Towards Automatic Significance Analysis for Approximate Computing

Vassilis Vassiliadis

CERTH & University of Thessaly,
Greece
vasiliad@uth.gr

Jan Riehme   Jens Deussen

RWTH Aachen University, Germany
{riehme, deussen}
@stce.rwth-aachen.de

Konstantinos Parasyris

CERTH & University of Thessaly,
Greece
koparasy@uth.gr

Christos D. Antonopoulos

CERTH & University of Thessaly,
Greece
cda@uth.gr

Nikolaos Bellas   Spyros Lalis

CERTH & University of Thessaly,
Greece
{nbellas,lalis} @uth.gr

Uwe Naumann

RWTH Aachen University, Germany
naumann@stce.rwth-aachen.de

Abstract

Several applications may trade-off output quality for energy efficiency by computing only an approximation of their output. Current approaches to software-based approximate computing often require the programmer to specify parts of the code or data structures that can be approximated. A largely unaddressed challenge is how to automate the analysis of the significance of code for the output quality.

To this end, we propose a methodology and toolset for automatic significance analysis. We use interval arithmetic and algorithmic differentiation in our profile-driven yet mathematical approach to evaluate the significance of input and intermediate variables for the output of a computation.

Our methodology effectively matches decisions of a domain expert in significance characterization for a set of benchmarks, and in some cases offers new insights. Evaluation of the software infrastructure on a multicore x86 platform shows energy reduction (from 31% up to 91% with a mean of 56%) compared to fully accurate execution, with graceful quality degradation.

Categories and Subject Descriptors   F.3.2 [*Logics and meanings of programs*]: [Program analysis];   G.1.0 [*Numerical analysis*]: General—Interval arithmetic;   G.1.4 [*Numerical analysis*]: Quadrature and numerical differentiation—Automatic differentiation

General Terms   Algorithms, Performance, Experimentation

Keywords   Significance analysis, Approximate computing, Energy efficiency

1.   Introduction

Big data applications, video analytics and scientific computing require computing capacity and energy efficiency beyond what is currently possible. Even if future applications have enough parallelism to exploit hundreds or thousands of cores, system performance will be restrained due to extreme power dissipation, hitting again the same power wall as single core architectures did more than ten years ago [16].

Part of the problem of energy inefficiency is that computing systems execute programs under the assumption that all code is equally significant for the quality of output. However, in several application domains it is not the precise result that matters to the user, but rather an approximation [11, 14].

Approximate computing is an emerging paradigm that allows trading-off performance and energy efficiency with accuracy [4, 33, 42]. It is based on the premise that specific phases of a computation may incur a high performance and energy toll without a corresponding contribution to the quality of the result. Approximation requires additional programming effort to specify which code segments are approximable and under which circumstances. Application developers also need to provide light-weight code versions, which produce the approximate results in each case. Typically, this kind of analysis is the product of a developer and domain expert collaboration.

This work is a step towards automatic significance analysis. We introduce the `dco/scorpio` framework which provides a quantitative assessment of the contribution of each

block of code to the quality of the final result. Intuitively, if an input or intermediate variable x is perturbed within an interval, and, as a result, the variation of the value of an output variable y is small, then x is insignificant to y. We employ interval analysis [24] and algorithmic differentiation [15, 26] to automatically quantify the significance of computations and to detect variations in significance among parts of code. Based on this analysis, the developer can then use our OpenMP-like task-based programming model to structure the computation in distinct tasks, specify their significance and provide faster, approximate versions for them.

We validate our method on five benchmarks from different domains, including video compression, imaging, scientific computing and financial engineering. We apply our methodology and execute the benchmarks at varying levels of approximation while recording energy gains and quality degradation. Our framework achieves energy reduction from 31% up to 91% with a mean of 56% when executing on a multicore x86 platform, by exploiting significance and approximations to produce acceptable results.

In summary, we make the following contributions: (i) We introduce an automated significance analysis methodology and toolset to rank computations in a program according to their contribution to the quality of program output. (ii) We integrate this significance ranking to a task-based programming model. (iii) We apply the methodology to a set of benchmarks from a variety of application domains to demonstrate its effectiveness.

The rest of the paper is organized as follows. Section 2 details the theoretical underpinnings of computational significance analysis and introduces the tool used to compute significance. Section 3 provides a description of the program development workflow and corresponding programming model. Section 4 discusses the experimental evaluation of our framework. We explore related work in Section 5. Section 6 concludes the paper.

2. Significance Analysis

In this section we introduce our hybrid (profile driven, yet mathematical) analysis and focus on the mathematical background of our definition of significance. For a given input range, significances of all intermediate computations for the final result can be obtained with a single analysis run. We assume that the application is given in the form of C++ code. We also asume that its execution trace can be abstracted as a differentiable function $f : I\!\!R^n \longrightarrow I\!\!R$ that computes a scalar output $y \in I\!\!R$ by evaluating $y = f(\mathbf{x})$ (e.g. running the code) for a given input vector $\mathbf{x} = (x_0, \ldots, x_{n-1})^T \in I\!\!R^n$. Generalization to vector functions is straightforward and is discussed briefly at the end of the section.

```
1  double f( double x0 ) {
2     return cos(exp(sin(x0) + x0) - x0);
3  }
```

Listing 1: Implementation of example function.

```
1   u0 = x0
2   u1 = sin(u0)       u2 = u1 + u0       u3 = exp(u2)
3   u4 = u3 − u0       u5 = cos(u4)
4   y  = u5
```

Listing 2: Elementary functions of `Listing 1`.

2.1 Significance as an Algorithmic Property

For input $\mathbf{x} \in I\!\!R^n$, the arithmetic evaluation of $y = f(\mathbf{x})$ given as a computer program can be written as a three part evaluation procedure [15] with internal variables u_j:

$$u_{k-n+1} = x_k, \qquad k = 0, \ldots, n-1, \qquad (1)$$
$$u_j = \phi_j(u_i)_{i \prec j}, \qquad j = 1, \ldots, p, \qquad (2)$$
$$y = u_p. \qquad (3)$$

The input values $\mathbf{x} = (x_0, \ldots, x_{n-1})^T$ are copied in Eq. 1 to internal variables $u_{1-n}, \ldots, u_0$, and Eq. 3 stores the final result in y. In Eq. 2 $\phi_j(u_i)$ represents an arithmetic operation (+,-,*,/), or an intrinsic function (sin, cos, exp, ...) of C++. Internal variables u_j are used to store the result of each elementary function evaluations, where $i \prec j$ denotes a direct dependence of u_j on u_i, $i < j$.

The different elementary functions ϕ_i are provided by the compiler run-time library[1]. We exploit the overloading capability of C++ to give these operations a new meaning with specially designed data types. Note that splitting complex expressions into elementary operations is also done by compilers, while internal variables are usually stored in processor registers.

The evaluation of the compiled program for a specific input $x \in I\!\!R^n$ leads to a unique sequence of elementary operations since all control flow decisions (branches, number of loop iterations) are determined uniquely. This sequence of elementary operations performed during such an evaluation is the basis of the three part evaluation procedure.

Listing 1 shows a concrete example for C++ function $y = f(x0)$ with scalar input x_0 and scalar output y. The respective code, broken down to single elementary functions, is given in *Listing 2*, where the first line corresponds to Eq. 1, the sequence of elementary functions in Eq. 2 is represented by lines 2–3 and line 4 corresponds to Eq. 3.

To evaluate the significance of variables u_j, $j = 1 - n, \ldots, p$, for the final result $y = u_p$, we need to answer two questions: (a) What is the influence of inputs $\mathbf{x}$ on u_j, and (b) what is the influence of u_j on output $y = u_p$. Answering (a) requires analysing the code which computes the variable u_j from its inputs $u_1, \ldots, u_{j-1}$ (Eq. 2). The answer to (b) requires analyzing the use of u_j in obtaining the final result y, through the computation of $u_{j+1}, \ldots, u_p$. We denote this

---

[1] They are side-effect free and take one or two arguments.

second part with $y = f(\mathbf{x}; u_j)$ to represent the dependency of y on u_j explicitly.

Interval arithmetic (IA) [24] is an appropriate tool to answer the first question: Given the range of possible input values as the input interval vector $[\mathbf{x}] = [\underline{\mathbf{x}}, \overline{\mathbf{x}}] = \{\mathbf{x} \in I\!R^n | \underline{\mathbf{x}} \le \mathbf{x} \le \overline{\mathbf{x}}\}$ with lower bound $\underline{\mathbf{x}} \in I\!R^n$ and upper bound $\overline{\mathbf{x}} \in I\!R^n$, an evaluation of f in interval arithmetic is obtained by replacing all variables and elementary functions ϕ_j with their interval version in Eq. 1-3:

$$[u_{k-n+1}] = [x_k], \qquad k = 0, \ldots, n-1, \quad (4)$$

$$[u_j] = \phi_j[u_i]_{i \prec j}, \qquad j = 1, \ldots, p, \quad (5)$$

$$[y] = [u_p]. \quad (6)$$

This will compute an interval enclosure $f[\mathbf{x}]$ of all possible values of $f(\mathbf{x})$ for $\mathbf{x} \in [\mathbf{x}]$, namely $f[\mathbf{x}] \supseteq \{f(\mathbf{x}) | \mathbf{x} \in [\mathbf{x}]\}$.

With IA, value ranges are propagated *forward* through the computation. For every variable $u_j, j = 1, \ldots, p$, we calculate an enclosure $[u_j]$ of all possible values for the given input range $\mathbf{x}$. The impact of all inputs $[x_k], k = 0, \ldots, n-1$, on a variable u_j is combined in $[u_j]$, and can be quantified by the width $w([u_j]) = \overline{u_j} - \underline{u_j}$ of interval $[u_j]$: if $w([u_j])$ is narrow, variation of the input within the given range causes only little variance in $[u_j]$ (small influence). On the other hand, if $w([u_j])$ is wide, the exact value of the input has large influence on variable u_j.

But the significance of u_j for the output y cannot be judged from this information alone. Subsequent operations during the evaluation of $y = f(\mathbf{x}; u_j)$ by computing u_k, $k = j + 1, \ldots, p$, may amplify or reduce the contribution of u_j to y. Therefore, we also need to evaluate how much y will change for different values of u_j, or more formally, to quantify the impact of $[u_j]$ on y (question (b) above). Pure IA evaluation of $f(\mathbf{x}; u_j)$ does not suffice, since in the final interval value $[y]$ the individual impact of variables $[u_j]$ cannot be obtained separately. Our answer to question (b) is inspired by the fact that the first order derivative of a differentiable function at a given point describes the function behavior in a neighborhood of that point. For a given point $\hat{x} \in I\!R^n$ and a function $f : I\!R^n \longrightarrow I\!R$ differentiable at point $\hat{x}$, $\nabla_{\mathbf{x}} f(\hat{x}) = (\frac{\partial f(\hat{x})}{\partial x_0}, \ldots, \frac{\partial f(\hat{x})}{\partial x_{n-1}})^T$ is the gradient of f at point $\hat{x}$. The elements of gradient $\nabla_{\mathbf{x}} f(\hat{x})$ quantify the rate of change in the function value near $\hat{x}$: if the absolute value of the partial derivative $\frac{\partial f(\mathbf{x})}{\partial x_i}$ is small, a disturbance in x_i will cause a small change in the function value $f(\mathbf{x})$.

Consider a computer program implementing a differentiable function $y = f(\mathbf{x})$. The gradient $\nabla_{\mathbf{x}} f$ of f at the evaluation point $\mathbf{x}$ can be obtained by *Algorithmic Differentiation (AD)* [15, 26] in adjoint mode. Based on the three part evaluation procedure Eq. 1-3, adjoint evaluation propagates the first order adjoint (denoted by subscript $_{(1)}$) $y_{(1)}$ of output y *backwards* through the computation towards first order

```
1   u_(1)5 = y_(1)
2   u_(1)4 = − sin(u_4) · u_(1)5          u_(1)3 = 1 · u_(1)4
3   u_(1)2 = exp(u_2) · u_(1)3            u_(1)1 = 1 · u_(1)2
4   u_(1)0 = (−1) · u_(1)4 + 1 · u_(1)4 + cos(u_0) · u_(1)1
5   x_(1)0 = u_(1)0
```

Listing 3: Adjoint code for *Listing 2*.

adjoints $x_{(1)k}, k = 0, \ldots, n-1$, of the inputs $\mathbf{x}$:

$$u_{(1)p} = y_{(1)}, \quad (7)$$

$$u_{(1)i} = \sum_{j:i \prec j} \frac{\partial \phi_j(u_i)_{i \prec j}}{\partial u_i} \cdot u_{(1)j}, \quad i = p-1, \ldots, 1-n, \quad (8)$$

$$x_{(1)k} = u_{(1)k-n+1}, \qquad k = 0, \ldots, n-1, \quad (9)$$

where $\frac{\partial \phi_j(u_i)_{i \prec j}}{\partial u_i}$ denotes the partial derivative of elementary function ϕ_j with respect to its argument u_i. After a single adjoint propagation with $y_{(1)} = 1$ the gradient $\nabla_{\mathbf{x}} y = \nabla_{\mathbf{x}} f = (x_{(1)0}, \ldots, x_{(1)n-1})^T$ can be harvested from the adjoints $u_{(1)k-n+1}, k = 0, \ldots, n-1$, of input $\mathbf{x}$. Moreover, derivatives $\nabla_{u_j} y = \frac{\partial f(\mathbf{x}; u_j)}{\partial u_j} = u_{(1)j}$ of y with respect to all internal variables $u_j, j = 1, \ldots, p$, are accumulated during the so-called reverse sweep.

Note that adjoint propagation expects that the original code (*Listing 2* for the example in *Listing 1*) has been evaluated beforehand. Thus intermediate variables $u_j, j = 1 - n, \ldots, p$, hold the actual values. The first line in *Listing 3* corresponds to Eq. 7, while the gradient harvesting of Eq. 9 is represented by the last line. The actual adjoint propagation (Eq. 8) is done in lines 2–4.

AD can be applied to interval functions [34] by replacing all variables and partial derivatives of elementary functions in Eq. 7-9 with their interval version. Therefore, we can compute an interval enclosure of the first order derivative $\nabla_{[u_j]}[y] = \frac{\partial f[\mathbf{x}; u_j]}{\partial [u_j]}$, namely the derivative of the function result $[y]$ with respect to the internal variable $[u_j]$:

$$\nabla_{[u_j]}[y] \supseteq \left\{ \frac{\partial f(\mathbf{x}; \hat{u}_j)}{\partial u_j} \middle| \hat{u}_j \in [u_j], x \in [x] \right\}. \quad (10)$$

In other words, the bounds of interval derivative $\nabla_{[u_j]}[y]$ are the steepest downward and upward slopes, respectively, of $y = f(\mathbf{x}; u_j)$ in the interval $[u_j]$, which quantify the impact of all possible values from $[u_j]$ on the final result y.

We can now define the significance $S_y(u_j)$ of variables $u_j, j = 1 - n, \ldots, p$, for the final result $y = f(\mathbf{x})$ over the input interval $[\mathbf{x}]$ as follows:

$$S_y(u_j) = w\left([u_j] \cdot \nabla_{[u_j]}[y]\right), \quad j = 1 - n, \ldots, p. \quad (11)$$

Note that the interval product of $[u_j]$ and the interval derivative $\nabla_{[u_j]}[y]$ is a worst case scenario, that might introduce a considerable overestimation of the significance of u_j.

2.2 Limitations

This approach comes with some limitations. A simple transformation of code with real variables into an interval version

184

might fail for various reasons (overestimation due to wrapping effect, special interval algorithms required, relational operators). Moreover, AD computes derivatives for a given evaluation point. The elementary function sequence in the implementation of function f is fixed and can be represented by a control flow free code. With IA, comparisons between values is no longer unique: for $c < [x]$ with $c \in [x]$, the answer is neither true nor false, since a part of interval $[x]$ is less than c whereas the remaining part is not. Since a fixed control flow is not guaranteed, in such scenarios the analysis is terminated and the relevant condition statement is reported to the user. Circumventing this issue by an automatic interval splitting approach is part of ongoing research.

Our method allows the developer to utilize all language tools including arrays, dynamically allocated memory, pointers, and nested loops. However, the analysed code must be differentiable, which might not apply to codes universally. Currently we consider that it is the responsibility of the developer to check the differentiability of the code/function to be analyzed.

2.3 dco/scorpio Framework

The significance analysis of Section 2.1 is implemented in the profile-driven tool `dco/scorpio` which is based on the template class library `dco/c++` (Derivative Code by Overloading in C++) [20, 27, 37] implementing tangent-linear and adjoint Algorithmic Differentiation. For any C++ code implementing $y = f(\mathbf{x})$, `dco/c++` exploits overloading of operators and intrinsic functions to compute derivatives $\nabla_{\mathbf{x}} y = \nabla_{\mathbf{x}} f(x)$ of outputs y with respect to input $\mathbf{x}$.

For the purpose of significance analysis, `dco/c++` templates were specialized with an interval base type [19] to obtain *dco::ia1s::type*, which enables AD on interval functions. An interval enclosure of $[y] = f[\mathbf{x}]$ can be obtained for a C++ implementation of $f(\mathbf{x})$ by defining all variables, required to compute the output y (including y), as *dco::ia1s::type* instances (*Listing 4*, compare to *Listing 1*) and performing a profile run. To compute the interval valued first order derivative $\nabla_{[x]}[y] = \nabla_{[x]} f[x]$, the `dco/scorpio` internal recording mechanism needs to be activated. During the evaluation of the code $f[\mathbf{x}]$ (with variables of type *dco::ia1s::type*), an internal representation of the computation sequence is stored within a Dynamic DFG (DynDFG). A DynDFG is a directed acyclic graph $G = (V, E)$, where each vertex u_j corresponds to a dynamically executed elementary function $u_j = \phi_j(u_i)_{i \prec j}$, and an edge $e_{i,j} \in E$ between vertex u_i and u_j means that u_i provided an input operand to operation ϕ_j during execution. Moreover, the edges are annotated with interval valued partial derivatives $\frac{\partial \phi_j[u_i]}{\partial [u_i]}$ which are computed during forward sweep.

Figure 1a shows the annotated DynDFG of an interval evaluation of the example function given in Listing 1 by the implementation given in *Listing 4*.

```
1  dco::ia1s::type f( dco::ia1s::type x0 ) {
2      return cos(exp(sin(x0) + x0) - x0);
3  }
```

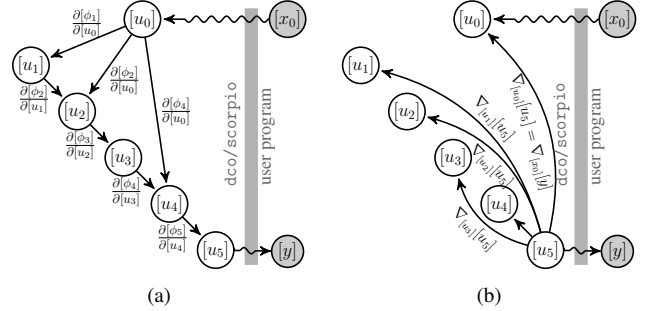Listing 4: The example of Listing 1 with *double* being replaced with *dco::ia1s::type*.

Figure 1: Annotated DynDFG and adjoint propagation : (a) DynDFG of $f(x)$ with local partial derivatives. (b) Derivatives available after evaluating $\nabla_{[x]}[y]$.

The scalar input $[x_0]$ of the user code will be associated with the internal variable $[u_0]$. Five internal variables $[u_j] = \phi_j[u_i]_{i \prec j}$, $j = 1, \ldots, 5$, are computed before the final value is stored in the output value $[y]$ of the user code. Edges towards a vertex $[u_j] = \phi_j[u_i]_{i \prec j}$ are annotated with local partial derivatives of the operation represented by $[u_j]$ with respect to its arguments $[u_i]$, $i \prec j$. Note that the interval operations, DynDFG recording, and adjoint propagation are hidden within the data type *dco::ia1s::type* of dco/scorpio.

With first order adjoint mode, AD derivatives are computed by propagating an initial adjoint $y_{(1)} = 1$ backwards through the DynDFG using the internally stored local partial derivatives according to Eq. 8. After the adjoint interval propagation (reverse sweep), the interval derivative $\nabla_{[x_0]}[y] = \nabla_{[x]} f[x_0] = \nabla_{[u_0]}[u_5]$ can be retrieved from dco/scorpio's internal representation along with the interval derivatives $\nabla_{[u_j]}[u_5] = \nabla_{[u_j]}[y]$ of the final result with respect to *all* internal variables $[u_j]$, $j = 1, \ldots, 5$ (*Figure 1b*). Using this information and Eq. 11 we can compute significances $S_{[y]}[u_j]$ of all variables $[u_j]$, $j = 0, \ldots, 5$.

`dco/scorpio` offers a set of macros (*Table 1*) to annotate source code for significance analysis. They establish a link between variables in the code and their internal representation in the tool and hide all implementation details from the user. All inputs need to be registered before the first intermediate user variable, intermediate user variables need to be registered straight after their computation, and output variables last. Moreover, for a vector valued function $\mathbf{y} = F(\mathbf{x})$ with $F : \mathbb{R}^n \longrightarrow \mathbb{R}^m$, significances $S_{\mathbf{y}}(u_j) = \sum_{i=0}^{m-1} S_{y_i}(u_j)$ can be obtained by a single run by registering all output $\mathbf{y}$ variables. .

| Macro	Description
INPUT(x,xl,xu, …)	Register input variable x, increment the number of inputs n, set $[x] = [xl, xu]$, associate x with the internal input variable $[u_{-n}]$.
INTERMEDIATE(z, …)	Register intermediate variable z, associate it with last computed internal variable $[u_j]$.
OUTPUT(y, …)	Register output variable y, associate it with the last computed internal variable $[u_p]$, set the adjoint $[u_p]_{(1)} = 1$,
ANALYSE()	Start adjoint propagation to obtain $\nabla_{[u_j]}[y], j = p - 1, \ldots, 1 - n$, compute significance of all registered inputs and intermediate user variables, report the significance of registered variables.

Table 1: Macros of the `dco/scorpio` tool
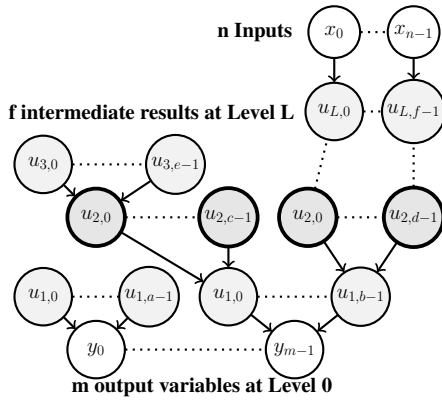
Figure 2: Dynamic DFG (DynDFG) of the application

3. Workflow for Systematic Significance Driven Programming

In this section we show how significance analysis is systematically used to guide the programmer towards source code annotation to expose significance information at the level of a task. We will be using a Maclaurin Series (*Listing 5*) as a running example to illustrate all aspects of the workflow:

$$f(x) = \sum_{i=0}^{n} x^i \approx \frac{1}{1-x}, x \in (-1, 1). \qquad (12)$$

3.1 Significance Analysis Framework

Our methodology is shown in Algorithm 1. Starting from the application source code we produce a simplified *Dyn-DFG* along with significance information for all nodes. An example *DynDFG* is shown in *Figure 2*. We use the notation introduced in the previous section. Nodes at the top are mapped to input vector **x**, leaf nodes at the bottom to the output vector **y** and the remaining nodes correspond to intermediate variables u_j.

Steps *S1* and *S2* identify the output and input data of the algorithm, respectively, and register the input data ranges. To annotate Maclaurin Series (*Listing 6*) we register **x** as the input data and set its value width equal to 1 ($\mathbf{x} \in [x - 0.5, x + 0.5]$ as seen in line 5). Step *S3* invokes the analysis

---

Algorithm 1: Significance Analysis Framework (dco/scorpio)

Input : Application source code
Output: G_{out} (*DynDFG*) along with significance tags

S1: $\mathbf{y} = (y_0, \ldots, y_{m-1})^T$
S2: $\mathbf{x} = (x_0, \ldots, x_{n-1})^T$
S3: $G = \texttt{dco/scorpio}(\mathbf{x}, \mathbf{y})$
S4: $G_s = simplify(G)$
S5: $G_{out} = findSgnfVariance(G_s)$

```
def findSgnfVariance(DynDFG G) {
    for ( L = 1; L<G.height; ++L )
        if ( SgnfVariance(L) >δ )
            call G.removeAbove(L+1)
            break
    return G
}
def simplify(DynDFG G) {
    for (L=0; L<G.height; ++L)
        nodes = G[L]
        foreach( v in nodes)
            inputs = v.InputDeps()
            call simplifyDep(G, inputs, v)
    return G
}
def simplifyDep(DynDFG G, nodes, parent) {
    foreach (v in nodes)
        if( v.AntiDependent(parent) )
            v.Parent = parent
            call simplifyDep(G, v.InputDeps(), parent)
        else
            call G.SetDependency(v, parent)
            call simplifyDep(G, v.InputDeps(), v)
}
```

---

toolset described in the previous section to produce a graph following the format shown in *Figure 2*. Each node u_j is annotated with the significance value of the corresponding intermediate variable to the output.

Step *S4* post-processes the graph produced by the significance analysis tool to eliminate internal nodes that express anti-dependencies such as: $res = res + term[i]$. These operations aggregate results, and are not really part of the computation. We illustrate these aggregation nodes using a darker color in *Figure 3*. Disregarding them is important for the next step *S5*, which is the main step of Algorithm 1.

Step *S5* traverses G_s using Breadth First Search (BFS) starting from the output nodes at level $L = 0$ and moving towards the input nodes, to construct G_{out}. The algorithm detects the level L whose nodes have significance values with statistical variance higher than δ.

Intuitively, when we detect nodes with high statistical variance in their significance values, we have reached a level in the *DynDFG* which can be used to partition the code into tasks of different significance. Nodes $u_{L,k}$ with high signif-
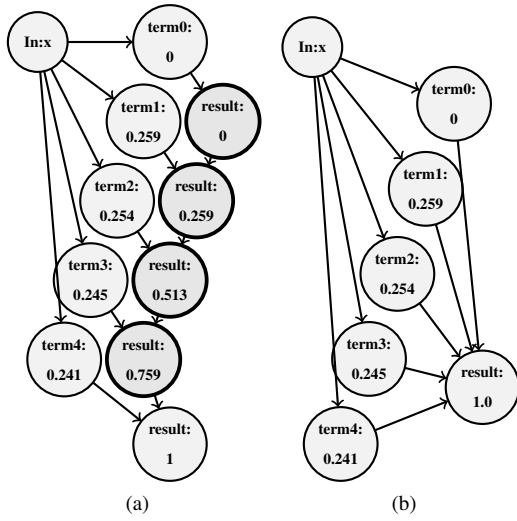
Figure 3: Figure (a) illustrates the Graph containing the significance values of the elemental computations as produced by `dco/scorpio` during *S3* and (b) The simplified graph after *S4* for the Maclaurin Series example.

icance for the program outputs **y** can be made to correspond to output variables of significant tasks, and the programmer can restructure the code around this information. On the other hand, if the algorithm terminates at the inputs **x** of the code without detecting any significance variations, it is guaranteed that nodes which reside in the same level are (almost) equally important. Parameter δ is dependent on application characteristics and the sensitivity to significance variations required by the programmer. When the analysis terminates, the method produces a *DynDFG* containing nodes up to level $L + 1$ with their significance value.

In *Figure 3* we show the *DynDFGs* produced in steps *S3* and *S4* for the Maclaurin Series, respectively. *Figure 3b* is also the result of *S5*. The last step of our analysis terminates at $L = 1$ since there are large variations between node significances at this level. Note that, the first term has a significance of 0. Note that a significance value of 0 means that the related computation can be substituted by a constant value which is 1 in this case, since $pow(x, 0) = 1$. The most significant term is the second one and every term computed afterwards is less significant than the one before it.

3.2 Programming Model for Code Significance

The next step of the workflow is to restructure the application into distinct tasks of work. The nodes of graph G_{out} at level L are the outputs of those tasks, where L denotes the task granularity according to which the programmer restructures the code. The developer uses this information to approximate the least significant intermediate results of level $L + 1$ to create approximate implementations of tasks that produce elements of level L. In the case of Maclaurin Series, `dco/scorpio` returns different significance for terms

```
1  double maclaurin_series(double x, int N)
2  {
3      double result = 0.0;
4
5      for (int i=0; i<N; ++i )
6      {
7          double term = pow(x, i);
8          result += term;
9      }
10
11     return result;
12 }
```

Listing 5: The Maclaurin Series original implementation.

```
1  double maclaurin_series(dco::ia1s::type x, int N)
2  {
3      dco::ia1s::type result = 0.0;
4
5      INPUT(x, x-0.5, x+0.5);
6      for (int i=0; i<N; ++i )
7      {
8          dco::ia1s::type term = pow(x, i);
9          result = result + term;
10     }
11
12     OUTPUT(result);
13     ANALYSE();
14
15     return result.toDouble();
16 }
```

Listing 6: Listing 5 enhanced with `dco/scorpio` macros.

```
1  void task(double *term, double x, int pos) {
2      *term = pow(x, pos);
3  }
4
5  void approx(double *term, double x, int pos) {
6      *term = pow_fast(x, pos);
7  }
8
9  double maclaurin(double x, int N, double waitRatio
       ) {
10     double temp[N], result = 0.0;
11
12     temp[0] = 1.0;
13     for (int i=1; i<N; ++i ) {
14         double significance = (N-i+1)/(double)(N+2);
15         #pragma omp task significance(significance) \
16           approxfun(approx) in(x, pos) out(temp[i:i]) \
17           label(maclaurin)
18             task(temp+i, x, i);
19     }
20     #pragma taskwait label(maclaurin) ratio(waitRatio)
21
22     for (int i=0; i<N; ++i)
23         result = result + temp[i];
24     return result;
25 }
```

Listing 7: The Maclaurin Series implementation after the analysis performed by `dco/scorpio`.

computed at *Listing 5*, line 7. The developer inspects G_{out} (*Figure 3b*) to identify tasks which compute a term.

We extend the latest version of OpenMP [29]. Both parallelism and significance are expressed via *#pragma* directives. Tasks are specified using the *#pragma omp task* directive (*Listing 7*, lines 15 to 17), and their significance is specified via the *significant()* clause. Depending on their significance, tasks may be approximated or dropped at runtime.

Benchmark	Domain	Lines of Code				
		Sequential	Parallel (P)	Approx. Function (A)	Significance clause (S)	Overhead(%)
Sobel Filter	Image Filter	143	174	35	1	20.7%
DCT	Multimedia	157	198	0	1	$\approx 0\%$
Fisheye	Multimedia	457	500	94	1	19%
N-Body	Physics	560	574	88	2	15.7%
BlackScholes	Finance	278	286	89	1	31.5%

Table 2: Lines of code of the sequential and parallel (task-based) version of the benchmark applications, and the extra code added to support significance-driven execution via our programming model. The overhead of the extra code is given relative to the parallel code $\frac{A+S}{P}$. Note that the approximations have been implemented by copying the original code and performing modifications which reduce its computational complexity as hinted by our analysis method.

The programmer may provide alternative, approximate task implementations, through the *approxfun()* clause. Task input and output are specified via the *in()* and *out()* clauses. Finally, *label()* can be used to group tasks under a common identifier to allow for task-group level synchronization.

Explicit barrier-like synchronization is supported via the *#pragma omp taskwait* directive. It can serve as a global barrier, or a barrier for a specific task group using the *label()* clause. *taskwait* can be used to control the minimum quality of application results. The *ratio()* clause instructs the runtime to execute (at least) the specified percentage of tasks accurately, while *respecting* task significance; more significant tasks should be executed accurately. The ratio serves as a single knob to *enforce* a minimum quality in the quality / performance-energy optimization space.

We now revisit the Maclaurin Series example, according to the analysis results depicted in *Figure 3b*. The algorithm is ported to our task-based programming model (*Listing 7*) so that each task computes a single term according to the analysis shown in *Figure 3b*. Noticing the monotonicity of the term significance values we choose to use the function in line 14 to communicate the significance of tasks to the runtime system. Approximations of the task significance values (e.g. via interpolation) may be used, with no penalty, as long as they capture the significance ranking of the tasks.

For tasks which produce multiple outputs, task significance corresponds to the maximum significance of its outputs. Finally, the developer provides approximate versions of the tasks (*Listing 7*, lines $5 - 7$), involving an approximate version of *pow()*; the latter offers higher performance and energy efficiency at the expense of lower precision.

4. Experimental Evaluation

We evaluate our approach with five applications, consisting of six different computational kernels. We intentionally include well-known kernels in our benchmark set, so that we can validate the results of the significance analysis. For each kernel we follow Algorithm 1, to restructure the kernel code in tasks, and come up with approximate versions. We evaluate the effectiveness of our significance tagging and approximation choices by studying the output quality and perfor-

mance of these kernels, while varying the percentage of tasks that are executed accurately. Then we compare our method with loop perforation [35], a compiler technique for skipping selected loop iterations. *Table 2* outlines our benchmarks and shows the overhead in terms of lines of code with respect to a task-based, implementation of the benchmarks.

4.1 Benchmarks

4.1.1 Sobel Filter

Sobel Filter is a 2D filter for edge detection in images which convolves the image with a 3x3 block filter, once in the horizontal and once in the vertical direction. Afterwards, it combines the results (t_x and t_y) of the two convolutions to compute an intermediate value $t = \sqrt{t_x^2 + t_y^2}$. The output pixels are then produced by clipping t to the range of $[0, 255]$.

For the significance analysis we use a set of images used in image compression benchmarking [5]. The analysis indicates that the first level of high variance between the significance of computations in the *DynDFG* is the one during which the convolutions take place. Three distinct blocks of computations are identified. The first one (A) uses the filter coefficients 2 and -2 and the other two (B, and C) use 1 and -1. The analysis shows that A is twice as significant as the other two. Finally, the computations which aggregate convolutions results and produce output pixels show little significance variance across all pixels.

Based on these results, we implement Sobel Filter using two groups of tasks. The first group consists of three types of tasks, one for each part of the convolution kernel. We approximate the tasks by dropping the respective computation. We set the significance of tasks computing A to 1.0, thereby forcing accurate execution regardless of the requested $ratio$ of computations. Tasks B and C receive a significance of 0.5; since one third of the tasks are significant (tasks A), tasks B and C will only be executed if the user requested $ratio$ is higher than 0.33. The second task group uses the results computed by the first one to produce the output image pixels. These tasks always execute accurately, since the analysis indicates that they have high significance for the output.
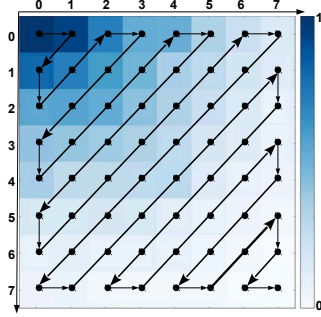
Figure 4: The DCT significance mapped on the 8x8 block of DCT coefficients. The top left corner has the highest value and drops in a wave-like pattern towards the opposite corner.
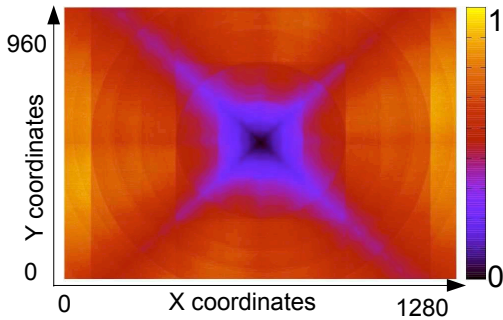

Figure 5: Significance values of the *InverseMapping* kernel.

4.1.2 Discrete Cosine Transformation

Discrete Cosine Transformation (DCT) is a module of video compression kernels, which transforms a block of 8x8 image pixels to a block of 8x8 frequency coefficients. Low frequency coefficients are closer to the upper left corner of the 8x8 block, whereas high frequency coefficients reside in the lower right corner.

Analysis reveals a variation in significance at level $L = 1$ of the *DynDFG* which corresponds to the computation of individual frequency coefficients. It takes into account the invocation of the DCT, quantization, de-quantization and inverse-DCT stages. The resulting coefficient significances are shown in *Figure 4*. The diagonal zig-zag path corresponds to the importance of coefficients according to the wisdom of image/video compression experts. The significance pattern that emerges from the analysis verifies this domain expert wisdom, thereby validating our approach.

We structure DCT using 15 tasks in total, one for each of the diagonals in *Figure 4*. Each task operates on coefficients of the same or similar significance. Task significance gradually drops with increasing distance from the top-left corner.

4.1.3 Fisheye Lens Image Correction

Fisheye transforms images distorted by fisheye-shaped lens back to the natural-looking perspective space [6]. The algorithm associates integer coordinates of the output image to
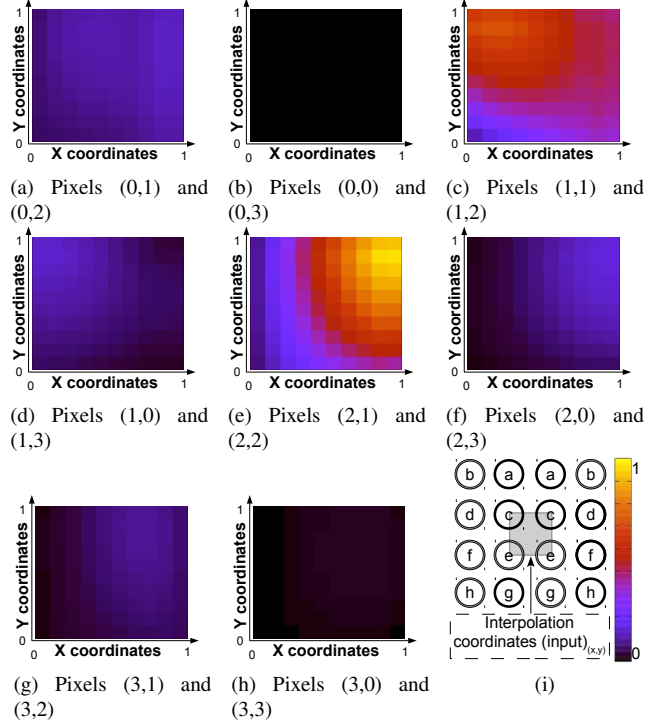

(a) Pixels (0,1) and (0,2) (b) Pixels (0,0) and (0,3) (c) Pixels (1,1) and (1,2)

(d) Pixels (1,0) and (1,3) (e) Pixels (2,1) and (2,2) (f) Pixels (2,0) and (2,3)

(g) Pixels (3,1) and (3,2) (h) Pixels (3,0) and (3,3) (i)

Figure 6: Significance graphs for the pixels in the $4x4$ block of *BicubicInterp* with respect to the interpolated output image; letters in (i) point to the corresponding graphs.

real-valued coordinates in the distorted image (*InverseMapping* kernel). Bicubic interpolation on a 4x4 window is then applied to calculate each pixel value of the output image from the neighbouring pixels of the corresponding point on the input image (*BicubicInterp* kernel).

Figure 5 depicts the output of the significance analysis for the *InverseMapping* kernel applied on an image whose dimensions in the natural looking space are 1280x960. The effect of fisheye-shaped lens is to expand the pixels closest to the boundary, and push together pixels that are near the center. Thus, computing coordinates for pixels near the border is more sensitive to imprecision than for those at the center.

BicubicInterp uses weighted averages to produce the interpolated pixel value. The grey rectangle of *Figure 6i* shows the area in which the interpolated pixel resides. *Figures 6a-6h* show the corresponding significance values of the pixel-pairs, mapped on the discretized input coordinate space. The results indicate that the inner 2x2 pixel block, which directly surrounds the coordinates of the input point, contains the two most significant pairs of pixels (*Figures 6 c* and *e*).

Each task of Fisheye computes a block of 128x64 output pixels. It invokes *InverseMapping* for each pixel of the block to calculate the coordinates in the distorted space and feeds them to *BicubicInterp* to interpolate the pixel value. We use the significance pattern of *InverseMapping* to assign higher significance values to tasks which are closer to the image border, and lower to those near the center. The ap-
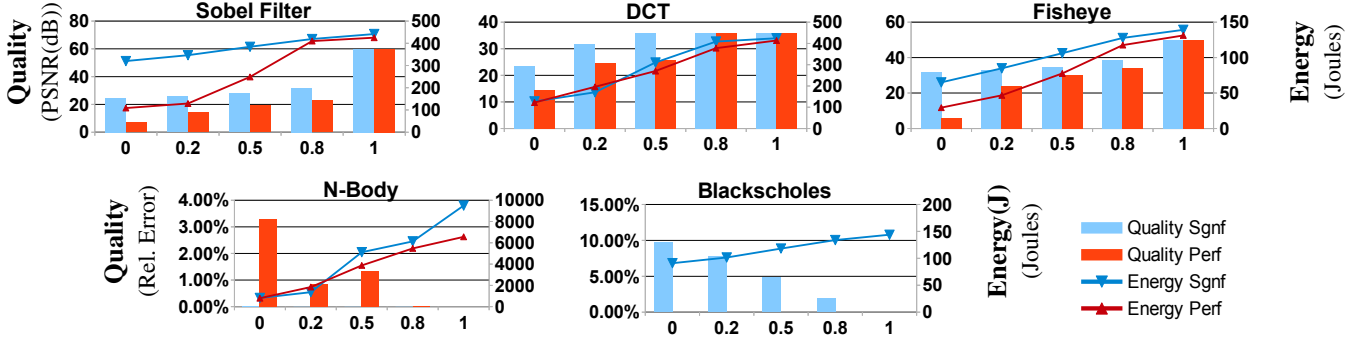
Figure 7: Output quality (blue bars, left y-axis) and energy consumption (blue lines, right y-axis) for the 5 benchmarks, as a function of the ratio of accurately executed tasks (x-axis). The results obtained by loop perforation are depicted in red.

proximate version of tasks invokes *InverseMapping* only for the pixels which lie on the border of the 128x64 block and uses interpolation to compute the coordinates of internal pixels. For *BicubicInterp* we exploit a transitive property of significance: it is sensible to opt for an approximate execution of computations which use approximate input data. In tasks where *InverseMapping* was executed approximately, *BicubicInterp* uses only the pixels pairs in *Figures 6 c* and *e*.

4.1.4 N-Body

This kernel simulates the kinematic behaviour (position and velocity) of liquid Argon atoms within a bounded space, under the effects of a force produced by a Lennard-Jones pair potential [17]. The potential is defined as a function of distance (r) and two material specific constants (σ and ϵ):

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right] \qquad (13)$$

We compute the significance of each atom's state with respect to the state of all other atoms. The results, once again, confirm domain expert wisdom: the significance is strongly correlated with the distance between atoms. The greater the distance between atom A and atom B, the less the kinematic properties of one affect the other.

The task-based version of N-Body partitions the 3D container of the particles into regions. Every few time-steps it assigns particles to regions based on their location. For each given atom, one task per region is instantiated to calculate the forces that operate on the atom due to the particles contained in that specific region. Neighboring regions to the one that envelopes the target atom are tagged with higher significance values than those which are further away.

4.1.5 BlackScholes

BlackScholes is a benchmark of the Parsec suite [7]. It implements a mathematical model for a market of derivatives, which calculates the buying and selling of assets so as to reduce the financial risk.

Significance analysis indicates that the computation of a stock price can be broken down to 4 blocks of code A,

B, C, D, with $sig(A) > sig(B) \gg sig(C) > sig(D)$. The least important parts (C and D) are approximated using less accurate but faster implementations of mathematical functions such as exp and $sqrt$ [22].

4.2 Loop Perforation

As a reference, we use versions of the benchmarks which utilize loop-perforation [35] to trade-off energy consumption with output quality. We perforate the loops in such a way that the same percentage of computations is skipped as the percentage of computations approximated by our runtime. Similarly to [35], we find that loop perforation is not applicable on BlackScholes since it does not use any loops within the computation of a stock's price. The perforated version of Sobel Filter skips the computation for a percentage of the rows of the image. In DCT we perforate the double nested loops which compute the coefficients of an 8x8 block of pixels. In Fisheye we drop the computation of some of the output image rows similarly to Sobel. Finally, the original version of N-Body computes the forces affecting a particle by iterating all other particles in a loop, whereas the perforated version skips some iterations of the loop.

4.3 Performance Results

We execute all applications for different degrees of approximation, varying the ratio of tasks that are executed accurately. The blue-colored bars in *Figure 7* show the effects on output quality. For Sobel Filter, DCT, and Fisheye we use Peak Signal to Noise Ratio (PSNR) with respect to the fully accurate execution as a quality metric (higher is better). Note that, PSNR is a logarithmic metric. For N-Body and BlackScholes we evaluate the relative error (lower is better) with respect to the fully accurate execution.

The quality of the output gradually increases with the number of tasks that are executed accurately, in all benchmarks. This shows that the significance-driven approach can indeed lead to well-behaved approximate applications. Note that DCT and N-Body produce high-quality output even for relatively low accurate task ratios. The more distinct the sig-

nificance properties of an algorithm are, the smaller the quality penalty due to approximation.

We measure performance and energy consumption for an Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz CPU with 14 cores and 128 GBs of RAM. The blue-colored lines in *Figure 7* show the energy cost of approximate executions (execution times are not shown here for brevity, they follow the same pattern). The energy for a fully accurate execution of each application corresponds to the rightmost data point of the respective plot. The results of loop-perforation correspond to the red-colored bars and lines in *Figure 7*.

Executing more tasks in (light-weight) approximate mode results in lower energy consumption, as expected. In some applications (Sobel Filter, Fisheye) perforated versions are more energy efficient that the corresponding significance-based ones due to the overhead of our task-based implementation. However, approximations driven by the analysis results lead to higher quality of results compared with perforation at the same ratio of accurate computations. This effect is more profound for DCT, Fisheye and N-Body. This difference in quality significantly outweighs overheads of the task-based implementation. For example in N-Body the significance-based approximation achieves a relative error of 0.006% already with a fully approximate execution, at an energy cost of 820 Joules. The perforated version requires 80% of the iterations to be executed accurately to achieve a relative error of 0.02%, at an energy budget of 5475 Joules. The same effect is observed in Sobel Filter.

Our methodology results in better quality for all benchmarks compared with loop-perforation. On average, Sobel Filter, DCT, and Fisheye produce images with 3.91 dB, 10.96 dB, and 6.9 dB higher PSNR compared to their perforated versions. Similarly, N-Body produces relative errors which are on average 6 orders of magnitude lower than those introduced by the perforated version.

5. Related Work

We classify related work into a) software frameworks which include programming models, compiler and runtime techniques, and b) hardware-based techniques.

5.1 Software Frameworks

The Chisel [23] framework, given a reliability and optionally a hardware accuracy specification, automatically selects approximate kernel operations to synthesize an approximate computation to a) minimize energy consumption, and b) satisfy reliability and accuracy requirements. Similarly to our approach, Chisel utilizes interval analysis and code differentiation. However our method is more efficient because only it requires a single analysis run for a fixed input range. Our analysis can also help developers gain insight to an application since it allows them to "visualize" the significance for different parts of the computation. Furthermore, our framework utilizes off-the shelf multi-core processors without any

necessity for instruction-level information. Also contrary to Chisel, it allows dynamic adaptation of the running code in the sense that a single binary may operate at different energy gain/output quality configurations using the $ratio$ knob [40].

Topaz [1] is a task-based framework for platforms which may be used to approximate computations by executing them unreliably. It uses an online outlier detection mechanism to detect and correct, unacceptable task results through re-execution on reliable hardware. Our framework fully automates significance analysis of individual operations and provides a first step towards automating the exploitation of analysis information to partition code in tasks. Topaz expects applications to be split into critical and non-critical sections.

Green [4] allows the programmer to write a precise version of a task and several versions of varying levels of precision. A runtime system monitors application QoS online and dynamically chooses the task version to use in order to provide a target QoS value. Green does not perform any automatic significance analysis: it is completely up to the user to specify candidate loops and functions amenable to approximation. ASAC is a sensitivity analysis technique which perturbs program variables and observes the effect on the outputs of the program [30]. A variable whose perturbation within a value interval does not affect the output beyond a user-defined threshold is considered non-critical and can be approximated. Our analytical framework presents a mathematically sound methodology to prove the significance of computations and assist in transforming the code to a collection of tasks with varying degrees of significance.

Abstract Interpretation [8, 9] is a compile-time framework to approximate programs' behavior, without executing them. It can be used to produce approximations of code regions. In [21], semantics for evaluation in IA and forward mode AD are given, that can be applied to simple examples. However, Abstract Interpretation can only answer about a potential interference to the output for any dataset but cannot characterize the impact of such interference for one particular input. No IA or AD tool based on abstract interpretation seems to exist that can compute interval enclosures or derivatives for production codes. Although there are reverse propagation techniques in abstract interpretation, no equivalent to adjoint mode AD seems to exist. Adjoint mode is the enabling technology for the efficient estimation of the impact of all intermediate variables to the final result.

Mutation [28] via a process akin to genetic algorithms identifies instructions within a program that impact its output quality. Much like typical profiling methodologies it is sensitive to the selection of test sets [10]. More importantly, it cannot provide a quantitative assessment of the effect of computations to the final result.

EnerJ [33] proposes approximate programming by allowing the developers to declare data structures that may be subject to approximate computation in return for increased performance. It allows operations to be computed in aggres-

191

sively voltage-scaled processors and data structures to be stored in DRAM with low refresh rate and SRAM with low supply voltage. Exposing approximate computing to the programmer requires expanding the processor ISA with unreliable instructions that offer no guarantee that a certain operation will be performed correctly [12].

Petabricks [2] is a parallel language that allows programmers to provide multiple alternative implementation of a code. These versions exhibit various performance vs. QoS behaviors and allow the user to select, at execution time, the most suitable one. Building on this idea, [3] uses genetic tuning algorithms to search the space of candidate algorithmic versions and accuracies in order to select the best approximate version. [36] provides guidelines for manual code transformation for approximate computing. These frameworks delegate the control of approximate code execution to the programmer, exploiting application-specific invariants. Paraprox [32] is a compiler and runtime platform that identifies common patterns found in data-parallel kernels, written in CUDA or OpenCL, and replaces them with approximate template-based kernels.

ApproxIt [42] is a framework for approximate computation of iterative-based methods, based on a lightweight quality control mechanism. The error-resilient and error-sensitive parts of each application are identified during an offline characterization phase by several simulations on representative inputs. Then, at online stage, reconfiguration of approximation modes are performed at certain iterations considering the time-varying resilience requirements of each application and respecting the user-specified output quality.

Loop perforation is a compiler technique that drops loop iterations deemed less significant for the output quality while keeping critical loop iterations that must always be executed [35]. It uses profiling to find the set of critical loop iterations that result to crashes and quality degradation if skipped. Our hybrid approach relies on mathematical foundations to identify the least significant portions of an application which may have a finer granularity compared to the body of an iteration.

5.2 Hardware Frameworks

ERSA [18] is a multi-core architecture where cores are either fully reliable or have relaxed reliability. ERSA uses an explicit and application-specific mapping of code to cores with different levels of reliability. Some additional ideas on hardware support for approximate computation are quality programmable vector processors using ISA extensions [41], ISA extensions with approximate semantics in general purpose CPUs [12], neural networks that approximate the results of a code region in hardware [13], and low-voltage probabilistic storage [31].

Architecture Vulnerability Factor (AVF) [25], Hardware Vulnerability Factor (HVF) [39], and Program Vulnerability Factor (PVF) [38] aim at providing metrics which indicate the level of error masking for a program which executes on unreliable hardware. They simulate applications and perform fault injections at various locations (on-chip structures, registers, memory, etc) to determine their respective Vulnerability Factor by judging if an error impacts the output results. These methods cannot be used to determine the relevant importance of computations with respect to output quality and require specialized hardware.

6. Conclusions

In this paper, we presented `dco/scorpio`, the first framework that produces a mathematically rigorous approach to automate specification of computational significance. Using interval analysis and algorithmic differentiation techniques, we developed a method that captures significance variations of operations with respect to the output of an application. A programmer uses this analysis to partition an application into OpenMP-like tasks and to rank these tasks according to their contribution to the output result.

We validated our methodology across a number of benchmarks by showing that `dco/scorpio` produces significance rankings that are close to what a domain expert would specify. Comparison of our approach against the compiler technique loop perforation [35], indicates that judiciously selecting approximate tasks and approximation conditions is crucial in order to achieve energy efficiency at a graceful degradation of the output quality.

As part of future work, we plan to improve the framework by extending significance analysis to a wider range of input intervals to accommodate the fact that code significance is input-dependent for some benchmarks. Automatic detection of light-weight functions to approximate tasks is another area we are planning to explore. Moreover, we intend to investigate alternative analysis scenarios by combining the robustness of algorithmic differentiation to Monte Carlo-based methodologies. Finally, our current approach expects a single kernel and its respective input-data ranges. We plan to expand our framework to treat kernels as reusable components in the spirit of libraries.

Acknowledgments

References

[1] S. Achour and M. C. Rinard. Approximate Computation with Outlier Detection in Topaz. In OOPSLA 2015, pages 711–730. ACM, 2015.

[2] J. Ansel, C. P. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. P. Amarasinghe. PetaBricks: a Language and Compiler for Algorithmic Choice. In PLDI '09, June 15-21, 2009, 2009.

[3] J. Ansel, Y. L. Wong, C. P. Chan, M. Olszewski, A. Edelman, and S. P. Amarasinghe. Language and Compiler Support for Autotuning Variable-Accuracy Algorithms. In CGO '11, April 2-6, 2011, 2011.

[4] W. Baek and T. M. Chilimbi. Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation. In PLDI '10, ACM, 2010.

[5] A. Becker, T. Richter, N. Fröhling, P. Fraser, T. Story, W. J. Cosshall, D. Coffin, and B. Lindbloom. *Image Compression benchmark*. 2015.

[6] N. Bellas, S. M. Chai, M. Dwyer, and D. Linzmeier. Real-time fisheye lens distortion correction using automatically generated streaming accelerators. In FCCM '09, 2009.

[7] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[8] P. Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics*, Springer, 2001.

[9] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In POPL 1977, ACM, 1977.

[10] M. Delamaro and J. Offutt. Assessing the Influence of Multiple Test Case Selection on Mutation Experiments. In ICSTW, March 2014.

[11] A. Doucet, S. Godsill, and C. Andrieu. On Sequential Monte Carlo Sampling Methods for Bayesian Filtering. *Statistics and computing*, 10(3):197–208, 2000.

[12] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture Support for Disciplined Approximate Programming. In ASPLOS XVII, ACM, 2012

[13] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural Acceleration for General-Purpose Approximate Programs. In MICRO-45, 2012.

[14] Í. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. Approx-Hadoop: Bringing Approximations to MapReduce Frameworks. In ASPLOS '15, ACM, 2015.

[15] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2nd edition, 2008.

[16] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward Dark Silicon in Servers. In MICRO-31(4):6–15, July 2011. ISSN 0272-1732.

[17] J. E. Jones. On the Determination of Molecular Fields. I. From the Variation of the Viscosity of a Gas with Temperature. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 106(738):441–462, 1924. ISSN 0950-1207.

[18] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra. ERSA: Error Resilient System Architecture for Probabilistic Applications. In DATE '10, European Design and Automation Association, 2010.

[19] M. Lerch, G. Tischler, J. W. von Gudenberg, W. Hofschuster, and W. Krämer. FILIB++, a fast interval library supporting containment computations. *ACM Trans. Math. Softw.*, 32(2):299–324, 2006.

[20] J. Lotz, U. Naumann, R. Hannemann-Tamás, T. Ploch, and A. Mitsos. Higher-order Discrete Adjoint ODE Solver in C++ for Dynamic Optimization . *Procedia Computer Science*, 51:256–265, 2015. ISSN 1877-0509. International Conference On Computational Science, ICCS 2015, Computational Science at the Gates of Nature.

[21] M. Martel. An Overview of Semantics for the Validation of Numerical Programs. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, January 17-19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 59–77. Springer, 2005.

[22] P. Mineiro. fastapprox. http://code.google.com/p/fastapprox/, 2012.

[23] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. In OOPSLA '14, ACM, 2014.

[24] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 1 edition, January 2009.

[25] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In MICRO-36, IEEE Computer Society, 2003.

[26] U. Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*, volume 24. Siam, 2012.

[27] U. Naumann, J. Lotz, K. Leppkes, and M. Towara. Algorithmic Differentiation of Numerical Methods: Tangent and Adjoint Solvers for Parameterized Systems of Nonlinear Equations. *ACM Trans. Math. Softw.*, 41(4):26:1–26:21, Oct. 2015. ISSN 0098-3500.

[28] J. Offutt. A mutation carol: Past, present and future. *Information and Software Technology*, 53(10):1098–1107, 2011. ISSN 0950-5849. http://www.sciencedirect.com/science/article/pii/S0950584911000838. Special Section on Mutation Testing.

[29] OpenMP Architecture Review Board. OpenMP Application Program Interface (version 4.0). Technical report, July 2013.

[30] P. Roy, R. Ray, C. Wang, and W. F. Wong. ASAC: Automatic Sensitivity Analysis for Approximate Computing. In LCTES '14, 2014.

[31] M. Salajegheh, Y. Wang, A. A. Jiang, E. Learned-Miller, and K. Fu. Half-Wits: Software Techniques for Low-Voltage Probabilistic Storage on Microcontrollers with NOR Flash Memory. *ACM Trans. Embed. Comput. Syst.*, 12(2s):91:1–91:25, May 2013. ISSN 1539-9087.

[32] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based Approximation for Data Parallel Applications. In ASPLOS '14, ACM, 2014.

[33] 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 '11, ACM, 2011.

[34] H. Schichl and A. Neumaier. Interval Analysis on Directed Acyclic Graphs for Global Optimization. Technical report, J. Global Optimization, 2004.

[35] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In ESEC/FSE '11, ACM, 2011.

[36] J. Sloan, J. Sartori, and R. Kumar. On Software Design for Stochastic Processors. In DAC '12, ACM, 2012.

[37] Software and Tools for Scientific Engineering, RWTH Aachen University, Germany. Derivative Code by Overloading in C++ (dco/c++). http://fsnew.stce.rwth-aachen.de/research/software/dco-c.

[38] V. Sridharan and D. Kaeli. Eliminating microarchitectural dependency from Architectural Vulnerability. In HPCA '09, IEEE Press, Feb 2009.

[39] V. Sridharan and D. R. Kaeli. Using Hardware Vulnerability Factors to Enhance AVF Analysis. In ISCA '10, ACM, 2010.

[40] V. Vassiliadis, C. Chalios, K. Parasyris, C. D. Antonopoulos, S. Lalis, N. Bellas, H. Vandierendonck, and D. S. Nikolopoulos. A Significance-driven Programming Framework for Energy-constrained Approximate Computing. In CF '15, ACM, 2015.

[41] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Quality Programmable Vector Processors for Approximate Computing. In MICRO-46, ACM, 2013.

[42] Q. Zhang, F. Yuan, R. Ye, and Q. Xu. ApproxIt: An Approximate Computing Framework for Iterative Methods. In DAC '14, ACM, 2014.