Oil & Gas Science and Technology - Rev. IFP Energies nouvelles **73**, 52 (2018)

https://doi.org/10.2516/ogst/2018049

Available online at:
www.ogst.ifpenergiesnouvelles.fr

Numerical methods and HPC
A. Anciaux-Sedrakian and Q. H. Tran (Guest editors)

REGULAR ARTICLE OPEN ACCESS

A language extension set to generate adaptive versions automatically

Maxime Schmitt*, Cédric Bastoul, and Philippe Helluy

University of Strasbourg and INRIA, ICube laboratory, 300 bd Sébastien Brant, 67412 Illkirch, France

Received: 25 February 2018 / Accepted: 31 July 2018

Abstract. A large part of the development effort of compute-intensive applications is devoted to optimization, *i.e.*, achieving the computation within a finite budget of time, space or energy. Given the complexity of modern architectures, writing simulation applications is often a two-step workflow. Firstly, developers design a sequential program for algorithmic tuning and debugging purposes. Secondly, experts optimize and exploit possible approximations of the original program to scale to the actual problem size. This second step is a tedious, time-consuming and error-prone task. In this paper we investigate language extensions and compiler tools to achieve that task semi-automatically in the context of approximate computing. We identified the semantic and syntactic information necessary for a compiler to automatically handle approximation and adaptive techniques for a particular class of programs. We propose a set of language extensions generic enough to provide the compiler with the useful semantic information when approximation is beneficial. We implemented the compiler infrastructure to exploit these extensions and to automatically generate the adaptively approximated version of a program. We provide an experimental study of the impact and expressiveness of our language extension set on various applications.

1 Introduction

The software development industry is constantly pushing research in language theory, compilers and tools to improve developer's productivity and the efficiency of applications. On a regular basis, new languages appear or new constructions in existing languages are developed to express a particular idea in a simpler and more concise manner and are included in new revisions of existing languages. In this paper, we present a new set of language extensions that allow approximation-specific knowledge to be integrated to any programming languages. Using these extensions, developers can express the approximate methods that they want to use, *e.g.* memory access or task skipping, without modifying their initial algorithm, and let the compiler optimize the code with the requested techniques automatically. To exploit these new extensions, we rely on state-of-the-art polyhedral representation of programs to apply code transformations on the input program and avoid unnecessary computations while keeping guarantees on the validity of the transformations.

With the rise of parallel processors, multiple vendors have formed a non-profit consortium to propose a new Application Programming Interface (API) which is known as Open Multi-Processing [1] (OpenMP). This API greatly reduces the time needed to implement parallel applications compared to using a specific library directly. In this paper we propose a similar approach for approximate computing and adaptive methods. Our method relies entirely on optional annotations for the compiler. Our "Adaptive Code Refinement" (ACR) annotations are put on top of compute intensive kernels that can benefit from approximate computing techniques. The compiler may exploit these annotations to generate multiple alternative code versions of the kernel which will be executed depending on the program's current state. In a nutshell, the software developer provides multiple "alternatives" of some computations and the conditions when they will be active. The area where these conditions are applicable can be selected among partitions of the computation kernel and may depend on runtime data. The developer can create its own static partitions or select the size of a regular partition. The compiler will exploit this information to enforce the use of the approximate alternative on the generated partitions dynamically thanks to specific compiler techniques and a specific runtime.

Compilers can use a multitude of strategies in order to optimize a program. However, traditional compilers are constrained by the input program semantics, *i.e.*, optimizing compilers can apply transformation as long as the program generates the same output as the initial program. The ACR extensions allow the compiler to relax the semantics

* Corresponding author: max.schmitt@unistra.fr

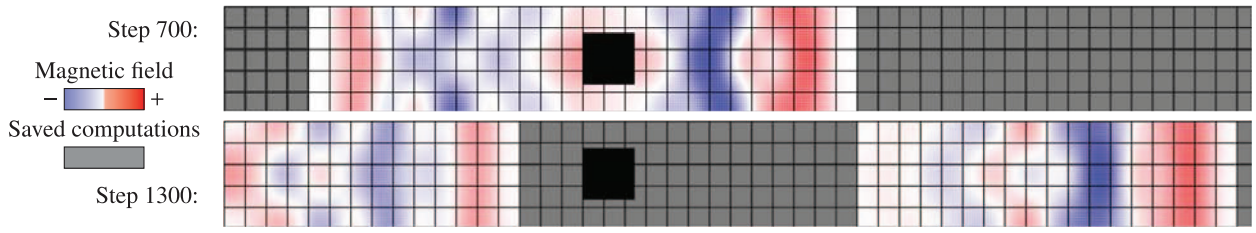


Fig. 1. Finite-difference time-domain magnetic field. Illustration of a 2D finite difference time domain simulation at two different time steps, 700 top and 1300 bottom. The magnetic field H is represented with a red and blue scale for the positive and the negative field values respectively, the white being neutral. An impermeable, square shaped, black object is present to reflect the waves. The grid overlay corresponds to the ACR cell decomposition. Each cell is composed of several points of the simulation domain and is transparent for the software developer. A grey cell is in an inactive state, *i.e.* the electric field computation in that cell is not done. The cell grid is dynamically updated at each simulation step and only the cells that have a constant magnetic field are disabled. The ACR compiler directives were used to express the cell size, how to identify a grey cell and what approximation to use when we are in a grey cell.

on chunks of the program where it is pertinent, without modifying the original algorithm itself. This results in an improved development workflow. The software developer can write and test his sequential or parallel algorithm implementation first, and then in a second phase use approximation techniques with ACR on the application's hot spots.

For example Figure 1 presents a visualization of a magnetic field during an ongoing simulation. The magnetic field is computed using the electric field. By observing the simulation data plot, we can notice that parts of the space have a uniform magnetic field, here represented as dark grey. This means that the electric field is constant and uniform in these areas. Software developers exploit this knowledge to accelerate their simulations manually by suppressing corresponding parts of the computation. Therefore, in this particular case, the electric field will not be updated inside these zones. Without ACR, the software developer will model his solver according to this knowledge and therefore, there will be a strong entanglement between the solver and its optimization strategy. With ACR, developers focus on the solver rather than architecture specific optimizations. Once the solver is finished and truthfully tested, developers can input the optimization information through specific ACR extensions. The original algorithm/program is not modified. In this example, the "alternative" will disable some targeted computations. The domain is partitioned with a regular grid and the alternative is active where the magnetic field is constant. The grid overlay in the figure corresponds to the ACR cell granularity. A cell is the data partition always sharing the same alternative. The code generated by the compiler controls the values of the magnetic field inside these cells to choose whenever the computation should be disabled or not. We show in Section 4 that implementing this dynamic optimization strategy may simply be a matter of specifying few ACR extensions.

Contributions: This paper introduces the ACR language extension set to enable approximate computing techniques by annotating compute-intensive parts of an application. We introduce these extensions in detail, and their effects to the application. We present the mathematical model on which we build our solution along with the code transformations and algorithms used to exploit our extensions. We introduce compile-time and run-time methods to achieve adaptive precision selection on explicitly targeted portions of a computation space. These methods are evaluated against a range of representative applications that support approximation, and are shown to allow for good performance improvement at the price of limited code modifications.

Outline: The remaining of the paper is organized as follows. Section 2 introduces the technical background and the theory used to construct our approximation model. Section 3 focuses on the language, the grammar and syntactic aspects of ACR. Section 4 provides an overview of the various categories of code that can be generated along with their construction algorithm. Section 5 demonstrates the expressiveness of our extension set and their effects on various example applications. Section 7 concludes and discusses ongoing work.

2 Background

In this paper we present a new set of extensions to use approximate computing in new or existing software. Extensions complement existing languages to provide additional information to compilers. Section 2.1 presents the use of libraries, extensions or constructions directly embedded into the languages to add semantic information to programs.

Approximate computing is a computation which may yield an inaccurate result rather than a guaranteed accurate one. Approximate computations can be used in cases where a possibly inaccurate result is sufficient for the application's purpose. Obtaining an approximate result may require less computation power than the accurate one. Therefore, approximate computing is used in constrained environments where reduced resource usage is required or valuable. Section 2.2 introduces approximate computing applicable environments, the available techniques to reduce the computation power of applications and methods to measure the accuracy of approximated versions.

In our work, we use the polyhedral representation of programs, a mathematical representation of programs that is used to apply transformations to the original code version and to generate the approximate code versions. Section 2.3 introduces the details of this representation

and demonstrates its advantages for program transformation and code generation.

2.1 Language extensions and specialized libraries

Programming languages are the main tools to create programs. The languages syntax and external libraries provide convenient abstractions to build or optimize programs. In this section, we review three ways to extend an existing language to support adaptive approximation: integrated language extensions, libraries and optional language extensions. We provide the advantages and drawbacks between the three approaches and motivate our choice for the optional language extension.

Integrated language extensions add new keywords or constructions inside the language grammar. The software developer may use these new constructions to provide additional semantics to variables or code blocks. The following table shows examples of additional constructions added to programming languages.

In the first example, the type system was enhanced with new keywords. A defined variable can be approximated [2] or accessing the variable be atomic with respect to the C11 extensions, *i.e.* accesses are safe between concurrent threads. The second example specifies a function reliability metric which states that the output precision for the function f must be at least as precise at 90% of its input parameter [3]. The last example provides additional control flow constructions where the number of iterations and stride of a for loop are determined by the compiler, at runtime, from an accuracy metric using information from a training dataset [4].

Specialized libraries are available in multiple domains and help developers by reuse of an existing code base. A library's interface consists in prototypes of functions that the developer may call. For specialized libraries, functions usually require an additional state parameter. This state contains the library internal problem representation, which is not accessible to the developer, and is carried between library function calls. To use this kind of library, the developer initializes the state, modifies the state using the library functions and recovers the information from the state using provided accessors. For example, in physics, there exists adaptive mesh refinement libraries [5] as well as specialized finite element libraries [6]. More generic libraries, *e.g.*, the Standard Template Library in C++, provide containers, algorithms, functions and iterators [7].

Optional language extensions are additions to the language that do not modify the language grammar. They can be expressed as annotations inside the comment section or with special constructions which will be ignored by the compiler when not recognized. These constructions use a dedicated syntax and can refer to information available inside the program source code, *e.g.*, identifier names and values. For example, OpenMP is a language extension for the C/C++/Fortran languages that provides parallelization and task generation extensions. *e.g.*, the following code does a parallel addition of all the values returned by a function and stores the result in the variable *a*. The first version

```
int reduceChunk (id,res){
  fst = id*tNum;
  lst = min(fst+tNum, N)
  res[id] = 0.;
  for(i=fst; i<lst; ++i)
    res[id] += f(i);
}
```

```
float a = 0.f;
float res[ceil(N/tNum)];
for(id=0; id<ceil(N/tNum);++id)
  thread(reduceChunk, id, res);
waitAllThreads();
for(id=0; id<ceil(N/tNum);++id)
  a += res[i];
```

Fig. 2. Parallel reduction of an array using a thread spawning library (right). Each thread is given an equal chunk of the initial array to reduce *via* the reduceChunk function (left). The master thread waits for each spawned threads to finish and finally adds the intermediary results together.

```
float a = 0.f;
#pragma omp parallel for reduction(+:a)
for (i = 0; i < N; ++i)
    a += f(i);
```

Fig. 3. Parallel reduction of an array using the OpenMP directives. "omp parallel" states that the following loop can safely be parallelized and the "reduction(+:a)" indicates that we want to do a reduction using the addition operator on the variable "*a*". The final result will be stored in the said variable.

in Figure 2 uses a thread spawning library to add every element of an array.

The second equivalent version in Figure 3 uses OpenMP directives instead of a library.

The OpenMP version is less verbose and allows for fast analysis of the main purpose of the algorithm while the version using the library contains more implementation details due to the library function interface. The parallel reduction algorithm is implicit for the OpenMP version. Internally, they may transparently use different optimized reduction strategies. Re-usability of existing libraries is not a problem when annotations are used as the compiler can issue a code targeting them. Hence, the code in Figure 3 may compile to the same code as in Figure 2.

Optional language extensions give flexibility to the developer. They can be ignored by compilers that do not implement the extensions or disabled at compile time, *e.g.* using a compilation flag. On the other hand, a missing library or unknown type/construct will throw a compilation error. Specialized libraries can reduce the maintainability, debugging and optimizations possibilities of complex programs, *e.g.*, interprocedural and code inlining optimizations are not possible for dynamic libraries. Language extensions are less invasive because code transformation is done automatically by the compiler. Optional extensions may be more verbose than integrated ones, but they share a common interface between many programming languages. Therefore, optional language extensions is the most suited abstraction for our adaptive approximation API based on the language agnostic polyhedral representation of programs.

2.2 Approximate computing techniques

The demand for computational power and storage capacity required to process data in multiple fields, *e.g.*, scientific computing, multimedia, social media, finance and health

care tends to grow faster than the resources that can be allocated to process them. The ratio of data traffic increase to computing power increase has been observed to be in the order of 3 in the early 2000's [8]. Therefore, approximate computing as well as approximate data storage becomes a particularly attractive solution to this ongoing phenomenon where the demand in data processing exceeds the capacity to process it. Gains proportional to the level of approximation can be expected with thorough selection of approximation strategies.

Approximate computing techniques are well suited for various kinds of applications [9]. Our language extension set is generic enough to target various application classes, including (see Sect. 5):

Noisy input systems. Storage and analysis of noisy data necessitate more resources to extract the relevant data. Each analog sensor is prone to interferences and comes with a specification of its capabilities and error margin. It is non-trivial to extract the signal from the background noise. Therefore, analysis of numerous noisy data streams becomes a challenge which can be addressed by using approximation techniques.

Error-resilient applications. In some cases, *e.g.* computer vision or sound synthesis, the program output quality is not impacted if we allow a small error margin. The quality of the result is bounded and approximation techniques can leverage the application's full potential while sticking to an acceptable output quality.

Iterative applications. Some commonly used algorithms of scientific simulations or machine learning use iterative methods in order to reduce the problem complexity. The runtime of those programs can greatly be reduced by allowing approximate portions of code inside each iteration.

Previous studies came up with various techniques to allow developers to be able to use approximation techniques in their programs. Most automatic techniques are based on error injection (*e.g.* use approximate algorithms, arithmetic operations and data representations) along with the monitoring of the output quality. Section 3.3 gives more details on various approximation techniques that can be used along with our language extensions to yield an approximate version of the program.

Whenever approximate computing is used, we also need the ability to measure the precision or quality of the output. There exists multiple quality metrics available in the literature. One of the most common metrics is the relative difference or error with respect to a standard output, *i.e.* obtained without the use of approximate techniques [10]. Application specific metrics are available, *e.g.* image and video processing have Peak-Signal-to-Noise-Ratio (PSNR) and Structural Similarity Index Measure (SSIM) [11], physics based simulations use energy conservation as a metric [12] and machine learning algorithms are validated against preprocessed datasets [13]. The developer should be able to select the best suited metrics for the problem at stake.

While the two previous points are crucial in order to use the right approximation methods and error metrics, the developers have yet to use these approximation techniques in their programs. Three main possibilities are available. Given that the developers are comfortable with approximation techniques, the first option is to write the application from the start with approximation in mind. This requires a high level of expertise of the application algorithms and in approximation theory. The second method uses domain specific libraries that already implement the approximation. With the last option, the developer provides code annotations for the compiler and lets the framework generate the approximate version automatically. In our work, we propose a new way relying on language extensions to enable a new compiler optimization based on approximate techniques.

2.3 Polyhedral representation of programs

The polyhedral model is an algebraic representation of programs where the points of a Z-polyhedron correspond to instances of a statement inside the application [14, 15]. Within this model, we can represent a wide variety of programs, *i.e.*, the model imposes some restrictions on loop bounds and data accesses: loop bounds, control statements and data accesses must be affine expressions of surrounding loop variables and constant parameters. In this work, we rely on the polyhedral model to transform, create new optimized versions and add annotations to produce approximate versions from the original code.

In Figure 4, the loop nest on the left can be represented in the polyhedral model whereas the one on the right cannot. In the left case, every loop bounds, data access expressions and control statements are affine functions. In the right case, the loop cannot be represented in the polyhedral model because the upper bound of the second loop is dependent on an array access and cannot be analysed statically by this model.

The polyhedral representation embodies the following information for each statement:

The iteration domain of the statement, which represents the number of times that the statement is executed. Here, each dimension represents a level of the loop nest. The domain is empty when the statement is not part of a loop nest.
The scheduling of the statement, which maps the statement instances to a time domain. Hence, it represents the order in which the various statement instances have to be executed with respect to each other.

```
for (i=0;i<=N;i++)
 for (j=i;j<=N;j++)
  if (i+j-P >= 0)
   A[i][j] = B[i+j-P];
```

```
for (i=0;i<=N;i++)
 for (j=0;j<=C[i];j++)
  A[i][j] = B[j];
```

Fig. 4. Loop nest example in C. The left loop can be raised to the polyhedral model because each data access and the control statement expression are affine functions. The right loop nest cannot be represented in this model because the upper loop bound of the j dimension is an array indirection dependent of i (C[i]). When considered alone, the j loop can be raised in the model because its upper bound can be treated as a constant parameter. In that case we miss the nesting information and reduce the set of potentially applicable code transformations.

The array access polyhedra, which are the read and write map functions from the iteration domain to array indices. They are used for dependency analysis, to assert that transformations made to the schedule are valid.

The polyhedral representation allows for code transformations through linear operations, *i.e.* reordering the statement instances with respect to each other in the schedule, and asserts the legality of these transformations with exact dependency analysis [14–16]. The polyhedral representation can be extracted from programming languages constructions [17–19]. Code generators can be used to translate it back to code [20, 21].

In the following of this section we give more details about the iteration domain, the scheduling, the code transformation, dependency analysis and code generation.

The iteration domain corresponds to the mapping between the point coordinates and the iterator values of each statement. This polyhedron is constructed using a set of inequalities control statements (bounds, ifs) of the loop nest. In Figure 4 left, there is only one statement, *i.e.* S_1: `{A[i][j] = B[i + j − P];}`, whose iteration domain is:

$$D_{S_1}(N,P) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \middle| \begin{matrix} 0 \le i \le N \\ i \le j \le N \\ P \le i + j \end{matrix} \right\}$$

We can easily recognize the two loop bounds and the guard expression inequations. The bounds depend on the two parameters *(N, P)* which are constant for the duration of the loop nest. The domain has two dimensions named *(i, j)* as in the initial loop nest.

The scheduling polyhedron is a function from the iteration space to a multidimensional time space. It defines a total order between the statement instances. The original loop scheduling function, which corresponds to the lexicographic ordering is the identity function. In our example, each instance of *(i, j)* is ordered lexicographically by construction of the for loop. Therefore, an instance precedes another one, $(i, j) < (i', j')$, if $(i < i') \vee (i = i' \wedge j < j')$. An identity scheduling function indicates that the instances execute in the same order as the dimension of the domain polyhedron.

Code transformations are linear functions that are used to modify the scheduling polyhedra of the statements. Transformations do not modify the number of instances but their relative order with respect to each other. The following example illustrates the matrix to perform a loop interchange for a two-dimensional loop nest with the dimensions i and j. The matrix represents the interchange loop transformation.

$$\text{Interchange} = \left\{ \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \right\}$$

Thanks to the linearity of the transformations, it is possible to compose them together to create more complex ones. It is nonetheless necessary to check that the transformation itself will not break any dependency of the data accessed by the loop.

Dependency analysis is one essential cornerstone of the model. It is used to verify transformation's legality and to assert that the new scheduling will not modify the program's behaviour with respect to the original one. Dependency analysis in the polyhedral model uses a dependency polyhedra to represent all the dependencies between two statements [14]. A dependency between two statements S and T exists if there is an instance $S(i)$ *happening before* $T(j)$, both accessing the same memory location [22]. Code transformation functions are applied to the dependency polyhedron which is formerly checked for the absence of backward dependency, *i.e.* a value from a statement is used before it has been computed. When there is no such backward dependency, the transformation is said to be legal.

Code generation translates the domain polyhedron and its associated optimized schedule to a programming language representation. The task is handled by a code generator, *e.g.* CLooG [20], that produces an imperative AST from the polyhedral representation. The new generated AST scans the iteration domain following the new schedule defined by the optimization phase. Code generation can also optimize the control overhead by lifting, whenever possible, the control statements outside the loop. Thus aspect is particularly important in our context (see Sect. 4.2). The code in Figure 5 is the result of a code generation pass on the Figure 4 original domain and schedule. The generated code does not include empty iterations contrary to the initial version. New loop bounds are used to ensure the nonemptiness of the iteration domain. Figure 6 pictures the iteration domain of the loop nests. The two lines represent the linear inequalities of the lower and upper loop bounds. The points that are present inside the cone pointing up are the valid instances. The generated loop in Figure 5 only visits the points inside the cone whereas the original loop in Figure 4 left visits points outside and uses the guard internal control statement to prevent the statements execution.

The code generation algorithm shifts the control statement of guarded empty iterations upwards, to the upper loop dimensions whenever possible, reducing the total loop overhead. This overhead could represent a significant part of the application's total computation time. Therefore, polyhedral code generation guarantees that the control overhead will be small even for complex polyhedra. Our solution benefits from the optimized loop scanning generated by the code generator to produce approximate versions with less overhead (see Sect. 4.2).

```
if ((N >= 0) && (2*N >= P))
  for (i = max(0,-N+P); i <= N; i++)
    for (j = max(i,-i+P); j <= N; j++)
      A[i][j] = B[i+j-P];
```

Fig. 5. Code generated from polyhedral representation**.** The code in Figure 4 generated back from the polyhedral model without code transformation. The code generator CLooG did lift the control statement to the upper loop nest dimensions with modifications to the loop bounds.

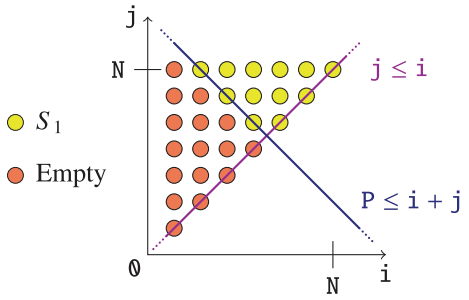


Fig. 6. Iteration domain graphical representation**.** Graphical representation of the iteration domain of the code in Figure 4. The original loop iterates over empty points because of the internal loop guard whereas the code generated in Figure 5 does not.

3 Adaptive Code Refinement

Compilers usually translate a high level program to a low level (optimized) program with special care in respecting the high level program semantics. In this work, we propose means to provide additional information to the compiler to allow it to relax the semantics through approximations in a controlled way.

In this section we present our language extension API to help developers to build applications exploiting approximate computing techniques. Our API can generate various approximate versions that can be ordered from the least approximative to the most approximative. We present a tool called "Adaptive Code Refinement" (ACR) which uses a directive set to generate approximate and adaptive versions of annotated compute kernels.

Our extensions define multiple constructions to help the developers willing to use approximate techniques during the software optimization phase. At the end of the optimization phase, the compiler produces multiple modified versions of the original code with the information provided by the developer through the extensions. We call these versions "strategies". A strategy is a composition of code transformations that are expressed as "alternatives", *i.e.* an alternative version of the original code. The final optimized kernel can be thought as a selection case between multiple modified versions. The compiler adds snippets of code to select the right version from information gathered from application data at runtime. We call this part "monitoring". The *strategy* selection can be achieved at various levels of "granularity". For example, it is possible to select one strategy for the whole loop duration, one strategy per iteration instance or an in- between, which selects a strategy for groups of iterations.

3.1 Granularity of approximations

Approximate computing techniques can be applied at different granularity levels: for the whole kernel, for each operation done or an in-between where we consider sets of operations. Our goal is to provide the compiler with the information it needs to use the right approximation at the right moment during the execution of the application. In order to take a decision, we need to gather information from the application. Therefore, we provide the "monitor" construct to gather information from the application data and the "grid" construct to select the granularity of the decision mechanism. Ideally, we want to select the best possible approximate version, *i.e.*, which guarantees that the output will stay within a given quality range.

3.1.1 Monitoring

The collect of information from the data is declared to the compiler in the form of a multidimensional array access within the ⟨*monitor-option*⟩ grammar rule. A simple example of the `monitor` extension format, as seen in Figure 7 is: `#pragma acr monitor(`*`data`*`[`$f(\vec{i})$`])`

The monitor's ⟨*array-access*⟩ grammar in Figure 8 declares an array where f (i) is the data access function. Our framework constrains the dimensions of the array specification to form an affine map between the iteration space and data space. Hence, they must be a linear function of the parameters, constants and iterators available inside the optimized kernel. The data domain does not necessarily need to match the loop's iteration domain, *e.g.* in Figure 7, the iteration and data domains do not match. Our system relies on polyhedral techniques to recover the iteration space from the data space by computing the map inverse or preimage [23]. The statement instances that access the array `A` in Figure 7 are pictured with the same color on the data and iteration domains.

3.1.2 Grid granularity

The granularity specifies the level at which *strategies* are selected. It corresponds to the ⟨*grid-option*⟩ grammar rule in Figure 8. With this "grid" construct, the data array is subdivided into *cells* using a tiling transformation [24, 25]. The transformation can be viewed as a function from $\mathbb{Z}^n \rightarrow \mathbb{Z}^{2n}$ where the first n dimensions are the tile index and the last n dimensions the data cells associated with that index. Hence, the technique subdivides the initial data domain into cells which can be selected individually. The size of the cell used for the tiling technique is selected with the grid construct.

A more generic meshing procedure using piecewise affine function could complement the tiling algorithm. Piecewise affine function allows for non-structured grids, *e.g.* a rectilinear or curvilinear grid, but usually requires a grid generator algorithm. For the time being we only consider structured Cartesian grids in our API. A tiling size of one is used whenever data share no similarities. The decision is then made per data basis.

3.2 Adaptivity specification

The ACR optimized kernel contains several modified versions of the initial kernel. With the monitoring and grid constructions defined before, we know where to extract the information but not how we should interpret them. In this section we introduce the ACR constructs which provide the decision mechanism.

Adaptivity means the capability to choose the precision dynamically at runtime depending on the state of the data.

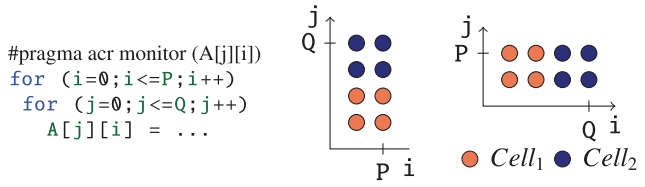


Fig. 7. A loop nest iteration and data domains**.** A loop nest and its accompanying iteration domain (middle) and data domain (right). The data domain is subdivided in two cells of four points each. The data and iteration points associated to a same cell are represented with the same color.

Several strategies are possible to select convenient approximate computing alternatives for given data cells. Two categories of strategies can be used. The static category specifies non-adaptive portion of the data that will unconditionally use a specified version of the code. The dynamic category uses different versions depending on the runtime context.

With a static strategy, the compiler can generate an optimized code at compile time with approximated computation embedded inside the kernel. The approximation remains the same during the whole application execution. The regions to be approximated are marked by the

⟨*pragma-defininion*⟩ ::= '`#pragma acr`' ⟨*acr-option*⟩

⟨*acr-option*⟩ ::= '`grid`' ⟨*grid-option*⟩
| '`monitor`' ⟨*monitor-option*⟩
| '`strategy`' ⟨*strategy-option*⟩
| '`alternative`' ⟨*alternative-option*⟩
| '`checker-select`' '`(`' ⟨*checker*⟩ '`,`' '`sync`' '`)`'
| '`checker-select`' '`(`' ⟨*checker*⟩ '`,`' '`async`' '`)`'

⟨*grid-option*⟩ ::= '`(`' ⟨*positive-integer-list*⟩ '`)`'
| '`none`'

⟨*monitor-option*⟩ ::= '`(`' ⟨*array-access*⟩ '`)`'
| '`(`' ⟨*array-access*⟩ '`,`' ⟨*folding-function*⟩ '`,`' ⟨*pre-processing*⟩ '`,`' ⟨*post-processing*⟩ '`)`'
| '`(`' ⟨*array-access*⟩ '`,`' ⟨*folding-function*⟩ '`,`' ⟨*pre-processing*⟩ '`,`' ⟨*post-processing*⟩ '`,`' ⟨*init-fold-val*⟩ '`)`'

⟨*folding-function*⟩ ::= '`min`' | '`max`' | '`mean`' | '`std-deriv`'
| ⟨*function-pointer*⟩

⟨*alternative-option*⟩ ::= ⟨*alternative-identifier*⟩ '`(`' '`parameter`' '`,`' ⟨*identifier*⟩ '`=`' ⟨*constant-value*⟩ '`)`'
| ⟨*alternative-identifier*⟩ '`(`' '`parameter`' '`,`' ⟨*identifier*⟩ '`=`' ⟨*identifier*⟩ '`)`'
| ⟨*alternative-identifier*⟩ '`(`' '`code`' '`,`' '`{`' ⟨*inline-code*⟩ '`}`' '`)`'
| ⟨*alternative-identifier*⟩ '`(`' '`code`' '`,`' ⟨*function-pointer*⟩ = ⟨*function-pointer*⟩ '`)`'
| ⟨*alternative-identifier*⟩ '`(`' '`zero-compute`' '`)`'
| ⟨*alternative-identifier*⟩ '`(`' '`interface-compute`' '`)`'
| ⟨*alternative-identifier*⟩ '`(`' '`step`' '`,`' ⟨*constant-value*⟩ '`)`'

⟨*strategy-option*⟩ ::= '`dynamic`' '`(`' ⟨*constant*⟩ '`,`' ⟨*alternative-identifier*⟩ '`)`'
| '`dynamic`' '`(`' ⟨*constant*⟩ '`,`' ⟨*constant*⟩ '`,`' ⟨*alternative-identifier*⟩ '`)`'
| '`static`' '`(`' '`global`' '`,`' ⟨*alternative-identifier*⟩ '`)`'
| '`static`' '`(`' ⟨*area*⟩ '`,`' ⟨*alternative-identifier*⟩ '`)`'

⟨*checker*⟩ ::= '`raw`'
| '`versioning`'
| '`stencil`'

Fig. 8. *Adaptive Code Refinement grammar.* The ACR grammar declares simple to use language extensions for approximate computing. The *alternative* defines a modification of the code to express various approximate blocks or statements. The *strategy* links these alternatives to a constant value for adaptive selection and applies them globally or restricted to an area when defined at compile time. The *monitor* serves as the link between the functions that pre-process and compute the strategy values at runtime and the data used to take a decision. The *grid* defines the granularity at which the decision is made. In the case where the granularity is more than one, the *folding function* is used to map multiple decision values into a single one for the whole cell.

developer with the ⟨*strategy-option*⟩ "`static`" construct. The approximation strategy can either be global, *i.e.*, covering the entire data space, or be defined on a localized area. In this case the area is specified with the set of iterator values corresponding to the desired data domain. We rely on widely used isl or Omega notations [26, 27]. For instance, to select half the data space of an array `data[i][j]`, where both `i` and `j` span from `0` to `N`, we may specify [N] → {[i,j] :0 <= i <= N/2 and 0 <= j <= N}.

With dynamic strategies, the approximation may vary at runtime depending on the state of the data. In ACR, this adaptive capability is inspired by adaptive mesh refinement (AMR) [28]. AMR technique defines a multi-level dynamically changing grid over the data domain. The deepest levels typically contain more data points than the outer levels and are located over active regions. This allows for a finer grained grid, *i.e.* more precise, only on the regions where it matters.

We propose a similar method to dynamically choose which approximation to use at a cell level granularity. Cell-level decisions implicitly require the retrieval of the most conservative level of approximation from the cell state during the program's execution. The components of each cell are aggregated together using the ⟨*folding-function*⟩ attribute of the *monitor* construct. The folding function defines a mapping from multiple data values to a natural integer (with prototype `int fold(datatype, int))`. This integer represents the maximum level of approximation that can be used while maintaining a good quality of result. The ⟨*preprocessing-function*⟩ is applied on every components of a cell before passing the value to the folding-function. The preprocessing allows an application specific datatype to be used along with the ACR predefined *folding functions.* If not required, the preprocessing function can be set as the identity function. The ⟨*init-fold-value*⟩ may be specified to initialize the value of the fold, otherwise a value of zero is used. This initial value allows for better flexibility with the folding function implementation and usage, *e.g.*, use of the `min` *folding function* with an initial value of zero will always return zero.

The compiler computes the affine mapping from the data space to the iteration space together with the transformation from the data space to a cell coordinate (see Sect. 3.1.1). Hence, the compiler can instrument the code to update the precision whenever a modification of the monitored data occurs. At the price of storage capacity, the compiler has the ability to select the adequate approximated code version to execute at a cell level granularity.

Adaptive Code Refinement does not allow the compiler to break data dependencies between the remaining iteration instances to limit the deviation of the results. When the loop carries a dependency between iterations, the potential schedules are constrained. In the context of ACR, this constrains the order in which the tiles are scanned. If automatic parallelization is applied, further dependency relaxations could unlock more parallelism at the price of larger deviations. For instance, Figure 10 presents the dependence graph that shows a dependency carried between the monitored cells. In this example, we propose two valid schedules with different properties. A close look at the first schedule and the dependence graph reveals that the computations could be parallelized within the diagonals (wavefront parallelism). This schedule, when diagonally parallelized depicts the maximum parallelism for this particular dependence graph. The second schedule, on the other hand, allows the *diagonal cells* to be parallelized. Hence, we trade a one level parallelism for nested parallelism with better data locality. The schedule which performs best on a given Central Processing Unit (CPU) depends on cache properties, memory bandwidth and data prefetching algorithm.

3.3 Approximate alternatives

Traditionally, a compiler must comply with the semantics of the input program and can by no means alter the precision of the result on its own. Therefore, the `alternative` extension provides, explicitly, the ability for the compiler to generate modified precision code. This extension allows alterations of the iteration domain, to partially or even totally replace the original computations. We only consider programs that run on a general purpose processor and do not take into account specialized hardware, *e.g.*, approximate arithmetical and logical unit or memory storage which can be used to reduce energy consumption, memory access time and computation time or resource sharing [3, 29, 30]. This specialized hardware is much less common than general purpose processors while our purpose is to provide language extensions for a wide range of problems and target platforms. The approximate strategies to perform on general purpose CPUs are listed below.

Loop perforation consists in the deletion of some loop iterations or by halting the loop prematurely. It efficiently drops computations but does it evenly or randomly over the loop nest [31]. The technique uses profiling data from previous execution of the kernel to select the perforation stride. While it requires training, this technique does not have any measurable runtime overhead as the loop perforation is done at compile time.

Memory access skipping removes superfluous data accesses [31–33]. The non-accessed values can either be interpolated using neighboring values or speculated/predicted using data regression algorithms. Accessing data from main memory may be much slower than computing a new one. Applications that process massive amount of data will reduce the CPU to RAM bus contention and distributed applications can use this technique to avoid the network transfers.

Task skipping avoids the execution of some tasks to save computation at the price of precision [33]. This technique is similar to loop perforation but applied at a task level. This technique requires the task output datatype to match its input datatype, otherwise, the next task in line will receive an erroneous input. Task based programming usually tends to split a program into tasks and construct a task graph. The tasks are represented as nodes in the graph and edges stand for the data flow between the tasks. After some graph analysis, it is possible to distribute the concurrent tasks between multiple processors or computers. Removing tasks from the graph can lead to better parallelism and reduced resource utilisation.

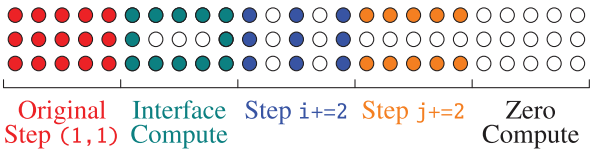


Fig. 9. Graphical representation of a two-dimensional domain with several alternatives applied. The domain consists of five cells delimited by the black-dashed line. Each cell consists of fifteen iteration instances represented as circles. An iteration is evaluated if the corresponding circle is filled and not executed otherwise. With `interface compute` only the instructions at the borders are kept. `Zero compute` disables them all. `Step` modifies the step count for the first or second dimension depending on the position of the annotation.

Memoization retains information of previously computed values of pure functions to skip the following same function call [34]. A pure function always returns the same value for given argument values and is free of side effects. To implement this mechanism, the compiler adds a lookup on the input values and returns the stored result if it has already been calculated, or computes and stores a new value for potential future use. This lookup is usually achieved with the help of a hash table [35]. Determining the number of return values to store, the retention policy and the lookup access time are critical for the usage of this solution. To be profitable, the function's execution time must be a magnitude higher than the arguments lookup and the function has to be called many times with the same values.

Function multi-versioning can be used to fine tune the precision of the result [4]. The developer could implement multiple algorithms to solve the same problem statement or write a version using less precise data representation (*e.g.* double to float). The implementation can also reuse the previously mentioned techniques. The only restriction of this technique is that the different function's versions must share the same prototype. Some specialized languages propose function multi-versioning in the form of data-flow oriented languages [4, 36].

In ACR, the dedicated extensions to specify a collection of means to achieve approximate computing is the *alternative* construct. Multiple approximate versions can be defined for the same portion of code. In the case of a static strategy, the compiler will be in charge of executing the appropriate version depending on the statically specified data space. In the case of a dynamic strategy, it will use the appropriate version depending on each grid value at runtime.

The *alternative* construct is described in Figure 8 with the ⟨*alternative-option*⟩ grammar rule. A simplified notation is: `#pragma acr alternative` *name(type, effect)*

The first parameter provides a name to the alternative using C identifier notation. Other constructs can reference this name to select a particular code alternative. This name is used to link with the strategy selection mechanism. The second parameter, "`type`", indicates which approximate strategy is used to generate the alternative code. To define an alternative, the developer may choose between the five following options[1]:

With "*parameter*", the constant parameter used inside the target loop nest will be replaced with the value of another parameter or a constant value specified by the developer. Changing a parameter value can result in loop perforation and task skipping. It is also possible to write boolean conditional code blocks to be executed only when a condition holds and set the parameter to true or false. This condition will be optimized by the compiler for the different versions with the boolean value computed by the *monitor* construct.
With "*Code*", the alternative is defined as an external function or code block. The developer can implement multiple algorithms for a given problem and name them under different alternatives. This provides the flexibility of code multi-versioning while leaving the compiler with the care of generating the runtime mechanism. Memory access skipping can be implemented by replacing a memory access instruction with an interpolating function.
With "*zero_compute*", the target computation is removed and will not be executed. This construct can be used to implement loop perforation and task skipping at the cell granularity.
With "*interface_compute*", the points of the kernel's iteration domain that do not intersect with the cell mesh are not visited, *i.e.* only the iterations at the border of a cell are visited. This allows stencil-like computation to be implemented in a more natural fashion (see Sect. 5.3).
With "*step*", the specified loop counter increment is modified according to the specified parameter (*e.g.* `i++` is replaced with `i+=step`). This construct can be used to implement loop perforation in a particular case of *inter-face_compute*.

Figure 9 shows a graphical representation of the last three aforementioned alternatives. These alternatives selectively reduce the amount of iterations inside the cells, *i.e.* the number and position of the empty circles. The "code" alternative modifies the code executed for the remaining colored instances of the cells. The 'parameter' alternative applies a different constant parameter value globally.

All the alternatives besides *parameter* can be defined on a particular statement or code block. By positioning the code annotation at the right place inside the kernel, the developer can selectively apply the alternatives. The code alternatives can be individually linked to one or multiple integer values using the *strategy* "dynamic" parameter. Affecting an integer value to strategies allows to order them from the most to the least precise and let the runtime select the most appropriate one dynamically.

It is possible for a unique strategy to compose multiple alternatives together whenever the following conditions are met:

The "parameter" alternative can only be positioned at the kernel top-level, because parameters are constants for the kernel's duration.

[1] This list may be extended in future versions of ACR.

The "code", "zero_compute", "interface_compute" and "step" alternatives cannot be nested when linked to the same strategy.

3.4 Approximation debugging information

Program debugging and patching is usually a demanding task that occupies a fair amount of the developer's time. We propose a workflow separated into two phases to help the debugging aspect of applications where the quality of results can be tuned. During the first phase, the developer writes her/his program as if she/he has an infinite amount of processing power at her/his disposal. Such a program would probably not scale well with a bigger problem size. But, the development is concentrated towards solving the main algorithm rather than architectural or approximation specific algorithm choices. Throughout this phase, the debugging is performed using ordinary debuggers and memory analysing tools. At the end of the first phase, the program should be ready for production but not yet optimized.

The second phase is dedicated to optimization and approximation of the program's hot spots, *i.e.* where most of the program's time is spent. The program will be annotated using our language extension set without having to modify the main algorithm. Selecting the appropriate approximation strategy is not an easy task but thanks to our language extensions, the user can investigate multiple strategies in a small amount of time.

To help with the evaluation of an implemented approximation strategy, we expose multiple metrics to the developer. The compiler introduces a per-thread storage for each optimized kernel and allows the user to query, *e.g.*, the following information:

- Total time spent in the optimized kernel.
- Time spent on each defined strategy.
- Number of times a strategy is selected.
- Time spent outside of this kernel between two calls.

These collected metrics can guide the developer in tuning her/his alternative choice and fine tune the decision criterion in the case of an adaptive approximation. The user could provide a function to evaluate the result's quality at the cell level. A compilation flag can serve as a switch to turn on the call duplication quality checks. In this case, the cell values are computed with the original and the approximate code version sequentially. The two results are then fed in the user evaluation function and the approximate data discarded. This online checking method provides insight into the deviation value at the cell granularity for one kernel call. This is especially suited when the application may call the optimized kernel multiple times.

Developers can also use existing quality checkers which compute the difference a posteriori [11]. Comparing the final result of both the optimized and the original application is also simplified with ACR. Indeed, it is straightforward with our API to disable parts or totality of the alternatives by switching off the ACR extension interpretation, or commenting some extensions.

4 Compiler reference implementation

In the following section we describe the backbone compiler algorithms to monitor the data and to generate the runtime to execute strategy code snippets [37]. The first section focuses on the code generation without runtime. The second section presents our dynamic approach where the code to be executed is generated at runtime using just-in-time (JIT) compilation: the program produces specialized approximated versions "on the fly" as the data and their corresponding need for precision evolves.

4.1 Static code generation

The first step of the compilation process raises the code to optimize to the polyhedral representation (see Sect. 2.3). Clan and pet [38, 39] are state-of-the-art front-end tools that transform a high level language into an equivalent polyhedral representation. Clan is a standalone application and library and pet is a library integrated into clang, the `C/C++` LLVM compiler suite's front end. The output format consists of information on parameters and on every statement present in the kernel. Each statement $S_i \in S$, where S is the set of kernel statements, it holds the following associated information gathered from the input program (see Sect. 2.3):

- The context polyhedron, which is the information gathered about the constant parameters.
- The iteration domain.
- The scheduling.
- The array access polyhedra.
- The statement itself, which is stored in an AST or string format. This is the part that will be copied back in place once the loop control has been generated.

During the compiler front-end execution, the language extensions are included in the program's abstract syntax tree. The statements in relation with an alternative are marked using a depth-first AST traversal. This is done with a recursive function managing a state that contains the active alternative set. Every time an alternative node is encountered, it is added to the state and will be removed before the function returns. When a node representing a statement S_i is visited, it is marked with the alternatives present in the state along with the strategies, alternatives, grid and monitoring information into a data structure for future processing.

The computation is altered according to the user's defined alternatives. Algorithm 1 describes how the statements are modified according to the alternative types. The function `ApplyAlt` takes a statement S_i along with an alternative definition and generates a new statement corresponding to the user alternative definition. The first two cases of the function match the parameter modification. If the user defined a constant value, the parameter value is set to the new value inside the context polyhedron. Otherwise, only the name of the parameter will be altered. The change of the code statement is handled by a replacement

of the original code in the compiler internal statement data structure. For *zero-compute*, the iteration domain is set to an empty domain, *i.e.* a statement that will never be executed. To generate the grid borders, we add a new dimension D^n without any constraints and set the dimensions D^j to be a multiple of each point $a \in \mathrm{D}^n$ times the `GridSize`. We project the newly created dimension D^n out and what remains in D^j is the multiple of the grid size. The operation is repeated with `GridSize` $-$ `1` to capture the left border of the cells. The "step" is handled in the same way as 'interface-compute' but with a multiple of the step instead of the grid size. The second function, `ApplyAllAlt`, iterates over all alternatives linked to a strategy and applies the alternatives in the order in which they are defined.

The following helper functions are used by the code generation algorithm. These functions are implementations of regular search algorithms and linear algebra functions found in the literature [40].

`DimsRelatedToData` takes a domain and returns the set of dimension identifier in relation to the data. This is done by application of the inverse of the monitor array access matrix.

`AddDimEq` takes a domain D, a dimension identifier D^j and a parametric constant C. It adds a dimension D^n to D and the constraint $D^j = C \times D^n$ to D ($\mathbb{Z}^n \rightarrow \mathbb{Z}^{n+1}$).

`ProjectOut` takes a domain and set of dimension identifiers D^{id} and projects these dimensions out ($\mathbb{Z}^n \rightarrow \mathbb{Z}^{n-|D^{id}|}$)

`StrategyAlts` takes a strategy and returns the alternatives related to this strategy.

Algorithm 1: Alternative application algorithm: Algorithm to apply the different alternatives to a statement S_i. The parameter alternatives are handled inside the context. The statement code, which is stored as a string literal or AST, is replaced if needed. *Zero* and *interface-compute* alternatives do modifications on the iteration domain for the dimensions in relation to the monitored data map function.

Algorithm 2 which generates the alternative statements operates in three phases:

The first phase iterates through the kernel statements $S_i \in S$ and generates two statements from each original statement. The statement $S_{i'}$ contains the static-defined strategy iteration domains mapped from the data space. The statement $S_{i'}$ domain corresponds to the remainder of the initial statement's domain without these static areas. The alternatives are applied to $S_{i'}$ and the monitoring code is inserted to monitor the data accesses in order to compute the maximum level of approximation. This statement is stored in a separate set for future use. The second statement, $S_{i''}$, is stored in a statement set used in the second phase. It is worth noting that a statement with an empty domain will not generate any code and can be removed at any point of this algorithm.

The second phase handles the dynamic strategies. For the dynamic version, the goal is to generate a kernel that scans the monitored cells one after the other. This allows for a code with low control overhead, *e.g.*, the second schedule present in Figure 10. The first operation applies a tiling transformation to the domain with respect to the dimensions used by the monitoring. A new statement is constructed from this tiled domain by projecting out the cell dimensions of the domain. Only one point per cell remains after the projection. This domain is used to generate a guard in front of each alternative code version. An alternative code is comparable to a new statement modified by all its associated alternatives. This new statement is scheduled immediately after the guard statement of the corresponding alternative. The `AddMonitoring` function ensures that all writes to the monitored data will be followed by a folding function call to update the maximum approximation monitor value. The monitor value is initialized by the guard statement before entering each cell. Once all the strategy versions along with their guards have been generated, a default guard is inserted to fall back to the unmodified statement which is executed by default.

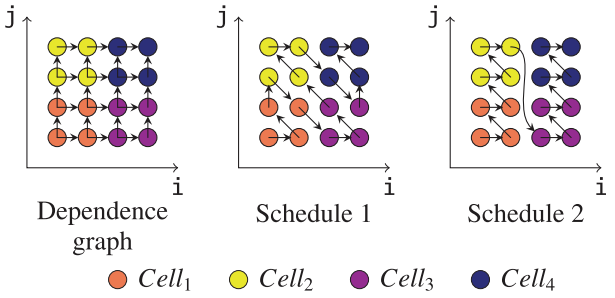


Fig. 10. *Loop Dependency.* Example of loop carried dependency between iteration instances with two valid schedules. A point represents the execution of one statement instance. For the dependence graph, an arrow represents a dependence: some instances have to be executed before the target instances. For the schedules, the arrows represent the execution order. Schedule 1 exhibits the maximum parallelism at iteration level whereas schedule 2 focuses on parallelism at the cell level. The code generated with the first schedule requires more cell-transition checks than it would with the second one.

The last phase of the algorithm deals with the static global strategies that must be enforced for the whole duration of the program. These alternatives must be applied to the previously generated statements, *i.e.* the union of the static area and the dynamic related statements. At the end of this last phase, the statement set is composed of all the polyhedral representation of the static alternative, dynamic alternatives with guards, the default code path, all with the global strategy enforced. Polyhedral code generation finally generates the actual code to be compiled.

Algorithm 2 is accompanied by the following set of helper functions:

`StaticGlobal`, `StaticArea`, `Dynamic` respectively return the set of static global, area and dynamic strategies.

`UniverseDomain` takes a domain $S \in \mathbb{Z}^n$ and returns a domain with the same number of dimensions that has no constraints, *i.e.*, an infinite polyhedron.

`TileDimensions` takes a domain D, a set of dimensions identifier D^{id} and a constant l and applies a tiling transformation on these dimensions.

`GenAlternativeGuard` takes a statement S_i and a strategy Strat$i$ and generates the guard statement that enters the branch if the alternative value matches the one of the folding function of the monitoring process.

`LowerTileDims` takes a domain D and a set of dimension identifier D^j used for tiling and returns the set of dimension identifiers of the tile itself and the one it encloses.

`AddMonitoring` adds the monitoring fold function to all access to the monitored array inside the statement.

`AddMonitorReset` adds the code to initialize the maximum approximation value. If not provided by the developer, defaults to lowest approximation value (original code).

4.2 Dynamic code generation

The static code generation algorithm is well suited for kernels that can be generated using the tiling process. However, dependencies can forbid the use of a tiling transformation and thus the use of static code generation. A small grid size can also cause extensive control overhead such as in schedule 1 of Figure 10, making the static code impractical. To remedy to these problems, we designed a dynamic method that generates a specialized approximate code at runtime. It is based on a runtime library that exploits cell monitoring information and polyhedral code generation techniques to generate a specialized version of the code with the lowest possible control overhead. For application which have strongly correlated data locality, the dynamic approach can reduce the number of guards tremendously as cells neighbors may more likely share the same maximum approximation level.

Algorithm 2: Code generation algorithm: Given a set of statements and strategies, this function returns a new set of statements corresponding to the annotated strategy versions with alternatives applied.

The code generation algorithm is briefly depicted in Figure 11. The domain to be translated is similar to the one in Figure 10 and contains four cells. In this example, during the execution of the application, three of those cells situated on the bottom and left part of the domain allow a maximum approximation corresponding to the alternative A_0 and the remaining one to the alternative A_1. There is no use for extra control overhead to be generated whenever the flow of execution goes from one cell to another one sharing the same alternative. Hence, the iteration domains of the cells that share the same alternative are merged together and treated as the same entity. This step is done with the information gathered by monitoring the data before the code generation algorithm is executed.

The code generation algorithm takes as argument a set of statements S in their polyhedral representation and outputs an AST. For dynamic code generation, each statement potentially represents a different alternative. The iteration domain for an alternative is computed as the union of domains of all the cells monitored with the same alternative value $D^{\text{Alt}_i} = \bigcup_{\text{Cell} \in \text{Alt}_i} D^{\text{Cell}}$. The loop generation algorithm acts dimension by dimension recursively. It starts with the outermost dimension where a projection is applied. This projection delimits the contiguous part of the domain in this dimension where the same statement is present. The projection over the 'i' dimension in Figure 11 results in two domains, one containing only the statement for the alternative A_0 and the other which contains both A_0 and A_1. In order to avoid a guard inside the loop nest to guard the execution of a given alternative, the loop is split in two parts and the algorithm used recursively on the two remaining polyhedra for the second dimension. The final result is an AST with a low control overhead compared to an equivalent compile time generated code which would contain many inside loop guards.

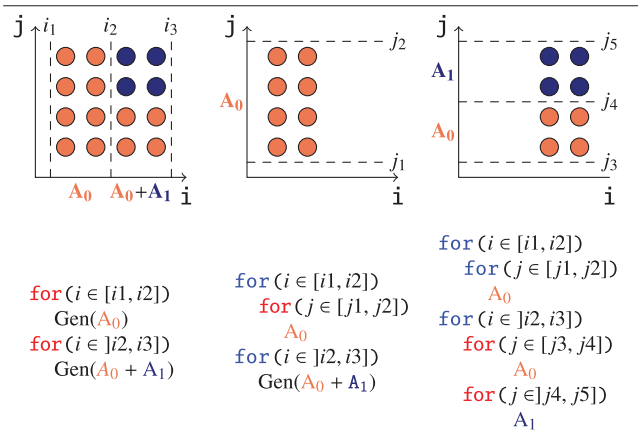


Fig. 11. *From polyhedral to code.* CLooG polyhedral code generation algorithm applied on the union of two alternative statement domains. The dimensions are processed one after the other recursively. The first dimension to be generated is the outermost loop (left). Then the second loop dimension is generated on the last two domains (center and right). The algorithm stops when there is no remaining domain to generate.

With dynamic code generation the iteration domain is split per alternative. Figure 12 stage 3 shows such partitioning, where the boxes match with the alternative's disjoint domains. The code generator will interleave the iteration instances of the various alternatives following the original code's schedule while keeping a low control overhead.

An illustration of the code generated for the static and dynamic methods is shown in Figure 13. The original version of the code (a) calls a function at each iteration of a three dimensional loop nest. Each white points of the iteration domain maps to a triplet values (i, j, k) passed to the function. The monitoring operates on 2×2 cells, on a data array related to the i and j dimensions. The cells are illustrated by the square tiling projected at the bottom of the domain. For this example we consider the following alternatives: the white cell is using the original kernel, the black cell is using the "`zero-compute`" alternative and the grey cells use the alternative "`parameter`" to set the number of iterations of the outermost dimension to one. The "*guarded*" code version (b) is a valid implementation of our extensions that, for each iteration instance, checks against the cell maximum approximation value collected by the monitoring. In this naive implementation the control overhead is high because of the internal guard. Each dark

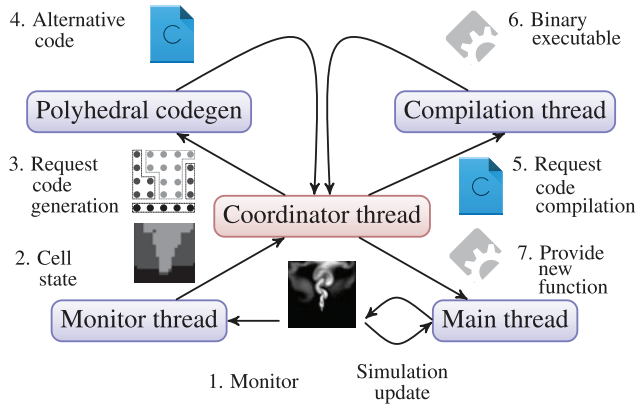


Fig. 12. ACR dynamic runtime optimizer**.** Dynamic runtime used to generate optimized strategies application at runtime.

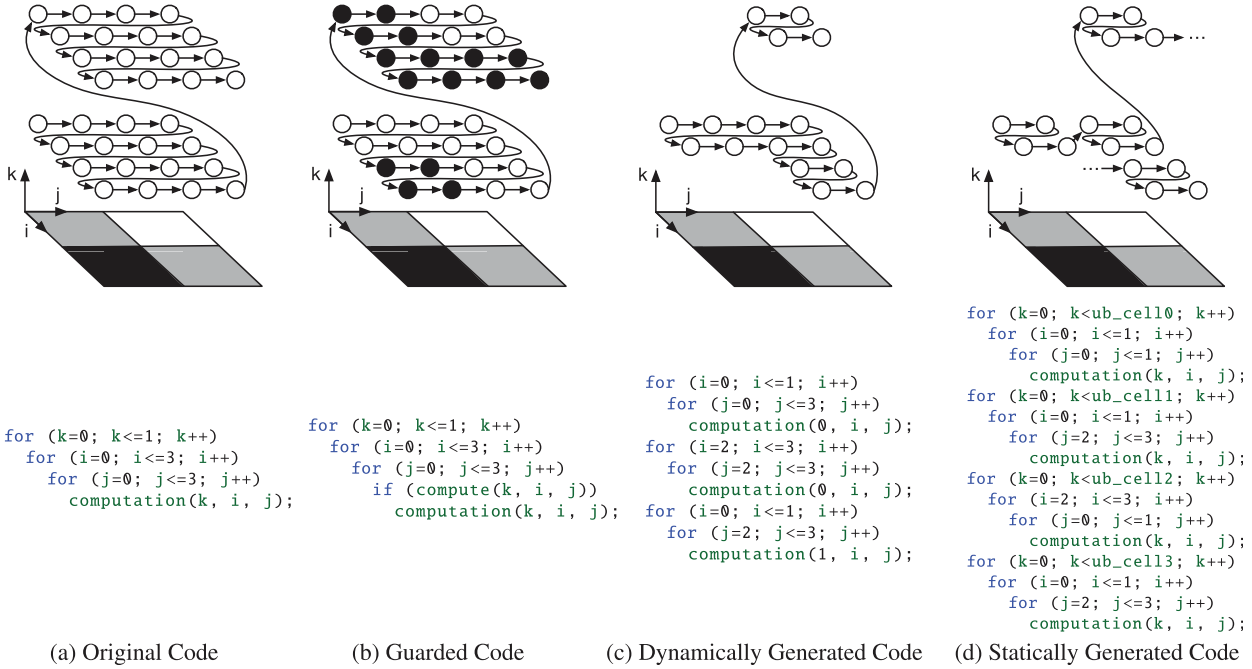
dot of the domain corresponds to the execution of a test without useful computation. On the other hand, the dynamic version (c) has the least control overhead as the runtime generated code matches exactly the state of the program data. The last version (d) is the statically generated code for whom the cells are entirely visited in series. This version only requires four checks, here represented by the k dimension upper bound, to select the number of iterations to do in the outer most loop from zero to one. Because the iteration ordering in the statically generated version is different from the original code, it is not always possible because of data dependencies.

The dynamic method requires a just-in-time compiler to be able to generate a specialized version of the code during the execution of the program. We choose to implement the runtime as a shared library that can be loaded whenever the program encounters a dynamic optimized section. At compile time, the compiler adds a call back to the library functions responsible for the compilation management. In Figure 12 we disclose the implemented architecture. The runtime is separated in four distinct functional units:

The monitoring thread constantly checks the current state of the data to select the most appropriate approximation strategy for each cell. Once a kernel has been fully executed, it provides the list of all the states (*i.e.* the suggested strategy identifier for each cell) to the coordinator thread.

The polyhedral code generation thread generates on demand an AST from the polyhedral representation given by the coordinator thread. Each generated statement corresponds to an alternative.

The compilation thread is responsible for generating an executable from the AST that has been generated from the polyhedral code generation unit. We choose to implement a compilation with two separate parallel paths. A fast lane using Tiny C Compiler (TCC) that has a fast JIT but has low code optimization capabilities and a second lane with the GCC compiler which is an order of 10–50 times slower but applies efficient code optimizations.



(a) Original Code (b) Guarded Code (c) Dynamically Generated Code (d) Statically Generated Code

Fig. 13. Code Generation Approaches. This figure presents the result of different code generation approaches for the original situation depicted in part (a). The top of the figure shows the code's iteration domain, where white points are executed while black points are not. Arrows represent the execution ordering. Under the iteration domain is a representation of the grid state. The approximation strategy is the following: in the black grid cell, no iteration is executed, in the grey grid cells, one iteration of the k-loop is executed and on the white grid cell, two iterations of the k-loop are executed. Parts (b), (c) and (d) show how the various approaches implement equivalent versions of the approximated code: with an internal guard for *guarded*, perfectly matching the situation for *dynamically* or on a grid-cell basis for *statically*.

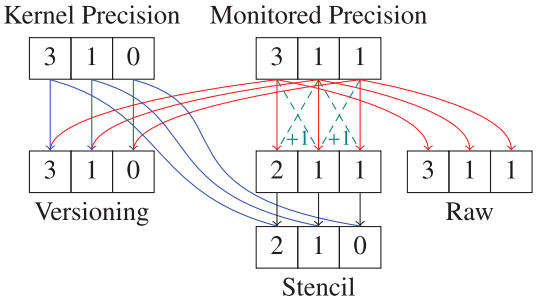


Fig. 14. Precision computation for the raw, versioning and stencil validity checks. The domain consists of three monitored cells. The precision of the optimized kernel is also visible. A vertex from one cell to another represents a value dependency. If multiple vertices enter a same destination cell, the lowest of the parent's values is chosen. The stencil check requires an intermediate step because the neighbor's values are also considered to compute a cell precision. The value from the neighbors is incremented by one (green dashed lines) before they enter the intermediate cell. This increment limits the influence of the neighbors when the stencil check is used.

The coordinator thread is the central part which orchestrates the other threads. It has the responsibility to check whether a specialized code version that is used for the kernel is still valid from the current cell state obtained from the monitoring thread. The validity can be expressed in three ways:

1. *Raw checking*: For each cell, compare the maximum approximate level, *i.e.* dynamic strategy number, used in the optimized kernel to the value obtained by the monitoring thread. If the two values do not match to the same alternative, the check fails.
2. *Versioning checking*: The same as "raw" checking but accepting monitored values that require less precise computation than the one the kernel is using. Reject cell values that require more precise versions than the one currently used by the kernel. If the percentage of values in an over-precise state exceeds a threshold, a new version is generated.
3. *Stencil checking*: The last checking is similar to "versioning" but it also considers that the neighbors of a cell can influence the precision required for a given cell. It sets the cell strategy value to the minimum approximation alternative possible between what the cell requires and the immediately greater approximation level than the most precise of its neighbors, for each cell, before applying the "versioning" checking algorithm.

The coordinator thread will use one of the three predefined validity checks before requesting a new version generation. An example of a validity check computation is shown in Figure 14. If the check fails, the monitor generates the alternative statement domain from the monitored data and sends a request to the polyhedral code generation thread. Once the AST has been generated it is directly forwarded to the compilation thread. The coordinator waits for the binary code from the compilation thread to modify the function pointer in the main executable to the newly optimized version instead of the previous one.

The coordinator thread may be configured in two modes, synchronous or asynchronous checking. Synchronous checking only allows the main application to ask for a new optimized version whenever the version is checked. If the coordinator requested a new version and the resulting binary code is not ready yet, the main application will be stalled in the mean time. This mode of operation is preferred to limit deviation of the output result. The asynchronous mode allows the application to run in parallel, with a non-appropriate approximation strategy, while the new optimized version is computed. If the main application arrives at the kernel call site before the new version is fully compiled, the code falls back to the original code version instead. Asynchronous mode may allow a less precise version to run temporarily while the monitoring and coordinator are processing the data cells. The original version can be used whenever no optimized version is available, allowing for the compilation to run concurrently to the slower but more precise version. To optimize the time required to generate new specialized code versions, the coordinator could speculatively generate new versions that could be called instead of the original version for the non-"raw" policies. The coordinator can also maintain a buffer of previously generated versions and use least-recently-used replacement policy whenever the buffer is full. Our runtime implements all these mechanisms. The user may select them through compilation option or through the ACR's 'checker-select' extension depending on the needs (speed *vs.* precision).

4.3 Compilation with polyhedral optimisations

One of the many advantages procured by the polyhedral representation of programs is the robustness of dependency analysis. A property of this model is to handle complete and exact dependency information [14]. Computing the dependency polyhedron requires solving a Diophantine equation that runs in the worst case in exponential time. Hence, this method is mostly reserved for complex dependency analysis as required when optimizing loop nests [22].

Compilation tools have been developed to automatically optimize loop nests [14, 41, 42]. Those tools search for a solution to an algebra problem. For example, find a schedule which minimizes the distance of reuse of written variables, *i.e.* optimizing for locality, or to push dependency on the lowest level of the loop, *i.e.* optimizing for parallelism. Such algorithm has been implemented in tools like PLUTO [41] or Traco [43].

These loop optimization techniques are orthogonal to ACR. They search for a different schedule for the loop nest that exhibits certain characteristics useful for performance. Our method focuses on slicing the input domain by alternatives and ensuring that the appropriate alternative is used. Hence, the automatic optimization tools can be used in a preprocessing stage to obtain the optimized schedule. In ACR, the original lexicographical schedule would be replaced by the optimized one before the alternative code

generation stage. The generated code could then exhibit parallelism or better locality. Ongoing work aims at mixing polyhedral optimizations with our ACR optimization engine.

5 Experimental study

This paper presents a new method to express static or adaptive approximation strategy to the compiler with the help of the ACR language extensions. Developers specify the approximation information with the *alternatives* construct and use the strategies to select the appropriate situation to use them. The dynamic alternative selection is tightly coupled to the *monitoring* and *grid* constructs for the definition of the cell grid. ACR's API is designed for quick testing of many approximation techniques.

In this section we put this API to work on various example applications. We detail the procedure to follow in order to find the alternatives and their applications on compute intensive kernels through the use of ACR's API. We also compare the final implementation with respect to the use of library or hand tuning. The following applications are representing several fields, from numerical simulations and cellular automaton to signal processing and data mining. We especially selected a broad range of algorithms to demonstrate the expressiveness of our language extension set in many situations.

5.1 Eulerian fluid simulation

We first evaluate ACR on a fluid simulation solving the Navier-Stokes equations [44]. It uses a non-conservative algorithm presented by Stam for the video-games industry [45]. He proposes an iterative solver to reproduce smoke or flames in real time for scenes rendering. Thus, the solver must be as fast and lightweight as possible. The quality of the output should also be accurate enough as too much approximation would lower the perceived resolution. We empirically observed that less than 5% of deviation of the result is not visually perceptible. Important portions of the simulation are empty of fluid until an advanced time in the simulation. A convenient approximation strategy is to monitor the values of the density and lower the number of iterations of the solver in parts of the simulation where the density is close to zero.

In this simulation, only two variables are used, the fluid density ρ and the fluid velocity v. The algorithm is divided in two phases. The first phase, the density movement, updates the grid fluid density ρ to the next time step depending on the density present in neighboring cells, computing the diffusion. The second phase, the streaming, moves the density and computes the velocity in the next time step with respect to the velocity vector. Using runtime analysis of the application, we discovered that approximately 60% of the runtime is spent in the first phase and the remaining in the second phase. Inside these two phases, there is one function where most of the computation time of the application is spent. This function is sketched in

```
static inline unsigned char density_val(float a) {
  if (a < 0.1f) return 2;
  if (a < 2.f) return 1;
  return 0;
}

void lin_solve (int M, int N,
    float x[M][N],
    float x0[M][N],
    float a, float c) {

#pragma acr grid(50)
#pragma acr monitor(x[i][j], min, density_val)
#pragma acr alternative high(parameter, P = 6)
#pragma acr alternative medium(parameter, P = 4)
#pragma acr alternative low(parameter, P = 1)
#pragma acr strategy dynamic(0, high)
#pragma acr strategy dynamic(1, medium)
#pragma acr strategy dynamic(2, low)
#pragma acr checker-select(versioning,async)
  for (int k=0 ; k < P ; k++ ) {
    for (int i = 1; i < M-1; ++i)
      for (int j = 1; j < N-1; ++j)
        x[i][j] =
          ( x0[i][j] +
            a * (x[i-1][ j ] +
                 x[i+1][ j ] +
                 x[ i ][j-1] +
                 x[ i ][j+1])
          ) / c;
    set_bnd(M, N, x);
  }
}
```

Fig. 15. A Gauss-Seidel iterative solver of linear equations. Three levels of approximation are used: low, medium and high which map to modified number of iterations of the solver. The number of iterations of the solver is lowered by the runtime in the cells where the density is low or medium.

Figure 15. It is a numerical solver using Gauss-Seidel relaxation iterative method to solve a linear equation.

Observation of the simulation data reveals that, for many scenarios, the smoke forms a funnel shape with a mushroom-like form at the top. These scenarios coincide with a source of fluid having a force applied at the base to simulate burning objects or particles moving with the air flow. The solver is used in the diffusion and streaming steps of the algorithm. Inside the collision, the parameter "a" of the function is set to the diffusion value of the fluid and inside the streaming step, "a" is set as the viscosity of the fluid. Therefore, if the updated value and its neighbors are roughly the same, the tolerance factor of the iterative solver should be reached faster. Hence, the number of iterations of the solver can in this case be lowered. The initial number of iterations is set to a value that is big enough for any kind of simulation. This high value reduces the overall speed of the simulation and can be improved.

Following our strategy to reduce the number of iterations of the iterative solver, the first extensions to specify are the code alternatives. The lines 14 to 16 of Figure 15 define the alternatives that set the number of iterations of the solver to 1, 4 or 6, which is used whenever there is a really low, medium or a high density of fluid respectively.

The next implementation step of the ACR extensions is to declare in which case the alternatives are used. We define three dynamic strategies lines 17–19 that orders the alternatives relatively to their level of approximation, fewer iterations of the solver is equivalent to a less precise solver. To select the right strategy at runtime, the *monitor* construct should point to the relevant data, here the density 'x' line 11. The density is defined on a 2D plane with respect to the loop dimensions i and j. The function `density_val` is a mapping from the density values to the strategies values in the range $[0, 1, 2]$ and is used as the pre-processing for the monitoring. Inside a cell, the minimum possible strategy value is kept using the "min" folding function. Lastly, the runtime enforces those alternatives on cells whose size is defined by the grid directive line 12.

An interesting aspect of this simulation is that it is not possible to generate the compile time tiled version due to the dependencies between iterations of the solver. A value at iteration k of the solver uses the neighbor values at the iteration $k-1$. Hence, it is not possible to use the tiling transformation as it would break the dependencies between the iterations of k. Therefore, the optimized code is generated using the dynamic runtime.

These alternatives reduce the number of times that the statement code line 32 has to be executed between 70% and 30%, after 100 and 2000 time steps respectively. The observed kernel speedup relative to these two time steps are 6.19 and 2.1. The mean deviation of the result stayed below our set limit of 5%. However, a maximum deviation of 50% can be observed for very small values but when taken proportionally to the density values range, the change represents a fraction of $\frac{1}{10000}$ which is not significant.

The ACR extensions lead to a lowered utilization of resources without modification of the original algorithm. Specializing this kernel without ACR would require the developer to write the monitoring and the just-in-time generation of the optimized kernel manually. Such task is not trivial and may require multiple days of development on its own. On the other hand, with ACR, it takes a maximum of few minutes to compile and test various alternatives.

This first example demonstrates the simplicity of use of the ACR extensions. The sole addition of twelve lines of language extensions was sufficient to add adaptivity to this application with the help of the compiler to generate low overhead optimized code at runtime.

5.2 Finite-difference time-domain

Finite-Difference Time-Domain (FDTD) is used in computational fluid dynamics to solve Maxwell-Faraday differential electrodynamics equations [46]. The software implements a solver which updates the magnetic integral of the equation as a first step and then updates the electric field at the same instant using the magnetic field.

Figure 16 includes the function which updates the electric field in a two dimensional (x, y) space. The magnetic field is orthogonal to the electric field and resides in the z dimension, orthogonal to the (x, y) plane. Figure 1 shows a representation of the magnetic field over the 2D space of the simulation. The simulation consists in a wave starting

```
1  unsigned char hz_to_monitor(double hzval) {
     if (hzval > 0.1 || hzval < -0.1) {
3      return 1;
     } else {
5      return 0;
     }
7  }

9  #pragma acr grid(J/5)
   #pragma acr monitor(Hz[i][j], min, hz_to_monitor)
11 #pragma acr alternative low(zero-compute)
   #pragma acr strategy dynamic(1, low)
13 #pragma acr strategy static([S,J] -> {[i,j] : \\
                          S+J/4 >= i >= S-J/4 and \\
15                        3J/4 >= j >= J/4}, low)
   #pragma acr checker-select(versioning,async)
17   for (int i = 0; i < I-1; ++i) {
       for (int j = 0; j < J-1; ++j) {
19       if (j == 0) // Borders
           Ey[i+1][0] += -alpha_Ey *
21                       (Hz[i+1][0] - Hz[i][0]);
         if (i == 0 && j > 1) // Borders
23         Ex[0][j] += alpha_Ex *
                       (Hz[0][j] - Hz[0][j-1]);
25
         Ex[i+1][j+1] += alpha_Ex *
27                       (Hz[i+1][j+1] - Hz[i+1][j]);
         Ey[i+1][j+1] += -alpha_Ey *
29                       (Hz[i+1][j+1] - Hz[i][j+1]);
       }
31   }
```

Fig. 16. Kernel updating the electric field in a 2D space of a FDTD simulation. The kernel is annotated with ACR compiler directives to disable the electric field computation whenever the magnetic field is close to null for all the values inside an ACR cell.

at the left of the domain and traveling to the right. When the wave encounters the impermeable block at the middle, part of it is reflected back and the remainder continues as seen on the top figure at step 700. Later in the simulation, *i.e.* the bottom part of the figure at step 1300, the waves on the right of the block did not reach the right wall yet while the ones on the left have already bounced back. However, it is clearly visible that many parts of the simulation retain a null magnetic field, *i.e.* that there is no current flowing in those parts. Therefore, for this application we consider the removal of the electric field computation where the magnetic field is null.

To nullify the computations inside a zone, the implementation uses the "zero-compute" ACR extension line 11 of Figure 16. The data of importance is the magnetic field Hz, which is set to be monitored at line 10. The strategy annotation line 12 adds a dynamic strategy which maps the "zero-compute" alternative to a value of 1. The preprocessing function "`hz_to_monitor`" returns one whenever the magnetic field is close enough to zero. The low alternative is active for cells filled with ones but a single zero triggers the original computation instead. The grid value is set to one fifth of the y dimension size and is drawn as overlay of the domain in Figure 1. The static strategy line 13 disables the computations for the portion of the simulation where impermeable object resides.

For this application the code uses dynamic code generation for maximum performance of the generated versions. The maximum observed deviation from the original output was 0.99% and a speedup of 1.35 after 5000 steps of the simulation. The ACR extensions allow reasonable speedup at the expense of very low deviation of the output. It is worth noting that after 4500 simulation steps, this technique does not help anymore because the magnetic field does not reach zero in a complete cell anymore. If every portion of the simulation requires precise computations, adaptive methods are of no use. However, this technique can be helpful to quick start the simulation and once the precise state is met, the runtime could be disabled and the original kernel would finish the computation.

This application was optimized using ACR extensions to generate an adaptive code which is specialized at runtime. Writing adaptive and just-in-time code would be time consuming and hard to debug without these extensions. For this application, we encountered a limitation of the adaptive methods where the runtime does not use approximate versions anymore. After a given simulation point the ACR runtime can be disabled to reduce the overhead.

5.3 Game of Life

Game of Life (GOL) is a cellular automaton invented by the British mathematician John Horton Conway. The automaton is constructed using a set of rules describing the evolution of a grid of cells from one state to another. The said rules are very simple and can be used to model complex behaviours:

- A cell *becomes* alive if there are at least three alive cells in its direct surrounding at the time step before.
- A cell *stays* alive if it had two or three alive neighbors at the previous time step.
- Otherwise a cell is considered dead at the next step.

The implementation uses a 2D grid which represents the cells domain and an update function. At each time step the state of all the cells is updated using the previous step values. The update algorithm is shown in Figure 17. Advanced algorithms use the particularity that life automatons have many recurring cell patterns. They use macro-cells and a hash table to store and access the pre-computed results for these patterns [47]. We reused the macro-cell approach with our framework as it matches our proposed ACR-cells almost perfectly.

This application's optimization uses the ACR-cells as macro-cell to determine whenever the macro-cells are updated or not. The life rules state that a cell can only become alive if it has exactly three alive neighboring cells. Therefore, if an ACR-cell contains zero alive cell, the only possible cells that can raise from the dead are the ones on the borders with an alive neighboring ACR-cell. The computation of the ACR-cell is enabled if there is at least one cell alive inside of it.

The "zero-compute" and "interface-compute" alternatives are defined lines 8 and 9 of the GOL kernel Figure 17. These alternatives are linked to strategies in their order of approximation on lines 10 and 11. The ACR-cell grid is generated over the cell grid and represents the macro-cells. The monitoring sets an ACR-cell to use the "zero-compute" alternative whenever no cells are alive using the user defined cell_to_precision and the "min" folding function line 6. The implementation uses the particularity of the "stencil checking" to set active cells neighbor's to update their border cells if they are not active. Therefore, with "stencil checking", a non-active ACR-cell of alternative value 2 is set to the alternative value 1 if it has an active neighbor. This results from the definition of "stencil" which computes the alternative value as $\min(2,0+1)$, where 2 is the cell raw value, 0 is the cell value of one of its neighbors and 1 is added according to the stencil definition. The resulting program only computes the new state for active ACR-cells and for the surrounding of these, only computes the borders with "interface-compute" if they are not active.

The equivalent of the "stencil checking" is not yet available with the static code generation version of our ACR implementation. Therefore, we used the dynamic version in synchronous mode. The cellular automaton state cannot be approximated, but the synchronous mode does not allow to compute a new state before the specialized code version is ready. Hence, there is no possibility for an ACR-cell state to be approximated. This method has been tested against an automaton simulating a digital clock and leads to a speedup between 1.4 and 2 after 20 and 160 generations respectively. With the help of ACR extensions we were able to quickly narrow the computations of the automaton in place where it matters, and to obtain respectable performance gains compared to the original code.

```
1  unsigned char cell_to_precision(cell i) {
     return is_cell_alive(i) ? 0 : 2;
3  }

5  #pragma acr grid(macroCellSize)
   #pragma acr monitor(previous_step_grid[i][j], \
7    min, cell_to_precision)
   #pragma acr alternative low(zero-compute)
9  #pragma acr alternative medium(interface-compute)
   #pragma acr strategy dynamic(1, medium)
11 #pragma acr strategy dynamic(2, low)
   #pragma acr checker-select(stencil,sync)
13   for (int i = 0; i < nb_row; ++i) {
       for (int j = 0; j < nb_col; ++j) {
15       current_grid[i][j] =
           compute_new_value_of_cell(
17           i, j, nb_row, nb_col,
             previous_step_grid);
19     }
     }
21   SWAP(current_grid, previous_step_grid)
```

Figure 17

Fig. 17. Game of Life (GOL) algorithm to update the state of the cell grid from one generation to the next one. The part of the cell grid where there is no living cell is not computed. The active ACR-cells are surrounded by cells that update the border of the ACR-cell and not their center because the GOL rule does not allow cell creation. This strategy both lowers the computations that are needed and allows for a non-approximate GOL update algorithm.

```
1  void complex_compute(...);
   void converged_compute(...);
3  unsigned char settleStrat(size_t a) {
     return a > 7 ? 1 : 0;
5  }

7  do {
     memset(centroids_point_num,0,k*sizeof(size_t));
9    has_converged = true; // Assume convergence
   #pragma acr grid(1)
11 #pragma acr monitor(time_settled[pos], min, \\
     settleStrat)
13 #pragma acr alternative low(function, \
     complex_compute = converged_compute)
15 #pragma acr strategy dynamic(1, low)
     for (int pos = 0; pos < pointsss; ++pos) {
17     complex_compute(dimension, k, pos, centroids,
           centroids_temp, data, centroids_point_num,
19         time_settled, &has_converged);
     }
21
     for (size_t centro = 0; centro < k; ++centro) {
23     for (size_t dim = 0; dim < dimension; ++dim)
         centroids[centro][dim] =
25         centroids_temp[centro][dim] / total_points;
     }
27 } while(!has_converged);
```

Fig. 18. K-Means core algorithm where the observation points are placed into the clusters and the center of the cluster is updated. The application skips the cluster assignment complex function whenever an observation has settled for more than seven iterations within the same cluster. In that case, it is assigned to the same cluster it was at the previous iteration.

5.4 K-Means clustering

K-Means is a partitioning algorithm which distributes n observations into k sets. The partitioning algorithm has the particularity to minimize the variance within clusters. Observations that are "related" or "closer" to each other have a greater chance to end up in the same cluster. The algorithm inside Figure 18 is actually an heuristic implementation of the NP-hard general problem. The algorithm has use cases in signal processing, cluster analysis and machine learning [48]. Most of them try to characterize noisy data using this algorithm.

The implementation uses an iterative algorithm. At the start, it chooses k points from the observations as the center of the clusters. Then at each iteration, each observation is attached to the center of the cluster that is at the lowest "distance". The "distance" is a function that takes two observations and returns lower values when the observation are closely related and bigger ones when they are less so. Once all the observations have been attributed to a cluster, the center of all the clusters are set at the barycentre of all the observations belonging to them. The algorithm loops unless the clusters stabilizes, *i.e.* such that points do not migrate between clusters, or when the ratio of unstable points is lower than a threshold (typically 5%).

Behavioural analysis of the algorithm shows that the cluster's center tends to migrate rapidly during a few iterations at the start of the algorithm, before staying approximately in the same position with fewer observation migrations [48]. Our ACR implementation can take advantage of this behaviour by managing two versions of the cluster affectation. Whenever an observation has already been part of a cluster for many iterations, the probability that it will migrate to another cluster becomes very low. Unfortunately, the original algorithm has no such information available and the code needs to be slightly modified to add the "settling" time of an observation inside a cluster. The additional code is straightforward and sets the counter to zero whenever the observation switches from cluster and increments the value otherwise.

The "complex_compute" function is the original algorithm which computes the distance between a given observation to every centers and assigns the observation to the cluster which yields to the lowest distance. If the observation migrates to another cluster, it sets the convergence to false and the settle value of this observation to zero. Otherwise, the observation is not moved and its settle value incremented. The alternative "converged_compute" is less compute intensive than the original algorithm. It only affects the observation to its previous cluster. The alternative is used whenever the number of iterations that the observation has settled within the same cluster exceeds a threshold. The alternative is defined line 13 of Figure 18. It changes the function called when linked to the strategy of value one. The monitoring oversees the settle information and selects the alternative 1 whenever an observation has settled for more than seven iterations. The approximation is done per observation as defined line 10 with a grid of only one element.

The application was evaluated on linearized 2D images of animals and fruits of various resolutions. ACR is used with the static code generation technique. The maximum deviation of the result is 4.29%. It was observed for the biggest data set of the benchmark which leads to a speedup of 1.54. The approximation tends to work better for larger datasets as the number of settled observations grows in correlation with the problem size. However, to use this approximation technique we needed additional information and construct a new structure to hold it. With the ACR extensions, we successfully leveraged this information to generate a specialized version with a low computational kernel version to achieve a valuable performance gain.

5.5 Overall performance and overhead

In this section we provide benchmark results, the deviation caused by the approximation and the overhead characterisation of ACR's runtime. In the first part, we focus on the application wall clock times and speedups with respect to the deviation of the results, then in the second part, we provide the methodology used to measure the overhead of the runtime. The experimental setup consists of a dodeca-core `Intel Xeon E5-2620 v3` with `16 GB` of `ECC ram`. The compilers and flags used are `GCC v7.1 (-O3-march=native)` and the `Tiny C Compiler v.9.26` for fast just in time compilation of kernels. Our ACR extension implementation is open source and available online [49].

Table 1. Performance of optimized kernel versions against the original code version. The deviation of the results present both the average and the maximum value measured.

Application	Wall time (s)		Speed-up	Deviation	
	Orig.	Opti.		Mean	Max
FDTD	24.53	18.67	1.31	0.29%	0.99%
GOL	170.47	93.58	1.82	None	None
Fluid sim.	108.67	82.47	1.32	0.11%	4.64%
K-Means	15.73	11.37	1.38	1.7%	4.9%

Table 1 shows the averaged results obtained from runs on multiple input problems and data sizes. Our technique achieves significant speedup while keeping the deviation of the result relatively low. The proportion of cells per alternative remained equal for smaller and bigger datasets, while runs on big datasets tend to have a better response to ACR's adaptive methods. We believe that this difference comes from cache and memory latency effects. Different problem statements do not react equally to the same set of alternative parameters. We are working on an auto-tuner that explores the parameters space to better guide the developer in this task. ACR's extensions allowed the compiler to generate the adaptive code automatically without modification to the original algorithm with little investment from the developer.

In order to measure the overhead induced by the runtime and code instrumentation, we saved the application parameters, the optimized kernels along with their utilization timeline while the application was running. Then, programs are run a second time with the same parameters but with the ACR runtime replaced by the saved versions in the same order. This simulates the ideal scenario, *i.e.* where an "oracle" knows ahead of time which version to run. The monitoring and compilation time is finally eliminated.

Table 2 shows the overhead for the FDTD application. It is the application that exhibits the highest overhead because of the high number of generated kernels. The measured overhead increases with the complexity of the generated kernels, because the code generation and compilation take more time for complex polyhedra. Complex polyhedra are typically generated where there are local perturbations. We ran the simulations with the application pinned on one or three CPU cores to identify the compilation and monitoring overheads. On one core, the application and runtime are competing for the CPU resources leading to a slowdown. On three cores, the runtime and the application can run concurrently with performance close to the optimal configuration. The overhead for three cores compared to the optimal is due to thread synchronizations and the asynchronous monitoring of the values. The ACR runtime has been designed to take advantage of modern multi-core architecture and shows low overhead on such systems.

6 Related work

Approximate Techniques Research on techniques to relax data dependencies and skipping computations led to various approximate strategies. Output-prediction (memoization) techniques have been applied on CPU and GPU (Graphics Processing Unit) to leverage the similarities in the output of some applications [34]. Other techniques reduce the number of computations to gain performance, *e.g.*, loop-perforation which removes some loop iterations based on profiling information [31]. Semantics relaxation can lead to better utilization of the capacity of the hardware, *e.g.*, by extracting more parallelism [50]. Byna *et al.* used approximation techniques on a sequential CPU algorithm to obtain a parallel algorithm with great performance on GPUs [51]. Similarly, SAGE (Self-tuning Approximation for Graphics Engines) allows for better parallelism on GPU [52]. Approximate techniques can also take advantage of approximate hardware capacity, *e.g.*, approximate data storage regions or approximate instruction set [53–55]. Less power-hungry approximate hardware can reduce the power required by applications in constrained area [56]. Hoffmann *et al.* provide a runtime that monitors systems metrics, *e.g.* load or power draw, to tune running application parameters to meet a desired application throughput in expense accuracy [57]. Paraprox uses pattern recognition techniques to automatically generate the approximate kernels [58]. Chippa *et al.* propose a framework to characterize the resilience of applications to approximate computing techniques [59]. This framework can help the developers to uncover the parts of an application where approximate techniques could be implemented. These techniques along with ACR allow for optimized usage of the available resources and architectural specificities of the hardware.

Language Extensions: Language extensions give new opportunities to relax or extend the semantics of existing programs. OpenMP annotations relaxe dependency constraints to allow the compiler to generate a parallel version of the program [1]. Similarly, the ACR extension set relaxes constraints to allow for approximate computing. Automatic compiler techniques for approximation purpose take advantage of in-language annotations to generate an approximate version. Ansel *et al.* provide annotations for the data flow language PetaBricks [4]. They use a feedback loop with a genetic algorithm which randomly alter some statements. The program's output deviation is analyzed and the approximation parameters are autotuned against a user provided dataset. EnerJ provides type system annotations in order to reduce the overall system power draw with the help of approximation techniques [29]. EnerJ also formalizes the semantics of a program executing on approximate data-type or code portions to isolate them from ordinary code

Table 2. Overhead for the FDTD application at different steps of the simulation on one core, three cores against the optimal "oracle" version.

	Application time(s)			Overhead / Opti	
Iterations	ACR 1C	ACR 3C	Opti.	1C	3C
500	15.47	12.47	12.20	27%	2.2%
1500	23.22	14.72	14.13	64%	4.2%
3000	36.69	19.31	18.42	99%	4.8%
5000	57.09	28.18	26.88	112%	4.8%

sections. With Green, developers specify the maximum acceptable deviation of result for loops and procedures [60]. Their framework provides a statistical guarantee that the value computed by the application will follow the quality of service requested. These previously mentioned techniques use a feedback loop to automatically alter the program semantics and check the output deviation against nonmodified training dataset. Bornholt *et al.* provide an uncertain datatype and operations to manipulate this object [61]. Eon provides a language and runtime system that integrates hardware information, *e.g.* battery level, to tune the quality of service depending on the environment [62]. With ACR, the programmer has full control on the transformations and the decision mechanism. Our extensions allow for automatic generation of localized adaptive approximation which, to the best of our knowledge, is not available in previous methods.

7 Conclusion

Adaptive Code Refinement language extension set provides approximation and adaptive capabilities to existing languages through specific annotations to be exploited by both a compiler and a runtime system. While being less invasive than writing an optimized code manually or using dedicated libraries, these extensions allow a software developer to annotate a computational kernel to transform it to an efficient and adaptive approximate version. ACR's annotation system is well adapted for a two phase software development workflow. First, the developer focuses on the development and validation of the main algorithm. Then, in the optimization phase, the algorithm is optimized to scale to large datasets using approximation techniques. With the help of ACR extensions and a compiler supporting them, the software developer can quickly evaluate various approximation strategies easily.

Approximation techniques are achieved, *e.g.*, with algorithmic tuning or removal of computations in selective conditions. Such possibilities are offered by ACR's *alternative* construct for a new level of flexibility and productivity. The alternatives can be enforced globally using static strategies or dynamically depending on the dynamic state of the data. ACR builds on the polyhedral representation of programs to generate monitoring and approximated code with low control overhead and limited deviation of the results. Furthermore, ACR's runtime includes several mechanisms for the user to control the precision.

We evaluated the expressiveness of the ACR extension set on various example applications, from fluid simulation to signal processing and data mining. The strategies were implemented on top of their compute intensive kernels with few ACR extensions and low or no modification of the original programs. All ACR-extended code show speedups at the price of a reasonable loss in precision.

ACR language extension set opens up new possibilities for software developers that need to exploit approximation to trade precision for speed. It allows for fast testing and prototyping of multiple approximation techniques. To the best of our knowledge, ACR is the first language extension set dedicated to approximation that covers such a wide range of approximation strategies.

Ongoing work includes mixing approximation with polyhedral optimization and parallelization techniques, improving ACR's runtime and extending ACR annotations with data compression capabilities to reduce memory footprint and adapt computation accordingly.

References

1 Dagum L., Menon R. (1998) OpenMP: an industry standard API for shared-memory programming, *IEEE Comput. Sci. Eng.* **5**, 1, 46–55.
2 Sampson A., Baixo A., Ransford B., Moreau T., Yip J., Ceze L., Oskin M. (2015) *Accept: A programmer-guided compiler framework for practical approximate computing*, University of Washington Technical Report UW-CSE-15-01, p. 1.
3 Carbin M., Misailovic S., Rinard M.C. (2013) Verifying quantitative reliability for programs that execute on unreliable hardware, *ACM SIGPLAN Notices* **48**, 33–52.
4 Ansel J., Lok Wong Y., Chan C., Olszewski M., Edelman A., Amarasinghe S. (2011) Language and compiler support for auto-tuning variable-accuracy algorithms, in: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 85–96.
5 Burstedde C., Wilcox L.C., Ghattas O. (2011) p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees, *SIAM J. Sci. Comput.* **33**, 1103–1133.
6 Kirk B.S., Peterson J.W., Stogner R.H., Carey G.F. (2006) libMesh: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations, *Eng. Comput.* **22**, 3–4, 237–254.

7 Musser D.R., Derge G.J., Saini A. (2009) *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, 3rd edn., Addison-Wesley Professional, London, UK.

8 Kambatla K., Kollias G., Kumar V., Grama A. (2014) Trends in big data analytics, *J. Parallel Distrib. Comput.* **74**, 7, 2561–2573.

9 Mittal S. (2016) A survey of techniques for approximate computing, *ACM Comput. Surv. (CSUR)* **48**, 4, 62.

10 Tornvist L., Vartia P., Vartia Y. (1985) How should relative changes be measured? *Am Statist.* **39**, 43–46.

11 Hore A., Ziou D. (2010) Image quality metrics: Psnr vs. ssim, *20th International Conference on Pattern Recognition (ICPR)*, pp. 2366–2369.

12 Feynman R.P., Leighton R.B., Sands M. (2011) *The Feynman lectures on physics, Vol. I: The new millennium edition: mainly mechanics, radiation, and heat*, Basic Books.

13 Bishop C.M. (2006) *Pattern Recognition and Machine Learning (Information Science and Statistics)*, Springer Verlag, Berlin, Heidelberg.

14 Feautrier P. (1992) Some efficient solutions to the affine scheduling problem. I. One-dimensional time, *Int. J. Parallel Program.* **21**, 5, 313–347.

15 Feautrier P. (1992) Some efficient solutions to the affine scheduling problem. II. multidimensional time, *Int. J. Parallel Program.* **21**, 6, 389–420.

16 Bastoul C. (2016) Mapping deviation: A technique to adapt or to guard loop transformation intuitions for legality, in: *Proceedings of the 25th International Conference on Compiler Construction, Barcelona, Spain*, pp. 229–239.

17 Grosser T., Groesslinger A., Lengauer C. (2012) Polly – performing polyhedral optimizations on a low-level intermediate representation, *Parallel Process. Lett.* **22**, 1250010.

18 Bastoul C. (2008) *Extracting polyhedral representation from high level languages*, Tech. Rep. Related to the Clan tool, LRI, Paris-Sud University.

19 Verdoolaege S., Grosser T. (2012) Polyhedral extraction tool, in: *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France, pp. 1–16.

20 Bastoul C. (2004) Code generation in the polyhedral model is easier than you think, in: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pp. 7–16.

21 Quilleri F., Rajopadhye S., Wilde D. (2000) Generation of efficient nested loops from polyhedra, *Int. J. Parallel Program.* **28**, 5, 469–498.

22 Banerjee U. (2007) *Loop transformations for restructuring compilers: the foundations*, Springer Science & Business Media, Massachusetts, USA.

23 Strang G. (2009) *Introduction to linear algebra*, Vol. 4, Wellesley-Cambridge Press, Wellesley, MA.

24 Wolfe M. (1989) More iteration space tiling, in: *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pp. 655–664.

25 Wolf M.E., Lam M.S. (1991) A data locality optimizing algorithm, *ACM Sigplan Notices* **26**, 30–44.

26 Verdoolaege S. (2010) isl: An integer set library for the polyhedral model, *ICMS* **6327**, 299–302.

27 Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T., Wonnacott D. (1996) *The omega calculator and library, version 1.1.0*, College Park, MD, 20742:18

28 Berger M.J., Colella P. (1989) Local adaptive mesh refinement for shock hydrodynamics, *J. Comput. Phys.* **82**, 64–84.

29 Sampson A., Dietl W., Fortuna E., Fortuna E., Gnanapragasam D., Ceze L., Grossman D. (2011) Enerj: Approximate data types for safe and general low-power computation, in: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI* '11, San Jose, California, USA, pp. 164–174.

30 Yeh T., Faloutsos P., Ercegovac M., Patel S., Reinman G. (2007) The art of deception: Adaptive precision reduction for area efficient physics acceleration, *40th Annual IEEE/ACM International Symposium on Microarchitecture*, 394–406.

31 Sidiroglou-Douskos S., Misailovic S., Hoffmann H., Rinard M. (2011) Managing performance vs. accuracy trade-offs with loop perforation, in: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 124–134.

32 Misailovic S., Roy D.M., Rinard M.C. (2011) Probabilistically accurate program transformations, in: Yahav E. (eds), *Static Analysis*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 316–333.

33 Rinard M. (2006) Probabilistic accuracy bounds for fault-tolerant computations that discard tasks, in: *Proceedings of the 20th Annual International Conference on Supercomputing, ICS* '06, ACM, Cairns, Queensland, Australia, pp. 324–334.

34 Rahimi A., Benini L., Gupta R.K. (2013) Spatial memoization: Concurrent instruction reuse to correct timing errors in simd architectures, *IEEE Trans. Circuits Syst. II: Express Briefs* **60**, 12, 847–851.

35 Michie D. (1968) ''memo'' functions and machine learning, *Nature* **218**, 5136, 19.

36 Ansel J., Chan C., Wong Y.L., Olszewski M., Zhao Q., Edelman A., Amarasinghe S. (2009) *PetaBricks: a language and compiler for algorithmic choice*, Vol. **44**, **ACM**.

37 Schmitt M., Helluy P., Bastoul C., Bastoul C. (2017) Adaptive code refinement: A compiler technique and extensions to generate self-tuning applications, *HiPC 2017 – 24th International Conference on High Performance Computing, Data, and Analytics,* Jaipur, India, pp. 1–10.

38 Bastoul C., Cohen A., Girbal S., Sharma S., Temam O. (2004) Putting polyhedral loop transformations to work, in: Rauchwerger L. (ed.), *Languages and Compilers for Parallel Computing*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 209–225.

39 Verdoolaege S., Grosser T. (2012, January) Polyhedral extraction tool, *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France.

40 Schrijver A. (1998) *Theory of linear and integer programming*, John Wiley & Sons.

41 Bondhugula U., Hartono A., Ramanujam J., Sadayappan P. (2008) A practical automatic polyhedral parallelizer and locality optimizer, *SIGPLAN Notices* **43**, 6, 101–113.

42 Pouchet L.-N., Bastoul C., Cohen A., Cavazos J. (2008) Iterative optimization in the polyhedral model: Part II. Multidimensional time, *ACM SIGPLAN Notices* **43**, 90–100.

43 Bielecki W., Palkowski M. (2016) Tiling of arbitrarily nested loops by means of the transitive closure of dependence graphs, *Int. J. Appl. Math. Comput. Sci. (AMCS)* **26**, 4, 919–939.

44 Stam J. (1999) Stable fluids, in: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH* '99, ACM Press/Addison-Wesley Publishing Co, New York, NY, USA, pp. 121–128.

45 Stam J. (2003) Real-time fluid dynamics for games, in: *Proceedings of the Game Developer Conference*, 25.

46 Oskooi A.F., Roundy D., Ibanescu M., Bermel P., Joannopoulos J.D., Johnson S.G. (2010) Meep: A flexible free-software package for electromagnetic simulations by the FDTD method, *Comput. Phys. Commun.* **181**, 3, 687–702.
47 Wm Gosper R. (1984) Exploiting regularities in large cellular spaces, *Phys. D: Nonlinear Phenom.* **10**, 1–2, 75–80.
48 Meng J., Chakradhar S., Raghunathan A. (2009) Best-effort parallel execution framework for recognition and mining applications, *IPDPS*'09, pp. 1–12.
49 Schmitt M. (2017) *ACR compiler and runtime*, http://gauvain.u-strasbg.fr/~schmitt/acr
50 Campanoni S., Holloway G., Wei G.-Y., Brooks D.M. (2015) HELIX-UP: Relaxing program semantics to unleash parallelization, in: *IEEE/ACM CGO,* San Francisco, USA, pp. 235–245.
51 Byna S., Meng J., Raghunathan A., Chakradhar S., Cadambi S. (2010) Best-effort semantic document search on gpus, in: *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 86–93.
52 Samadi M., Lee J., Jamshidi A., Anoushe Hormati D., Mahlke S. (2013) Sage: Self-tuning approximation for graphics engines, in: *MICRO*'13 IEEE/ACM Intl. Symp. on Microarchitecture, Davis, California, pp. 13–24.
53 Chippa V.K., Mohapatra D., Raghunathan A., Roy K., Chakradhar S.T. (2010) Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency, in: *Proceedings of the 47th Design Automation Conference*, pp. 555–560.
54 Fang Y., Li H.W., Li X.W. (2012) Softpcm: Enhancing energy efficiency and lifetime of phase change memory in video applications via approximate write, in: *Test Symposium (ATS), 2012 IEEE 21st Asian*, pp. 131–136.
55 Sampson A., Nelson J., Strauss K., Ceze L. (2014) Approximate storage in solid-state memories, *ACM Trans. Comput. Syst.* **32**, 3, 1–9.
56 Misailovic S., Carbin M., Achour S., Qi Z.C., Rinard M.C. (2014) Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels, *ACM SIG-PLAN Notices* **49**, 309–328.
57 Hoffmann H., Sidiroglou S., Carbin M., Carbin M., Misailovic S., Agarwal A., Rinard M.C. (2011) Dynamic knobs for responsive power-aware computing, in: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVI*, ACM, Newport Beach, California, USA, pp. 199–212.
58 Samadi M., Jamshidi D.A., Lee J., Mahlke S. (2014) Paraprox: Pattern-based approximation for data parallel applications, in: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS* '14, ACM, Salt Lake City, Utah, USA, pp. 3550.
59 Chippa V.K., Chakradhar S.T., Roy K., Raghunathan A. (2013) Analysis and characterization of inherent application resilience for approximate computing, in: *Proceedings of the 50th Annual Design Automation Conference*, pp. 113.
60 Baek W., Chilimbi T.M. (2010) Green: A framework for supporting energy-conscious programming using controlled approximation, in: *Proceedings of the 31st ACM SIG-PLAN Conference on Programming Language Design and Implementation, PLDI* '10, ACM, Toronto, Ontario, Canada, pp. 198–209.
61 Bornholt J., Mytkowicz T., McKinley K.S. (2014) Uncertain: A first-order type for uncertain data, in: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS* '14, ACM, Salt Lake City, Utah, USA, pp. 51–66.
62 Sorber J., Kostadinov A., Garber M., Brennan M., Corner M.D., Berger E.D. (2007) Eon: A language and runtime system for perpetual systems, in: *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems, SenSys* '07, ACM, Sydney, Australia, pp. 161–174.