Adaptive Code Refinement: A Compiler Technique and Extensions to Generate Self-Tuning Applications

Maxime Schmitt, Philippe Helluy, Cédric Bastoul

Adaptive Code Refinement: A Compiler Technique and Extensions to Generate Self-Tuning Applications

Maxime Schmitt, Philippe Helluy and Cédric Bastoul
University of Strasbourg, Inria
Strasbourg, France
{max.schmitt,helluy,bastoul}@unistra.fr

***Abstract*—Compiler high-level automatic optimization and parallelization techniques are well suited for some classes of simulation or signal processing applications, however they usually don't take into account domain-specific knowledge nor the possibility to change or to remove some computations to achieve "good enough" results. Differently, production simulation and signal processing codes have adaptive capabilities: they are designed to compute precise results only where it matters if the complete problem is not tractable or if computation time must be short. In this paper, we present a new way to provide adaptive capabilities to compute-intensive codes automatically. It relies on domain-specific knowledge provided through special pragmas by the programmer in the input code and on polyhedral compilation techniques to continuously regenerate at runtime a code that performs heavy computations only where it matters. We present experimental results on several applications where our strategy enables significant computation savings and speedup while maintaining a good precision, with a minimal effort from the programmer.**

I. Introduction

A large range of compute-intensive applications are calculating approximate results. This is especially true for simulation codes which are based on inherently imperfect models that try to emulate as precisely as possible a real world object or phenomenon. This is also true for signal processing applications which are limited by the precision of, e.g., input sensors or processing algorithms. Some other applications may also compute approximate results for functional reasons, e.g., to meet a deadline such as in real-time video decoding, or because the precise result is not tractable or valuable such as late earthquake prediction, or simply because the user needs a rough result to drive further precise investigations such as in geophysics. For such applications, "ideal" naïve computation kernels, that would be convenient with an infinite computation power, are often designed at first for algorithmic tuning and debugging purposes. Then they are optimized to a "production" version exploiting possible approximations to scale to the actual problem size or to meet the deadline. Translating an ideal code to a production code is complex, time consuming, leads to less maintainable codes and must be redone when a major change in the initial strategy arises. In this paper, we present a new compiler technique that automates the conversion from an ideal code to a version that exploits approximations dynamically, by adjusting computations with respect to the current state of the program. It aims at improving both developer's productivity and approximation's quality.

State-of-the-art automatic optimization and parallelization compiler techniques heavily rely on the *polyhedral model* to manipulate computation-intensive kernels to aggressively restructure them at the iteration level [1], [2], [3]. They are usually based on exact, or over-approximated, data dependence analysis to ensure that the optimization preserves the original program semantics. Relaxed semantics models are possible, e.g., to support commutativity to enable vectorization [4], however all input code iterations are to be executed in the optimized code. On the other hand, more aggressive techniques to automatically compute approximations have been designed, e.g., by ignoring some dependencies to enable parallelization [5], by providing alternative implementations of some code parts [6], or by skipping computations [7], [8]. In our work, we investigate a new way, called *Adaptive Code Refinement* (ACR), inspired by Adaptive Mesh Refinement [9], a classical numerical analysis technique providing the ability to dynamically tune a computational grid to achieve precise computation only where it matters. We achieve this goal by exploiting domain-specific information provided by the user, a dynamic optimization strategy and state-of-the-art polyhedral code generation techniques.

Our strategy is supported by high-level information provided by the user and a static-dynamic code generation approach. First, we designed a set of pragmas to provide the user with the means to express both static and dynamic approximation-related information. It allows the user to focus only on an "ideal" version of the computational kernels. Those kernels can actually compile and run while ignoring the pragmas, but they may only produce results in a reasonable amount of time for small problems. The pragma set is quickly described in Section II. Next, ACR uses polyhedral code generation techniques to generate a code to compute approximate results according to static information, with the ability to regenerate itself according to the evolution of the computed values and to pre-defined approximation strategies. In a nutshell, ACR decomposes the computation space into a grid and monitors specific values at the grid level. Depending on their evolution, it will tune the computation precision onto each grid cell to ensure complex computation is done only where and when it matters. According to this monitoring, it will (1) generate an optimized version of the code specific to the current situation, (2) compile it dynamically and (3) switch the execution to the new optimized code when it is ready. Specific threads are devoted to monitoring and code generation to minimize overhead. The continuous monitoring and code generation process enables strong computation savings while limiting accuracy loss. The process is depicted in Section III.

In Section IV we present a case study on an Eulerian fluid simulation implementation which belongs to a class of applications that are naturally well suited for our technique. Experimental results on this case study and other benchmarks from simulation and iterative algorithms are detailed in Section V. The study shows empirical evidence that ACR allows significant time and computation savings while maintaining accuracy with minimal efforts from the programmer, even compared to a hand-tuned version. Related work is presented in section VI.

II. ACR Pragma Set

ACR offers a set of high-level pragmas that allow the user to provide high-level, domain-specific approximation information. The ACR optimizing algorithm will exploit them to automatically compute quality approximate results. ACR pragmas are language extensions to be inserted before a computational loop to compute approximations using either alternative implementations to those provided inside the loop or a reduced number of computations in some areas of the computation space. The pragma set is divided into four constructs:

1. The `monitor` construct specifies the data to monitor for the dynamic optimization strategy and how the monitoring is summarized for a complete cell. Its format is:
`#pragma acr monitor(`*data*$[f(\vec{\imath})]$`,`*summary*`[, `*filter*`])`
where *data* is the data array to monitor (using the access function $f(\vec{\imath})$ of the iteration vector $\vec{\imath}$), *summary* specifies how monitored values should be summarized into one value at the cell level (using predefined policies, e.g., `mean`, `max`, `min`, etc.) and *filter* is an optional function that may be used to preprocess values being monitored (e.g., to classify them into categories): the actual value used to drive the dynamic optimization strategy will be *filter(data)*.

2. The `grid` construct defines the granularity of the dynamic approximation strategy: the `data space` is decomposed into cells of a size equal to the specified grid value. The monitoring as well as the approximation strategy will be done at the cell-level granularity. Its format is:
`#pragma acr grid(`*size*`)`
where *size* is a constant number: if the computation space is 2-dimensional, the cell size will be $size \times size$.

3. The `strategy` construct specifies in which conditions which alternatives should be used for each cell. They may be either static (specifying areas of the computation space where to apply a given alternative) or dynamic (specifying which alternative to use in a cell depending on the monitored value or its evolution in that cell). Its format is:
`#pragma acr strategy direct(`*value*`, `*alternative*`)`
where *value* is a value that can be reported by a monitor for a grid cell during execution and *alternative* is the alternative to apply to a cell if the monitor reports that value for that cell. An example of a static (compile time defined) strategy is:
`#pragma acr strategy zone(`*area*`, `*alternative*`)`
where *area* is a part of the computation space expressed using set notations (e.g., `{i | 0<=i<=3}` to express the part of the space where `i` is between 0 and 3) and *alternative* is the name of an alternative defined with the `alternative` construct. Many other strategies are possible, however describing the complete set is out of the scope of this paper. All the benchmarks presented in Section V use a dynamic strategy with a set of direct links between dynamic monitored values and alternative computations.

4. The `alternative` construct defines an alternative computation to the one provided in the code. Its format is:
`#pragma acr alternative `*name*`(`*type*`, `*effect*`)`
where *name* is a user-defined strategy name and *type* specifies the strategy type as follow:

1) *parameter* to keep the original code but with a new parameter values to be defined in the *effect* field,
2) *code* to provide an alternative code block to be defined in the *effect* field
3) *zero_compute* to cut out the computation entirely
4) *interface_compute* to only maintain computation at the grid cell interface.

Fig. 1 shows an example on how this pragma set may be used to specify approximate computation on a simplified version of a loop of our case study, detailed in Section IV.

```
// Loop iterating over frames of the fluid simulation
while(true) {
  ...
  // lin_solve kernel
  #pragma acr grid(10)
  #pragma acr monitor(density[i][j], max, filter)
  #pragma acr alternative low(parameter,    MAX = 1)
  #pragma acr alternative medium(parameter, MAX = 4)
  #pragma acr alternative high(parameter,   MAX = 8)
  #pragma acr strategy direct(1, low)
  #pragma acr strategy direct(2, medium)
  #pragma acr strategy direct(3, high)
  for (k = 0; k < MAX; k++)
    for (i = 1; i <= N; i++)
      for (j = 1; j <= N; j++)
        lin_solve_computation(k, i, j);
  ...
}
```

Fig. 1: **Pseudo-code depicting the application of ACR to the `lin_solve` kernel of the fluid simulation case study**. Here, the `filter` function returns in which range a given density value is: near zero (1), medium (2) or high (3). The monitor value for a cell corresponds to the maximum of all ranges in the cell. This range is used to apply a different alternative computation for each cell. In this example, the loop `k` will iterate more or less depending on that range.

III. Adaptive Code Refinement

Computational loops annotated with user-provided ACR pragmas are candidates for approximate computation using our approach. In such kernels, the rigid data-dependence model used by compilers is explicitly relaxed and alternative computation can be used to make the computation faster, according to the user-defined strategy. This strategy may be static, i.e., independent of the computed values in specific areas of the computation space, or dynamic, i.e. dependent on the monitored values in parts of the computation space where no static strategy applies. ACR makes a strong use of state-of-the-art polyhedral compilation techniques to generate efficient codes computing approximations with good precision. In particular we represent the computation space and its cells as polyhedra in the same way polyhedral compilers represent iteration domains [1], [2], [3], but we rely on a less restrictive representation. The background details on the polyhedral model, and our application domain are detailed in Sections III-A. The ACR optimizing algorithm is based on three main components. First it continuously evaluates a "state grid" of the data space according to the user-defined strategy as detailed in Section III-B. Second, it generates an optimized code according to the state grid, as explained in Section III-D. Finally, a runtime ensures the best version of the code for both performance and accuracy is used. It is discussed in Section III-E.

A. Background

The application domain of ACR corresponds to computational loop-based kernels such that the possible values of each loop iterator can be modeled by a set of affine constraints on that loop iterator, outer loop iterators and fixed parameters. A simple and useful case corresponds to loops such that the loop stride is a known constant and the loop bounds are linear functions of the outer loop counters and fixed parameters.

The corresponding class of applications is a superset of the polyhedral model [1] which fuels modern loop optimizers such as in GCC [10] or LLVM [11]. The polyhedral model is an algebraic representation of programs, complementary to, e.g., abstract syntax trees. It allows to achieve precise analyses of the code and to apply aggressive loop transformations. For a code to be raised to a polyhedral representation, all loop bounds, branch conditions and subscript functions must be affine expressions of outer loop iterators and constant parameters. Despite these constraints, many computational loops in scientific applications may be modeled through this representation either directly [12] or relying on existing model extensions [13], [14]. In our work, we consider only restrictions on the kernel loop bounds and on the subscript function of the monitored data as provided in the `monitor` construct (see Section II). Hence, we are able to deal with more general loop bodies than the strict polyhedral model, where any kind of data access is possible and data dependent conditions are allowed as well.

The key polyhedral model property we are exploiting is the ability to represent the iteration domain of a multidimensional loop through a system of linear inequalities. This system defines a polyhedron in a multidimensional space where each integer point corresponds to an iteration of the loop. E.g., the computation space of the `lin_solve` loop in Fig. 1 is:

$$\mathcal{D}_{lin\_solve}(\mathrm{MAX},\mathrm{N}) = \left\{ \left(\begin{array}{c} k \\ i \\ j \end{array} \right) \middle| \begin{array}{lll} 0 \leq & k & < MAX \\ 1 \leq & i & \leq N \\ 1 \leq & j & \leq N \end{array} \right\}.$$

Representing iteration spaces in this way, it is quite easy to specify subsets of that space (inserting additional constraints), to compose them (applying polyhedral unions) or to decompose them (splitting the polyhedra into unions of polyhedra) them depending on our algorithm. To manipulate polyhedra, we rely on `isl` [15] to achieve, e.g., polyhedral unions and differences and on CLooG [16] to generate back a code from a polyhedral representation.

Another key polyhedral model concept we are using is *scheduling*. It specifies the relative ordering of the iterations with linear constraints. Scheduling is a polyhedral relation from the iteration space to a multidimensional time space. In the final code, the iterations are ordered with respect to the lexicographic ordering of their time dimension (the first dimension is the most significant, the next one is less significant and so on, like hours, minutes, seconds, etc.). For instance, the ordering of the iterations of the `lin_solve` loop in Fig. 1 is:

$$\theta_{lin\_solve}(\mathrm{MAX},\mathrm{N}) = \left\{ \left(\begin{array}{c} k \\ i \\ j \end{array} \right) \rightarrow \left(\begin{array}{c} t_1 \\ t_2 \\ t_3 \end{array} \right) \middle| \begin{array}{l} t_1 = k \\ t_2 = i \\ t_3 = j \end{array} \right\}.$$

This scheduling corresponds to the identity with respect to the initial code: the i^{th} time dimension is equal to the i^{th} loop iterator, hence the iterations should be executed in the same order as in the initial loop. In this work, we rely on polyhedral scheduling to guarantee the remaining or simplified iterations are executed in the same order with respect to the original code to minimize result deviation. We use CLooG [16] to generate a code that respects a scheduling specification.

B. Monitoring Data

ACR needs a way to gather information about the computation at runtime to identify different approximation regions, i.e., the parts of the computation space where the approximation strategy should be different. This task should be simple enough to avoid adding significant computation overhead with regard to the time saved by the optimized code. To achieve that, a uniform grid is embedded into the data space to represent different zones of the simulation, according to the `grid` pragma. Every cell in the grid represents an (hyper-)square shaped portion of the data space. The dimension of the grid may affect the accuracy and the efficiency of the generated code in contradictory ways, hence a good tradeoff is preferable.

The information stored in the grid, or its evolution during execution, should allow to decide about the desired accuracy in the corresponding portion of the computation space. The grid is refreshed by a specific thread according to the user-provided monitoring specification. Moreover, to avoid too frequent changes in the grid (which would translate to code generations that may be obsolete when they become available), we did study different post-processing policies (evaluated on our case study in Section V):

Raw (no post-processing): the grid reflects exactly the approximation strategy which should be applied according to ACR pragmas, with the risk that some regions oscillate rapidly between various approximation strategies, not leaving enough time to generate optimized codes between two changes.

Versioning: grid cells are updated when a higher precision is needed whereas precision downgrade is ignored. When the difference between the current grid and the `raw` grid is more than a given threshold, we restart with the `raw` grid. Hence some grid cells are set to a more precise alternative with respect to ACR pragmas, with the risk of a less efficient computation. However this policy reduces the need for a new code compilation because we allow some cells to use higher precision temporarily.

Stencil: each grid cell is evaluated not only according to the monitored values in that cell, but also according to neighboring cells. For instance if a grid cell is set to low precision but is surrounded by high-precision cells, it is switched to high-precision as well to anticipate a probable change. Here again, some grid cells are tagged to be more precise with respect to ACR pragmas, with the risk of a less efficient computation.

Once the grid is filled with the information gathered, and post-processed if requested, the regions are constructed by joining grid cells with similar approximation strategy.

Technically, each grid cell is represented as a polyhedron and the region is constructed by aggregating cells with polyhedral unions thanks to `isl`. Fig. 3 shows a snapshot of a fluid simulation and the corresponding grid state: each shade of grey corresponds to a region where a similar approximation strategy should be applied.

C. From Data Space to Computation Space

The monitoring builds a grid of the data space where each grid cell is linked to an alternative. In order to generate the optimized code, we need to know which part of the computation space contributes to each grid cell.

When the access function of the monitored data is an affine function of the outer loop iterators and constant parameters, which is a restriction of the `monitor` construct, and the iteration space can be represented through a system of linear inequalities as described in Section III-A, then the relation between the computation space and the data space is an affine relation that maps each point of the iteration domain to the data it is accessing. Finding the computation space contributing to some data is a matter of inverting the access function, a classical *preimage* polyhedral computation [17]. For instance, in Fig. 3 the monitored data is `density[i][j]`. A grid cell will specify bounds for `i` and `j`, and the preimage will report all the original iterations for `i` and `j` within those bounds and `k` within the original bounds.

D. Code Generation

Once a state grid has been computed to reflect the current mapping of approximation strategies onto different regions of the computation space, the next step is to generate on the fly an optimized code to replace the current one (if the current state grid actually differs from the previous one) to implement the approximation. Several approaches with different properties are possible. The *hand-tuned* approach is the naïve, simple way to build an adaptive code. It respects the original iteration ordering but suffers from a high control overhead. We describe it in Section III-D1. The *dynamic* approach, further explained in section III-D2, is capable at runtime to generate a code that respects the original iteration ordering but with a low control overhead. Finally, the *static* approach generates an adaptive code at compile time with low control overhead but without respecting the original iteration ordering. This version may be used only when no inter-cell data dependency exists to avoid high deviation of the approximation.

1) Hand-tuned Code: A trivial way to build an adaptive code from an original loop to approximate and the monitoring information is to insert a guard inside the computational loop to select the strategy corresponding to the current iteration. Because the adaptive loop itself is easy enough to be written manually, we refer it as the *hand-tuned* code. Using this strategy, only the dynamic selection has to be generated at runtime according to the monitoring. It simply computes to which grid cell contributes a given iteration, then it executes the corresponding approximation strategy. The hand-tuned code respects the original iteration ordering since it scans the original iteration domain in the initial order. However its control overhead is quite high since a selection test has to be performed at the innermost loop level.

2) Dynamic Code Generation: To generate a very efficient code with no control overhead, we translate the problem to a code generation in the polyhedral model task. Tools like CLooG [16] are able to generate an efficient code from a polyhedral representation made of two set of objects: *iteration domains* which describe the set of statement instances to execute, and *scheduling relations* which describe the relative order of the statement instances.

We build the iteration domains by aggregating grid cells with similar approximation strategies together using polyhedral union operations. Hence, each region is modeled as a union of convex polyhedra. Those regions are then mapped back to the computation space by considering the iteration subspace that updates those regions as detailed in Section III-C. We associate each subspace with the corresponding computation that reflects the approximation strategy, i.e., a block of code or some new constraints such as parameter values (as in our example in Fig. 1). Those subspaces form the input iteration domains of the code generation problem. Then, to ensure the approximated computations are processed in a similar order than the original one to preserve accuracy, we enforce the lexicographic ordering of the original computation space dimensions as the input scheduling relations of the code generation problem. Then CLooG is able to generate a code with extremely optimized control overhead to, e.g., avoid costly tests at the innermost level of the computational loop to choose the right approximation strategy, which could not be possible with a static approach.

Once CLooG has generated an approximation code, it is compiled and loaded dynamically to the computation process. The automatically generated code has no costly internal tests to decide about the optimization strategy, and the remaining computations are done with respect to the initial ordering to preserve accuracy. It executes the same iterations on the hand-tuned version and in the same order, but without any internal test to drive the computation. The cost to get such a code is a polyhedral code generation at runtime because it is necessary to get the grid state to generate a code that matches the current situation. In Sec. V, we show that the performance benefits of the generated code is much greater than its generation cost.

3) Static Code Generation: Our last code generation approach generates a code at compile time with low control overhead, but that does not respect the original iteration ordering. We rely again on polyhedral manipulation techniques to decompose the original iteration domains to disjoint parts that contribute to different grid cells. We also rely on polyhedral code generation techniques to generate a code corresponding to each grid cell. The loop bounds may depend to parameters that will be updated dynamically to reflect the current approximation strategy for that grid cell. At runtime, each grid cell code will be executed independently, hence without respecting the initial iteration order. This may severely impact the approximation quality but may be adapted for codes with simple data dependences where iterations are independent.

E. Runtime

The ACR runtime for the dynamic code generation is decoupled into five threads to exploit multicore architectures and to reduce the technique's overhead:

(1) the *computation thread* is responsible for the main computation itself, (2) the *monitoring thread* computes the state grid, (3) the *CLooG thread* provides a code generation service: it waits for polyhedral code generation requests and generates the corresponding C codes with low control overhead (several server threads may coexist to process several requests concurently), (4) the *compilation thread* provides a compilation service: it waits for C code compilation requests and generates the corresponding object codes, finally (5) the *coordinator thread* creates and manages all the other threads.

The runtime operates as summarized in Fig. 2. At the beginning of the computation, no optimized code is available. Hence, the computation thread executes the original code for the first iteration and updates the internal data structures. The monitoring thread constantly watches the monitored data as specified by ACR pragmas. When necessary, it updates the state grid and signals the coordinator thread. When the coordinator thread is signaled about the availability of a new state grid, it builds a code generation request to get an optimized code corresponding to the current situation. Then it sends it to the CLooG thread. When the CLooG thread answers, the coordinator thread sends a compilation request to the compilation thread, who answers with an object code. In the meantime, the computation thread continues the iterations with its current code. When the coordinator thread receives a new compiled optimized code, it checks whether the code generated still fits the current state of the grid or not. If yes, it updates the code of the computation thread for the next kernel call. If not, it ignores it, updates its request for an optimized code and lets the computation thread continue with the original code. The same happens if the state grid evolves while an unsuitable optimized code is being used by the compute thread: the computation code is switched back to the original.

The runtime is optimized in several ways to ensure a convenient optimized code is available for the computation thread as soon as possible. First, the coordinator thread requests two different compiled codes for the same C input: a non-optimized one which may be generated and used quickly (we use TCC, the Tiny C Compiler) and a very optimized code that may be available later and that will replace the non-optimized one (we used GCC with aggressive optimization options for this). Second, the coordinator thread is using a cache of generated codes to immediately use an already generated code for a known state grid. Finally, the coordinator accepts over-approximations instead of switching back to the original code: what is needed is that the optimized code performs the same or more complex computations than the levels specified in the current grid, for every grid slot. In that way we can say that the computations done are "safe" and they do at least what was specified by the domain-specific information.

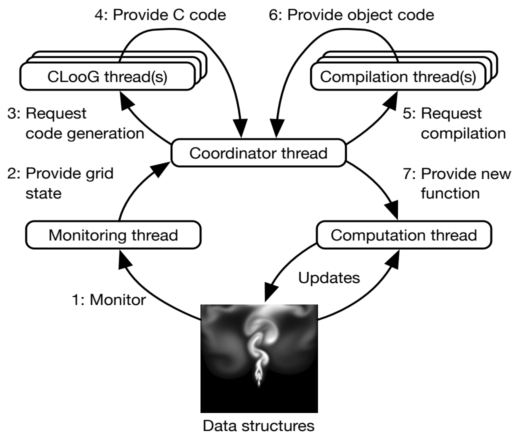


Fig. 2: ACR runtime thread interaction diagram.

When using the hand-tuned or the static approaches, the runtime is simply the same without CLooG and compilation threads since there is no need for runtime code generation. Instead, the monitoring updates a selection function or a parameter array that drives the loop execution.

IV. Case Study: Fluid Simulation

To illustrate how Adaptive Code Refinement can be used, we detail how it may be applied to a typical representent of its application domain, a fluid simulation application [18]. This program, called Eulerian fluid simulation, has the characteristic of being a grid based simulation. A snapshot of such

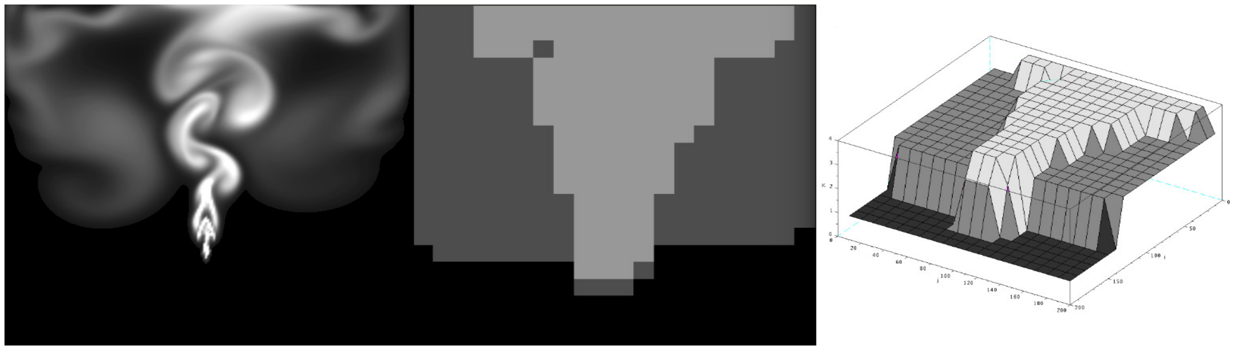


Fig. 3: Snapshot of the fluid simulation on the left, the corresponding grid state on the center and the volume of total computations (denoted by `k`) on the right.

simulation is shown in Fig. 3(left). Particle-based simulations and grid-based simulations are the most effective ways of simulating the behavior of fluids. Grid-based methods respond to the so-called Eulerian approach, where fluids are represented by fixed points in the space with information about the fluid in time and they are updated at every time step of the simulation. Grid based techniques often suffer from mass loss and are slower than particle based methods, but they usually have higher accuracy and better tracking of smooth fluid surfaces. They form a very suitable family of codes to apply ACR, because on one hand the simulation is an approximation of a physical phenomenon and on the other hand the processing is done on a highly regular computation space where each element requires complex computations.

A. Exploiting Domain-Specific Knowledge

In fluid simulation, the state of a fluid is typically represented by a velocity vector and a density value for every point in the space. The density in a given point represents the amount of fluid concentrated and the velocity vector represents the direction and intensity of the flow in that point. The evolution of the simulation is described by the Navier Stokes equations [19]. The simulation steps can be decomposed in Advection, Diffusion and External Forces influence. Advection is the phenomena that describes how velocity moves the fluid and other objects in the space along with the flow. Diffusion describes the resistance of a fluid to flow because of its viscosity. The influence of External Forces describe local or body forces applied to a specific region or all the fluid like a fan blowing air, gravity, etc. Density is carrying the pertinent information for efficient monitoring: precise computations should be done in regions where this value is high and conversely. This domain-specific knowledge is encoded for ACR through the `monitor` pragma to maximize the accuracy of the approximation.

B. Applying ACR

To apply ACR to the simulation, the grid state is filled according to the density values. The value of a grid cell is the level of the maximum density point in that cell. The specific target of the optimization by approximation is the portion of the simulation code dedicated to the diffusion phase. Diffusion computations corresponds to a significant part of the total computations of the simulation.

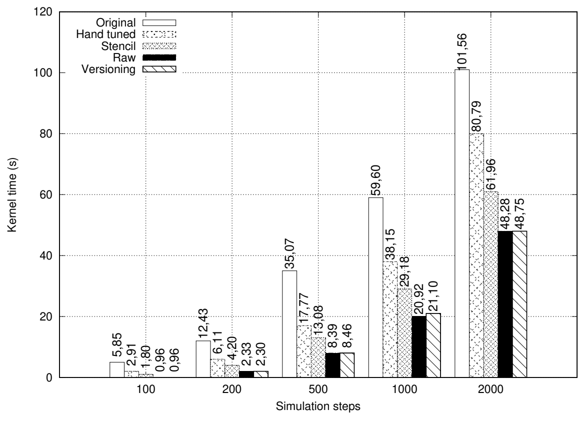
The diffusion phase is computed with a numerical iterative method to obtain a solution. The numerical algorithm gets better solutions the more iterations it does. The original code is programmed to do a fixed amount of iterations in the whole space. To do diffusion, the iterations of the numerical method are done one by one in the complete simulation space. That is because the particles need to know about the solution of its neighbors to compute the next approximation of its own solution. We have used the ACR approach to make the numerical algorithm perform less iterations while maintaining dependencies as much as possible on areas with little amount of fluid or no fluid at all. We have chosen to have 3 levels of complexity for the regions: the optimized algorithm will do the first basic iterations over the entire iteration step, then the other iterations over a more restricted region where there is more than a negligible amount of fluid, and will end doing more iterations only where there is a considerable amount of fluid that needs extra iterations to reach a good enough solution. A simplified excerpt of the corresponding code with the ACR pragmas is shown in Fig. 1.

V. Experimental Results

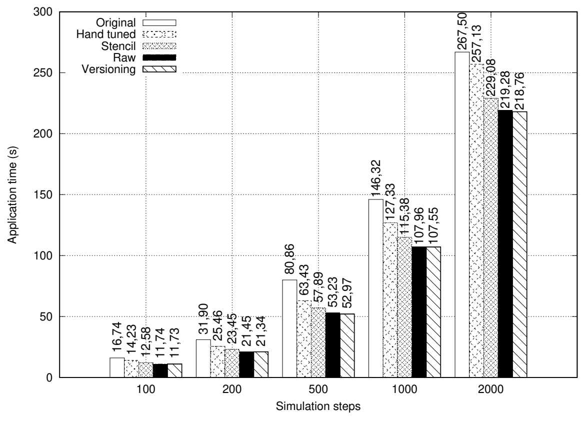
The experimental setup is a quad-core Intel Core 2 Quad Q6600 system with 4 GB of memory. All codes are compiled using GCC 6.2.1 with `-O3 -march=native` option (which builds the best performing original and hand-tuned versions). In addition to GCC, the compilation thread also uses the Tiny C Compiler 0.9.26 to minimize dynamic compilation time.

A. Eulerian Fluid Simulation

ACR [20] was evaluated against a single threaded implementation of a 2D and 3D Eulerian fluid simulation described by Stam [18] of 400×400 particles. We compared the approximate computation code relying on ACR with various grid updating policies (detailed in Section III-B) against the original code as well as a hand-tuned version that mimics the ACR strategy without dynamically generated code. We observe performance, computation savings and accuracy over



(a) Execution time for the ACR optimized kernel alone



(b) Execution time for the whole application

Fig. 4: Execution time for the optimized kernel and the complete fluid simulation application. `Original` is the original application; `Hand tuned` is a manually written version that mimics ACR without dynamically generated code; `Stencil`, `Raw` and `Versioning` are ACR versions with corresponding state grid post-processing policies (detailed in Section III-B).

a range of simulation iterations. During simulation, fluid is injected regularly in the iteration space together with a directional force to make it acquire velocity. The test cases have different types of regions with similar appropriate approximation policy and they evolve over time.

Overall execution time is reported in Fig. 4 at different steps of the simulation, for the original code, we compared a hand-tuned version with a dynamic code generation version with three different state grid post-processing strategies. The statically generated version is not appropriate with this benchmark because the data dependencies requires to use the original iteration ordering to avoid high deviation. Performance results after 1000 simulation steps show a speedup of 2.85 for the optimized kernel for ACR with `raw` state grid update strategy (resp. 2.82 for `versioning` and 2.04 for `stencil`) with respect to the original code and 1.82 (resp. 1.81 and 1.31) with respect to the manually optimized code. The `versioning` policy achieves equivalent performance and better precision compared to the `raw` policy: it requires 3% more computations but saves 43% code compilations on average.

It is worth noticing that the performance improvement is only due to the approximation strategy: the generated computation thread is sequential and no other polyhedral optimization has been applied. ACR is complementary to existing optimization techniques and will be composed with them in future work. The present paper only evaluates the benefits of "pure" ACR.

Computation savings of the diffusion part for a complete simulation step (where it is used 3 times) with respect to the original code are shown in Tab. I. Our metric for computation savings is the difference of the number of iterations of the original code (corresponding to the number of calls to the `lin_solve_computation()` in the pseudo-code in Fig. 1, i.e., $3 \times MAX \times N \times N$, with MAX set to the same value as the high precision ACR alternative) and the number of iterations actually executed by the computation thread. Results show very significant computation savings compared to the number of computations of the original diffusion, ranging from 30 to 74%.

TABLE I: Saved computations with ACR - `raw` policy

Iterations	100	200	500	1000	2000
% saved comput.	73.45	69.05	59.72	48.73	30.61

Accuracy results are shown in Tab. II. We measured the difference in density for every particle in fixed snapshots of both simulations, with the original application as reference. We measured both the mean and maximum difference of every particle. Values of the density fluid vary between 0 and 20. We observe that even after 1000 iterations and removing 77% of the computations, the mean difference for all particles is only a fraction of the maximum density value. The `stencil` policy shows the best precision property at the price of efficiency as shown in Fig. 4. Moreover, the maximum measured deviation is below 0.05 while in the application, two particles may be displayed using different colors only if they do not belong to the same plateau, each separated by 0.08 and starting at 0. Hence, those results show ACR has a limited impact on the simulation despite its aggressive computation savings.

B. Evaluation on Multiple Applications

Conway's Game of Life [21], belongs to the family of cellular automata. Those automata consist of a regular grid of cells, each having a finite number of states. The automaton

TABLE II: Precision results for ACR after 1000 iterations (density values range from 0 to 20)

Grid update policy	Raw	Versioning	Stencil
Average deviation	0.0017	0.0011	0.0003
Maximum deviation	0.0484	0.0464	0.0247

evolves using a set of rules in order to determine the state in which each cell will be at the next generation. Game of Life particular automata consists of a 2D space of cells being either *alive* or *dead*. This automaton rules follows: if we consider as neighbours the eight cells surrounding any individual cell, a previously *dead* cell becomes *alive* if there were exactly three neighbours *alive* at the previous generation. If the cell was already *alive*, it will survive to the next generation if it has exactly two or three *alive* neighbours. Otherwise the cell *dies* or stays *dead*.

The particularity of `Conway's Game of Life` is that it does not tolerate any approximation in the computation of the living cells however we can still optimize with smart computation savings and by getting rid of useless data accesses. Big cellular automatons contain usually a lot of "dead zones" where none of the cells are alive. This emphases a perfect way to save computation but we can even avoid accessing those regions. It is indeed impossible for those dead cells to become alive in a time that is lower than the distance separating them from a breathing cell. We use this property to only visit cells that can be potentially lit in the near future. We use the dynamic version with stencil in order to activate the neighbouring cells in advance. So only part of the domain filled with alive cells and their neighbouring are active at a given time. We are using the *alternative zero_compute* in the domain where these dead zones are present. We take advantage of the stencil policy as it enables computation on dead regions having active neighbour cells and anticipates their migration.

Experimental results shown in Tab. III is the result of running the algorithm on an existing automaton simulating a digital clock. Notice the increase in speedup over time. This is due to the application behaviour. It does indeed require less and less computation as the automaton evolves to form the first minute after midday (i.e. passing from hour 0:00 to 0:01). As this automaton is evolving, also does the grid representing its computations.

TABLE III: Game of Life application simulating a clock

Iterations	Original (s)	ACR (s)	Speedup	Deviation
20	51.1	36.5	1.4	None
40	94.49	57.1	1.7	None
80	181.3	99.0	1.8	None
160	355.0	181.7	1.9	None

K-Means Clustering [22] partitions multiple observations inside *clusters*. In order to group observations together we need a function $f(obs1, obs2)$ that computes the "distance" between the two observations. The higher this value is the less likely they are to be in the same cluster. The main goal is to pack observations together and to minimize each cluster sum of distance altogether. The problem is NP-hard but there exists a heuristic named "k-means algorithm" having two repeating steps. Supposing each cluster has a center and can be parameter of the previous distance function. The first step simply assigns all observations to the nearest cluster center and the second recomputes the cluster center as the mean of each observation position. The two steps repeat until a local minimum is reached.

For `K-Means Clustering` we took advantage of the following properties of the algorithm: at the beginning, the algorithm solution is very volatile and objects will change cluster really often, although after a small number of iterations some clusters seems to have already converged to a local minimum [23]. Thus only a small part of the objects will continue to change from one cluster to another, in particularly the one which are on the edge between two cluster zones. The input for this benchmark is essentially pictures. Pixels have the particularity that if they do switch from one cluster to another, chances are high that pixels nearby will do the same. For a maximum accuracy of the algorithm we used the stencil strategy in order to activate the clustering on neighbours pixels. We decided to skip the computation if the objects inside the cells have settled for more than a certain number of algorithm iterations. We implemented a version of the kernel that only reassigns pixels to their previous cluster and used the *alternative function* to specify how many times pixels have to stay inside the same cluster to switch to the new function.

The algorithm is fed with three different pictures, the first picture having 2560×1600 pixels representing a bird on a branch. The second is a picture taken by a telescope from a Nebula (3000×2785) and the last is a picture of a wolf (4288×2848). Those images are serialized and passed as a single entity to the algorithm. Tab. IV shows the result of this benchmark. We can notice that with ACR the number of iterations of the algorithm is reduced as we would expect because we locked some pixels in place.

TABLE IV: K-Means clustering on three images

Picture	Original	ACR	Speedup	Deviation
Bird	4.18s	3.52s	1.18	0.0073
Nebula	16.08s	13.05s	1.23	0.0001
Wolf	26.93s	17.53s	1.54	0.0429

Finite-difference Time Domain (`FDTD`) Electro-Magnetic Simulation [24] is intended to solve the Maxwell equations and simulate the propagation of electric and the induced magnetic field in space. `FDTD` is a grid-based model using finite-difference expressions to approximate the equation derivatives. The resulting expressions are solved in two steps, first by solving the electric field component and the next instant in time the magnetic field component.

To optimize this application we choose to skip the electric field computation where the magnetic field is relatively low.

We can do this because a low magnetic field does not induce any current and thus leaves the electric field intact. To achieve this, the ACR monitoring pragma will be pointed to the magnetic field array and the alternative "zero_compute" be used whenever the field is low enough. The algorithm is based on neighbour propagation of values, this is why we also used the stencil version for this benchmark.

The simulation runs in a 2D space with an obstacle in the middle. That will reflect the wave as it travels from the left to the right and back, creating more waves as it spreads. The results are shown Tab. V. This application requires high accuracy or the divergence elevates really quickly.

TABLE V: `FDTD` simulation of an electro magnetic wave bumping on an obstacle.

Iterations	Application time (s)		Speedup	Deviation
	`Orig`	`ACR`		
500	13.94	12.47	1.11	0.0000
1500	18.82	14.72	1.27	0.0000
3000	27.15	19.31	1.41	0.0017
5000	38.23	28.18	1.35	0.0099

C. Runtime overhead estimation

In order to evaluate the overhead of the ACR runtime we pre-generated, for each call to the target kernel, its optimized version with respect to precision. The simulation was then run with the exact same setup having the ACR runtime replaced with the pre-generated versions. This represents the best case scenario where we already have the perfect version ready when the kernel is called.

In Tab. VI we present the results obtained for `FDTD`. It is the simulation requiring the most kernel versions hence where the overhead is the highest. The results show a very small overhead compared to the optimal version. The ACR 1C column gives the time when both the ACR runtime and the computation thread are bound to the same CPU core. Those results support our multithreaded runtime approach that allows the computation thread to continue with the precise version while the code generation is done in the background.

TABLE VI: `FDTD` application time with the ACR runtime enabled on 1 and 3 CPU cores compared to the precomputed versions

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

D. Comparison Against Loop Perforation

We compared our technique with a simplified version of loop perforation [7]. Our simplifications removes the profiling step of loop perforation which is highly data dependent, for a better comparison to ACR which adapts to any input data. We tuned loop perforation to obtain the same speedup as ACR and compared the deviation of both approaches. Tab. VII shows ACR has a significantly lower deviation due its dynamic nature which selects the most pertinent iterations to approximate.

TABLE VII: Percentage of iterations saved and the data deviation compared to the original at constant iteration. The deviation is computed by taking the difference between two values belonging to the same spot of the simulation and expressing the ratio from the initial value and this difference.

Bench	Saved iterations		Deviation	
(Iterations)	`ACR`	`Perfor`	`ACR`	`Perfor`
2D Fluid (1000)	48.7	50.0	0.0011	1.024
3D Fluid (600)	52.2	50.0	0.0051	0.9847
FDTD (3000)	51.9	50.0	0.0000	6.1979

VI. Related Work

Compiler Techniques The idea of relaxing data dependences and skipping computation to trade accuracy for performance has been studied in the past in various ways. Loop perforation [25], [7] is a static technique which removes complete loop iterations selected with respect to a training phase. Contrary to loop perforation, ACR is a dynamic approach which may remove only selected parts of a given loop execution and is designed to produce more accurate results with end-user guidance. Our work shares the concept of high-level information including alternative implementations provided through pragmas with Green [6], however Green is based on an offline training to select the final code while ACR is continuously recomputing the best code. EnerJ [26] uses the type system to specify approximate variables to save energy. SAGE [8] is a GPU-oriented technique skipping or simplifying processing with respect to performance impact while we primarily focus on accuracy. HELIX-UP [5] ignores some dependences to enable code parallelization, which is complementary to our approach. Power savings is also studied by Misailovic et al. [27] who provide a language interface to rely on hardware providing approximate instruction and memory storage that draws less power but may produce a wrong result at a given rate.

Numerical Analysis Techniques ACR has been inspired by numerical analysis techniques, in particular Adaptive Mesh Refinement [9] which can maintain the consistency of a solution for a bounded error in the minimum possible amount of time in simulation problems. [28] extends the use of a multiscale analysis for grid adaptation to incorporate locally varying time stepping. Space filling curves are also used to transform multidimensional data for better parallelization of multiscale adaptation methods [29]. The differences between our approach and these works is the manipulation of the simulation space since they modify the shape of the simulation space at runtime. Its underlying iteration structure is a hierarchical grid which incorporates and looses points during the simulation. On the opposite, the ACR approach keeps the original iteration space intact while the generated code that achieves the computation may change dynamically.

VII. Conclusion

In this paper, we introduced Adaptive Code Refinement (ACR), a new compiler technique to improve performance at the price of accuracy. Starting from a reference program and high-level approximation information provided by the user, it generates automatically a program that continuously adapts the optimization strategy to the most appropriate one in regions of the computation space. ACR provides a unique set of features to offer performance, accuracy and flexibility. Performance is provided by building on state-of-the-art polyhedral code generation techniques to generate an optimized code and by exploiting multicore architecture with several specialized threads to minimize the runtime overhead. Accuracy is preserved as much as possible by relying on a dynamic strategy that achieves precise computation only when and where it matters and by preserving the original computation ordering. Finally, flexibility is achieved through a simple yet powerful set of pragmas to drive the approximation strategy, allowing the user to focus on a simple ideal computation kernel while the approximation is managed by the ACR system.

To evaluate ACR, we built its compiler extensions and runtime, and we applied it on several representative benchmarks from simulation and iterative algorithms. The experimental results demonstrate significant performance improvement with low accuracy deviation, with a minimal effort from the programmer. We showed that ACR outperforms a manual optimization that would mimic the same dynamic approximation but which cannot have an optimized control flow that can only be achieved by our runtime code generation. It is also more precise than simplified existing loop perforation technique since ACR is able to select relevant approximation at runtime. Finally, ACR is complementary to other optimizations such as polyhedral loop parallelization: our results only report improvements due to approximations with low control overhead.

ACR is a first step towards generic compiler-assisted generation of self-tuning applications. Many studies and extensions are possible to improve it, including ways to increase the flexibility of the grid, to make the technique even more automatic or to reduce the mechanism overhead. However it is clearly a very promising new way to explore for any application where approximation is either desirable or possible.

References

[1] P. Feautrier, "Some efficient solutions to the affine scheduling problem: one dimensional time," *Intl. Journal of Parallel Programming*, vol. 21, no. 5, pp. 313–348, october 1992.

[2] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *PLDI'08 ACM*, Tucson, USA, Jun. 2008.

[3] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative optimization in the polyhedral model: Part II, multidimensional time," in *PLDI'08*. Tucson, Arizona: ACM Press, June 2008, pp. 90–100.

[4] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet SIMD code generation," in *PLDI'13*, Seattle, USA, Jun. 2013, pp. 127–138.

[5] S. Campanoni, G. Holloway, G.-Y. Wei, and D. M. Brooks, "HELIX-UP: Relaxing program semantics to unleash parallelization," in *IEEE/ACM CGO*, San Francisco, USA, Feb. 2015, pp. 235–245.

[6] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *PLDI'10*, Toronto, Canada, Jun. 2010, pp. 198–209.

[7] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *European Conference on Foundations of Software Engineering*, Szeged, Hungary, Sep. 2011, pp. 124–134.

[8] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *MICRO'13 IEEE/ACM Intl. Symp. on Microarchitecture*, Davis, California, Dec. 2013, pp. 13–24.

[9] M. J. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics," *J. Comput. Phys.*, vol. 82, no. 1, pp. 64–84, May 1989.

[10] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, and R. Upadrasta, "Graphite two years after: First lessons learned from real-world polyhedral compilation," in *GCC Research Opportunities Workshop (GROW'10)*, Pisa, Italy, 2010.

[11] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Grösslinger, and L.-N. Pouchet, "Polly-polyhedral optimization in llvm," in *International Workshop on Polyhedral Compilation Techniques*, France, 2011.

[12] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[13] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *ETAPS CC'10*, ser. LNCS, Paphos, Cyprus, Mar. 2010, pp. 283–303.

[14] A. Venkat, M. Shantharam, M. Hall, and M. Mills Strout, "Non-affine extensions to polyhedral code generation," in *IEEE/ACM CGO'14*, Orlando, FL, USA, February 2014, pp. 185–194.

[15] S. Verdoolaege, "*isl*: An integer set library for the polyhedral model," in *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software*, Kobe, Japan, Sep. 2010, pp. 299–302.

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

[17] D. K. Wilde, "A library for doing polyhedral operations," *Parallel Algorithms and Application*, vol. 15, no. 3-4, pp. 137–166, 2000.

[18] J. Stam, "Real-time fluid dynamics for games," in *Proceedings of the Game Developer Conference*, 2003, p. 25.

[19] A. J. Chorin, "Numerical solution of the Navier-Stokes equations," *Mathematics of computation*, vol. 22, no. 104, pp. 745–762, 1968.

[20] M. Schmitt, "ACR compiler and runtime." [Online]. Available: http://gauvain.u-strasbg.fr/%7Eschmitt/acr

[21] M. Gardner, "Mathematical games: The fantastic combinations of John Conway's new solitaire game "life"," *Scientific American*, vol. 223, no. 4, pp. 120–123, 1970.

[22] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[23] J. Meng, S. Chakradhar, and A. Raghunathan, "Best-effort parallel execution framework for recognition and mining applications," in *IPDPS'09*. IEEE, 2009, pp. 1–12.

[24] A. F. Oskooi, D. Roundy, M. Ibanescu, P. Bermel, J. D. Joannopoulos, and S. G. Johnson, "Meep: A flexible free-software package for electromagnetic simulations by the FDTD method," *Computer Physics Communications*, vol. 181, no. 3, pp. 687–702, 2010.

[25] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using code perforation to improve performance, reduce energy consumption, and respond to failures," MIT, Tech. Rep. MIT-CSAIL-TR-2009-042, Sep. 2009.

[26] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *PLDI*, San Jose, California, USA, Jun. 2011, pp. 164–174.

[27] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels," in *ACM SIGPLAN Notices*, vol. 49, no. 10, 2014, pp. 309–328.

[28] S. Müller and Y. Stiriba, "Fully adaptive multiscale schemes for conservation laws employing locally varying time stepping," *J. of Scientific Computing*, vol. 30, no. 3, 2007.

[29] K. Brix, M. Silvia Sorana, S. Müller, and S. Gero, "Parallelisation of multiscale-based grid adaptation using space-filling curves," *ESAIM*, vol. 29, pp. 108–129, Dec. 2009.