ALONA: Automatic Loop Nest Approximation with Reconstruction and Space Pruning

Daniel Maier[1], Biagio Cosenza[2], and Ben Juurlink[1]

[1] Technische Universität Berlin, Germany
[2] University of Salerno, Italy

Abstract. Approximate computing comprises a large variety of techniques that trade the accuracy of an application's output for other metrics such as computing time or energy cost. Many existing approximation techniques focus on loops such as loop perforation, which skips iterations for faster, approximated computation. This paper introduces ALONA, a novel approach for automatic loop nest approximation based on polyhedral compilation. ALONA's compilation framework applies a sequence of loop approximation transformations, generalizes state-of-the-art perforation techniques, and introduces new multi-dimensional approximation schemes. The framework includes a reconstruction technique that significantly improves the accuracy of the approximations and a transformation space pruning method based on Barvinok's counting that removes inaccurate approximations. Evaluated on a collection of more than twenty applications from PolyBench/C, ALONA discovers new approximations that are better than state-of-the-art techniques in both approximation accuracy and performance.

1 Introduction

Many real-world applications can trade the accuracy of an application's result for other metrics, typically performance or energy. Approximate computing is an emerging paradigm that explicitly exploits this gap. Prior work investigates both hardware and software techniques [23]. Software techniques include soft slices [31] and mixed-precision tuning [8]. A popular approach is to attack the problem at loop level. Loop perforation [33] is a general-purpose technique that lowers the accuracy of loops by skipping loop iterations. Similarly, loop perforation can also be applied to data-parallel kernels executed on GPUs. Paraprox [30] is a framework for the approximation of data-parallel programs that operates on commodity hardware systems. More recently, these perforation techniques have been augmented with a reconstruction phase that improves the accuracy of perforated kernels [21].

While loops are generally an important target for both approximation and optimization techniques, e.g., loop-level parallelization, existing loop-level approximation techniques focus on one loop and a specific dimension instead of taking the whole iteration domain of all loop nests into consideration.

Polyhedral compilation has been proven to be an effective way to reason about loops. Polyhedral techniques can potentially target any affine loop nest, and perform a variety of tasks such as the efficient application of code transformations [27, 4], the accurate modeling of performance and other metrics [15, 1], and automatic parallelization [5, 3].

We explore how the polyhedral model can be effectively used for the implementation of automatic approximation techniques targeting loop nests. In particular, we use it to efficiently handle a sequence of code transformations, which in our case includes loop perforation, and to filter the large set of possible approximations using a metric based on Barvinok's counting of the number of the exact and approximate loop iterations [2]. Furthermore, we provide a signal reconstruction step implemented as a post-processing technique that improves the accuracy by reconstructing missing values in the final result and provides an interface for application-specific reconstruction.

Our approach is fully compatible with existing polyhedral techniques that focus on automatic parallelization, and, in fact, extends existing state-of-the-art perforation frameworks such as Paraprox [30] and Sculptor [20] in terms of supported approximation schemes, and improves them in terms of accuracy and performance.
The contributions of this work are:

1. We introduce ALONA, the first compiler approach for automatic approximation of affine loop nests based on polyhedral techniques. ALONA generalizes existing state-of-the-art perforation techniques and can model multi-dimensional approximation schemes (Section 3.2, evaluated in Section 6.2).
2. ALONA's accuracy is significantly improved by supporting signal reconstruction techniques that mitigate the error by reconstructing missing values in the output (Section 4, evaluated in Section 6.3).
3. To efficiently handle the large transformation space, ALONA proposes an approximation space pruning technique based on Barvinok's counting of loop iterations (Section 5, evaluated in Section 6.4).
4. We experimentally evaluate the proposed polyhedral perforation framework on a collection of more than twenty benchmarks from PolyBench and Paraprox (Section 6). Results show that ALONA discovers new approximations and outperforms state-of-the-art methods.

2 Related Work

Approximate computing is an emerging paradigm where accuracy is traded for a gain in performance or a reduction in energy consumption. This trade-off is feasible because there is often a gap between the accuracy provided by a system and the accuracy required by an application. Research in approximate computing exploits this gap using a variety of approaches, ranging from hardware-supported techniques such as Truffle [11], CADE [16] and Replica [12], to pure software methods, including compilers and APIs, which we review in this section.

The Approxilyzer framework [35] quantifies the significance of instructions in programs when considering the output accuracy; given a program and an end-to-end quality metric, it selects a set of instructions as candidates for approximation by introducing bit errors. TAFFO [8] is a dynamic assistant for floating to fixed point conversion. A general-purpose software technique, extended and refined in many recent works [19–21], is *loop perforation* [33]. Loops are the bottleneck in many applications. Loop perforation skips loop iterations (or part of loop iterations) in order to reduce the compute load and to gain performance. Paraprox [30] is a framework for the approximation of data-parallel programs. It uses approximation techniques specifically tailored to specific data-parallel patterns. For instance, stencil patterns are approximated using center, row, and column value approximation. Maier et al. [21] extend these patterns with an additional reconstruction phase that exploits GPU's local memory for better approximation accuracy. Sculptor [20] moves the scope of perforation from loop iterations to individual statements and explores dynamic perforation, e.g., a perforation pattern that changes during the runtime of a program. Different phases during program execution with individual sensitivity towards approximation were explored by Mitra et al. [22]. Lashgar et al. [19] extend the OpenACC programming model with support for loop perforation. Relaxation of synchronizations is studied in the parallelizing compiler HELIX-UP [7] and by Deiana et al. [10] who propose a C++ language extension for non-deterministic programs. Some approaches focus on other parallel semantics, e.g., speculative lock elision [18], task discarding [29], and breakable dependence [34].

Polyhedral compilation has been proven as an effective way to reason about loops, both in terms of transformation and modeling. Polyhedral techniques target affine loop nests and can be used for a variety of tasks such as the efficient application of code transformations or the design of performance models. The polyhedral approach is nowadays in use by several automatic parallelization compilers, notable examples are Pluto [6, 3], Pluto+ [5], PPCG [36], Polly [14], and speculative parallelization [17]. High-level loop transformations are critical for performance; however, finding the best sequence of transformations is difficult, and resulting transformations are both machine- and program-dependent. As optimizing compilers often use simplistic performance models, iterative compilation [26, 25, 13] is widely used to maximize performance. Adaptive Code Refinement [32] is a semi-automatic approach that relies on the polyhedral representation of stencil programs to apply transformations and to generate an approximated version of the code.

3 ALONA

We show the compilation workflow of our framework in Fig. 1 and it comprises of eight steps. (1) Starting from the accurate program the SCoPs are extracted. (2) Create the approximation space by generating perforated SCoPs for each SCoP. (3) In the next step, the Barvinok score of each approximation is calculated. (4) The approximated SCoPs are sorted based on the score. (5) The approximation space is pruned. (6) From the remaining approximated SCoPS, the source code is synthesized. (7) The selected programs are compiled and executed while recording performance and accuracy. (8) Finally, Pareto-optimal solutions are selected. Optionally, a reconstruction phase is performed after step 6.

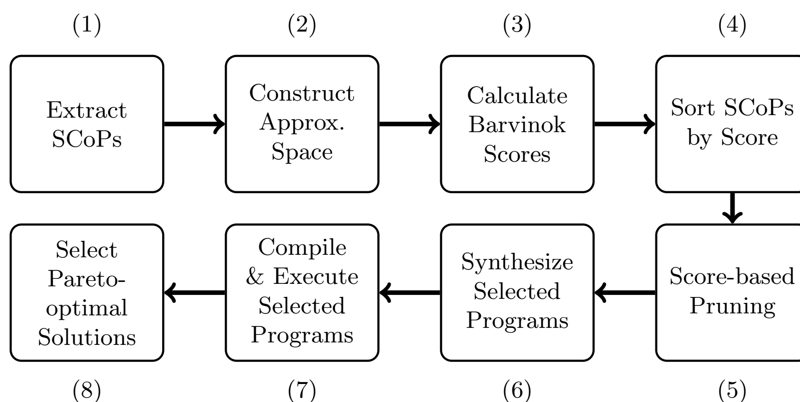


Fig. 1: ALONA's Compilation Workflow.

3.1 Polyhedral Loop Nest Perforation

Loop perforation is the technique of skipping parts of a loop. ALONA implements loop nest perforation with the tools provided by the polyhedral model. Loop perforation transforms loops to execute a subset of their iterations in order to reduce the amount of computational work [33]. This transformation can be accomplished, e.g., by changing the increment of the loop variable or by modifying the loop start and end. Listing 1.2 shows the perforated loop of Listing 1.1: the loop variable `i` is incremented by 2 instead of 1 and, therefore, every other loop iteration is perforated.

```
for(i=0; i<=7; i++) {
    /* work() */
}
```

Listing 1.1: Original Loop.

```
for(i=0; i<=7; i+=2) {
    /* work() */
}
```

Listing 1.2: Perforated Loop.

3.2 Polyhedral Model

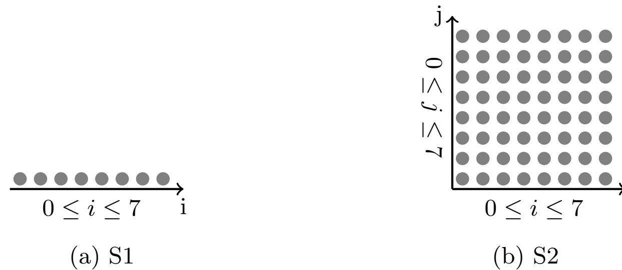
The polyhedral model represents loops by convex polyhedra and uses parametric integer programming for analysis and transformation. A polyhedron is a convex set of points in a lattice, i.e., a set of points in a $\mathbb{Z}$ vector space that is bounded by affine inequalities. Loop nests, in their algebraic representation, are called *static control parts* (SCoP). A SCoP contains all information about control and data flow. Loop nests are usually required to have a statically defined control flow. A SCoP is a maximal set of consecutive statements where loop bounds and conditionals only depend on invariants and global parameters. The global parameters are constant but statically unknown and only available during runtime.

```
for(i=0; i<=7; i++) {
S1: C[i] = 0;
    for(j=0; j<=7; j++) {
S2:     C[i] += A[i][j] * B[j];
    }
}
```

Listing 1.3: Accurate Loop Nest.

Listing 1.3 shows an exemplary loop nest that computes the matrix-vector product $C = A \cdot B$, and which we use throughout this section to provide examples of iteration domain and memory access functions. The loop nest contains two SCoPs: $S1$ and $S2$. The outer loop runs from $i = 0$ to $i \leq 7$, incrementing i for every iteration and initializing $C[i] = 0$. The inner loop runs from $j = 0$ to $j \leq 7$, incrementing j, reading from $C[i]$, $A[i][j]$ and $B[j]$ and writing to $C[i]$.

The *iteration domain* represents the iterations of a statement in a loop nest and describes the dynamic instances of all statements in the SCoP. A set of affine inequalities defines all possible values of the surrounding loop iterators. Each instance of a statement S is identified by (S, i) where i is the *iteration vector* that contains the values of the loop indices of the surrounding loops.



(a) S1 (b) S2

Fig. 2: Iteration Domain for the Loop Nest in Listing 1.3.

The set of inequalities for SCoP $S1$ in Listing 1.3 consists of $0 \le i \le 7$. The inequalities bounding the iteration domain of $S2$ are $0 \le i \le 7$ and $0 \le j \le 7$. The iteration domain defined by these inequalities is shown in Figure 2. While $S1$ is one-dimensional, $S2$ has two dimensions spanned by i and j.
Memory access functions describe the locations of data that is accessed by statements. Memory accesses are performed through array references. For each statement S, two sets $\mathcal{L}^S$ and $\mathcal{R}^S$ exist, each containing (M, f) pairs. Each pair is a reference to a variable M that is accessed (written $\mathcal{L}$ or read $\mathcal{R}$) and a data access function f that maps the iteration vector in $\mathcal{D}^S$ to the memory location in the variable M. The arrow $\rightarrow$ denotes the mapping of a statement $S(i)$ with iteration vector i to variable $M(i)$ where i denotes the location in M. The memory access functions for Listing 1.3 are $\mathcal{L}^{S1} = S1(i) \rightarrow C(i)$, $\mathcal{R}^{S1} = \emptyset$ and $\mathcal{L}^{S2} = S2(i) \rightarrow C(i)$, and $\mathcal{R}^{S2} = S2(i,j) \rightarrow A(i,j), B(j), C(i)$.

Polyhedral Transformations are accomplished in the polyhedral representation of a loop nest. These transformations are constructed out of a set of transformation primitives that mostly correspond to simple polyhedral operations. By composition of an arbitrary number of transformations, complex optimizations can be built. The actual code generation happens after all transformations have been applied and this step is independent of the actual transformations.

3.3 Polyhedral Loop Perforation

Polyhedral loop perforation is implemented in two steps: shrinkage of the iteration domain, and adjustment of the memory accesses.
(1) The iteration domain is bounded by affine inequalities and, in order to shrink the size of the iteration domain, we alter these inequalities. First, the dimension which should be reduced in size has to be identified together with the loop variable, i.e., `i`. Next, we alter the inequalities to reflect the new size of the iteration domain. Although this can be done in many ways, we explain here for reasons of simplicity the following approach: In the inequalities, we multiply all occurrences of `i` with the perforation factor $f = 2$. Consider again SCoP `S2` in Figure 2b. First, we shrink the iteration domain by the perforation factor. The

resulting iteration domain is visualized in Figure 3a. The iteration domain is now half the size in the i-dimension and the inequality for i is updated.
(2) We adjust the memory accesses. Otherwise, a whole chunk of loop instances is perforated instead of every other instance. The adjustment of the memory accesses is closely related to the perforation factor f. In fact, in each memory access, all occurrences of the loop index variable are multiplied by the loop perforation factor, i.e., `i` becomes `2*i`. Using this approach, we achieve the effect of a perforated iteration domain, while the iteration domain is in fact dense and convex. Recall the memory access functions for SCoP S2 from Section 3.2: $\mathcal{L}^{S2} = S2(i) \rightarrow C(i)$, and $\mathcal{R}^{S2} = S2(i,j) \rightarrow A(i,j), B(j), C(i)$. We multiply every occurrence of `i` with `2` in order to adjust the memory accesses. We use $\mathcal{L}_p$ and $\mathcal{R}_p$ to denote the perforated memory access functions: $\mathcal{L}_p^{S2} = S2(i) \rightarrow C(2i)$, and $\mathcal{R}_p^{S2} = S2(i,j) \rightarrow A(2i,j), B(j), C(2i)$. The result is depicted in Figure 3b.

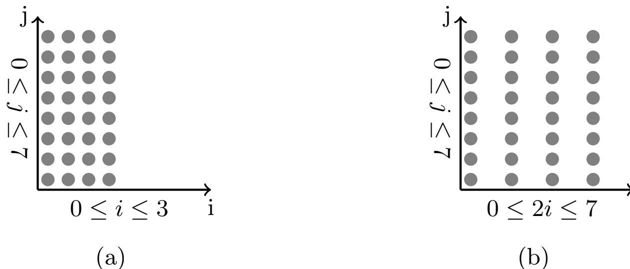


Fig. 3: The Intermediate Steps of Polyhedral Loop Perforation.

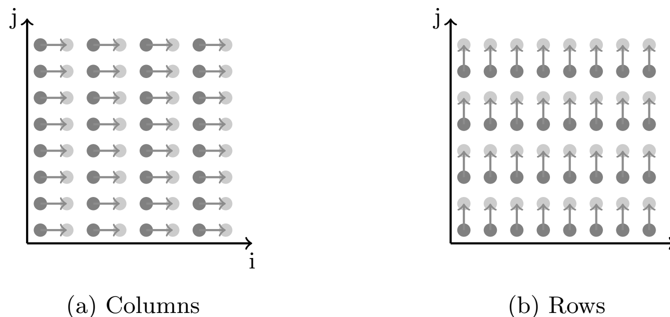
3.4 Extensions to Classical Loop Perforation

Classical loop perforation [33] has been refined to include parallel programs [30], to perforate on statement-level and using non-static pattern [20], or to perforate threads on GPUs [21]. This section shows how our approach generalizes existing approximation schemes, which are all covered by our framework.
Paraprox [30] uses three schemes to approximate stencil data access patterns: the center scheme that uses the center value in a stencil to approximate the neighboring values; the rows scheme that perforates two out of three rows by using one row to approximate the row above and the row below; and the column scheme that is a 90° rotated row scheme. The rows and columns scheme can be implemented straightforwardly using polyhedral perforation on the corresponding inner or outer loop. The center scheme can be obtained by applying first a perforation of rows and successively a perforation of columns.
Sculptor [20] extends the perforation from loop iterations to individual statements and uses dynamic perforation patterns instead of being limited to fixed patterns for the whole loop. Sculptor operates on LLVM bitcode, while we operate on high-level C code. Our approach is able to perforate on an individual

statement level, and it is able to employ different perforation patterns for different statements. We support different perforation patterns in a loop by first applying loop splitting, which splits a loop into two sub-loops. Then, we perforate the new loops individually.
Multi-dimensional Schemes: As a result of using the polyhedral model and performing perforation at SCoP level, ALONA supports multi-dimensional perforation of loop nests, which is not supported by previous work.

4 Reconstruction

By post-processing the results of an approximated application, the error introduced by the approximation technique can be mitigated. As this process depends heavily on the application, the *reconstruction phase* requires an application-specific or data-specific approach to achieve optimal results. However, in some cases, even a general approach can be beneficial. The impact of a general and simple reconstruction approach is also shown by related work [21]. Therefore, we introduce our reconstruction phase that prevents uninitialized values in the results of applications.



(a) Columns (b) Rows

Fig. 4: Reconstruction.

Consider a loop where one output element is calculated in each iteration. After perforating every other loop iteration, every other output element is correct, but the remaining output elements are not written at all and, therefore, have high error. Data often contains redundancy, e.g., spatial redundancy in images, where pixels close to each other are very likely to have a similar value. We exploit this redundancy to reconstruct missing data. In our approach, a reconstruction phase is implemented together with the perforation of SCoPs as this allows us to easily identify the perforated elements and replace them with nearby approximations. We apply the reconstruction only to the output SCoPs, i.e., when the final result is written, while we apply normal (non-reconstructed) perforation to all other SCoPs. As the reconstruction is performed in a post-processing step outside the polyhedral model, it does not influence polyhedral transformations.

The reconstruction phase comprises two steps: (1) identify perforations of the output SCoP of the application, and (2) synthesize code to reconstruct all missing output elements. This can be done, e.g., using the nearest neighbor output element that was not perforated. However, also more complex interpolation is possible. For a two-dimensional loop nest that is perforated in both dimensions, the perforation is separated for each dimension. This example is depicted in Figure 4. First, the `i`-dimension is reconstructed (a); then, the `j`-dimension of the loop is reconstructed (b). Arrows indicate the source and target of the reconstruction. Bright points indicate reconstructed data copied from adjacent dark points. While ALONA includes a generic framework for reconstruction, it is also possible to integrate application-specific reconstruction phases.

5 Approximation Space Pruning

ALONA generates many approximated code versions. The number of approximated code versions depends on the number of loop nests, their depth, and the number of supported skip factors. For example, in Listing 1.3, the approximation space for a given skip factor and scheme, consists of three approximations: (1) perforate S1; (2) perforate S2; (3) perforate S1 and S2. Considering different scheme and factor, the number of possible approximated code versions can be very large. This section introduces ALONA's approximation space pruning technique, which is able to find approximated code versions with low error, therefore reducing the number of version to be executed, e.g., by search-based autotuners.

5.1 Approximation Ordering

Accurately modeling the error of an approximated program is a complex task, because the error profile of an application is not only application-specific, but it depends also on the input data. Instead of trying to accurately model the error, our approach solves an easier task: ranking each approximated code version so that, e.g., only those with higher rank are evaluated. The idea to focus on an ordering problem instead of an accurate (regression) model has been previously used for performance tuning [9]; here, we use it to filter the most accurate code versions that are later executed.

Our method establishes an ordering of the possible loop nest approximations based on the observation that a code version with more perforated iterations is likely to have a higher error than a code version with fewer perforated iterations. We calculate this ordering by computing, for each code version, the ratio of loop iterations that are perforated, which ranges from 0 (all loop iterations perforated) to 1 (accurate program). For instance, for a program with one loop that is perforated for every other loop iteration, this ratio equals to 0.5.

5.2 Barvinok Counting Based Pruning

We use pruning to reduce the number of code versions to be tested. Our approach is based on the observation that there is a connection between the

amount of work removed and speedup for many applications. We use Barvinok counting to calculate both the size of the original accurate iteration domain and the size of the approximated perforated domain. We use the ratio of the cardinality of the iteration domain of the perforated program and the cardinality of the iteration domain of the original program. In order to calculate this score, we count the number of instances in the iteration domain of each SCoP. We compute the sum of the cardinality of $\mathfrak{D}_p^i$ of the perforated program and divide it by the sum of the cardinality $\mathfrak{D}^i$ for the original program. The resulting ratio is the amount of perforated loop instances of all SCoPs of the program. This number is independent of the problem size.

$$Score(\mathfrak{D}_p) = \frac{\sum_{\forall i} |\mathfrak{D}_p^i|}{\sum_{\forall i} |\mathfrak{D}^i|}$$

Pruning is used to reduce the optimization time by lowering the number of configurations considered. First, we apply score-based pruning where configurations below a certain score are discarded. Then, exhaustive search is used to select optimal solutions.

6 Experimental Evaluation

We evaluate the accuracy and performance of ALONA and compare it to Paraprox [30], the state-of-the-art perforation technique. Our set of configurations also include those used in Sculptor [20]; however, we are unable to label the specific configurations, as we do not know Sculptor's algorithmic decisions.

We use a set of well-known and well-tested tools for the experimental evaluation of our approach. We use the Polyhedral Compiler Collection [28] for the extraction of the SCoP (clan), which is extended by our framework to implement the SCoP perforation. Furthermore, we use the Barvinok library to retrieve the size of the iteration domain. Finally, Cloog is used for code generation and gcc 7.4.0 (-O3) to compile the synthesized code.

6.1 Benchmark Setup

We measure runtime and accuracy of 22 applications from PolyBench/C 4.1 [24] and three additional applications also used by Paraprox. Our measurements were conducted on an Intel Core i7-3930K (3.20 GHz). The accuracy of an application is heavily influenced by the input data and can span a wide range. We evaluate the applications with their default input and, therefore, the error is not representative for all possible inputs. However, when comparing different perforation configurations, the accuracy trend shows which perforation schemes deliver higher accuracy. We report the *mean relative error* (MRE) of the element-wise difference of true and approximated results.

6.2 Discovered Solutions

In Figure 5, we show detailed speedup and error for six applications. Orange and red points indicate state-of-the-art techniques. Blue points show approximations

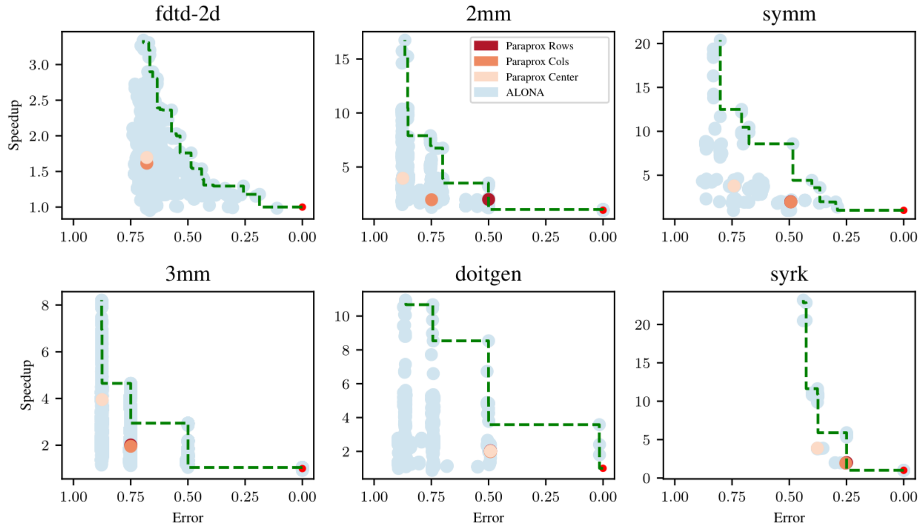


Fig. 5: Configurations Discovered by ALONA and State-of-the-art Techniques.

discovered by ALONA. Using the green dashed line, we indicate Pareto-optimal configurations. We show how ALONA is able to outperform Paraprox both in terms of accuracy and speedup and, in fact, that many superior solutions are discovered.

For the FDTD-2D application, there are 26 Pareto-optimal configurations. ALONA generates approximated programs that are nearly twice as fast when compared to Paraprox for the same error budget. Many configurations outperform Paraprox in both speedup and error. 2MM has 17 Pareto-optimal approximations, and they result in three clusters of approximately the same error. The application is very sensitive to approximation and all points result in an error of at least 50%. Here, ALONA outperforms state-of-the-art techniques in terms of speedup. There are 12 Pareto-optimal configurations for SYMM. ALONA yields a more than 4× higher speedup than state of the art, and it is also able to provide a lower speedup for the same performance. The clustering of the points of 3MM is similar to the 2MM application as it performs a similar calculation. 12 configurations are Pareto-optimal and ALONA improves on both error and speedup. While most configurations generated by ALONA for DOITGEN have an error of 50% or higher, there are 3 configurations that have a much lower error of around 2% and a speedup that outperforms Paraprox' by 2×. 14 configurations are Pareto-optimal. SYRK has 11 Pareto-optimal configurations and ALONA is able to improve on the speedup.

Overall, the results show that as our approach generalizes existing perforation schemes, it is capable to discover new approximation schemes, and many of those newly found solutions are also Pareto-optimal.

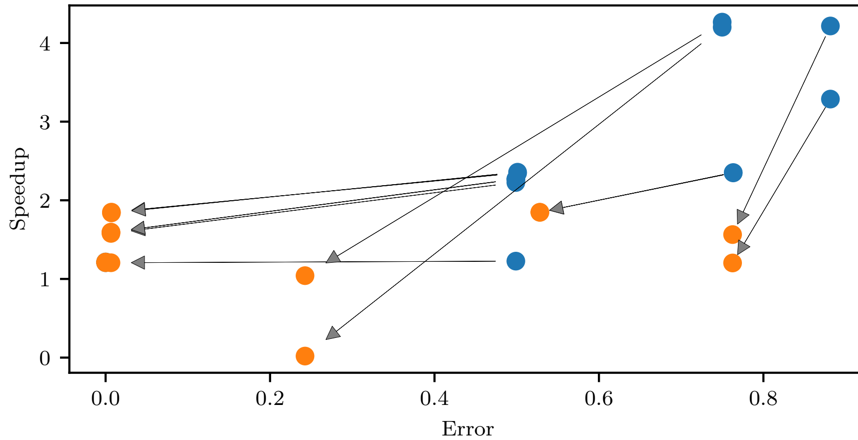


Fig. 6: Effects of Reconstruction.

6.3 Reconstruction

We evaluate a case study detailing the effects of the reconstruction for the BICG application. The prospects of reconstruction and the importance of application-specific reconstruction techniques are outlined in Section 4. In our case study, we use a post-processing step to replace all missing data values in the result with adjacent data values.

The experimental results are shown in Fig. 6. There are 14 different approximated code versions. For each of the different approximated programs, we measure error and speedup, respectively with and without reconstruction. Some approximations being very similar in performance and accuracy and thus the points are overlapping. Blue points indicate approximations *without* reconstruction. Orange points connected by arrows show the same approximations *with* reconstruction enabled. Ideally, the arrows are horizontal lines (same speedup, accuracy improvement). All arrows are pointing to the bottom left, as the error is reduced and the speedup is affected. In all cases where the non-reconstructed results are affected by error, utilizing the reconstruction lowers the error, in many cases significantly. The speedup is affected moderately in many cases. However, there are also cases where the speedup decreases sharply, or the approximated application is slowed down in comparison to the baseline. On average, we are able to improve the error by approx. 30% while retaining approx. 60% of the speedup by employing reconstruction.

These results emphasize the importance of reconstruction in order to minimize the error: even when using a basic reconstruction technique, there is big potential to improve the accuracy by an order of magnitude while retaining most of the performance.

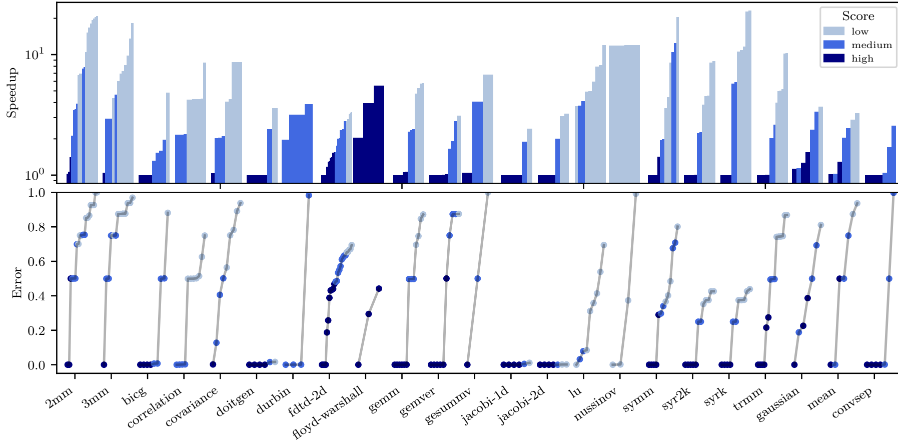


Fig. 7: Speedup and Accuracy of Pareto-Optimal Configurations Discovered by ALONA.

6.4 Evaluation of Perforation Configurations

Depending on the application, i.e., the number of nested loops, the space of possible perforation configurations can be very large. There is no single configuration that is superior to all other configurations, because the target of optimization is multi-objective: the two goals are high performance and low error.

In Figure 7, we compare Pareto-optimal perforation schemes for all applications. The figure is built from two related subplots: In the upper bar chart, we show the speedup of all Pareto-optimal configurations; and in the lower part we show the *corresponding* error. We use three color shades from light to dark that indicate the score. Darker points/bars have a higher score and brighter points have a lower score. All configurations use no reconstruction. For most of the applications, the majority of the Pareto-optimal solutions has a low score. A notable exception is FLOYD-WARSHALL, which has only a small number of Pareto-optimal solutions. Solutions with a small speedup have a darker shade, a medium speedup is indicated by a medium shade and high speedups are usually brighter shaded. This can be explained by the rationale that perforating a smaller part leads to a smaller speedup and perforating a larger part of an application leads to a larger speedup. A similar trend can be observed for the accuracy.

It should be noted that PolyBench is not curated as an approximate computing benchmark suite. Therefore, its applications are not necessarily resilient to approximation. Furthermore, in this specific experiment, the results are not improved by our reconstruction approach as we do not have application-specific reconstruction routines for all applications. This can be clearly observed, as there are some applications where the error for most of the configurations is rather high, e.g., applications with low resilience like 2MM and 3MM.

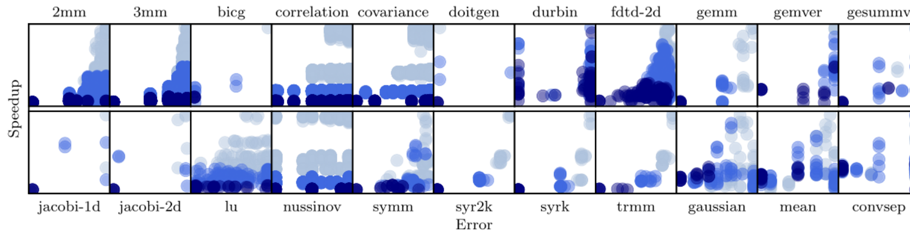


Fig. 8: Speedup and Accuracy in Relation to Score.

Nonetheless, ALONA is able to find some solutions with low error and a speedup larger than 1 for almost all applications. There are also applications, where almost all solutions have a low error, e.g., BICG, DOITGEN, DURBIN, JACOBI-1D, JACOBI-2D, or CONVSEP, which indicates that these applications are more resilient to approximation.

6.5 Approximation Space Pruning

In Figure 8, the results of our Barvinok-based approximation space pruning are shown. Each subfigure plots speedup (y-axis) and error (x-axis) for all configurations individually for each application. We calculate the Barvinok-based score as detailed in Section 5. The brightness of the points indicates the score. Notably, for most applications, the set of solutions can clearly be divided into three large sets with respectively low, medium, and high speedup. This observation is useful in the context of performing a static pre-filtering of the approximation space to significantly reduce the number of approximations, e.g., to one third of all approximations.

Exemplary, for the 2MM application, the *time to solution* is 99 s for the compilation and 4213 s (approx. 72 min) for executing all application variants. By using our score-based filtering, we are able to reduce this time to 768 s (approx. 13 min). This time is composed of 39 s for compilation and 729 s for the execution of the applications.

7 Conclusions and Future Work

ALONA is a new approximation framework that automatically approximates affine loop nests. By using polyhedral analysis, ALONA extends state-of-the-art loop perforation techniques and applies perforation and reconstruction on multidimensional loop nests. It also features an optional processing step that reconstructs missing results and significantly improves the accuracy of the approximated applications. The large number of approximated code variants constructed by ALONA are tackled with an approximation space pruning technique, which filters code variants with larger error thanks to a Barvinok's counting-based

approximation metric. This technique can be combined with search heuristics and iterative compilation, providing an efficient tool to identify fast code versions with low error and overall reducing the time to solution. Experimental results show that ALONA is capable of discovering new approximations that are Pareto-dominant with respect to approximations found by state-of-the-art approximation schemes. In particular, whenever we have the same approximation accuracy as state-of-the-art approaches, ALONA's performance is better.

In future work, we plan to integrate ALONA with existing polyhedral optimizers and in combination with automatic parallelization.

Acknowledgement

This research has been partially funded by MIUR PON Ricerca e Innovazione 2014-2020 (grant number AIM1872991-1).

References

1. Bao, W., Krishnamoorthy, S., Pouchet, L.N., Sadayappan, P.: Analytical Modeling of Cache Behavior for Affine Programs. ACM Program. Lang. (2017)
2. Barvinok, A.I.: A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. Mathematics of Operations Research (1994)
3. Baskaran, M.M., Ramanujam, J., Sadayappan, P.: Automatic C-to-CUDA Code Generation for Affine Programs. In: Proc. CC (2010)
4. Benabderrahmane, M., Pouchet, L., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: Proc. CC (2010)
5. Bondhugula, U., Acharya, A., Cohen, A.: The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests. ACM TOPLAS (2016)
6. Bondhugula, U., Baskaran, M.M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In: Proc. CC (2008)
7. Campanoni, S., Holloway, G.H., Wei, G., Brooks, D.M.: HELIX-UP: relaxing program semantics to unleash parallelization. In: Proc. CGO (2015)
8. Cherubin, S., Cattaneo, D., Chiari, M., Agosta, G.: Dynamic Precision Autotuning with TAFFO. TACO (2020)
9. Cosenza, B., Durillo, J.J., Ermon, S., Juurlink, B.: Autotuning stencil computations with structural ordinal regression learning. In: Proc. IPDPS (2017)
10. Deiana, E.A., St-Amour, V., Dinda, P.A., Hardavellas, N., Campanoni, S.: Unconventional Parallelization of Nondeterministic Applications. SIGPLAN Not. (2018)
11. Esmaeilzadeh, H., Sampson, A., Ceze, L., Burger, D.: Architecture support for disciplined approximate programming. In: Proc. ASPLOS (2017)
12. Fernando, V., Franques, A., Abadal, S., Misailovic, S., Torrellas, J.: Replica: A wireless manycore for communication-intensive and approximate data. In: Proc. ASPLOS (2019)
13. Ganser, S., Größlinger, A., Siegmund, N., Apel, S., Lengauer, C.: Speeding Up Iterative Polyhedral Schedule Optimization with Surrogate Performance Models. TACO (2018)

14. Grosser, T., Größlinger, A., Lengauer, C.: Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. Parallel Proc. Letters (2012)
15. Gysi, T., Grosser, T., Brandner, L., Hoefler, T.: A fast analytical model of fully associative caches. In: Proc. PLDI (2019)
16. Imani, M., Garcia, R., Huang, A., Rosing, T.: CADE: Configurable Approximate Divider for Energy Efficiency. In: Proc. DATE (2019)
17. Jimborean, A., Clauss, P., Pradelle, B., Mastrangelo, L., Loechner, V.: Adapting the polyhedral model as a framework for efficient speculative parallelization. In: Proc. PPoPP (2012)
18. Khatamifard, S.K., Akturk, I., Karpuzcu, U.R.: On approximate speculative lock elision. IEEE Transactions on Multi-Scale Computing Systems **4**(2) (2018)
19. Lashgar, A., Atoofian, E., Baniasadi, A.: Loop Perforation in OpenACC. In: Proc. ISPA (2018)
20. Li, S., Park, S., Mahlke, S.: Sculptor: Flexible approximation with selective dynamic loop perforation. In: Proc. ICS (2018)
21. Maier, D., Cosenza, B., Juurlink, B.: Local Memory-aware Kernel Perforation. In: Proc. CGO (2018)
22. Mitra, S., Gupta, M.K., Misailovic, S., Bagchi, S.: Phase-aware Optimization in Approximate Computing. In: Proc. CGO (2017)
23. Mittal, S.: A Survey of Techniques for Approximate Computing. ACM Comput. Surv. (2016)
24. Pouchet, L.N.: Polybench/c 3.2, http://www.cse.ohio-state.edu/ pouchet/software/polybench/
25. Pouchet, L.N., Bastoul, C., Cohen, A., Cavazos, J.: Iterative optimization in the polyhedral model: Part ii, multidimensional time. In: Proc. of PLDI. ACM (2008)
26. Pouchet, L.N., Bastoul, C., Cohen, A., Vasilache, N.: Iterative optimization in the polyhedral model: Part i, one-dimensional time. In: Proc. CGO (2007)
27. Pouchet, L., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P., Vasilache, N.: Loop transformations: convexity, pruning and optimization. In: Proc. POPL (2011)
28. Pouchet, L.N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, R., Sadayappan, P.: Hybrid iterative and model-driven optimization in the polyhedral model (2009)
29. Rinard, M.: Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In: Proc. ICS (2006)
30. Samadi, M., Jamshidi, D.A., Lee, J., Mahlke, S.: Paraprox: Pattern-based approximation for data parallel applications. In: Proc. ASPLOS (2014)
31. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., Grossman, D.: Enerj: approximate data types for safe and general low-power computation. In: Proc. PLDI (2011)
32. Schmitt, M., Helluy, P., Bastoul, C.: Automatic Adaptive Approximation for Stencil Computations. In: Proc. CC (2019)
33. Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H., Rinard, M.: Managing performance vs. accuracy trade-offs with loop perforation. In: Proc. ESEC (2011)
34. Udupa, A., Rajan, K., Thies, W.: ALTER: Exploiting Breakable Dependences for Parallelization. In: Proc. PLDI (2011)
35. Venkatagiri, R., Mahmoud, A., Hari, S.K.S., Adve, S.V.: Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In: Proc. MICRO (2016)
36. Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., Catthoor, F.: Polyhedral Parallel Code Generation for CUDA. ACM TACO (2013)