APOLLO
Automatic speculative POLyhedral Loop Optimizer

Juan Manuel Martinez Caamaño
INRIA CAMUS, ICube Lab.
Univ. of Strasbourg, France
jmartinezcaamao@gmail.com

Aravind Sukumaran-Rajam
Dpt of Computer Science and Engineering
Ohio State University, USA
sukumaranrajam.1@osu.edu

Artiom Baloian
INRIA CAMUS, ICube Lab.
Univ. of Strasbourg, France
artiom.baloian@inria.fr

Manuel Selva
INRIA CAMUS, ICube Lab.
Univ. of Strasbourg, France
manuel.selva@inria.fr

Philippe Clauss
INRIA CAMUS, ICube Lab.
Univ. of Strasbourg, France
philippe.clauss@inria.fr

ABSTRACT

A few weeks ago, we were glad to announce the first release of Apollo, the Automatic speculative POLyhedral Loop Optimizer. Apollo applies polyhedral optimizations on-the-fly to loop nests, whose control flow and memory access patterns cannot be determined at compile-time. In contrast to existing tools, Apollo can handle any kind of loop nest, whose memory accesses can be performed through pointers and indirections. At runtime, Apollo builds a predictive polyhedral model, which is used for speculative optimization including parallelization. Being a dynamic system, Apollo can even apply the polyhedral model to nonlinear loops. This paper describes Apollo from the perspective of a user, as well as some of its main contributions and mechanisms, including the just-in-time polyhedral compilation, that significantly extends the scope of polyhedral techniques.

1. INTRODUCTION

Is it legal to parallelize the code in Listing 1? Can you apply tiling? Can your polyhedral compiler handle it?

Listing 1: Sparse Matrix-Matrix multiplication

```
for(row = 1; row <= left->Size; row++) {
  Pelem = left->FirstInRow[row];
  while(Pelem) {
    for(col = 1; col <= cols; col++) {
      result[row][col] +=
        Pelem->Real * right[Pelem->Col][col];
    }
    Pelem = Pelem->NextInRow;
  }
}
```

IMPACT 2017
Seventh International Workshop on Polyhedral Compilation Techniques
Jan 23, 2017, Stockholm, Sweden
In conjunction with HiPEAC 2017.
http://impact.gforge.inria.fr/impact2017

The polyhedral model [9], or polytope model, is well-known for performing aggressive loop transformations devoted to parallelism and data-locality. Although very effective, compilers relying on this model [3, 10] are restricted to a small class of compute-intensive codes that can only be handled at compile-time. However, most codes are not amenable to this model, because they use dynamic data structures accessed through indirect references or pointers, which prevent a precise static dependency analysis. On the other hand, Thread-Level Speculation (TLS) [18] is a promising approach to overcome this limitation: regions of code are executed in parallel before all the dependencies are known. Hardware or software mechanisms track register and memory accesses to determine if any dependency violation occurs. But traditional TLS systems implement only a straightforward loop parallelization strategy, consisting of slicing the target loop into consecutive parallel threads, where each thread follows the original serial schedule of loop iterations and statements.

A few weeks ago, we were glad to announce the first release of Apollo [1], which is a TLS software framework implementing a speculative and dynamic adaptation of the polyhedral model, where parallelizing and optimizing transformations are performed on-the-fly for loops exhibiting a polyhedral-compliant behavior at runtime. This software is the outcome of seven years of research and developments, three PhD theses [11, 20, 15] and several Master theses. It is based on an initial prototype called VMAD [13, 12], which was implementing some seminal concepts, that were later improved and extended with Apollo [21, 22, 23, 5].

We begin the next Section with an overview of the framework in Subsection 2.1, then some performance results in Subsection 2.2, and finally in Subsection 2.3, we describe Apollo from the user's perspective and define the kinds of codes that may be good candidates. Then, we present two key concepts: Section 3 details the *prediction model*, which enables to detect polyhedral-compliant runtime behaviors and to perform speculative polyhedral optimization; and Section 4 details Apollo's hybrid code generation mechanism based on *code bones*, which allow to generate on-the-fly, optimized code resulting from any polyhedral transformations. Section 5 describes the pitfalls that we overcame to make the polyhedral model usable at runtime, and addresses to

the polyhedral model community some new challenges for making runtime polyhedral optimization even more effective and polyhedral techniques' scope even larger. Conclusions are given in Section 6.

2. THE APOLLO FRAMEWORK

This section gives an overview of the Apollo framework along with the achieved speedups and how to use it.

2.1 Overview

Apollo is capable of applying polyhedral loop optimizations to any kind of loop-nest[1], even if it contains unpredictable control and memory accesses through pointers or indirections, as long as it exhibits a polyhedral-compliant behavior at runtime, at least in phases. A polyhedral-compliant behavior is characterized as follows:

- linear loops: when the target loop nests are running, (1) every memory instruction references a series of memory addresses that can be defined as an affine function of the surrounding loop counters; (2) the loop trip count of each loop of the nest can be defined as an affine function of the surrounding loop counters; (3) every scalar variable depending on scalar variables defined in previous iterations behaves as an induction variable, making its series of values definable as an affine function of the surrounding loop counters.

- nonlinear loops: either the same characteristics hold, or (1) when not linear, a memory instruction references a series of memory addresses which can be approximated by a couple of parallel regression hyperplanes of dimension $d-1$, defined by an affine function of the d surrounding loop counters, and forming a *tube* that is *sufficiently narrow*, inside which every memory address that is accessed occurs; (2) when not linear, a loop trip count can also be approximated as in (1). More details regarding this modeling are given in Section 3.

The framework is made of two components: a static compiler, whose role is to prepare the target code for speculative parallelization, and implemented as passes of the Clang-LLVM compiler [14]; and a runtime system, that orchestrates the execution of the code.

New *virtual iterators* (or loop counters), starting at zero with step one, are systematically inserted at compile-time in the original loop nest. They are used for handling any kind of loop in the same manner, and serve as a basis for building the prediction model and for reasoning about code transformations.

Apollo's static compiler analyzes each target loop nest regarding its memory accesses, its loop bounds and the evolution of its scalar variables. It classifies these objects as being static or dynamic. For example, if the target addresses of a memory instruction can be defined as a linear function of the iterators at compile-time, then it is considered as static. Otherwise, it is dynamic. Dynamic instructions require instrumentation so that their memory access patterns can be observed and analyzed at runtime. The same is achieved for the loop bounds and scalars. This classification is used to build an instrumented version of the code, where instructions collecting values of the dynamic objects are inserted, as well as instructions collecting the initial values of the static objects (e.g. base addresses of regular data structures).

At runtime, Apollo executes the target loop nest in successive phases, where each phase corresponds to a slice of the outermost loop (see Figure 1):

- First, an on-line profiling ① phase is launched, executing serially a small number of iterations, and recording memory addresses, loop-trip counts and scalar values.

- ② From the recorded values, linear equalities and inequalities are obtained, through interpolation or regression, to build a polyhedral prediction model. This process is further addressed in Section 3. Using this model, the loop optimizations to be applied are determined by invoking Pluto [4] on-line. From the Pluto-suggested transformation, the corresponding parallel code is generated, with additional instructions devoted to the verification of the speculation. These last steps for code generation are detailed in Section 4. In order to mask the time overhead of these steps, the original serial version of the loop is launched in parallel ③.

- A backup ④ of the memory regions, that are predicted to be updated during the execution of the next slice, is performed. An early detection of a potential misprediction is also performed, by checking that all the memory locations that are predicted to be accessed are actually allocated to the process.

- A large slice of iterations is executed ⑤ using the parallel optimized version of the code. While executing, the prediction model is continually verified by comparing the actual reached values against their predictions. At the end of the slice, if no misprediction was detected, Apollo performs a new backup for the next slice ⑥, and executes this slice using the same optimized version ⑦. If a misprediction is detected, memory is restored ⑧ to cancel the execution of the current slice. Then, the execution of the slice is re-initiated using the original ⑨ serial version of the code, in order to overcome the faulty execution point. Finally, a profiling slice is launched again to capture the changing behavior and build a new prediction model.

2.2 Performance results

In Figure 2, we show the speed-up obtained by using Apollo over the best serial version generated among the gcc-4.8 or clang-3.4 compilers with optimization level 3 (-O3). Our experiments were ran on two machines: a general-purpose multi-core server, with two AMD Opteron 6172 processors of 12 cores each; and an embedded system multi-core chip, with one ARM Cortex A53 64-bit processor of 8 cores. The benchmarks were executed using 8 threads.

The set of benchmarks has been built from a collection of benchmark suites, such that the selected codes include a main kernel loop nest and highlight Apollo's capabilities: `SOR` from the Scimark suite [17]; `Backprop` and `Needle` from the Rodinia suite [7]; `DMatmat`, `ISPMatmat`, `SPMatmat` and `Pcg` from the SPARK00 suite of irregular codes [24]; and `Mri-q` from the Parboil suite [19]. In Table 1, we identify the characteristics for each program that make it impossible to parallelize at compile-time, where:

- *Has indirections* means that the kernel loop accesses memory through array indirections (e.g., `A[B[i]]`).

[1]for-loops, while-loops, do-while-loops, goto-loops, ...

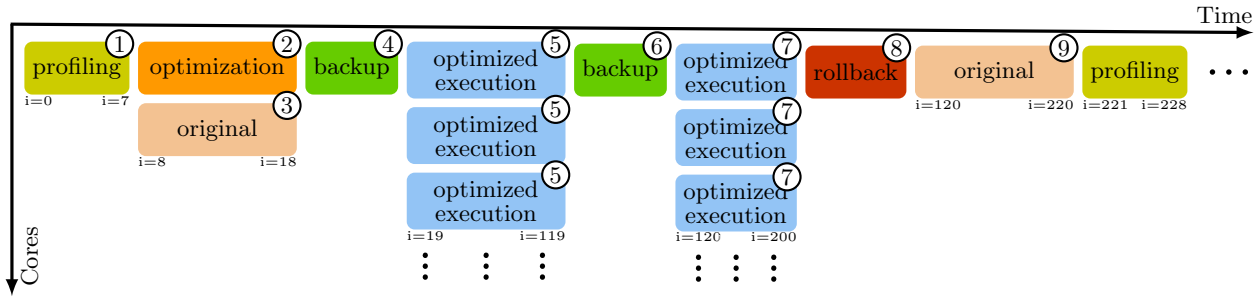

Figure 1: Execution in slices of iterations from iteration 0 to 228 of the outermost original loop

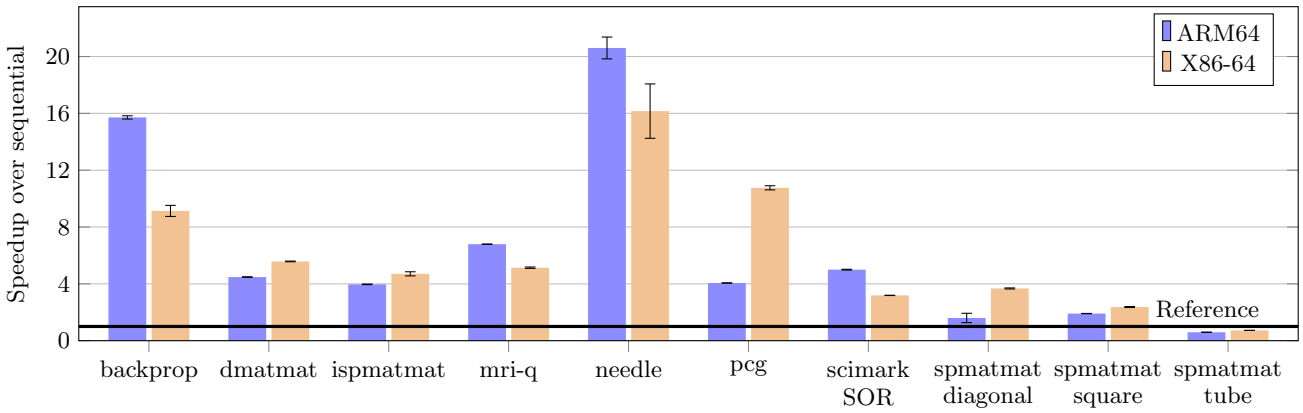

Figure 2: Speedup of Apollo, using 8 threads, over the best sequential version generated with clang/gcc.

- *Has pointers* means that the kernel loop accesses memory through pointers (e.g., linked list).
- *Unpredictable bounds* means that some loop bounds cannot be known at compile-time (e.g., while loops or for loops bounded by runtime variables).
- *Unpredictable scalars* means that the values taken by some scalars cannot be known at compile-time.

Both "has indirections" and "has pointers" leads to memory accesses that cannot be identified as linear statically.

Benchmark	Has ind.	Has pointers	Unpredict. bounds	Unpredict. scalars
Mri-q		✓		
Needle		✓		
SOR	✓	✓		
Backprop	✓	✓		
PCG	✓	✓	✓	✓
DMatmat	✓	✓		
ISPMatmat	✓	✓	✓	✓
SPMatmat	✓	✓	✓	✓

Table 1: Characteristics of each benchmark.

For the `SPMatmat` kernel, five inputs with different data layouts were used to highlight some key features of Apollo.

In Table 2, we show the transformations that were selected by Pluto at runtime. Reported results are obtained by computing the mean and standard deviation from the outcome of five runs. We can observe that the versions optimized with Apollo can be up to 20× faster than the original serial version. In many cases, speed-ups over 8× (more than the number of threads being used) were reached thanks to transformations that also improve data locality.

Benchmark	Selected Optimization
Mri-q	Interchange
Needle	Skewing + Interchange + Tiling
SOR	Skewing + Tiling
Backprop	Interchange
PCG	Identity
DMatmat	Tiling
ISPMatmat	Tiling
SPMatmat	Tiling

Table 2: Transformations suggested by Pluto at runtime.

2.3 How to use it?

To use Apollo, the programmer has to enclose the target loop nests using a dedicated #pragma directive (shown in Listing 2). This #pragma does not imply any semantics, it is only used to identify loop nests that may be interesting to optimize with Apollo. Inside the #pragma, any kind of loops are allowed, although there are still some restrictions:

1. The target loops must not contain any instruction performing dynamic memory allocation, although dynamic allocation is obviously allowed outside the target loops.
2. Since Apollo does not handle inter-procedural analy-

ses, the target loops should not generally contain any function invocation. Nevertheless, a called function may be inlined in some cases, thus annihilating this issue.

Listing 2: Example of the #pragma directive.

```
#pragma apollo dcop
{
  for(row = 1; row <= left->Size; row++) {
    ...
  }
}
```

Then, the programmer compiles the code using our specialized compiler which is based on LLVM. Two commands showing how to compile a source code with Apollo are presented in Listing 3. Any compiler flag available with clang may be used. The result of these commands are specialized executables containing invocations to the runtime system of Apollo, as well as static analysis results and important data that are required for runtime code generation.

The generated executables can be launched by the user in the standard way. Once the execution reaches a loop nest previously marked with the special #pragma, the runtime system of Apollo takes control of the execution and speculative execution starts, as described in the previous Section. When the original loop exit is reached, the execution returns to its normal flow.

Listing 3: Usage of the static component.

```
apolloc -O3 source.c -o myexecutable
apolloc++ -O3 source.cpp -o myexecutable
```

3. THE PREDICTION MODEL

In contrast to most TLS systems, Apollo builds a model that predicts the behavior of the loop nest. This is the key for performing speculative polyhedral transformations. By assuming that this prediction is valid, Apollo deduces dependencies between iterations and instructions, and applies aggressive code optimizations, that involve reordering the execution of iteration and instructions.

This model predicts: (1) memory accesses, (2) loop bounds, and (3) basic scalars. Intuitively, memory accesses and loop bounds must be predicted to enable precise dependency analysis for selecting an optimizing transformation. Basic scalars correspond to the ϕ-instructions in the header of each loop. Typically, they correspond to loop iterators and accumulators updated at each loop iteration. They introduce very restrictive dependencies that may forbid any optimization. By predicting the values that are taken by these scalars, Apollo removes these dependencies.

Memory accesses.

Apollo embeds three possible modelings for memory accesses: (i) linear, (ii) tube and (iii) range. In Figure 3, we depict each model.

When it is possible to interpolate a linear function from the registered memory addresses, a memory access is predicted as (i) linear. Future accesses are predicted to follow exactly the interpolating function. However, memory accesses may not follow a perfect linear pattern. In this case, a regression hyperplane is calculated using the least squares

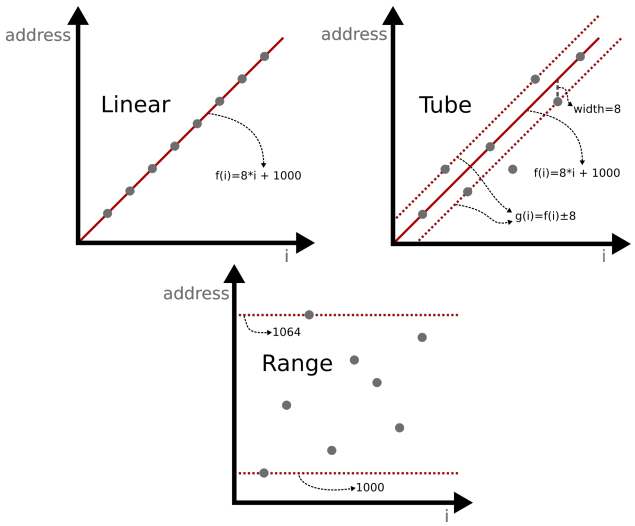

Figure 3: Different modelings for memory accesses.

method. The regression hyperplane coefficients (initially of type real) are then rounded to their nearest integer value. The Pearson's correlation coefficient is then computed. If it is greater than 0.9, future memory accesses are predicted to happen inside a (ii) tube, otherwise inside a (iii) range. This criterion is derived from experimental evaluation [20, 22, 23]. The tube consists of the regression hyperplane, a tube width and a predicted alignment. The tube width is the maximum observed deviation from the regression hyperplane, rounded to the next bigger multiple of the word size. The range consists of a maximum and a minimum memory address between which all the memory accesses are predicted to occur. In many cases, if a memory access occurs outside the predicted region, the transformation may still remain valid if no new dependencies are introduced. When a misprediction occurs, a more complex verification mechanism checks if this is the case, to avoid any useless rollback.

Loop bounds.

There are two possible modelings, (i) linear, or (ii) tube. Figure 4 visually depicts the different types of predictions for loop bounds. Notice that the lower bound is always 0, and only the upper bound is predicted. The linear prediction mirrors the memory accesses linear prediction. For the tube case, a regression hyperplane is computed and its coefficients are rounded to the nearest integer values. Then two hyperplanes are derived, predicting a maximum and minimum number of iterations for a loop to execute. These new hyperplanes are parallel to the regression hyperplane, but passing through the maximum positive and negative deviations from the number of executed iterations. When selecting a transformation, the iteration space is divided in two, and all the iterations situated below the minimum predicted number of iterations may be aggressively transformed; on the other hand, the iterations between the predicted minimum and maximum must be executed sequentially, until the loop exit has been reached.

Basic scalars.

There are three possible modelings, (i) linear, (ii) semi-linear and (iii) reduction. Again, the linear case resembles

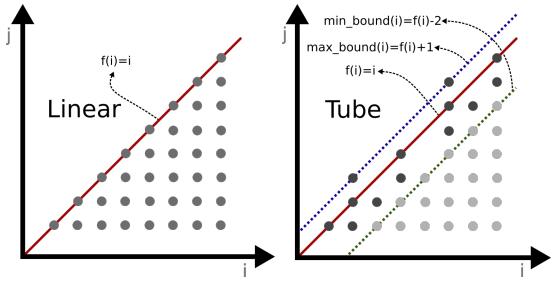

Figure 4: Different modelings for the loop bounds.

the memory access linear case. A semi-linear scalar is characterized by a constant increment at each iteration of its parent loop, although the initial value of the scalar at the beginning of the loop cannot be predicted. The memory locations used for computing the initial value of the basic scalar must be predicted to remain unmodified during the execution of the loop. Any other behaviors are considered as being reductions. Unfortunately, the underlying polyhedral tools used in Apollo are not able to handle reductions at all, preventing the generation of optimized code when they occur.

Once the prediction model is obtained, Apollo is ready for selecting a polyhedral optimizing and parallelizing transformation, and to generate binary executable code from it. These tasks are detailed in the next Section.

4. HYBRID CODE GENERATION

A key component of Apollo is its optimized code generation mechanism. This mechanism intervenes both at compile-time and at runtime.

At compile-time, some building blocks, common to every transformation that may be selected at runtime, are extracted from the original source code. We call them *Code-Bones* [5]. These are embedded in the binary executable to be used later by the runtime system. At runtime, Apollo's code generation mechanism is in charge of: encoding the Code-Bones and the prediction model in a polyhedral representation; passing it to Pluto and CLooG [2, 8] to obtain an optimizing and parallelizing polyhedral transformation and its associated scan of iteration domains; and finally generating optimized binary code. In this Section, we provide an overview of this code generation mechanism.

Any speculatively optimized code is generally composed of two types of operations: (1) operations extracted from the original target code, whose schedule and parameters have been modified for optimization purposes; and (2) operations devoted to the verification of the speculation, whose role is to ensure semantic correctness and to launch a recovery process in case of wrong speculation. From the control-flow graph (CFG) of the target loop nest, we extract the different Code-Bones that reflect both roles:

(1) Each memory write instruction in the original code yields an associated code bone, that includes all instructions belonging to the *backward static slice* of the memory write instruction. In other words, these are all the instructions required to execute an instance of the memory write. Notice that memory read instructions are also included in Code-Bones, since the role of any read instruction is related to the accomplishment of at least one write instruction. This first set of Code-Bones is called *computation bones*.

(2) For each memory instruction (read/write) of the computation bones, an associated *verification bone* is created. Additionally, verification bones for the scalars (one for the initial value and one for the increment), and for the loop bounds, are also created. These bones contain instructions devoted to verifying the validity of the prediction model.

```
computation_bone.mem7(vi,vj,vk):

  Pelem.real.pred = 24000*vi + 48*vj + 223724352
  right.pred = 64016*vi + 64016*vj + 8*vk + 409724360
  result.pred = 64016*vi + 8*vk + 921916376
  result.store.pred = 64016*vi + 8*vk + 921916376

  Pelem.real = load Pelem.real.pred
  right = load right.pred
  result.load = load result.pred

  val = result.load + Pelem.real * right
  store val, result.store.pred
  ret
```

Figure 5: Code-Bone computing a store instruction, for the code of Listing 1, after runtime patching.

```
verification_bone.mem7(vi,vj,vk):

  col = 0*vi + 0*vj + 1*vk + 1
  result.base.pred = 8*vi +921852360
  result.store.pred = 64016*vi + 8*vk + 921916376

  result.base = load result.base.pred
  original_ptr = result.base + 8*col

  verif_result = original_ptr == result.store.pred
  ret verif_result
```

Figure 6: Code-Bone verifying the prediction of a store instruction, for the code of Listing 1, after runtime patching.

All the generated bones are embedded in the executable in their LLVM intermediate representation form.

Recall the code in Listing 1. In Figures 5 and 6, we depict the computation bone associated to the unique store instruction of the code and the verification bone in charge of verifying this access. These have been simplified for pedagogical purposes. We depict a single verification bone among many others, that are dedicated to the verification of loop bounds, scalars and the rest of the memory accesses.

The first four instructions in the computation bone compute the memory addresses that will be accessed, by using the predicting linear functions. The linear function's coefficients are computed and appended in the IR by the runtime system, from the interpolation of the collected addresses, when profiling a small slice of the target loop nest. Then, the value to be written to memory is calculated and a store instruction is finally executed. Similarly, the verification bone calculates memory addresses and basic scalar values using the predicting linear functions. Then, the original pointer is computed. If both values are different, then misprediction is detected. The code bone returns the misprediction status.

At runtime, once the prediction model has been obtained, the bones embedded in the executable are loaded and parsed to identify the memory access instructions, scalars and verification instructions, which characterize them. For now, other internal computations can be completely ignored, since they do not affect the polyhedral representation.

Using the available bones, a loop structure that mimics the original nest is created, complemented with dynamic information obtained thanks to the prediction model. The reconstructed nest is verified against the predicted dependencies in the original nest to ensure their equivalence. The final result is a loop nest, with well-defined statements, with memory accesses and loop bounds described by linear equalities and inequalities. This code can now be encoded into a polyhedral representation.

Then Apollo invokes Pluto[2] to determine an optimizing and parallelizing transformation. The transformed representation is passed to CLooG to compute scanning loops. A dedicated compiler pass transforms CLooG's output to LLVM-IR. Then, this IR is optimized and passed to the LLVM Just-In-Time (JIT) compiler for generating binary code, which is then launched in a next chunk. This binary code is reused in the successive optimized chunks that are launched, until a misprediction occurs, which invalidates the previous prediction model, or until the completion of the loop nest.

Instructions devoted to the verification of the predictions are something unique to Apollo. These instructions exhibit some properties that can be exploited to achieve better performance. The optimizations detailed below exploit some of these properties.

The first optimization consists of moving all the verification bones that do not participate in dependencies into a separate loop nest, to be executed before the rest of the code. This verification loop nest is encoded in its own polyhedral representation and optimized through Pluto, Cloog and LLVM JIT separetly from the computation loop nest. This enables an *inspector-executor* way of launching optimized chunks, thanks to an early detection of any misprediction. In some cases, this can completely eliminate the need of performing memory backups and rollbacks.

Other optimizations can reduce the algorithmic time-complexity of the verification code. Consider a verification bone that does not participate in any dependency, and where all the coefficients of the predicting linear functions at a given loop level are equal to zero. For this loop level, the input of the code bone remains the same, since all the predictions and original address computations are not affected by changes of the corresponding virtual iterator. Hence, verifying a single iteration for this loop level is enough. We depict this optimization with the example in Figure 7. This bone verifies an access to an array that is performed through an indirection. The iterator vj does not participate in any computation of this bone, since the coefficients multiplying it are equal to zero. Hence we can safely remove this loop.

5. POLYHEDRAL CHALLENGES

Using polyhedral tools at runtime raises several challenges. We describe in this section what are these challenges and how Apollo handles them.

[2]Pluto is used as a shared library.

```
for(vi=0; vi<N; ++vi)
 for(vj=0; vj<N; ++vj)
  for(vk=0; vk<N; ++vk)
   if(&(A[vi] + vk) != 400*vi+0*vj+8*vk)
    rollback()

for(vi=0; vi<N; ++vi)
 for(vk=0; vk<N; ++vk)
  if(&(A[vi] + vk) != 400*vi+8*vk)
   rollback()
```

Figure 7: Verification code: before (Top) and after (Bottom) optimization.

5.1 Apollo's internal solutions

Time overhead.

The motivation behind Code-Bones is to provide a good trade off between the set of supported transformations and the time taken by the invoked polyhedral tools to work with such blocks. Instead of Code-Bones, we could have considered basic block nodes of the control-flow-graph as polyhedral statements. This is the approach adopted by Polly [10]. However, such regions are too coarse and would restrict the set of supported transformations. For example, it would be impossible to schedule differently instructions which belong to a same basic block of the original code.

A further approach providing even finer schedules would be to directly consider individual memory instructions of the LLVM IR as polyhedral statements. Unfortunately, due to exponential complexity in the number of statements, the polyhedral tools would be too slow and no more suitable for the runtime usage of Apollo.

Quality of the optimizations vs. time overhead.

Pluto exposes multiple options that must be tweaked to result in the best optimizing transformation. There is no unique setup of options that always outperforms other options. Furthermore, the best set may depend on the target code or the hardware. However, numerous experiments lead us to define a set of options yielding well performing optimized code in most cases. Intra-tile-optimization (–intratileopt) is always activated since it enables Pluto to do loop interchanges to improve data locality. We always enable parallelization (–parallel), unless there is a single processor core. Loop unrolling (–unroll) is enabled with a factor of 2; larger factors did not yield any significant performance improvements, while also greatly increasing the code size, harming the LLVM JIT performance. Maximum fission is always set (–nofuse); this configuration provides the best performance results and keeps CLooG's and the LLVM JIT compilation times low. Additionally, by relying on a simple heuristic, Apollo automatically decides when tiling should be beneficial. However, we never found level 2 tiling (–l2tile) to be profitable, and it greatly increases CLooG's execution time. For the rest of the options, we kept the default behavior. Notice that, in a dynamic context, it is not mandatory to obtain the best performing optimized code. One must consider a trade-off between the time taken to obtain optimized code and its execution performance, since

global performance is no more solely depending on the target code itself, but also on the time-overhead of the runtime system.

CLooG embeds an option to optimize the control of the generated code, at the price of increasing the code size. We are compelled to deactivate this option since it greatly increases CLooG's total execution time. Additionally, such larger code size would also increase the time taken by the LLVM JIT compiler to generate binary code, the last step in our code generation pipeline.

Integer overflows.

The inequalities predicting the memory accesses, that are obtained from interpolation or regression, cannot generally be used as they are, to obtain good optimizing transformations. Since they reference memory as a one-dimensional array by addressing bytes, polyhedral tools often generate integer overflows, thus crashing the user's application. This happens due to some large values taken by the coefficients participating in these inequalities. To overcome this issue, multiple analyses are performed in Apollo to recover high level information about memory accesses. This not only helps to prevent integer overflows, but also improves the quality of the selected transformation, especially if a skewing transformation is involved. In this purpose, several steps are performed: (i) aliasing groups are identified, each associated to its own array; (ii) for each array, it is sometimes possible to recover the dimensions and access functions of a multi-dimensional array. If successful, the arithmetic complexity of the computations related to dependency analysis and transformation selection is significantly lowered. Our implementation, which has to be fast, is derived from Maslov's delinearization technique [16]. Since all the values of the coefficients in the linear access functions are known at runtime, this task is greatly simplified: some coefficients may explicitly exhibit some dimension sizes. Finally, notice that implementations of polyhedral tools that use the GNU Multiple Precision Arithmetic Library (GMP) are not suitable for a runtime usage, due to excessive time-overhead.

5.2 Dynamic polyhedral kernels

Apollo extends significantly the scope of polyhedral techniques. First, polyhedral approaches are no more limited to codes having a convenient syntactic structure, that explicitly exhibit affine loop bounds and array references. Now, a runtime polyhedral behavior elects any kind of loop for polyhedral optimizations. Second, even nonlinear loops can be handled thanks to smart runtime modelings of the memory and iterative behaviors. Apollo can apply polyhedral transformations to nonlinear loops by representing series of nonlinear values as *tubes*, defined by affine inequalities. However, to go further in spreading the use of polyhedral techniques for general programs, polyhedral kernels[3] that are better adapted to a dynamic usage are required. This opens interesting perspectives for many new research developments. Despite the faced challenges, it has been shown that Apollo is able to benefit from the entire polyhedral model stack at runtime. However, some of the threats are mitigated and not completely solved.

When using Pluto, some parameters cannot be set through the library interface: the tile sizes cannot be set to a size different from the default, and it is impossible to add arbitrary constraints to the transformation. Even worse, it is impossible to describe a *tube* or *range* of memory accesses, although it is possible with tools like *Candl* [6]. To overcome this latter issue, we did the following: first, the OpenScop representation is passed to Candl, to perform the dependence analysis. Then, in the OpenScop representation, all tube and range accesses are replaced by accesses using their regression hyperplanes, and the computed dependencies are attached to the OpenScop representation. We modified Pluto to use the attached dependencies and to ignore the access functions encoded in the OpenScop representation. In this way, the transformation is finally selected by Pluto based on the dependencies computed by Candl, and using the equations describing nonlinear memory accesses.

More generally for our dynamic context, polyhedral kernels that purpose sub-optimal solutions, but that guarantee a smaller time overhead, could be advantageous. Current kernels, as Pluto, seek the best solution regarding their heuristics, although a more straightforward solution, determined in short time, would already provide interesting speed-ups. A polyhedral scheduler working incrementally could be a good approach: a first straightforward solution could be produced, and, while it has been launched, a next better solution could be computed in a separate thread, which would in turn be launched as soon as it has been fully determined, and so on. When several solutions cannot be ranked regarding their predicted performance, they could be evaluated dynamically by executing them in successive chunks, to finally select the best performing one.

More generally, polyhedral kernels may not stay exclusively static. Their heuristics may be assisted and strengthened by runtime analysis. For example, it has been shown in some works that it is difficult to predict the effectiveness of one or another code transformation. In some cases, the resulting control complexity of the new loops may hamper the benefits of the transformation. Runtime analysis would provide the actual provided performance.

Current polyhedral schedulers consider statements as the smallest entity to be scheduled, as they usually apply on source codes. However, data dependencies are related to memory references, which occur in elementary memory instructions of intermediate code representations as the LLVM IR. Being able to schedule such instructions would yield more efficient solutions. A typical example of good candidate is a stencil computation, where one unique statement embeds many inter-dependent memory references. Nevertheless, since schedulers' complexity is exponential in the number of statements, the scheduling granularity could be different and adjusted according to the memory and computing costs of the statements. More generally, polyhedral kernels should operate on compilers' intermediate forms, and no more exclusively on source codes. This has been initiated by Polly. But polyhedral kernels should simultaneously operate at runtime, and no more exclusively at compile-time. This has been initiated by Apollo.

Regarding polyhedral code generators, it may be useless to address some code optimizations that are handled anyway by lower-level JIT compilers, as for example, loop-invariant code motion. The focus should be put on what is exclusively provided by polyhedral concepts. The rest should be transferred to general underlying optimizing mechanisms. Such

[3]We call "polyhedral kernels" fundamental tools performing code analyses and transformations, using mathematical operations on polyhedra, like schedulers and code generators.

an approach may lower the time-overhead of polyhedral code generators.

Finally, the polyhedral model can be viewed as the most accurate and efficient model of program analysis and optimization. Thus, one of its important goals is to extend its scope to general-purpose programs, in order to be used in modern applications. By being usable at runtime, maybe supported by speculative techniques or new behavior modelings, it is likely to provide very good answers to the multicore and processor heterogeneity challenges.

6. CONCLUSION

Apollo is proof that polyhedral techniques are effective at runtime on more general loops than traditional fortran-like loops. The target loops may be while-loops with memory references through pointers and indirections, exhibiting a linear or nonlinear behavior at runtime. A few weeks ago, the first release of this framework was made available.

We expect *you* to contribute in further developments related to runtime polyhedral techniques, for making the polyhedral model's benefits available to every programmer and user.

7. REFERENCES

[1] APOLLO: Automatic POLyhedral speculative Loop Optimizer. http://apollo.gforge.inria.fr.

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. PLDI, 2008.

[4] U. K. R. Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, Ohio State University, 2008.

[5] J. M. M. Caamaño, W. Wollf, and P. Clauss. Code bones: Fast and flexible code generation for dynamic and speculative polyhedral optimization. In *Euro-Par 2016 Parallel Processing - 22th International Conference, Grenoble, France. Proceedings. (Best paper mention)*, 2016.

[6] Candl: Data dependence analysis tool in the polyhedral model. http://icps.u-strasbg.fr/~bastoul/development/candl.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[8] Cloog: the chunky loop generator. http://www.cloog.org.

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[10] T. Grosser, A. Größlinger, and C. Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4), 2012.

[11] A. Jimborean. *Adapting the polytope model for dynamic and speculative parallelization*. PhD thesis, Université de Strasbourg, Sept. 2012.

[12] A. Jimborean, P. Clauss, J. Dollinger, V. Loechner, and J. M. M. Caamaño. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *International Journal of Parallel Programming*, 42(4):529–545, 2014.

[13] A. Jimborean, L. Mastrangelo, V. Loechner, and P. Clauss. *VMAD: An Advanced Dynamic Program Analysis and Instrumentation Framework*, pages 220–239. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[14] LLVM compiler infrastructure. http://llvm.org.

[15] J. M. Martinez Caamaño. *Fast and Flexible Compilation Techniques for Effective Speculative Polyhedral Parallelization*. Theses, Université de Strasbourg, Sept. 2016.

[16] V. Maslov. Delinearization: An efficient way to break multiloop dependence equations. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 152–161, New York, NY, USA, 1992. ACM.

[17] SciMark benchmark suite. http://math.nist.gov/scimark2.

[18] J. Steffan and T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, HPCA '98, Washington, DC, USA, 1998. IEEE Computer Society.

[19] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, Mar. 2012.

[20] A. Sukumaran-Rajam. *Beyond the Realm of the Polyhedral Model: Combining Speculative Program Parallelization with Polyhedral Compilation*. PhD thesis, Université de Strasbourg, Nov. 2015.

[21] A. Sukumaran-Rajam, J. M. M. Caamaño, W. Wolff, A. Jimborean, and P. Clauss. Speculative program parallelization with scalable and decentralized runtime verification. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 124–139, 2014.

[22] A. Sukumaran-Rajam, L. E. Campostrini, J. M. M. Caamaño, and P. Clauss. Speculative runtime parallelization of loop nests: Towards greater scope and efficiency. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 245–254, 2015.

[23] A. Sukumaran-Rajam and P. Clauss. The polyhedral model of nonlinear loops. *ACM Trans. Archit. Code Optim.*, 12(4):48:1–48:27, Dec. 2015.

[24] H. L. A. van der Spek, E. M. Bakker, and H. A. G. Wijshoff. SPARK00: A benchmark package for the compiler evaluation of irregular/sparse codes. *CoRR*, abs/0805.3897, 2008.