# Code Bones: Fast and Flexible Code Generation for Dynamic and Speculative Polyhedral Optimization

Juan Manuel Martinez Caamaño[✉], Willy Wolff, and Philippe Clauss

INRIA CAMUS, ICube Laboratory, University of Strasbourg, Strasbourg, France
jmartinezcaamao@gmail.com

**Abstract.** In this paper, we present a new runtime code generation technique for speculative loop optimization and parallelization, that allows to generate on-the-fly codes resulting from any polyhedral optimizing transformation of loop nests, such as tiling, skewing, fission, fusion or interchange, without introducing a penalizing time overhead. The proposed strategy is based on the generation of *code bones* at compile-time, which are parametrized code snippets either dedicated to speculation management or to computations of the original target program. These code bones are then instantiated and assembled at runtime to constitute the speculatively-optimized code, as soon as an optimizing polyhedral transformation has been determined. Their granularity threshold is sufficient to apply any polyhedral transformation, while still enabling fast runtime code generation. This strategy has been implemented in the speculative loop parallelizing framework Apollo.

## 1 Introduction

The polytope model (or polyhedral model) [7] is a powerful mathematical framework for reasoning about loop nests, and for performing aggressive transformations which improve parallelism and data-locality. Although very powerful, compilers relying on this model [3,8] 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, due to dynamic data structures accessed through indirect references or pointers, which prevent a precise static dependence analysis. On the other hand, Thread-Level Speculation (TLS) [14] is a promising approach to overcome this limitation: regions of the code are executed in parallel before all the dependences are known. Hardware or software mechanisms track register and memory accesses to determine if any dependence violation occur. While traditional TLS systems implement only a straightforward loop parallelization strategy consisting of slicing the target loop into consecutive parallel threads, TLS frameworks implementing a speculative and dynamic adaptation of the polytope model have been recently proposed: VMAD [9] and Apollo [17], where parallelizing and optimizing transformations are performed for loops exhibiting a polyhedral-compliant behavior at runtime. A main limitation of these frameworks relies on the dynamic code generation mechanism:

for each target loop nest, some code skeletons, which are incomplete optimized code versions that will be completed at runtime, are generated at compile-time and embedded in the final executable file. This approach has several limitations: (1) Each skeleton only supports a limited set of loop optimizing transformations; for example, while a given skeleton enables a combination of skewing and interchange, it cannot support any additional transformation as tiling or fission. (2) The impact of some code transformations regarding the structure of the resulting loop nest cannot be predicted; for example, loop fission may result in an arbitrary number of loops; another example is loop unrolling, where the best unroll factor may only be known at runtime. (3) With code skeletons, the same schedule must be applied to all the statements of a target loop, while the polytope model considers scheduling per statements. (4) The complicated structure of generic code skeletons hampers the application of some compiler optimizations, as for example automatic vectorization.

In this paper, we present a dynamic code generation mechanism for speculative polyhedral optimization, that allows to apply on-the-fly any combinations of transformations to a target loop nest, similarly to what is achieved at compile-time by static polyhedral compilers as Pluto [3]. It is based on the compile-time generation of *code bones*, which are code snippets either made of instructions of the target loop nest, or of speculation verification instructions. These code bones are then instantiated and assembled at runtime, according to an optimizing transformation that has just been determined from runtime profiling. The resulting assembled code is then further optimized and compiled using the LLVM just-in-time compiler to generate the final executable code. Our contribution has been implemented in the speculative parallelization framework Apollo [17]. We show on a set of benchmark codes that this code generation technique enables: (1) significant parallel speed-ups, thanks to (2) various automatic runtime loop optimizations that are traditionally only possible at compile-time, (3) on loops that cannot be handled at compile-time.

The paper is organized as follows. An overview of Apollo is presented in Sect. 2. Section 3 details the proposed code generation mechanism (the main contribution of this paper). Section 4 present the empirical results regarding performance and time overhead of the proposed approach. Section 5 compares our mechanism against other approaches. Finally, conclusions are given in Sect. 6.

## 2    Speculative Parallelization

Apollo[1] [17] is a framework capable of applying polyhedral loop optimizations on any kind of loop-nest[2], even if it contains unpredictable control and memory accesses through pointers or indirections, as soon as it exhibits a polyhedral-compliant behavior at runtime. 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 [10]; and a

---

[1] Automatic POLyhedral speculative Loop Optimizer.
[2] for-loops, while-loops, do-while-loops.

runtime system, that orchestrates the execution of the code. New *virtual itera-tors*, 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 address of a memory instruction can be defined as a linear function of the iterators, then it is considered as static. Otherwise, it is dynamic and thus requires instrumentation to be analyzed at runtime to take part of the prediction model. The same is achieved for the loop bounds and for 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 Fig. 1):

1. First, an on-line profiling ① phase is launched, executing only a small number of iterations, and recording memory addresses, loop-trip counts and scalar values.
2. ② From the recorded values, linear functions are interpolated to build a linear prediction model. Using this model, a loop optimizing and parallelizing transformation is determined by invoking, on-line, the polyhedral compiler Pluto. From the transformation, the corresponding parallel code is generated, with additional instructions devoted to the verification of the speculation.
3. A backup ③ of the memory regions, that are predicted to be updated during the execution of the next slice, is performed. An early detection of a mis-prediction is possible, by checking that all the memory locations that are predicted to be accessed are actually allocated to the process.
4. A large slice of iterations is executed ④ using the parallel optimized version of the code. While executing, the prediction model is also verified by comparing the actual values against their linear predictions. If a misprediction is detected, memory is restored ⑤ to cancel the execution of the current slice. Then, the execution of the slice is re-initiated using the original ⑥ 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. If no misprediction was detected during the run of the parallel code, a next slice of the loop nest using the same parallel code is launched.
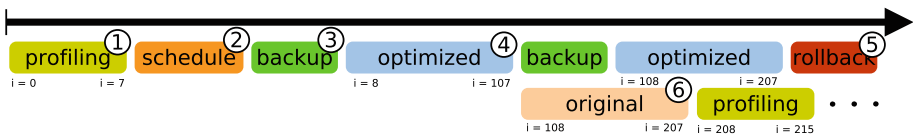


**Fig. 1.** Execution in slices of iterations

## 3   Code Generation Strategy

Until now, in order to achieve fast code generation, the Apollo framework has been using *code skeletons* [9]. Code skeletons are incomplete transformed versions of the target loop nests that are generated at compile-time, and completed at runtime as soon as the necessary information has been discovered and computed. Each of such skeletons supports a fixed combination of loop transformations, related to a fixed loop structure. This approach becomes impractical when supporting combinations of polyhedral transformations that may alter the loop structure, such as loop fission, loop unrolling or even simple statement reorderings. To cover every possible combination of loop transformations, we propose a new fast code generation strategy based on *code bones*, which are parametrized code snippets generated at compile-time, and assembled at runtime to result in the transformed code.

Any speculatively optimized code is generally composed of two types of computations: (1) computations of the original target code, whose schedule and parameters have been modified for optimization purposes; and (2) computations related to the verification of the speculation, whose role is to ensure semantic correctness and to launch a recovery process in case of wrong speculation. These computations are generated in two phases: (i) a compile-time phase where code bones of each type are built, and (ii) a runtime phase where complex transformations are determined and instantiated using the code bones.

*Generation of Code Bones:* At compile time, code bones are extracted from the control-flow graph (CFG) of the target loop nest. Each memory write instruction 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. Starting from this first set of code bones (called *computation bones*), a second set of bones devoted to the verification of the predictions (called *verification bones*) is generated. For each memory instruction of the computation bones, that may be a write or a read, an associated verification bone is created. These verification bones contain a verification instruction comparing the actual accessed address to the generic predicting linear function that will be instantiated at runtime. Hence, the backward slice computing the target address of the verified memory instruction is also inserted in the verification bone. In the corresponding computation bone, all these instructions are removed and replaced by the computation of the predicting linear function. This provides better opportunities for the compiler to optimize the computation bones thanks to simpler address computations. Similar verification bones are also created for dynamic scalars and loop bounds. Each code bone is then optimized independently of the rest of the code by the compiler. Finally, the so-built code bones are embedded in a fat binary code in their LLVM intermediate representation form (LLVM-IR).

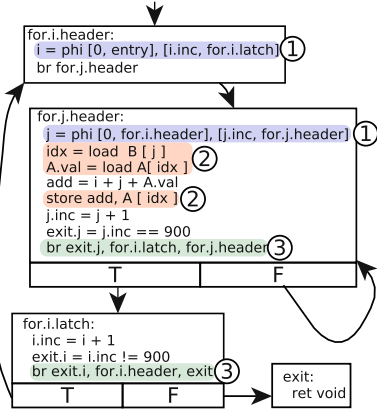They will be used later by the runtime system for code generation.

**Example:** As an illustration, consider the loop nest in Listing 1.1. Since array A is accessed through an indirection using array B, whose values are unknown at compile-time, it is impossible to determine what elements of A will be updated. The corresponding CFG is shown in Fig. 2. To make the examples clearer for the reader, instructions are shown in a simplified SSA intermediate representation. Instructions defining original loop iterators are identified by number ① , memory accesses by ② and loop exit conditions by ③. Loop iterators are identified as phi-nodes in the header of each loop, recalling that a phi-node is an instruction used to select an incoming value depending on the predecessor of the current basic block. In order to handle this loop nest at runtime for speculative optimization, code bones are generated. Since there is only one memory write, one computation bone is built. Array B is accessed through a linear memory reference that is identified at compile-time. Hence, only one verification bone is built, which is related to the access of array A. The computation bone is shown in Fig. 3. The computations of the predictions for the original iterators and addresses are identified by number ①, while the memory access instructions using the predicted address by ②. The associated verification bone is shown in Fig. 4, where the computations of the predicted addresses are identified by number ①, number ② points out the load of B[j] using the predicted memory address, original_ptr calculates the actual address of A[B[j]], while the verification instruction is identified by ③, which compares original_ptr against the prediction stored in A.pred. Notice that this bone includes the original address computations. Variables vi.0 and vi.1 stand for the virtual iterators that are used as a basis for building the prediction model. They are passed as parameters to the code bone. The linear functions of the prediction model are interpolated in terms of these iterators. Variables coef_i.0-1, coef_j.0-2, coef_a.0-2 and coef_b.0-2 are the coefficients of the linear functions. These coefficients will be instantiated and replaced by constant values at runtime.

*Runtime Composition of Code Bones:* The runtime code generation process is depicted in Fig. 5. When linear interpolating functions have been successfully built from the on-line profiling phase, they are used to build the encoding of a loop nest which is compliant with the polyhedral model, using the OpenScop format [2]. This polyhedral representation is then given as input to Pluto to perform dependence analysis, and to compute an optimizing and parallelizing transformation. Pluto's result, also in OpenScop, is then passed to the code generator Cloog [1] to obtain the polyhedral scan, *i.e.*, the new loops and iterators for the statements. Then, a dedicated translation proccess generates LLVM-IR from Cloog's output. This translation process is straightforward: Cloog's output recalls some constructs in C code like for-loops, simple if-conditions and statement invocations. This code invokes the code bones and instantiates the embedded linear functions according to the schedule provided by Cloog. Notice that a given bone may be invoked several times, but with different parameters to instantiate the linear functions. This may happen in case of loop fission
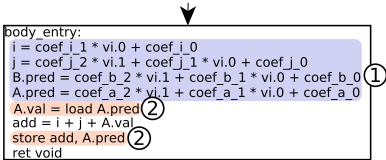
**Fig. 2.** CFG of a simple loop nest

```
for ( i = 0;  i  <  900;  ++i )
    for ( j = 0;  j  <  900;  ++j )
        A[ B[ j ] ]  += i + j ;
```
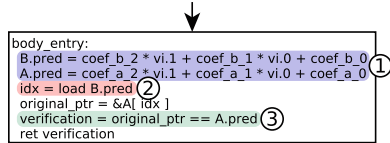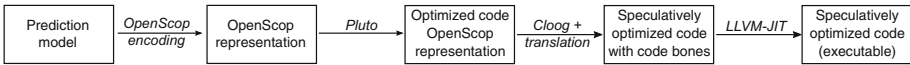
**Listing 1.1.** A simple loop nest



**Fig. 3.** Computation bone



**Fig. 4.** Verification bone



**Fig. 5.** Runtime code generation

for example. Finally, the resulting code is optimized further and converted into executable form using the LLVM just-in-time compiler.

*Transformation Selection Overhead:* The selection of a loop transformation is performed using the polyhedral compiler Pluto. However in [18], it has been shown that the execution time of Pluto increases in a roughly $n^5$ complexity in the number of statements in the system. In consequence, for complex kernels involving many dependences, Pluto can introduce a high time overhead which is inadequate for a runtime usage. Meanwhile the availability of a just-in-time polyhedral compiler, we bypass this issue by handling codes that may yield a high overhead of Pluto in a specific way. Kernels which are associated with more than 5 computation bones are classified as *complex*, while the others are classified as *simple*. For simple kernels, Pluto's time is masked by the execution of a slice of the original serial code. Apollo then starts executing the parallel code as soon as it is ready. For complex kernels, contiguous code bones in the same iteration domain are fused into a single bone. The execution is then equivalent

to their serial executions. Notice that this fused bone can perform verification and multiple memory-writes. Simplification of complex kernels is achieved by fusing bones until the total number reaches a given threshold. From our experiments, this threshold has been fixed to 15 bones. Then Pluto is invoked to select a transformation. If the threshold cannot be reached, Pluto is used solely for dependence analysis, to determine which loops in the original code may be parallelized, without any additional transformation.

```
parallel for (t2=0;t2<=899;t2++)
    VerifBone(slice_lower,t2);
parallel for (t2=0;t2<=899;t2++)
    for (t1=slice_lower;t1<=slice_upper;t1++)
        CompBone(t1,t2);
```
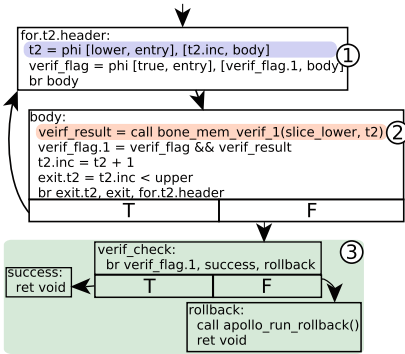
**Listing 1.2.** Generated code



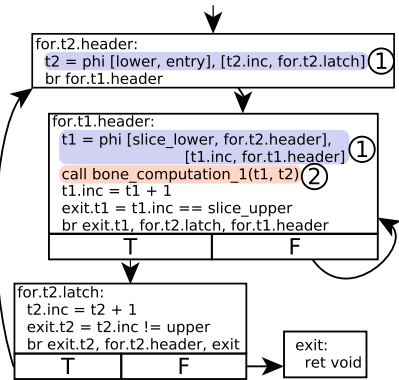**Fig. 6.** CFG of the verification code



**Fig. 7.** CFG of the computation code

*Optimization of the Verification:* Verification bones exhibit three specific optimization opportunities: (1) While computation bones often participate in dependencies, verification bones may not. Such verification bones are extracted into a separate loop nest, which is run before the rest of the bones in a inspector-executor fashion. (2) For these latter verification bones, it is possible to identify dimensions of the iteration domain (*i.e.*, loop indices), for which the the predicting linear functions is invariant. This is achieved by checking the linear functions: if the coefficients multiplying the iterator of a dimension are zero, the computation remains invariant for this dimension. Thus, only the first iteration is run. (3) When some operands of the computing functions are detected as being necessarily linear, while some others require some memory accesses to be performed, outer loops embedding the memory accesses are fully executed, while the inner loops computing linear operands only require the first iteration to be executed.

**Example (Continued):** Let us now assume that at runtime, each array element B[j] is assigned with j. Thus, Apollo discovers, through profiling and interpolation, that addresses touched by the store instruction can be represented by a linear function of the form: $j + base\_address$. The dependence spawned is an output dependence carried by the outermost loop. The large reuse distance between elements of A is penalizing regarding temporal data locality. Since the store instruction is going to use the predicting linear function as its target address, accesses to array B and A are not dependant anymore. Notice also that all the linear functions for verifying the memory access have 0 as coefficient for the outermost loop iterator. Then, it is sufficient to execute a single iteration of this loop. Both loop nests are sent to Pluto: a first single-loop nest accessing array B and verifying that its elements yield addresses equal to the predicting linear function, and a second two-loop nest computing and storing elements of array A using the linear function as the target address. Pluto suggests to parallelize the single verification loop, to interchange loops i and j in the second nest to improve temporal data locality, and also to parallelize the outermost loop. For clarity, the resulting code built by the code generator is represented in C, in Listing 1.2, although every operation is performed on LLVM-IR. Parameters slice_lower and slice_upper are the bounds of the slice of the original loop nest that will be run speculatively by Apollo. The CFGs of both generated loop nests are represented in Figs. 6 and 7. The invocations to the respective code bones are marked with number ②. Notice the branch (marked with number ③) to the rollback procedure in case of misprediction in the verification code. Number ① marks the iterators of the new loops.

## 4    Experiments

Our experiments were ran on two AMD Opteron 6172 processors of 12 cores each. Reported results are obtained by averaging the outcome of three runs. The tile sizes were always set to Pluto's default (32 for each dimention). The set of benchmarks has been built from a collection of benchmark suites, such that the selected codes include a main loop nest and highlights Apollo's capabilities: SOR from the Scimark suite[3], backprop and needle from the Rodinia suite [5]; dmatmat, ispmatmat, spmatmat, djacit and pcg from the SPARK00 suite of irregular codes [13], mri-q and stencil from the Parboil suite [16], and finally seidel-2d, which is a special version of the code belonging to the Polybench suite[4], in which the arrays are allocated dynamically, thus yielding pointer aliasing issues. The input problem sizes are as follows: dmatmat, ispmatmat and SOR: $3000 \times 3000$ matrices; spmatmat (square): $2500 \times 2500$ matrices; spmatmat (diagonal): $8000 \times 8000$ matrices; spmatmat (random): $2000 \times 2000$ matrices with 3000000 non-zero elements; spmatmat (worst case scenario): $4000 \times 4000$ matrices with 1600000 non-zero elements; pcg: $1100 \times 1100$ matrices; seidel-2d: $20,000 \times 20,000$ matrices; needle: $24,000 \times 24,000$ matrices;

---

[3]   http://math.nist.gov/scimark2/.

[4]   http://sourceforge.net/projects/polybench.

**Table 1.** Number of code bones and applied transformations.

| Benchmark | #comp-bones | #verif-bones | Applied transformation |
|---|---|---|---|
| needle | 1 | 1 | Tile + Skew + Vectorize + Unroll |
| SOR | 1 | 6 | Tile + Skew + Vectorize + Unroll |
| seidel-2d | 1 | 10 | Tile + Skew + Unroll |
| dmatmat | 1 | 5 | Tile + Unroll |
| ispmatmat | 1 | 8 | Tile + Unroll |
| spmatmat | 1 | 11 | Tile + Vectorize + Unroll |
| stencil | 1 | 2 | Tile + Interchange + Vectorize + Unroll |
| djacit | 7 | 5 | Skew + Unroll |
| mri-q | 2 | 1 | Interchange + Unroll |
| backprop | 2 | 4 | Interchange + Vectorize + Unroll |
| pcg | 21 | 33 | Identity + Unroll |

`stencil`: 4000 × 4000 matrices; `mri-q`: two vectors of sizes 2048 and 262,144; `backprop`: a neural-network with 80,000 input units, 512 hidden units and 16 output units.

Table 1 shows the number of generated bones and the optimizing transformations that were applied on-the-fly, in addition to parallelization. For every benchmark, except `spmatmat`, mispredictions are detected in advance, during backup, and before launching the speculatively-optimized code, as explained in Sect. 2. Codes `djacit` and `pcg` are both classified as *complex*: code bones where automatically fused to accelerate Pluto's transformation selection. Code `djacit` was simplified to 6 bones, while `pcg` was simplified at maximum to 21 bones. For the latter, the identity loop transformation was used since the total number of bones remained over the threshold of 15.

To emphasize different features of Apollo, four different inputs were used for `spmatmat`: (i) a square matrix, (ii) a band matrix, (iii) a randomly distributed matrix, and (iv) a matrix yielding the worst-case scenario for Apollo. Input (i) exhibits a single linear phase, which is conducive to Apollo. Input (ii) yields two different phases: the input band matrix has a constant number of elements per rows, excepting in the very last rows where this number is decreasing. Apollo is successful in optimizing the first large phase where rows have a constant number of elements. But since this number decreases in the last rows, the change of memory accesses and loop bounds yields a rollback, followed by instrumentation and serial loop completion. For input (iii), Apollo is not able to interpolate linear functions to build the prediction model, and continuously switches between instrumented and original executions of the loop. The last input, (iv), represents the worst case scenario for Apollo, consisting of multiple phases of a few iterations. After each instrumented execution for profiling, Apollo successfully builds a prediction model and generates an optimized code; but when executing the speculatively-optimized version, a misprediction is detected, and
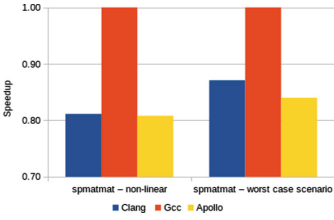
**Fig. 8.** Slow-downs compared to clang and gcc in worst-case scenarios.
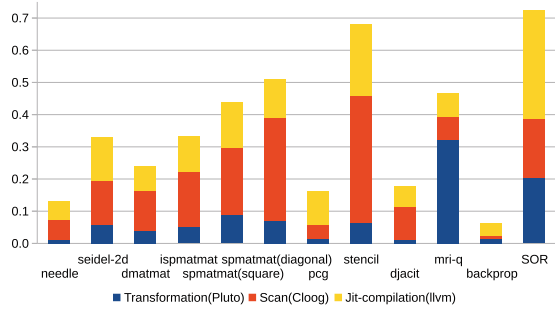


**Fig. 9.** Total code-generation times.

a rollback occurs. For this last input, the execution yields 6 rollbacks. Figure 8 shows a comparison with inputs (iii) and (iv), where Apollo is not able to execute any optimized slice of iterations successfully. Caption `spmatmat - non-linear` stands for the previously described input (iii), while caption `spmatmat - worst case scenario` stands for input (iv). Even if Apollo imposes an overhead, it does not influence execution time significantly from clang's serial version. Moreover, even if input (iv) yields the worst case scenario – where the framework continuously performs code generation, backup, and fails during the optimized code execution, and rollbacks – the performance impact is weak.

As mentioned, we can distinguish four parts in the code generation process of Apollo: (i) encoding the code bones in an OpenScop object, (ii) determining a polyhedral transformation (Pluto), (iii) generating the scan (Cloog), and finally (iv) generating executable code (LLVM-JIT). In Fig. 9, we only depict the time overheads for the last three stages. Indeed, the time spent encoding the code bones information (i) is negligible when compared to the other parts (0.012 s at maximum, 0.004 s on average). Figure 10 shows the percentage of the total execution time which is spent in the main steps of Apollo. Interpolation, code generation and transformation selection phases are grouped with caption 'code-generation'. The backup time remains lower than 5 % of the execution time for most of the benchmarks. Instrumentation is one of the most time-consuming phases of Apollo, however mostly as fast as executing the original serial code. It is particularly large for `backprop`, since the original serial code is slow due to poor data locality (column-major array access). Since the code generation phase is always executed in parallel with the original code, the percentage corresponding to the original code execution is always higher than the one corresponding to the code generation phase. Figure 11 show a comparison between the code generation strategy using skeletons used by Apollo and the one proposed in this paper with code bones. To compute the speed-ups, we selected the best serial version generated among the gcc-4.8 or clang-3.4 compilers with optimization level 3 (-O3). The code bones approach outperforms the code skeletons approach for
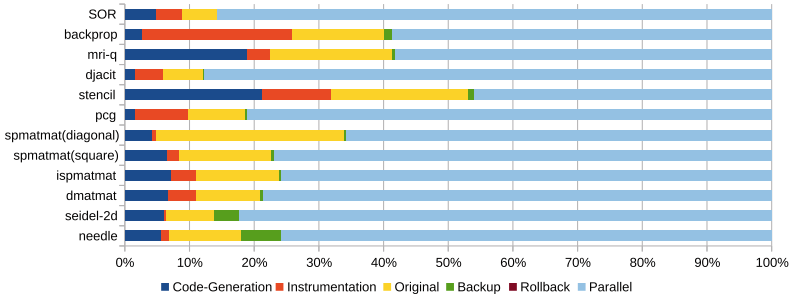
**Fig. 10.** Overheads of Apollo among the total execution time(24 threads).
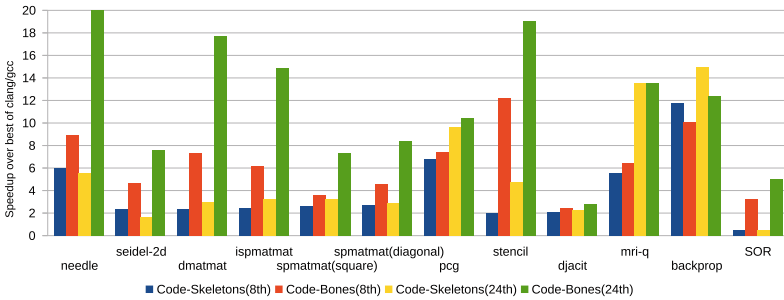


**Fig. 11.** Speedup against the best of clang/gcc (8 and 24 threads)

codes that benefit from transformations not supported by the code skeleton approach, such as tiling. For benchmarks where the applied transformation is the same with both approaches, similar speed-ups are obtained. For SOR, tiling is required for parallelization.

## 5    Related Work

The Apollo framework is a major revision of a previous framework called VMAD [9]. To generate parallel code on-the-fly, VMAD builds *code skeletons* at compile-time, whose limitations have been addressed in this paper.

Most TLS systems [11,12,15] are limited to simple parallelization schemes: the outermost loop of the original loop nest is optimistically sliced into speculative parallel threads. Such a scheme does not consider complex reordering of iterations and statements, thus, the implemented code generation mechanisms are reduced to different statically generated and simple code versions, and a runtime system that switches between them. Softspec [4] represents preliminary ideas of our approach. However, no code transformations are performed, only slicing the loop for parallel execution.

Polly [8] may be seen as the static counterpart of our proposal. Polly is a polyhedral compiler built on top of LLVM. However, since Polly operates only at

compile-time, without any coupled runtime system, it is limited to codes where precise information is available in the LLVM-IR. SPolly [6] is an extension to enlarge its applicability, by detecting common expression values and aliasing properties that prevent polyhedral optimization. During a first execution of the program, a profile is generated; values and aliasing properties are deduced, and specialized versions of the loop are created. These specialized code versions are not generated at runtime. There is no speculation and thus no verification code.

## 6    Conclusion

The proposed runtime code generation strategy offers the opportunity of applying any polyhedral loop transformation on-the-fly, without paying a penalizing time overhead. It also enlarges the scope of speculative parallelization by bringing it closer to what a static optimizing compiler may achieve. The CFG abstraction using code bones could also be employed for other goals related to dynamic optimization, as soon as the runtime process consists in scheduling, guarding and instantiating some sub-parts of the target code.

## References

1. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT (2004)
2. Bastoul, C.: Openscop: A specification and a library for data exchange in polyhedral compilation tools. Technical report (2011)
3. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: PLDI (2008)
4. Bruening, D., Devabhaktuni, S., Amarasinghe, S.: Softspec: software-based speculative parallelism. In: ACM FDDO (2000)
5. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.H., Skadron, K.: Rodinia: a benchmark suite for heterogeneous computing. In: IISWC (2009)
6. Doerfert, J., Hammacher, C., Streit, K., Hack, S.: Spolly: speculative optimizations in the polyhedral model. In: IMPACT (2013)
7. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part 2: multidimensional time. IJPP **21**(6), 389–420 (1992)
8. Grosser, T., Größlinger, A., Lengauer, C.: Polly - performing polyhedral optimizations on a low-level intermediate representation. PPL **22**(4), 28 (2012)
9. Jimborean, A., Clauss, P., Dollinger, J.F., Loechner, V., Martinez, J.M.: Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. IJPP **42**(4), 1–17 (2014)
10. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO 2004 (2004)
11. Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., Torrellas, J.: Posh: a tls compiler that exploits program structure. In: PPoPP (2006)
12. Rauchwerger, L., Padua, D.: The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. In: SIGPLAN Not. (1995)
13. van der Spek, H., Bakker, E., Wijshoff, H.: Spark00: A benchmark package for the compiler evaluation of irregular/sparse codes. (2008). arXiv:0805.3897

14. Steffan, J., Mowry, T.: The potential for using thread-level data speculation to facilitate automatic parallelization. In: HPCA 1998 (1998)
15. Steffan, J., Colohan, C., Zhai, A., Mowry, T.: The stampede approach to thread-level speculation. ACM TCS **23**, 253–300 (2005)
16. Stratton, J.A., Rodrigues, C., Sung, I.J., Obeid, N., Chang, L.W., Anssari, N., Liu, G.D., Hwu, W.-M.W.: The Parboil technical report. Techical report, IMPACT (2012)
17. Sukumaran-Rajam, A., Martinez Caamaño, J.M., Wolff, W., Jimborean, A., Clauss, P.: Speculative program parallelization with scalable and decentralized runtime verification. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 124–139. Springer, Heidelberg (2014)
18. Upadrasta, R., Cohen, A.: Sub-polyhedral scheduling using (unit-)two-variable-per-inequality polyhedra. In: POPL 2013 (2013)