytope model.

Code patterns have a long history of addressing runtime code specialization and simple optimizations. Noël et al. [17] use templates with "holes" declared as external global variables that are filled at runtime. They perform optimizations based on constant propagation, strength reduction and loop unrolling. This proposal sets the premises of our work of performing more advanced loop optimizations dynamically using skeletons. More recent works [11] extend this approach and use value prediction based on affine functions to build their templates. They are specialized by statically precomputing values of the affine functions at some specific points. Runtime instantiation represents selecting one of the precomputed values, thus the overhead is negligible. This proposal invites to more advanced loop optimizations, however, in this form, template instantiation relies on statically known values. We extend this, by using skeletons performing re-scheduling of loop iterations, on codes for which the parameters cannot be pre-computed statically. In another proposal [27], patterns are seen as pre-compiled fragments of code stitched together dynamically.

To our knowledge, the closest work addressing skeletons generated at compile-time is the Bones source-to-source compiler [18]. Taking as input C code with some algorithm class information provided by the user, Bones generates CUDA or OpenCL codes for GPU and x86 processors through skeleton instantiation, helped by a library of parameterized host code and kernel code. The main difference is that our skeletons (also generated at compile-time), still contain parameterized code which is instantiated at runtime. Furthermore, our skeletons help also in orchestrating the speculative parallelization.

3 Overview of Our System

This proposal extends our work on designing a TLS framework able to apply polyhedral transformations at runtime [7], 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 [8]. 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. All code manipulations are performed in the intermediate representation (LLVM IR), hence our framework is both programming language and target agnostic.

Using the chunking mechanism presented in Fig. 2, we slice the iteration space of the outermost loop in successive chunks. The bounds of the chunks are determined dynamically to adapt to different execution phases of a loop. Execution starts with a

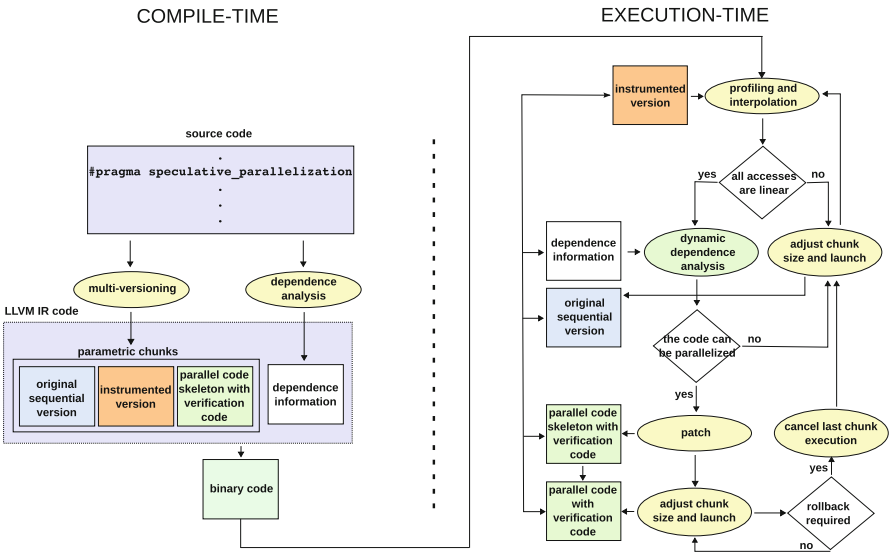


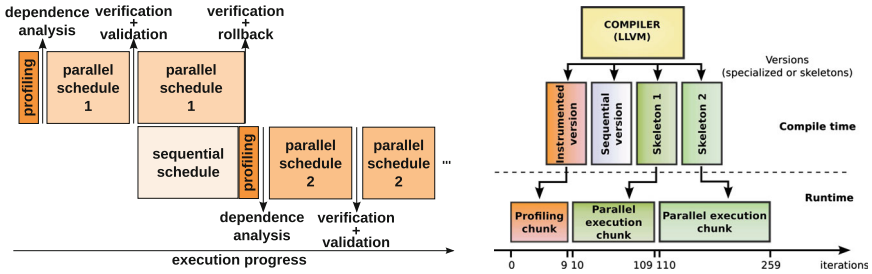
Fig. 1 Static-dynamic collaborative framework

profiling phase, whose results are used to validate a suitable polyhedral transformation. Next, a new transformation is proposed for each phase, and a customized parallel version is generated, by patching the skeleton. If several parallelizing transformations are validated, VMAD includes a module dedicated to select the best performing code version. This is achieved by launching successively small chunks embedding one of the possible parallel versions, and selecting the best performing one, according to the resulting average of execution time per iteration. Phase detection is detailed in the end of this section and relies on the code dedicated to monitor the speculations.

During the speculative execution, the predictions are verified, initiating a rollback upon a misspeculation and resuming execution with a sequential chunk. If validation succeeds, a new parallel chunk is launched. The implementation of VMAD consists of two parts: a *static* part, implemented in the LLVM compiler [16], designed to prepare the loops for instrumentation and parallelization, and generate customized parallel skeletons, 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, instantiate the parallel skeleton code and guide the execution, as illustrated in Fig. 1. Since the compiler is target agnostic, the framework is independent of the target architecture (as long as LLVM provides a back-end), nevertheless, the runtime system has to be ported on the new architecture.

Static component Our modified LLVM compiler generates customized versions of each loop nest of interest: original, instrumented and several algorithmic skeletons each supporting a specific class of polyhedral transformations, together with a mechanism for switching between the versions (a decision block preceding the code versions).

To complete the loop's execution and adapt to the current phase, we automatically link up at runtime the different versions of the original code. Each version is launched in a chunk to execute a subpart of the loop and another one continues, as in relay races.



(a) The chunking mechanism

(b) Alternate execution of different versions during one loop nest's run

Fig. 2 Multiversioning

The support for chunking the outermost loop and linking distinct versions is illustrated in Fig. 2b. The instrumented, original and two skeletons are built at compile time. At runtime, one or another version is automatically selected to be executed for a number of iterations.

We build skeletons from which several parallel code versions can be generated at runtime, by patching predefined code areas. The skeletons 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. Also, patching a binary skeleton is considerably faster than fully dynamic code generation using just-in-time (JIT) compilation. However, the limitation of this approach is that it can only support a subset of the possible polyhedral transformations, namely those preserving a given loop structure and avoiding to reorder the statements inside the body of the loops. More details on the design of the skeletons are given in the next section.

A set of polyhedral loop transformations, defined as matrices, is generated statically and is encoded in the data section of the binary file. Their computation follows a static dependence analysis, which ensures that the dependences which can be statically identified will not invalidate the schedules, thus preventing useless scheduling alternatives. Generally, even if no static dependence can be identified to guide the selection of transformations, it is convenient to consider "classic" transformation matrices that are unimodular and resulting in combinations of loop exchange and skewing.

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 to launch parallel code as soon as possible. Thus, the computed linear functions speculatively characterize the behavior of the loop. Instrumentation is followed by a dependence analysis which evaluates whether any of the proposed polyhedral transformations can be efficiently applied. If successful, the runtime system assigns values to the coefficients of the linear functions in the corresponding code skeleton and launches it.

The dynamic dependence analysis is an incremental process that computes the distance vectors used in validating polyhedral transformations, based on the actual

memory addresses accessed during the run of an instrumented chunk, and on the functions interpolating them. In order to ensure that the computed distance vectors entirely characterize the current code behavior, the interpolating functions are used to check if any memory instruction couple, where at least one is a write, and for which no distance vectors have been computed, may carry a dependence. This is achieved by a fast value range analysis of the touched memory addresses and a GCD test. Additional information is available in [9].

The speculative execution is monitored by the runtime system. As soon as a misspeculation is identified, it is followed by a rollback which restores the memory to a correct state. For this purpose, the runtime system creates proactively a copy of the memory area that is going to be modified by the next parallel chunk, since it can be predicted using the interpolating linear functions. In case a rollback is performed, the memory is overwritten with the content of the copy, and the rollbacked iterations are re-executed using the original sequential schedule; next instrumentation is re-initiated. Otherwise, a new parallel chunk is launched.

Our system dynamically adapts to each phase of the loop by launching the corresponding code version for a subset (chunk) of iterations. Thus, the size of each chunk is computed based on the currently observed behavior of the loop: a stable behavior leads to an increase of the chunk size until a fixed threshold, whereas a change in the behavior resets the size to a default starting value.

Next, we focus on the dynamic, lightweight code generation of parallel optimized code using skeletons.

4 Binary Skeletons

The skeletons are prepared statically (corresponding to Skeleton 1 and Skeleton 2 versions in Fig. 2b) and instantiated at runtime to generate distinct code versions (parallel schedule 1 and parallel schedule 2 in Fig. 2a). For the purposes of clarity, we start with a pedagogical example in which the code can be statically analyzed, and we detail in what follows the implications on non-statically analyzable code and our solutions to handle it.

First, consider the simple two-loop nest, with indices i, j in the first column of Table 1. Performing the affine transformation (i, j) → (x, y)=(i+j, i) on the original loops, one obtains a new version. We can then rewrite the code to loops on x and y instead of i and j, obtaining the skewed routine from Table 1, second column. As one can notice, the loop structure remained the same (except the loop bounds and the initialization code), despite the affine transformation that has been applied. By

Table 1 Simple loop transformations

```
do i = 1,6
 do j = 1,5
  A(i,j) =
  A(i-1,j+1)+1
```

```
do x = 2,11
 do y = max(x-5,1),
        min(6,x-1)
  i = y
  j = x-y
  A(i,j) =
  A(i-1,j+1)+1
```

```
do x = low_x, upp_x
 low_y = max(a*x+b,cst)
 pp_y = min(c*x+d,cst)
 do y = low_y,upp_y
  i = e*x+f*y+g
  j = h*x+k*y+l
  A(i,j) =
  A(i-1,j+1)+1
```

rewriting the loop bounds and the memory accesses as generic affine functions of the enclosing loop indices, we can build a skeleton from which an infinite number of parallelizing transformations can be applied, provided that the loop structure and the order of the statements remain unchanged, as shown in the third column of Table 1. At runtime, the coefficients of the linear functions computing the loop bounds and the original iterators are assigned values according to the affine transformation to be applied. Each set of coefficients is equivalent to a new polyhedral transformation. A skeleton could, for example, support loop skewing combined with loop interchange, in which the *first loop level* is parallel, while another skeleton can combine the same two types of transformations with *the second loop level as parallel*. By assigning different values to the coefficients in the same skeleton, different skewed and/or interchanged loop versions can be obtained. Similarly, one can design new skeletons to support other classes of polyhedral transformations.

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 2, column 1 (original version) and its equivalent form in column 2 with virtual iterators (sequential version in Fig. 2b). The virtual iterators are part of our chunking mechanism, allowing the runtime system to control the number of executed iterations in each chunk, independently of the executed version's nature: instrumented, sequential or parallel.

To preserve the correct semantics of the original code and to perform the required speculation management tasks, the skeletons are completed with *guarding, initialization* and *verification code*, as shown in column 3 and detailed below. Additionally, skeletons include explicit calls to the GOMP/OpenMP library [5] to spawn parallel threads. All these code manipulations are performed at the intermediate representation level.

Table 2 Simplified skeletons

```
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 ≠ αx+βy+γ)
      rollback();
     else continue;}
    else {
     p->val += q->val;
     q = q->next; }
    if (guarding code)
     p=p->next;
} } } }
```

4.1 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 [26]. For this purpose, we use an implementation of the algorithm available in the software library FM [21]. Note that the conditions of the original while loops are preserved in the code by copying the original loop bodies in the skeleton, 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. We call this *guarding code* and it is aimed to verify our speculation on the loop bounds, as detailed in Sect. 4.3. The not perfectly nested instructions are embedded in conditionals which ensure that they are executed only at the right iterations. The *guarding code* is inserted in the innermost loop, thus allowing various affine transformations combined with loop interchange.

4.2 Initialization Code

We use the linear functions obtained from the profiling phase, to initialize the basic scalars at runtime. In Table 1, the basic scalars are the original iterators i and j, preserved in the skeleton code to ease the computations, while in Table 2, the basic scalars are p, q and $j : p$ and q correspond to the phi-nodes in the original code and their values are employed in the computation of accessed memory locations, whereas j contributes to the computation of the exit conditions of the subloop.

Our value prediction mechanism is similar to the ones presented in [22,28]. 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 T. For example, for a loop nest of depth 2:

$$T \cdot \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \Leftrightarrow \begin{pmatrix} i \\ j \end{pmatrix} = T^{-1} \cdot \begin{pmatrix} x \\ y \end{pmatrix},$$

we obtain a new loop version in x and y. The bounds of the loops and the coefficients of the linear functions are assigned values dynamically. The coefficients of the linear functions are computed by applying the transformation matrix T on the predicting linear functions obtained from profiling and linear interpolation.

4.3 Verification Code for Speculative Parallelization

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, our speculations consist of the linear functions that predict the memory addresses being accessed. 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 skeletons 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.*

We divide the type of verification in three categories, depending on the instances being verified:

1. *Basic scalars.* When the execution of the iteration completes, we verify that the value computed by the code of the loop body and the value we predict coincide. For this verification, *we compare the actual value with the one expected for the next iteration according to the sequential order*. Validation of basic scalars ensures that all values computed in the loop reach the predicted values. Hence, the result of the dependence analysis is preserved as long as the interpolating linear functions used for initializations and verifications remain valid.
2. *Memory accesses.* Note that some iterations might execute before being validated by the preceding iteration according to the sequential order. Hence, one is required to verify all memory accesses performed in the current iteration to ensure that each targeted location has been correctly predicted. This has twofold consequences. First, it ensures that no invalid access is performed. And second, it guarantees that the memory state can be safely restored, as no modification outside the predicted memory is done. Although memory accesses are verified prior to being performed, indirect array addressing can still be handled by our system, since it is modelled as two separate memory accesses, each of them verified independently.
3. *Loop bounds.* The iteration counts of the subloops have a direct role in the polyhedral transformation being applied. The verification code relies heavily on the *guarding* code, presented previously. The transformed loop bounds control the number of iterations to be executed by each loop of the nest and together with the *guarding code*, it must be verified that: (i) each loop executes *all* its iterations, (ii) but no loop executes more iterations than it should. Due to the out-of-order execution of the iterations, the code must allow the execution of the last iteration of a loop (according to the sequential order) without exiting, as it might be followed by other iterations according to the parallel schedule. As an example, consider the transformed loop in the code snippet in Table 2. The bounds of the outermost loop cannot be predicted, therefore a rollback is triggered when the original outermost condition becomes false. In contrast, the subloops' bounds are interpolated, and, thanks to this prediction, the execution order of their iterations can be changed, while precisely controlling the loop exiting iterations.

5 Experimental Results

In this section we present the experiments we conducted to evaluate our approach of applying the polyhedral model at runtime, in the view of speculatively parallelizing the loop nests. Our benchmarks were run on two architectures. The first platform embeds two AMD Opteron 6172, of 12 cores each, at 2.1 GHz, running Linux 3.2.0-27-generic x86_64, while the second platform is an Intel Xeon X5650 at 2.67 GHz, with 12 cores hyper-threaded, running Linux 3.2.0-24-generic x86_64. We have selected a set of benchmarks from different sources: the polyhedral benchmark suite [19], the Rosetta codes [25], the Rodinia benchmark suite [3] and the DSPstone benchmarks [6]. Notice that although some of these codes could have been handled statically, they are used to show the effectiveness of our system. Hence, we have modified them to use dynamically allocated arrays or pointers, which would prevent static analysis. Our measurements are given in Table 3. We compare the speed-up of our

Table 3 Code speculatively parallelized with VMAD, compared to OpenMP

Program	No. of threads	AMD Opteron 6172		Intel Xeon X5650		Polyhedral Transf.
		Speed-up VMAD	Speed-up OpenMP	Speed-up VMAD	Speed-up OpenMP	
adi	12	1.78	13.49	1.80	5.21	$\begin{pmatrix}1&0\\0&1\end{pmatrix}$
	24	1.82	13.34	4.09	4.75	Identity 1st loop par
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 1st loop par
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. 2nd loop par
floyd	12	0.77	N/A	0.73	N/A	$\begin{pmatrix}1&0\\0&1\end{pmatrix}$
	24	1.43	N/A	0.73	N/A	Identity 2nd loop par
fir2dim	12	2.74	N/A	3.29	N/A	$\begin{pmatrix}1&0\\0&1\end{pmatrix}$
	24	2.61	N/A	2.93	N/A	Identity 1st loop par
covariance	12	4.30	6.19	4.03	5.86	$\begin{pmatrix}1&0\\0&1\end{pmatrix}$
	24	7.45	12.07	4.55	8.92	Identity 1st loop par
correlation	12	4.29	6.26	3.88	5.73	$\begin{pmatrix}1&0\\0&1\end{pmatrix}$
	24	7.47	12.18	4.64	8.55	Identity 1st loop par
qr_decomp	12	2.87	12.02	2.11	11.03	$\begin{pmatrix}1&0\\0&1\end{pmatrix}$
	24	4.69	20.02	2.48	12.22	Identity 1st loop par
grayscale	12	1.81	8.73	1.13	3.59	$\begin{pmatrix}1&0\\0&1\end{pmatrix}$
	24	2.03	6.61	1.25	2.28	Identity 1st loop par

system for the target loop nests relatively to manual parallelization using OpenMP, when such a parallelization is possible, on both architectures, with 12 and 24 threads. Note that the speed-up obtained by OpenMP code is the highest that can be reached with straightforward parallelization (identity transformation matrix), since it does not require any dynamic analysis, dynamic code generation or support for speculative execution. In contrast, it places the burden on the programmer to analyze and parallelize the code, and to ensure its correctness. On the other hand, VMAD is entirely automatic and does not rely on any hardware support for speculative executions. Thus, it is readily applicable on any target architecture, merely by porting the runtime system, at the cost of an inherent overhead for offering a purely software support for speculative parallelization.

One of the interesting outcomes is that the behavior of the same code on the two different architectures is very different, in terms of scalability with the numbers of threads. We observed that this behavior is not specific to VMAD, as we obtained similar results when parallelizing the codes manually, with OpenMP. Each processor is better adapted for a particular type of applications, and the codes benefit differently from the hardware support such as the hardware prefetcher, the branch predictor, etc.

Additional experiments with a varying number of threads showed that even without any transformation other than straightforward parallelization of the outermost loop, our system outperforms OpenMP, thanks to the execution in chunks of the loops, which is similar to strip-mining, having a positive effect on data locality (in Table 3 we show only the results with 12 and 24 threads). With `adi` on the Intel processor, the execution with 16 threads using our system is slightly better than the code parallelized with OpenMP. However, the benefits of chunking and parallelization are hidden by the overhead when running `adi` on the AMD processor. We analyzed the overhead and concluded that it stems from copying the data prior to parallelization using `memcpy`. More details on the overhead of our system are given in the end of this section.

An interesting example is `backprop`, which can be parallelized in its original form using OpenMP. This code is handled similarly by any traditional TLS system, by parallelizing the outermost loop. On the other hand, our system discovers that a loop interchange is possible, which brings significant performance gains, by improving data locality. This benchmark underlines the fact that our system can significantly improve embarrassingly parallel codes, unlike traditional TLS systems, even outperforming manual parallelization. Additionally, it shows that the runtime overhead of the system is hidden by the gains provided by applying the polyhedral transformation.

Another example highlighting this contribution to the state of the art is `cholesky`, which is not parallel in its original form. Therefore, previous TLS systems cannot handle this code, nor can it be manually parallelized with OpenMP, since every loop carries dependences. In contrast, our system analyses the runtime behavior of the code and finds a suitable polyhedral transformation which allows the loop from the second level in the transformed code to be executed in parallel, as shown in the last column of Table 3. Performance can be further improved by generating a skeleton handling tiling transformations, which is one of the first targets of our future work.

The example `floyd` illustrates the capability of our system to adapt dynamically to the behavior of the code and to exploit partial parallelism. This benchmark embeds a conditional that does not allow parallelization because it does not have a predictable

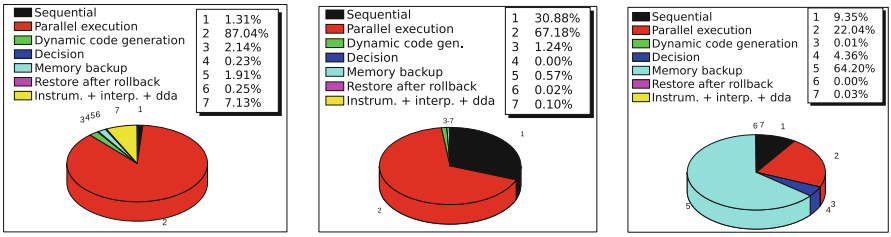
behavior during the first iterations. Nevertheless, our system executes a sequential chunk and monitors again the loop. The second profiling phase identifies that one branch of the conditional is now always executed, and parallelization becomes possible. Moreover, the result of the dependence analysis indicates the second loop level as being parallel, which is not handled by traditional TLS systems. Although the inherent overhead of the system stemming from the support for speculative execution hides the benefits of parallelization, the benchmark is a suitable candidate to exhibit VMAD's capabilities of performing partial parallelization. We detail on the sources of overhead in the end of the section, emphasizing the main bottlenecks and suggestions on how the penalties can be reduced.

The benchmark `fir2dim` contains a loop nest of depth 3 performing memory accesses via pointers. Arrays are represented as dynamically allocated pointers and their parsing is performed by using pointer arithmetic. OpenMP fails to parallelize such codes, due to the impossibility of predicting the starting value of the pointers for each thread. On the contrary, our system is successful in parallelizing these examples, thanks to its instrumentation phase, which builds interpolating linear functions.

Other examples, such as `covariance` and `correlation` show that codes parallelized with our system have a good overall performance. Nevertheless, most of the figures indicate that the speed-up could be considerably improved by reducing the overhead, thus we plan to revise some of our implementation and design choices.

Various strategies could be employed in order to validate or guide the optimizations applied dynamically, such as guarding the loop nests with tests and proposing one code version or another, depending on the some key values unknown statically. However, we aimed to design a more general approach for codes on which such conditionals would not suffice. Codes such as `fir2dim`, `qr_decomp` and `grayscale` use dynamic allocation and pointers that cannot be handled properly using pointer analysis, since memory access functions depend on the input size unknown at compile-time, thus making them non-linear, but linear at runtime. The same issue about non-linearity arises with `cholesky`. Hence, accurate static dependence analysis is impossible. A conditional code would even correspond to something close to our system itself. For `backprop`, loop interchange is not always beneficial, depending on the size parameters translating to loop bounds. Flexibility is required. Our system is adapted to component programming and the use of library code which are becoming prevalent, and not in favor of pointer analysis requiring the whole program. Also, spurious and infrequently occurring dependences are treated conservatively with static analysis, to produce sound results across all inputs. It could not handle codes like `floyd`. All these facts argue for our dynamic approach for maximizing parallelism in the multicore era. We ran the well-known polyhedral static parallelizer Pluto [2] on the benchmark codes when possible (for-loops, linear array accesses): on `cholesky`—that had to be rewritten with linear accesses–, we outperform Pluto thanks to our chunking system; Pluto is not exchanging loops in `backprop`; similar execution times are obtained with the other codes, except `adi`, where our system is outperformed.

The systems's runtime overhead As expected, the overhead's impact strongly depends on the characteristics of the code, since it is relative to the time of the total execution. Thus, for loop nests in which the outermost loop has a large number of iterations, the profiling phases, consisting in instrumentation, interpolation of



(a) covariance **(b)** backpropagation **(c)** adi

Fig. 3 Runtime overhead

memory accesses and dynamic dependence analysis, have almost a negligible overhead. In practice, we noticed that in many situations, this overhead did not pose significant problems. In Fig. 3a–c we depict the time taken by each phase of executing codes with our system, relative to the total execution. The `Sequential` phase refers to the execution of the last chunk, which is always run sequentially, to ensure that all iterations of the loop were executed. `Dynamic code generation` is the time taken to specialize the skeletons, using runtime information. `Decision` is the time taken by the runtime system to select the code version run by the next chunk and to set its size. `Memory backup` is the safe copy that is performed before launching a speculative chunk. `Restore after rollback` is the time required to restore the correct state of the memory, upon a misprediction. And finally, the time taken to instrument the code, interpolate the results and run the dynamic dependence analysis is depicted as `Instrumentation + interpolation + dda`. In each column of Fig. 3, the pie chart illustrates the total execution time divided in the time taken by each action, when running on the AMD Opteron 6172. One can notice that dynamic code generation is negligible, which argues in favor of using binary skeletons. Similarly, the overhead incurred by instrumentation and dependence analysis is minimal, thanks to our sampling mechanism. Nevertheless, our strategy to back-up memory is costly, and requires further refinements. A first improvement would be to include back-up each location independently prior to be speculatively accessed. This strategy would parallelize the memory back-up process and thus provide better performance.

6 Conclusions and Perspectives

In this paper, we showed a new use of algorithmic skeletons, as an efficient support for speculative and dynamic parallelization, and proved that they can be automatically generated and then specialized to target codes and cover a large class of advanced runtime code transformations at a low cost.

Thanks to this automatic skeletal parallelization, 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. The system can handle codes in any form and is not hindered by the type of memory allocation, being capable of handling pointers, static or indirect array accesses, multi-dimensional or linearized arrays, or any types of linked data

structures, as soon as a linear memory behavior has been detected in some execution phases. Moreover, unlike OpenMP, we can handle multiple exit loops and pointer-chasing loops. We conclude by reminding the main contributions, underlined by the benchmarks.

1. VMAD is able to automatically parallelize codes which do not exhibit parallelism in their original form. Thus, they cannot be handled efficiently by existing TLS systems (due to numerous rollbacks) and cannot be parallelized manually, unless they are transformed.
2. VMAD 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.

In the near future, we plan to extend the use of skeletons in order to provide more freedom to the runtime system on the kind of optimizing and parallelizing transformations. It will consist in building elementary skeletons at compile-time that will be assembled at run-time, following an enclosing algorithmic skeleton associated to a specific class of transformations. Transformations that change significantly the structure of the original sequential code will then also be handled efficiently by the system.

Moreover, since the framework follows a modular approach, it can be easily extended to target new types of optimizations (e.g. vectorization) following the results of analysis phases.

References

1. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: a transparent dynamic optimization system. In: PLDI '00. ACM (2000)
2. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: PLDI '08. ACM (2008)
3. 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, pp. 44–54. IEEE (2009)
4. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput. **30**(3), 389–406 (2004)
5. GOMP An OpenMP implementation for GCC—GNU Project. http://gcc.gnu.org/projects/gomp
6. http://www.ice.rwth-aachen.de/research/tools-projects/entry/detail/dspstone/
7. Jimborean, A., Clauss, P., Pradelle, B., Mastrangelo, L., Loechner, V.: Adapting the polyhedral model as a framework for efficient speculative parallelization. In: PPoPP '12 (2012)
8. Jimborean, A., Mastrangelo, L., Loechner, V., Clauss, P.: VMAD: an advanced dynamic program analysis and instrumentation framework. In: OBoyle, M. (ed.) Compiler Construction, Lecture Notes in Computer Science, vol. 7210, pp. 220–239. Springer, Berlin, Heidelberg (2012)
9. Jimborean, A.: Adapting the polytope model for dynamic and speculative parallelization. PhD Thesis, University of Strasbourg, France (2012). http://tel.archives-ouvertes.fr/tel-00733850
10. Johnson, T.A., Eigenmann, R., Vijaykumar, T.N.: Speculative thread decomposition through empirical optimization. In: PPoPP '07. ACM (2007)
11. Khan, M.A., Charles, H.P., Barthou, D.: Improving performance of optimized kernels through fast instantiations of templates. Concurr. Comput. Pract. Exp. **21**(1), 59–70 (2009)
12. Kim, H., Johnson, N.P., Lee, J.W., Mahlke, S.A., August, D.I.: Automatic speculative doall for clusters. In: CGO '12. ACM (2012)
13. Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., Cox, D.: Design of the java hotspot client compiler for java 6. ACM Trans. Archit. Code Optim. **5**, 7–32 (2008)

14. Li, C., Gava, F., Hains, G.: Implementation of data-parallel skeletons: a case study using a coarse-grained hierarchical model. In: ISPDC, pp. 26–33 (2012)
15. Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., Torrellas, J.: POSH: a TLS compiler that exploits program structure. In: PPoPP '06. ACM (2006)
16. LLVM compiler infrastructure. http://llvm.org
17. Noël, F., Hornof, L., Consel, C., Lawall, J.L.: Automatic, template-based run-time specialization: implementation and experimental study. In: International Conference on Computer Languages. IEEE Computer Society Press (1998)
18. Nugteren, C., Corporaal, H.: Introducing 'Bones': a parallelizing source-to-source compiler based on algorithmic skeletons. In: Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, GPGPU-5, pp. 1–10. ACM, New York, NY, USA (2012). doi:10.1145/2159430.2159431
19. Polybenchs. (2010). http://www-rocq.inria.fr/pouchet/software/polybenchs
20. Pouchet, L.N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P., Vasilache, N.: Loop transformations: convexity, pruning and optimization. In: POPL '11. ACM (2011)
21. Pouchet, L.N.: FM: the Fourier-Motzkin library. (2008). http://www.cse.ohio-state.edu/pouchet/software/fm
22. Prabhu, M.K., Olukotun, K.: Using thread-level speculation to simplify manual parallelization. In: PPoPP '03. ACM (2003)
23. Raman, E., Vachharajani, N., Rangan, R., August, D.I.: Spice: speculative parallel iteration chunk execution. In: CGO '08. ACM (2008)
24. Rauchwerger, L., Padua, D.: The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In: PLDI '95. ACM (1995)
25. Rosetta Codes. (2011). http://rosettacode.org/wiki/Rosetta_Code
26. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, NY, USA (1986)
27. Smith, F., Grossman, D., Morrisett, G., Hornof, L., Jim, T.: Compiling for template-based run-time code generation. J. Funct. Program. **13**(3), 677–708 (2003)
28. Tian, C., Feng, M., Gupta, R.: Speculative parallelization using state separation and multiple value prediction. In: International Symposium on Memory Management, ISMM '10. ACM (2010)