Approximate Loop Unrolling

Marcelino Rodriguez-Cancio
Vanderbilt University, USA
marcelino.rodriguez-cancio@vanderbilt.edu

Benoit Combemale
University of Toulouse / Inria, France
benoit.combemale@irit.fr

Benoit Baudry
KTH, Sweden
baudry@kth.se

Abstract

We introduce Approximate Unrolling, a compiler loop optimization that reduces execution time and energy consumption, exploiting code regions that can endure some approximation and still produce acceptable results. Specifically, this work focuses on counted loops that map a function over the elements of an array. Approximate Unrolling transforms loops similarly to Loop Unrolling. However, unlike its exact counterpart, our optimization does not unroll loops by adding exact copies of the loop's body. Instead, it adds code that interpolates the results of previous iterations.

CCS Concepts • **Software and its engineering** → *Just-in-time compilers.*

Keywords approximate computing; compiler optimizations

ACM Reference Format:
Marcelino Rodriguez-Cancio, Benoit Combemale, and Benoit Baudry. 2019. Approximate Loop Unrolling. In *Proceedings of the 16th conference on Computing Frontiers (CF '19), April 30-May 2, 2019, Alghero, Italy*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3310273.3323841

1 Introduction

Approximate computing exploits the fact that some computations can be made less precise and still produce good results [25]. Previous works in the field have proven that when small reductions in accuracy are acceptable, significant improvements in execution times and energy consumption can be achieved [40]. Opportunities for approximation exist across the whole stack and previous works have explored this concept in both hardware [6, 33, 36] and software [3, 28, 30, 34].

In this work, we describe Approximate Unrolling, a novel compiler loop optimization that uses the ideas of approximate computing to reduce execution times and energy consumption of loops. The key motivation of Approximate Unrolling relies on the following observation: data such as time series, sound, video, and images are frequently represented as arrays where contiguous slots contain similar values. As a consequence of this neighboring similarity, computations producing such data are usually locally smooth functions. Our technique exploits this observation: it searches for loops where functions are mapped to contiguous array slots in each iteration. If the function's computation is expensive, we substitute it by less costly interpolations of the values assigned to nearby array values. In exchange for losses in accuracy, we obtain a faster and less energy consuming loop.

Approximate Unrolling combines static analysis and code transformations in order to improve performance in exchange for small accuracy losses. Static analysis has two objectives: determine if the loop's structure fits our transformations (counted loops mapping computations to consecutive arrays slots); estimate if the transformation could actually provide some performance improvements (computation mapped to the array elements is expensive). The code is transformed using two possible strategies. One that we call *Nearest Neighbour* (NN), which approximates the value of one array slot by copying the value of the preceding slot. The other strategy, called *Linear Interpolation*(LI) transforms the code so the value of a slot is the average of the previous and next slot. A key novel aspect of these static transformations is that they ensure accuracy for non-approximated iteration of the loop, avoiding error to accumulate. This is a unique feature that is not found on other approximate loop optimization techniques.

We have implemented a prototype of Approximate Unrolling, inside the OpenJDK C2 compiler. This design choice prevents phase ordering problems and allows us to reuse internal representations that we need for our analysis. An implementation in a production-ready compiler is also key to assess the relevance of our optimization on already highly optimized code, such as the one produced by the JVM.

We experimented with this implementation using a four real-world Java libraries. The objectives of our experiments are to determine whether Approximate Unrolling is able to provide a good performance vs. accuracy trade-off and to compare Approximate Unrolling with Loop Perforation [34], the state of the art approximate computing technique for loop optimization.

Our results show that Approximate Unrolling is able to reduce the execution time and CPU energy consumption of the x86 code generated around 50% to 110% while keeping the accuracy to acceptable levels. Compared with Loop Perforation, Approximate Unrolling preserved accuracy better

CF '19, April 30-May 2, 2019, Alghero, Italy

ACM ISBN 978-1-4503-6685-4/19/05…$15.00
https://doi.org/10.1145/3310273.3323841

in 76% of the cases and raised less fatal exceptions than Loop Perforation (2 *vs.* 7).

The contributions of this work are: (i) **Approximate Unrolling**, an approximate loop optimization. (ii) An efficient implementation of Approximate Unrolling inside the OpenJDK Hotspot VM (iii) An empirical assessment of the effectiveness of Approximate Unrolling to trade accuracy for time and energy savings. (iv) An empirical demonstration of Approximate Unrolling's novel contribution to the state of the art, as an effective complement to Loop Perforation.

2 Approximate Unrolling

In this section we describe the main contribution of our paper. We use the example of Listing 1 to illustrate the kind of loops targeted by the optimization as well as the process for the approximate transformation.

```
for (int i = 0; i < N ; i++) {
  phase = phase+phaseInc;
  double v = Math.sin(phase)*ampl;
  A[i] = v; }
```

Listing 1. Illustrative Example: A loop mapping a sine wave into an array.

```
for (int i = 0; i < N ; i+= 2) {
  phase = phase + phaseInc;
  double v = Math.sin(phase) * ampl;
  A[i] = v;
  // UNROLLED ITERATION:
  phase = phase + phaseInc;
  double v1 = Math.sin(phase)*ampl;
  A[i + 1] = v1; }
```

Listing 2. Approximate Unrolling first unroll the loop as Loop Unrolling would normally do. In this listing, the Illustrative Example is unrolled using Loop Unrolling.

Approximate Unrolling replaces the unrolled iteration's original instructions by others that interpolate results. We perform this transformation in such a way that errors do not accumulate as the loop runs. We currently implement two interpolation strategies: Linear Interpolation (LI) and Nearest Neighbor (NN). A *policy* determines which strategy, if any, is likely to optimize each loop. In the following paragraphs, we discuss each step.

Like normal unrolling, Approximate Unrolling can unroll more than one loop iteration. For simplicity, we only show the transformation where 1 iteration out of 2 is unrolled/approximated.

Step 1. Target Structure: Every compiler optimization is able to target only certain code structures. In the case of Approximate Unrolling, these are `'for'` loops that (i) have an induction variable incremented by a constant stride – i.e. counted loops – (ii) contain an array assignment inside their body, and (iii) the indexing expression of the array assignment is value-dependent on the loop's induction variable.

The example at Listing 1 is indeed a `'for'` loop where (i) the induction variable `'i'` is incremented by a constant stride, i.e. `'i++'`; (ii) `'A[i]'` is an array assignment inside the loop's body and (iii) the indexing expression of `'A[i]'` is value-dependent on the induction variable `'i'`.

```
for (i = 0; i < N - 1; i += 2) {
  phase = phase + phaseInc;
  double v = Math.sin(phase) * ampl;
  A[i] = v;
  phase = phase + phaseInc;
  //v1=Math.sin(phase)*ampl; REMOVED
  A[i + 1] = A[i]; }
// Guard Loop g
for (j = i; j < N; j ++) {
  phase = phase + phaseInc;
  double v = Math.sin(phase) * ampl;
  A[i] = v;}
```

Listing 3. After unrolling, our optimization approximates the unrolled iteration. The loop of Listing 2, transformed using nearest neighborg interpolation.

```
//Initial iteration peeled:
phase = phase + phaseInc;
double v0 = Math.sin(phase) * ampl;
A[0] = v0; int i = 1;
if ( N > 2 )
 for (i = 2; i < N - 1; i += 2 ) {
  phase = phase + phaseInc;
   //v=Math.sin(phase)*ampl; REMOVED
   phase=phase+phaseInc;//Exact iteration:
   double v = Math.sin(phase)*ampl;
   A[i] = v;
   //Approximate iteration:
   A[i - 1] = (A[i]+A[i-2])*0.5f; }
for (j=i; j<N; j++) {//Guard Loop
 phase = phase + phaseInc;
 double v = Math.sin(phase) * ampl;
 A[i] = v;}
```

Listing 4. The loop of Listing 2 transformed by Approximate Unrolling using Linear Interpolation.

Step 2. Approximation Policy: In compiler optimization is often the case that is possible to predict whether the optimization will have a beneficial impact on speed. This estimate is called *policy* in C2's terms. Approximate Unrolling also features a policy that can predict speed and energy efficiency gains. The indicator is computed over the C2's internal representation, a graph where each node roughly represents an assembler instruction. This policy determines the approximation strategy to use (if any) as follows: let $|O|$

be the predicted number of nodes to remove from the original code, while $|LI|$ and $|NN|$ represent the number of nodes needed to perform the LI and NN interpolations respectively. If $|O| > |LI|$, resp. $|O| > |NN|$, the policy indicates that LI, resp. NN, should be used to approximate the loop. Is possible for a small loop that $(|O| > |LI|) = (|O| > |NN|) = false$. If so, the loop is not optimized. If both $(|O| > |LI|) = (|O| > |NN|) = true$, LI is preferred since it produces better accuracy results in 70% of the cases in our dataset.

This simple policy model proved to be accurate enough in our experiments, even when performance prediction is a quite challenging topic. We believe this is because the code introduced by Approximate Unrolling is very small and limited to basic instructions such as `mov`, `add`, and `mul`. The policy also applies to function calls, as several nodes are devoted to resolving the method's owner address and pushing and popping data from the stack that are removed as result of removing the call. This gives little room for false positives such as loop bodies containing more instructions than the ones introduced actually running faster due to requiring less CPU cycles or having better branch prediction, etc.

Notice that: being a compiler-level concern, the policy can only predict speed and energy efficiency improvements. Detecting forgiving zones is an algorithm-level concern.

Step 3. Loop Unrolling: If the policy predicts that the loop will gain performance, the optimization unrolls the loop, just like a regular optimizing compiler would perform Loop Unrolling. Just like Loop Unrolling, Approximate Unrolling can unroll the code multiple times, we keep our analysis and examples to the case it unrolls the loop once for brevity sake.

Step 4(A). Approximating the loop with Linear Interpolation: Approximate Unrolling adds code to the loop's body that interpolates the computations of the odd iterations as the mean of the even ones (the computations of iteration one are interpolated using the results of iterations zero and two, computations of iteration three are interpolated using the results of two and four and so on). Initially, Approximate Unrolling'peels' iteration zero of the loop. Then, it modifies the initialization and updates the statements to double the increment of the loop's counter variable. The approximate loop is terminated earlier and a guard loop is added to avoid out-of-bounds errors. Listing 4 shows the example loop using linear interpolation. Notice that some computations have been deleted. This is actually done in Step 5.

Step 4(B). Approximating the loop with Nearest Neighbor: If the policy determines that NN should be used, Approximate Unrolling modifies the update statement to double the increment of the loop's counter variable. It also adds code with the nearest neighbor interpolation at the end of the loop's body. The loop is terminated earlier and a guard loop is also added to avoid out-of-bounds errors. Listing 3 shows the loop interpolated using nearest neighbor. Again, some computations are removed in the example, as discussed in the next step.

Step 5. Dead code removal: Once the approximate transformation is performed, some code is left dead (i.e. its computations are no longer used). In our example, the computation of the sine (`'v1=Math.sin(phase)*ampl'`) is no longer needed in the interpolated iteration once the interpolation code is added. This last transformation step removes such unused code. The expected improvement in performance comes precisely from this removal, as less code is executed. Listings 4 and 3 illustrate our example after dead code elimination.

Is important to notice that *only dead code resulting from the approximation is removed.* Therefore, loop-carried dependencies are not removed, since their values are needed in the following iteration. This provides a guarantee that the exact iteration of the loop will always produce precise results. This way, the error does not accumulate as the loop executes in the approximate iterations. In the context of approximate loop optimizations, this is a unique feature to Approximate Unrolling.

Indicating approximable areas to the compiler: Approximate techniques act on parts of the system that can endure accuracy losses. As said before, this is an algorithm-level concern and the compiler can do very little to detect approximate zones. This is why such detection is out of the scope of this paper. Finding zones that will endure approximation have been addressed in numerous previous works through automated tools [5, 34] or language support [3, 8, 23, 30]. Automated tools work by injecting modifications to the approximate system and then evaluating the impact through automated testing and accuracy metrics. Language support provides keywords that allow the programmer to indicate approximate zones. Our technique can work with any language having counted '`for`' loops, being agnostic to the detection mechanism.

3 Implementation

This section describes our experimental implementation of Approximate Unrolling in the C2 compiler of the OpenSDK Hotspot VM, a Java compiler used by billions of devices today. We choose to implement Approximate Unrolling directly in the C2 to avoid the Phase Ordering problem [17], reduce implementation's effort by means of code reuse, and most importantly, this implementation showed that we could improve the performance of a production-ready compiler. Our modified version of the Hotspot JVM is available on the webpage of the Approximate Unrolling project [1]

3.1 The Ideal Graph

The C2's internal representation is called the *Ideal Graph* (IG) [11]. It is a *Program Dependency Graph* [12] and a *Value Dependency Graph*[2]. All C2's machine independent optimizations work by reshaping this graph.

[1] https://github.com/approxunrollteam

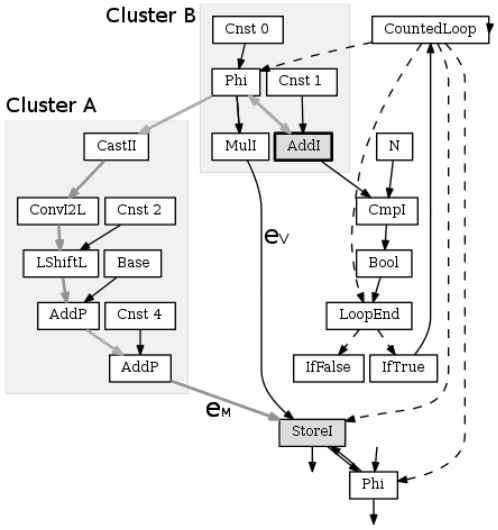

Figure 1. The ideal graph of the example loop of Listing 5. Dashed arrows represent control edges and solid arrows represents data edges.

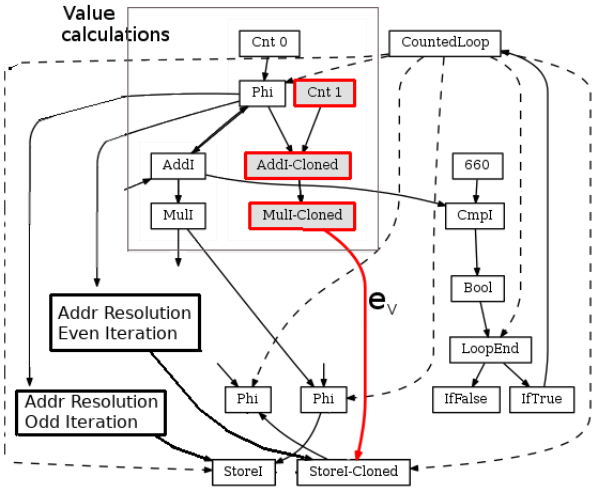

Figure 2. The ideal graph for the unrolled loop of Listing 5. Nodes to be deleted by the optimization, along with all their related data edges, are marked with a gray background.

In the *Ideal Graph* (IG), nodes are objects and hold metadata used by the compiler to perform optimizations. Nodes in this graph represent instructions as close as possible to assembler language (i.e. `AddI` for integer addition and `MulF` for float multiplication). The IG metamodel has nearly 380 types of nodes. We deal with four to illustrate Approximate Unrolling in the example: `Store`, `CountedLoop`, `Add` and `Mul`.

`Store` nodes represent storages into memory. Their metadata indicates the type of the variable being written, making it easy to identify `Store` belonging to arrays. `CountedLoop` represents the first node of a counted loop (`'for'` loop with a constant stride increment). This node's metadata contains a list of all nodes in the loop's body instructions and a list with all nodes of the update expression. Finally, `Add` and `Mul` nodes represent the addition and multiplication operations.

Nodes in the IG are connected by *control edges* (which describes control flow) and *data edges* (which describes data dependencies). *Data edges* are the most important ones for our implementation, and we will refer to this kind of node exclusively. If a node B receives a *data edge* from a node A, it depends on the value produced by A. Edges are pointers to other nodes and contain no information. Edges are stored in nodes as a list of pointers. The edge's position in the list usually matters. For example, in `Div` (division) nodes the edge in slot 1 points to the dividend and the edge in slot 2 points to the divisor.

The `Store` requires a memory address and a value to be stored in the address. The memory edge e_M is stored at index 2 and the value edge e_V at index 3. Edge e_M links the `Store` with the node computing the memory address where the value is being written, while e_V links the `Store` with the node producing the value to write.

Let us consider the very simple example of Listing 5. The resulting IG for this loop is shown in Figure 1. In the figure, the `StoreI` represents the assignment to `A[i]`, the `MulI` node represents the `i*i` expression. The address is resolved by the nodes in the Cluster A (containing the `LShift` node).

```
for (int i = 0;i < N;i++) A[i]=i*i;
```

Listing 5. Example loop for the implementation

3.2 Detecting Target Loops

Java `for` loops with a constant-increment update expression are targeted by other C2 optimizations. Therefore, the C2 recognizes these loops [10, 39] and marks them using `CountedLoop` nodes. We reuse this analysis for our optimization. Once counted loops are detected, we determine if there is an array whose index expression value depends on the loop's update expression. This is done by searching for a `Store` writing to array-owned memory on the `CountedLoop` body's node list. In the example of Figure 1 we find the `StoreI` node (in gray). Finally, we check if the array index expression value depends on the loop's update expression by looking for a data-edges path between the `Store` node representing the array assignment and any node on the loop's update expression. In the example of Figure 1 this path is highlighted using bold gray edges and is composed by the following nodes: `AddI` → `Phi` → `CastII` → `ConvI2L` → `LShiftL` → `AddP` → `AddP` → `StoreI`.

3.3 Unrolling

Our implementation piggybacks on the Loop Unrolling already implemented in the C2 compiler. At the point Approximate Unrolling starts, the compiler has already unrolled the

loop. While unrolling, the compiler clones all the instructions of the loop's body. Figure 2 shows the IG once the loop of Listing 5 has been unrolled. The cloning process introduces two Store nodes: StoreI and StoreI-Cloned. The cloned nodes belong to the even iteration of the loop (by C2's design). Once the loop is unrolled, Approximate Unrolling reshapes the graph to achieve the interpolated step by modifying one of the two resulting iterations.

```
1  B8: #
2  movl [rcx + #16 + r9 << #2], R8 # int
3  movl r9, r10 # spill
4  addl r9, #3 # int
5  imull r9, r9 # int
6  movl r8, r10 # spill
7  incl r8 # int
8  imull r8, r8 # int
9  movl [rcx + #20 + r10 << #2], r8 # int
10 movl rdi, r10 # spill
11 addl rdi, #2 # int
12 imull rdi, rdi # int
13 movl [rcx + #24 + r10 << #2], rdi # int
14 movl [rcx + #28 + r10 << #2], r9 # int
15 addl r10, #4 # int
16 movl r8, r10 # spill
17 imull r8, r10 # int
18 cmpl r10, r11
19 jl,s 7
```

Listing 6. Assembler code generated for the example loop without using Approximate Unrolling

3.3.1 Nearest Neighbor Interpolation

As mentioned in section 3.1, a Store node takes two input data edges e_M and e_V. Edge e_M links with the node computing the memory address, while e_V links with node producing the value to write.

Approximate Unrolling performs nearest neighborhood interpolation by disconnecting the cloned Store node from the node producing the value being written (i.e. it deletes e_V). In Figure 2 this means to disconnect node MulI-Clone (in gray) from node StoreI-Clone by removing edge e_V.

This operation causes the node producing the value (in the example MulI-Clone) to have one less value dependency and potentially become *dead* if it has no other dependencies. A node withoutBy desconecting value dependencies means that its computations are not being consumed and therefore is useless *dead code*. In this case, the node is removed from the graph. We recursively delete all nodes that do not have dependencies anymore, until no new dead nodes appear. In Figure 2, we delete MulI-Cloned and then AddI-Cloned. This simplification of the IG translates into fewer instructions when the IG is transformed in assembler code. After the removal, Approximate Unrolling connects the node producing the value for the original Store into the cloned Store.

Listing 6 shows the code generated by C2, without performing Approximate Unrolling: the compiler has unrolled the loop twice, generating four storages to memory (lines 2, 3, 9, 10) and four multiplication instructions (imull, lines 5, 8, 12, 17). Listing 7 shows the code generated for the same loop using our transformation: there are still four storages (lines 4, 5, 9, 10), but only two multiplications (Lines 8, 13).

```
1  B7: # B8 <- B8 top-of-loop Freq: 986889
2  movl rbx, r8 # spill
3  B8: #
4  movl [r11 + #16 + rbx << #2], rcx # int
5  movl [r11 + #20 + r8 << #2], rcx # int
6  movl rbx, r8 # spill
7  addl rbx, #2 # int
8  imull rbx, rbx # int
9  movl [r11 + #24 + r8 << #2], rbx # int
10 movl [r11 + #28 + r8 << #2], rbx # int
11 addl r8, #4 # int
12 movl rcx, r8 # spill
13 imull rcx, r8 # int
14 cmpl r8, r9
15 jl,s B7
```

Listing 7. Assembler code for the example loop using Approximate Unrolling

3.3.2 Linear Interpolation

To use linear interpolation, Approximate Unrolling performs the same disconnect-and-then-remove-dead-code analysis than with nearest interpolation. Is important to remark that this is not a simple iteration skip. By detaching the cloned Store and then performing a dead-code elimination, we guaranty that loop-carried dependencies stay alive, ensuring the non-approximate iteration exactitude. In a second step, another loop is added to the program that interpolate between every two elements of the array.

4 Evaluation

To evaluate our approach we created a dataset of loops that Approximate Unrolling could target. These were extracted from well-known libraries belonging to different domains where performance is paramount: OpenImaJ [15](a computer vision Library), Jsyn[7](a framework to build software-based musical synthesizers), Apache Lucene[22] (a text search engine) and Smile[14](a Machine Learning library).

The objective of our evaluation is to assess whether Approximate Unrolling can increase performance and energy efficiency while keeping accuracy losses acceptable. To answer this question, we use a set of JMH[1] microbenchmarks to measure speedups for each of the loops in our dataset. We then observe the energy consumption of the microbenchmarks to evaluate energy reductions. Energy was assessed

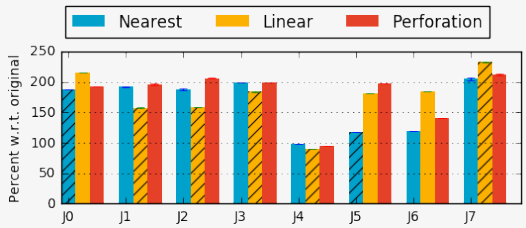

Figure 3. JSyn performance improvements

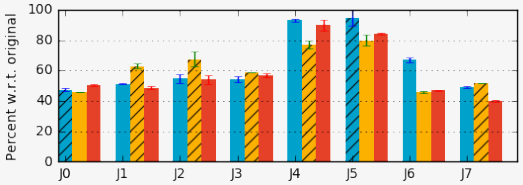

Figure 5. JSyn energy reductions

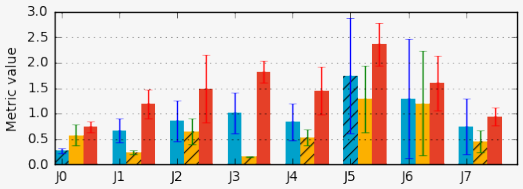

Figure 7. JSyn Accuracy metric (Lower is better)

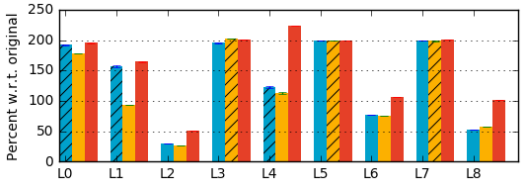

Figure 4. Lucene. Performance improvement

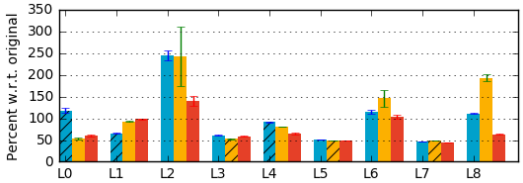

Figure 6. Lucene. Energy consumption

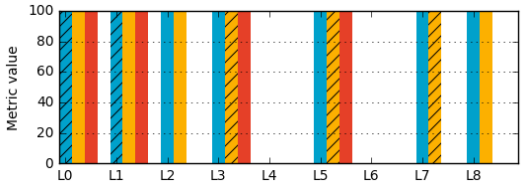

Figure 8. Lucene. Accuracy metric's value (Higher is better, zero means crash)

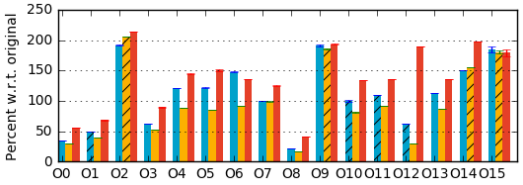

Figure 9. OpenImaJ. Performance improvement

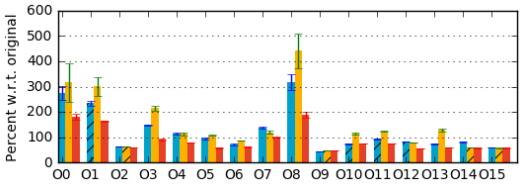

Figure 11. OpenImaJ. Energy consumption

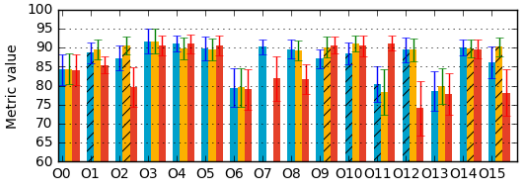

Figure 13. OpenImaJ. Accuracy metric's value (Higher is better)

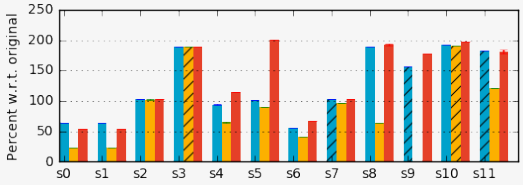

Figure 10. Smile. Performance Improvements

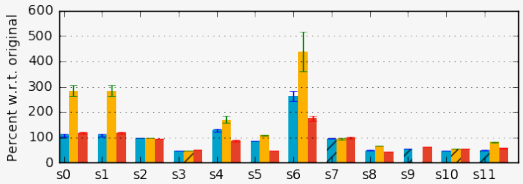

Figure 12. Smile. Energy consumption

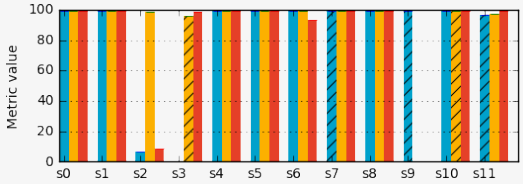

Figure 14. Smile. Accuracy metric's value (Higher is better)

by estimating the total CPU energy consumption of our microbenchmarks using JRALP [19].

To evaluate the impact of each loop on accuracy, we designed individual workloads for each domain. In other words, we evaluated impact in the accuracy of loops in OpenImaJ using a workload that performs real-time face recognition in videos featuring people of different sex, age and ethnicity speaking directly to the camera. The workload to evaluate Jsyn's loops builds a clone of the popular 3xOsc[2] synthesizer and use it to render a set of scores into WAV sound files. Lucene's workload indexes a set of text files and then returns the results of multiple text search performed over these files. Finally, Smile's workload consisted of a classifier able to recognize zip-code handwritten digits.

Row 'Loops' of Table 1 shows the total number of loops in each case study that Approximate Unrolling could target, while 'Covered' shows the number of loops Approximate Unrolling could target in each case that were also covered by the workload's execution. We use for our experiments only the latter, as performance, energy and accuracy measurements can only be made running the loop with a workload.

Table 1. Case studies for our experiments

	OpenImaJ	Jsyn	Lucene	Smile
Loops	118	8	151	73
Covered	16	8	9	12

The accuracy impact for a given loop was measured by comparing the output produced by the workload having this single loop optimized *vs.* the original output (without our optimization). This was done using a specific accuracy loss metric for each project in our dataset. This is common in approximate computing [5, 30], as each domain the result's quality is perceived differently and the degree of tolerance w.r.t. accuracy losses vary from case to case.

Also, for each case, we define a function $\alpha(x)$ that defines the acceptable application's accuracy loss. We do so to investigate how many loops reduce the accuracy below tolerable levels. In other words, we say that a loop reduces the accuracy of an application to unacceptable levels if value x of the accuracy metric makes $\alpha(x) = false$. The accuracy loss metric and acceptable accuracy loss function $\alpha(x)$ used for each project is described as follows:

OpenImaJ: We use the Dice metric: $D(A, B) = \frac{2*|A \cap B|}{|A|+|B|}$ to compare the pixels inside the rectangle detected by the face recognition application without approximation against the pixels obtained with the approximated version. A value of 0.85 produces results not distinguishable by humans, therefore we define the tolerable accuracy loss function for this application as: $\alpha(x) \rightarrow x > 0.85$

[2] https://bit.ly/2FKjSEy

Jsyn: We use the Perceptual Evaluation of Audio Quality (PEAQ) [41] metric. This is a standard metric designed to simulate the psychoacoustic of the human ear, which is not equally sensitive to all frequencies. PEAQ assigns a scale of values depending on how well the modified audio matches the original: 0 (non audible degradation), -1 (audible but not annoying), -2 (slightly annoying), -3 (annoying) -4 (degraded). Therefore: $\alpha(x) \rightarrow x < -2.5$

Lucene: We use a similar metric to Baek et.al [5] to evaluate the impact of approximate loops in the Bing search engine. We give 1 point to each hit that the approximate engine returned in the same order as the non-approximated one, 0.5 if returned in a different order, 0.0001 points if the hit is not in the original version and 0.0 points if the program crashes. We define acceptable as: $\alpha(x) \rightarrow x > 0.95$.

Smile Accuracy metric: We use the classification's recall (i.e. the number of properly classified elements over the total number of elements). According to [16] a 2% error is a good value for this dataset, so we define acceptable level as: $\alpha(x) \rightarrow x > 0.98$

4.1 Performance improvements

For each of the covered loops we measured performance, energy consumption and impact on application's accuracy *without* Approximate Unrolling and used this data as baseline. We then repeated the same observations for the approximated versions of all covered loops. This was done using both approximation strategies (LI and NN).

Figures 3 to 14 show the performance, energy and accuracy results of our experiments. In all figures, each loop is represented using three bars that show (from left to right) the results obtained using Nearest Neighbor (NN) with ratio 1/2 (i.e. approximating one out of two iterations), Linear Interpolation (LI) with ratio 1/2 and Perforation (PERF). Figures 3, 4, 9, 10 show the performance gains for each loop in our dataset. Figures 5, 6, 11, 12 represent the percent of energy consumption w.r.t to the precise version of the loop. Figures 7, 8, 13, 14 show the impact of each loop in its host application's accuracy. In the performance graphs, the Y axis represents the percent of gain w.r.t to the original loop. For example, a 200% value means that the approximate version of the loop run twice as fast. The same applies to energy consumption, a 50% value means that an approximate loop needs only half of the energy to finish than its precise counterpart. In the accuracy graphs, the Y axis represents the accuracy metric's value. When the bar is missing, it would mean that the approximation crashed the program. This is the case for every approximation made to loop L4 in the Lucene's workload.

In Section 2, we illustrate how the policy recommends the approximation strategy to approximate a loop. The policy can also suggest not to optimize the loop at all. Stripped bars represent the strategy recommended to approximate a particular loop. For example, Figure 3 shows that the strategy

Table 2. Resume of the performance, energy and accuracy results of our experiments.

Project	Loops	Speed ↑	Enrg↓	α	Crash	Rec.	Rec. Speed ↑	Rec. Enrg↓	Rec α	Rec. Crash
JSyn	8	7	8	8	0	7	7	7	6	0
OpenImaJ	16	9	8	10	0	8	6	7	8	0
Lucene	9	6	6	7	2	6	6	6	5	1
Smile	12	5	6	10	0	5	5	5	5	0
Total	45	27	28	35	2	26	24	24	24	1

proposed for loop J0 is NN while for loop J1 is LI. The policy also indicated not to approximate some loops in our dataset, for example, Figure 4 shows that this is the case for loops L2 and L8 (no striped bars). We include all loops in our dataset in the plots, even those rejected by policy. We do so to learn about the policy's quality and to observe the optimization's impact in all cases.

Table 2 resumes the data presented in the figures. In the table, column 'Loops' show the number of loops amenable for Approximate Unrolling in our dataset, while columns 'Speed↑' and 'Energ↓'show the number of loops that actually improved speed and consume less energy respectively. Column 'Rec.' shows the number of loops the policy recommended to optimize and columns'Rec. Speed↑' and 'Rec. Energ↓' how many among these loops actually improved performance and energy efficiency respectively. Table 2 also shows for how many loops accuracy looses were acceptable and the number of crashes caused by approximate loops. Column 'α' shows the number of loops with acceptable performance, while 'Rec. α' indicates how many recommended loops reduced the accuracy of the application to acceptable levels. Column "Crash" shows how many loops crashed the application and 'Rec. Crash' how many recommended loops made the application crash.

In our experiments, we can observe that Approximate Unrolling can in fact increase the speed and energy efficiency of loops being optimized. The improvements are due to loops executing fewer instructions. Some results are interesting, such as LI having better speed than NN on J6. This is due to factors that influence performance beyond our optimization such as hardware architecture, machine-dependent optimizations, etc. In the case of J6, the LI interpolation loop is subject to vectorization, a machine-dependent optimization performed on the C2 after machine-independent techniques such as Approximate Unrolling.

Policy's accuracy In order to obtain a better grasp of the policy's role in the result's quality, Table 2 is divided into two segments. The left segment shows the improvements *without* any policy, while the right segment shows a full-fledged Approximate Unrolling working with all its components. This shows that for a random loop in our dataset there is a 60% chance of performance improvement. However, if the policy recommends the loop, there is a 92% chance of gains. The policy indicated two false positives (O1 and O10) that failed to gain performance in our dataset. This was due to Approximate Unrolling's C2 implementation inhibiting Loop Vectorization in later phases of the compiler. The policy had a 96% recall: a single case was not recommended and still gained speed (loop J6). This occurs since the current implementation is conservative w.r.t. branching and always assume the worst case.

4.2 Comparison with Loop Perforation

We compared our optimization against Loop Perforation [34] a state-of-the-art approximate loop optimization. The comparison is done using our implementation of Loop Perforation in the C2 compiler.

Scope of application: Perforation and Approximate Unrolling differ in the code transformation. The key difference is that Approximate Unrolling replaces instructions in the approximate iteration, while Perforation skips them completely. This means that Perforation does not preserve the loop-carried dependencies. Hence, as the loops unfold, Approximate Unrolling preserves accuracy better. They also differ in the scope of applications. Perforation works better in patterns that can completely skip some task[34], like Monte-Carlo simulations, computations improving a result, or data filtering and search. Otherwise, our technique should behave better in situations when no value of the array can be left undefined, such as sets of random numbers that follow a distribution, or sound and video data processing.

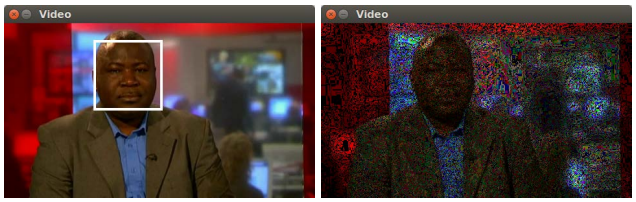

Figure 15. Approximation using NN (left). The recognition system is still able to detect the face. With Perforation (right), the image is too noisy image and the detection fails.

To exemplify how Approximate Unrolling is best suited for signal data and that it can complement the series of cases Loop Perforation can target, Figure 15 represents the results of applying Nearest Neighbor and Perforation to a Color Lookup Table in OpenImaJ. The pictures show that the recognition algorithm fails when using Perforation as a

consequence of the noisy image. Notice that while 12 out of 15 loops in OpenImaJ lose less accuracy using Approximate Unrolling, three loops (O5, O9, and O11) are more accurate using Perforation. This happens because these loops are updating already good results, a situation for which Perforation is known to work well [34].

Figure 5 shows that Approximate Unrolling's accuracy in the Musical Synthesizer was always higher due to the continuous and smooth nature of the sound signal, which is perfect for our optimization. As Approximate Unrolling keeps loop-carried dependencies, it does not accumulate error. On the other hand, error accumulation with Perforation caused the frequencies to drop, changing the musical note being played. It also introduced noise.

In the Lucene use case, Perforation crashes the application five times, while Approximate Unrolling crashes it twice. Every time Approximate Unrolling crashes the application, so does Perforation. Similar results with the Classifier: Perforation reduces the accuracy to zero twice, while Approximate Unrolling does it once. Again, every time Approximate Unrolling reduces recall to zero, so does Perforation.

Performance, Energy, Accuracy & Trade-Off: We optimized all the loops in our dataset using both optimizations to find out which one produced the best results in speedups, energy, and accuracy. From Figures 3 to 14 we gather that from 45 loops in our dataset, Approximate Unrolling maintained better accuracy in 29 vs. 9 Loop Perforation. Meanwhile, Perforation provides a better performance in 31 loops *vs.* 14 our technique. A similar result is obtained when comparing energy consumption (16 Perforation *vs* 29 Approximate Unrolling). The figures also show that Perforation crashed the application 7 times *vs.* 2 Approximate Unrolling. Every time our technique crashed the program, so did Perforation.

Table 3. Approximate Unrolling Vs Perforation

Parameter	Aprx. Unrol.	Tie	Perf.
Better Accuracy	29	7	9
Higher speedup	14	0	31
Lower consumption	16	0	29

We were expecting Perforation to be always faster. However, figures 3, 4, 9, 10 show that Linear Interpolation leads to faster loops than Perforation in 7 loops. The reason is that linearly interpolated loops undergo two more optimizations: Loop Fission and Loop Vectorization. Also, Nearest Neighbor can the same performance than Loop Perforation. This is due to the architecture used for our experiments with Nearest Neighbor, which introduces only one assembler instruction and because when there is no loop-carried dependencies in a loop, all instructions become dead code after the interpolation. These factors allowed Nearest Neighbor to match Loop's Perforation performance in 10 cases.

4.3 Best Fit Loops for Approximate Unrolling

The manual analysis of the results reveal the presence of patterns in loops that benefit from Approximate Unrolling. We found that Approximate Unrolling works best when it must remove more instructions than the ones it inserts as performance gains come from removing instructions and inserting others that perform fewer operations. As an example of a loop that worked well with Approximate Unrolling in our dataset, we show the loop in Listing 8. On the other side, we encountered loop bodies so small that no instructions could be inserted so the modified loop would perform less operations than the original. Listing 9 is an example of such small loops. We included this learning in our policy implementation. Secondly, the data being mapped to the array must be a locally smooth function. This happens naturally in signal-like data such as sound and images. However, other types or data behaves also this way, such as arrays containing starting positions of words in a text.

```
for (int i = 0; i < N; i++) {
float sum = 0.0F;
for (int j=0,jj=kernel.length-1;j<kernel.
    length;j++,jj--)
sum += buffer[i + j]) * kernel[jj];
buffer[i] = sum;}
```

Listing 8. This OpenImaJ loop works well with as the body is computational intensive with no loop-carried dependencies

```
for(i=49;(i--)>0;){jjrounds[i]=-2147483648;}
```

Listing 9. This loop belongs to Lucene. It does not work well with Approximate Unrolling.

4.4 Impact of approximation ratio

We now report our experiments with three out of four(3/4) and one out of four 1/4 approximation ratios.

We hypothesize that the approximation ratio's impact in performance is roughly proportional to the number of approximate iterations. For example, if by using a 1/2 ratio we doubled performance, we can expect roughly a 300% improvement in performance using the 3/4 (three approximate iterations out of four) and a 33% improvement using the 1/4 ratio (one approximate out of four). The opposite also applies, if a loop's performance degrades with the 1/2 ratio, it will be even slower as we increase the approximate iterations. Figure 16 shows the relationship between different ratios. The figure was obtained using full-fledged Approximate Unrolling, including policy selection of optimization. We observed similar results with energy consumption.

Accuracy wise, the results were more dependent on the context and purpose of the loop, making difficult to draw general conclusions. With Jsyn and OpenImaJ the accuracy is proportional to the amount of approximation, with the exception of loops updating already good results, which

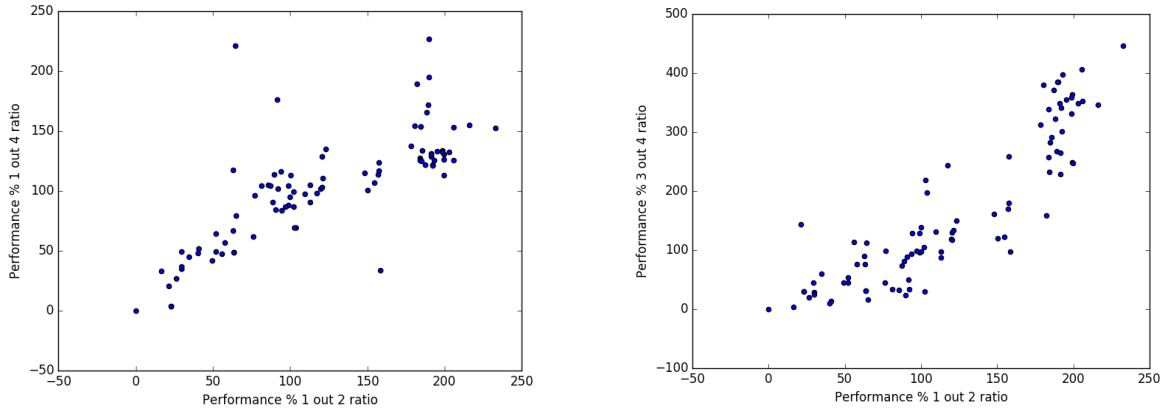

Figure 16. Each point represents the loop's speedup percent. X axis shows the improvement percent using the 1/2 ratio, while axis Y represents the improvement using 1/4 (left) and 3/4 (right).

have mixed behaviors. In the Text Search, many loops are not dealing with signal-like data, therefore the accuracy depends more on the strategy than on the ratio, and Nearest Neighbor leads to better results than Linear Interpolation. In the Classifier use case, some loops are extremely forgiving and have no impact on the classifiers' recall, while others do not allow for approximation at all.

5 Related work

The quest for energy savings and performance has made Approximate Computing an attractive research direction in the last few years[25, 29, 40]. The approaches are numerous and diverse, as they exploit opportunities in both hardware and software designs. Hardware-oriented approaches approximation techniques have proposed dynamically adaptable mantissa widths [37], approximate adders [13, 33], voltage scaling [9, 18], circuit design to exploit the accuracy-performance trade-off [4, 27, 38], and the non-determinism of hardware [6, 20, 32, 35, 36]. On the other hand, software-oriented techniques propose ways to approximate existing algorithms automatically [21, 24, 28, 31, 34] or provide support for programming using approximation [3, 5, 8, 23, 30].

Related to our technique is Paraprox [28], which also works assuming that nearby array elements are similar. However, Approximate Unrolling uses the neighbor similarity to avoid computations, while Paraprox uses it to avoid memory accesses. We did not compare both techniques as Approximate Unrolling is a scalar optimization, while Paraprox is meant for parallel computing, making difficult an objective assessment.

Finally, Loop Unrolling is a well known compiler optimization, whose gains are obtained by reducing branch jumps and loop-termination operations. Approximate Unrolling also provides gains in such way, as it also unrolls the loop. Performance gains with Approximate Unrolling are much higher as about half of the operations are removed, with the added benefit of smaller code. On the down side, Loop Unrolling increases code size, while Approximate Unrolling reduces the system's accuracy.

6 Conclusions and Future Work

In this paper, we describe Approximate Unrolling, an approximate compiler optimization. We describe the shape of the loops that Approximate Unrolling can target, the policy that can determine the opportunity for performance gains and the transformations performed to trade accuracy for speed and energy efficiency. Also, we propose an implementation of Approximate Unrolling inside the OpenJDK Hotspot C2 Server compiler. To evaluate Approximate Unrolling, we apply our implementation on four real-world Java libraries, which are representative to different application domains using intensive iterative processes (e.g., signal processing). The assessment demonstrates the ability of Approximate Unrolling to effectively trade accuracy for resource gains. We also demonstrate that Approximate Unrolling supports a more balanced trade-off between accuracy and performance than Loop Perforation, the current state of the art approximation technique. Finally, we reason on the impact of different approximation ratios and which are loops better suited for Approximate Unrolling.

To effectively program with approximation, developers must be given language, runtime and compiler support. Approximate languages allow the programmer to indicate forgiving code areas, enabling the compiler to optimize based on these hints. Similar to the way the C2 optimizes on the presence of the `volatile` or `final` java keywords, approximate compilers can act on keywords such as `@Approx` from EnerJ[30] or `loosen` from FlexJava[26]. We envision Approximate Unrolling as part of the optimization toolbox of compilers for approximate languages such as EnerJ, FlexJava or Rely[8].

References

[1] Aleksey Shipilev. Necessar(il)y Evil dealing with benchmarks, ugh, July 2013.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 177–189, New York, NY, USA, 1983. ACM.

[3] 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 *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[4] L. Avinash, C. C. Enz, J.-L. Nagel, K. V. Palem, and C. Piguet. Energy parsimonious circuit design through probabilistic pruning. In *Design, Automation and Test in Europe (DATE)*, 2011.

[5] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[6] J. Bornholt, T. Mytkowicz, and K. S. McKinley. Uncertain\textlessT\textgreater: A First-Order Type for Uncertain Data. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[7] P. Burk. JSyn - A Real-time Synthesis API for Java. In *Proceedings of the International Computer Music Conference*, pages 252–255. International Computer Music Association, 1998.

[8] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[9] I. J. Chang, D. Mohapatra, and K. Roy. A Priority-Based 6t/8t Hybrid SRAM Architecture for Aggressive Voltage Scaling in Video Applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 21(2):101–112, 2011.

[10] C. Click. Global Code Motion/Global Value Numbering. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 246–257, New York, NY, USA, 1995. ACM.

[11] C. Click and M. Paleczny. A Simple Graph-based Intermediate Representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, pages 35–49, New York, NY, USA, 1995. ACM.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[13] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy. IMPACT: Imprecise adders for low-power approximate computing. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2011.

[14] L. Haifeng. Smile - Statistical Machine Intelligence and Learning Engine, 2017.

[15] J. S. Hare, S. Samangooei, and D. P. Dupplaw. OpenIMAJ and ImageTerrier: Java libraries and tools for scalable multimedia analysis and indexing of images. In *Proceedings of the 19th ACM international conference on Multimedia*, MM '11, pages 691–694, New York, NY, USA, 2011. ACM.

[16] T. Hastie. *The Elements of Statistical Learning - Data Mining, Inference*. Springer.

[17] S. Kulkarni and J. Cavazos. Mitigating the Compiler Optimization Phase-ordering Problem Using Machine Learning. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 147–162, New York, NY, USA, 2012. ACM.

[18] A. Kumar, J. Rabaey, and K. Ramchandran. SRAM supply voltage scaling: A reliability perspective. In *International Symposium on Quality Electronic Design (ISQED)*, 2009.

[19] K. Liu, G. Pinto, and L. Yu David. Data-Oriented Characterization of Application-Level Energy Optimization. 2015.

[20] C. Luo, J. Sun, and F. Wu. Compressive Network Coding for Approximate Sensor Data Gathering. In *IEEE Global Communications Conference (GLOBECOM)*, 2011.

[21] L. McAfee and K. Olukotun. EMEURO: A Framework for Generating Multi-purpose Accelerators via Deep Learning. In *International Symposium on Code Generation and Optimization (CGO)*, 2015.

[22] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0.* Manning Publications Co., Greenwich, CT, USA, 2010.

[23] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014.

[24] S. Misailovic, D. Kim, and M. Rinard. Parallelizing Sequential Programs With Statistical Accuracy Tests. Technical Report MIT-CSAIL-TR-2010-038, MIT, Aug. 2010.

[25] S. Mittal. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.*, 48(4):62:1–62:33, Mar. 2016.

[26] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris. FlexJava: Language Support for Safe and Modular Approximate Programming. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 745–757, New York, NY, USA, 2015. ACM.

[27] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan. ASLAN: Synthesis of Approximate Sequential Circuits. In *Design, Automation and Test in Europe (DATE)*, 2014.

[28] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based Approximation for Data Parallel Applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[29] A. Sampson. Approximate Computing: An Annotated Bibliography, 2016.

[30] 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 *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[31] E. Schkufza, R. Sharma, and A. Aiken. Stochastic Optimization of Floating-Point Programs with Tunable Precision. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[32] S. Sen, S. Gilani, S. Srinath, S. Schmitt, and S. Banerjee. Design and implementation of an "approximate" communication system for wireless media applications. In *ACM SIGCOMM*, 2010.

[33] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel. A Low Latency Generic Accuracy Configurable Adder. In *Design Automation Conference (DAC)*, 2015.

[34] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM.

[35] P. Stanley-Marbell and M. Rinard. Approximating Outside the Processor. 2015.

[36] P. Stanley-Marbell and M. Rinard. Lax: Driver Interfaces for Approximate Sensor Device Access. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[37] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3), 2000.

[38] S. Venkataramani, K. Roy, and A. Raghunathan. Substitute-and-simplify: A Unified Design Paradigm for Approximate and Quality Configurable Circuits. In *Design, Automation and Test in Europe (DATE)*,

2013.

[39] C. A. Vick. *SSA-based reduction of operator strength*. Thesis, Rice University, 1994.

[40] Q. Xu, T. Mytkowicz, and N. S. Kim. Approximate Computing: A Survey. *IEEE Design Test*, 33(1):8–22, Feb. 2016.

[41] M. Åăalovarda, I. Bolkovac, and H. DomitroviÄĞ. Estimating perceptual audio system quality using PEAQ algorithm. In *ICECom (18; 2005)*, 2005.