Research Status Report: Approximate Loop Unrolling

Marcelino Rodriguez-Cancio
IRISA
263 Avenue Général Leclerc
35000 Rennes, France
marcelino.rodriguez-cancio@irisa.fr

Benoit Combemale
INRIA/IRISA
263 Avenue Général Leclerc
35000 Rennes, France
benoit.combemale@irisa.fr

Benoit Baudry
INRIA/IRISA
263 Avenue Général Leclerc
35000 Rennes, France
benoit.baudry@inria.fr

ABSTRACT

This report describes the status of the research on **Approximate Loop Unrolling** (**A-Roll**), an approximate loop optimization. **Approximate Loop Unrolling** transforms loops in a similar way **Loop Unrolling** does. However, unlike its exact counterpart, **A-Roll** does not unrolls by adding copies of the loop's body. Instead, it adds interpolations. We also describe our experimental implementation of **A-Roll** in the Server Compiler (C2) of the Hotspot Java Virtual Machine, as well as our current research roadmap.

Keywords

approximate computing; compiler optimizations; loop unrolling

1. INTRODUCTION

Several applications in computer science such as multimedia, machine learning, numerical analysis and ubiquitous computing can endure some degree of inexactitude. Indeed, the data used by these applications can have many levels of precision, allowing to trade-off accuracy for speed, energy consumption or even financial costs. This is exploited adhoc by developers using techniques specific to a domain, for example by changing the bit rate in sound or varying the time step length in simulations.

In this report we describe **Approximate Loop Unrolling** (**A-Roll**) a domain-agnostic optimization to trade-off accuracy for speed, code size and potential energy savings.

The optimization works over a common pattern consisting in a *counted loop* storing computations result's into an array being indexed by the loop's counter variable. *Counted loops* are those incrementing or decrementing a single variable (namely the *counter variable*) and stopping when the counter variable reaches certain value.

The subset of counted loops over which **A-Roll** can act represents an important part of the total existing loops in several projects. As an instance, in Apache Common Math nearly 19% of all loops can potentially be optimized using **A-Roll**.

2. APPROXIMATE UNROLLING

A-Roll works in a similar way to Loop Unrolling, but instead of unrolling the loop by adding exact copies of the loop's body, it unrolls adding interpolations. The objective of the optimization is to increase the loop's speed while reducing its size and the power consumption needed to obtain a result. As example we will use the loop of listing 1, which fills an array with a sine wave. The same loop, transformed using **A-Roll** is shown in listing 2.

```
double stride = Math.PI * 2 / N;
for ( int i = 0; i < N; i++ ) {
  A[i] = Math.sin(i * stride);
}
```

Listing 1: Motivation example: A loop filling an array with a sine wave

```
double stride = Math.PI * 2 / N;
for ( int i = 0; i < N; i++ ) {
  A[i] = Math.sin(i * stride);
  A[++i] = A[i - 1];
}
```

Listing 2: Motivation example: The loop modified using Approximate Unrolling

A-Roll has added a nearest neighbor interpolation at the end of the loop, adding code that assigns the value of `A[i]` to `A[i + 1]`.

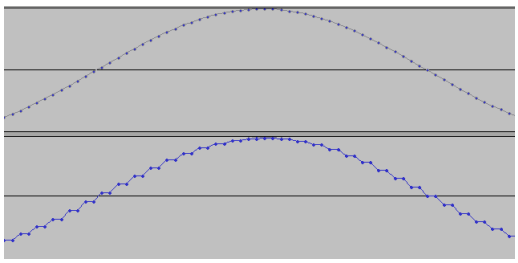

Figure 1: Sine wave generated by the motivation examples of listing 1 and 2. The upper wave is generated by the original loop, while the lower wave is generated by the degraded one.

The output of both loops is depicted in Figure 1. The wave in the top is generated by the original loop, while the

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

wave at the bottom is generated using the interpolated one. The interpolation selected is the most basic one and still acceptable results are achieved in this case. The interpolated wave introduces a harmonic distortion that is almost not perceivable by human ears.

Constraints.

A-Roll works interpolating data. This enforces the notion of distance between two data elements (in this case two array slots). **A-Roll** assumes that two consecutive data slots are logically close to each other in the application's data model. This is the case in numerous data representa tions, like sound, 3D graphics, sensor data, market trends, etc. Establishing a connection between the array and the counter variable allows us to realize that the calculation's results stored in the array are somehow close to each other in the data model. Optimally, these calculation should also have the form of a locally smooth function, so the accuracy loses remains acceptable.

This arguments forces us to determine some constraints to the application of **A-Roll** to a loop:

1. The loop to be optimize is a counted loop storing computation's results into an array.
2. Is possible to relate the loop's counter variable to the array index.
3. The calculations can be typified as a locally smooth function.

We implemented a tool to find loops abiding to these constrains that can be found in our Github repository [1].

3. IMPLEMENTATION

In this section we describe our experimental implementation of the proposed optimization in the C2 compiler of the OpenJDK Hostpot. **A-Roll** is a machine independent optimization. In the C2 compiler, all these optimizations works by reshaping the Ideal Graph, which is the internal representation (IR) of the C2 compiler. To exemplify our implementation, we will use a very simple loop:

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

Listing 3: Example loop for the implementation

The Ideal Graph.

The C2's internal representation (IR) is graph called the *Ideal Graph*. All C2's machine independent optimizations work by reshaping this graph. A detailed description of the Ideal Graph (IG) can be found elsewhere [3], we only describe what is needed to understand our work.

The IG is quite similar to the Program Dependency Graph (PDG)[4]. It also contains information on both the control and data flow, Region nodes define basic blocks and instructions inside the basic blocks have no specific order, only the one enforced by their data dependencies.

The Model of Computation of the IG works as a Petri net for Region nodes (i.e. control). Region propagates control signal into data nodes and other control nodes. Data nodes do not propagate control and their execution is supposed to be instantaneous.

The nodes in the graph represents instructions as closest as possible to assembler language (i.e. `AddI`, `MulF`), with some exceptions such as control flow nodes (i.e. `Region`, `CountedLoop`) or IO instructions (i.e. `LoadI`, `StoreL`)

The `CountedLoop` is a type of Region node that holds special interest for us because it represents the head of counted loops and contains important metadata for our implementation (i) the list of instructions in the loop's body (ii) the exact instruction that increments the counter variable.

Another specially interesting type of node is `Store`. `Store` nodes represents storages into memory and is possible to know when an storage is performed onto memory hold by an array thanks to metadata contained by the node. This information is saved into the `Store` node while the C2 parses the bytecode of the input program.

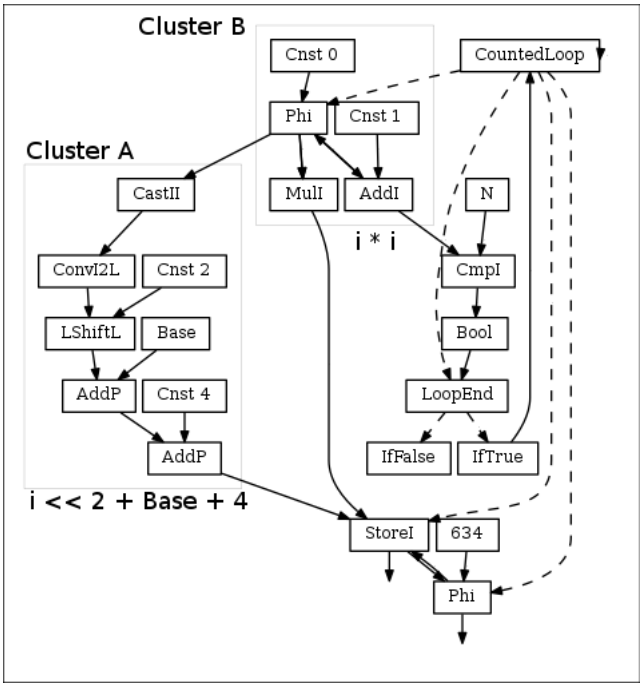

Figure 2: The ideal graph of the example loop of listing 3.

Figure 2 represents the IG for the loop of listing 3. There, the `CountedLoop` node sends control signal (dashed edges) to the `StoreI` node, which takes two inputs: the value to store in memory and the memory address to store the value to. The address is resolved by the nodes in the cluster A, containing the `LShift` node, while the value is calculated by the cluster B, including the `MulI` node.

Relating the array indexed to the loop's counter variables.

Our optimization operate only if there is an arrays indexed by loop counter variable. Therefore we need to (i) detect an array storage (ii) relate this storage to the loop counter variable.

The C2 compiler is able to detect loops using an algorithm by Vick [7] based on the work of Tarjan [6]. When our

[1]https://github.com/DIVERSIFY-project/approx-loop-counter

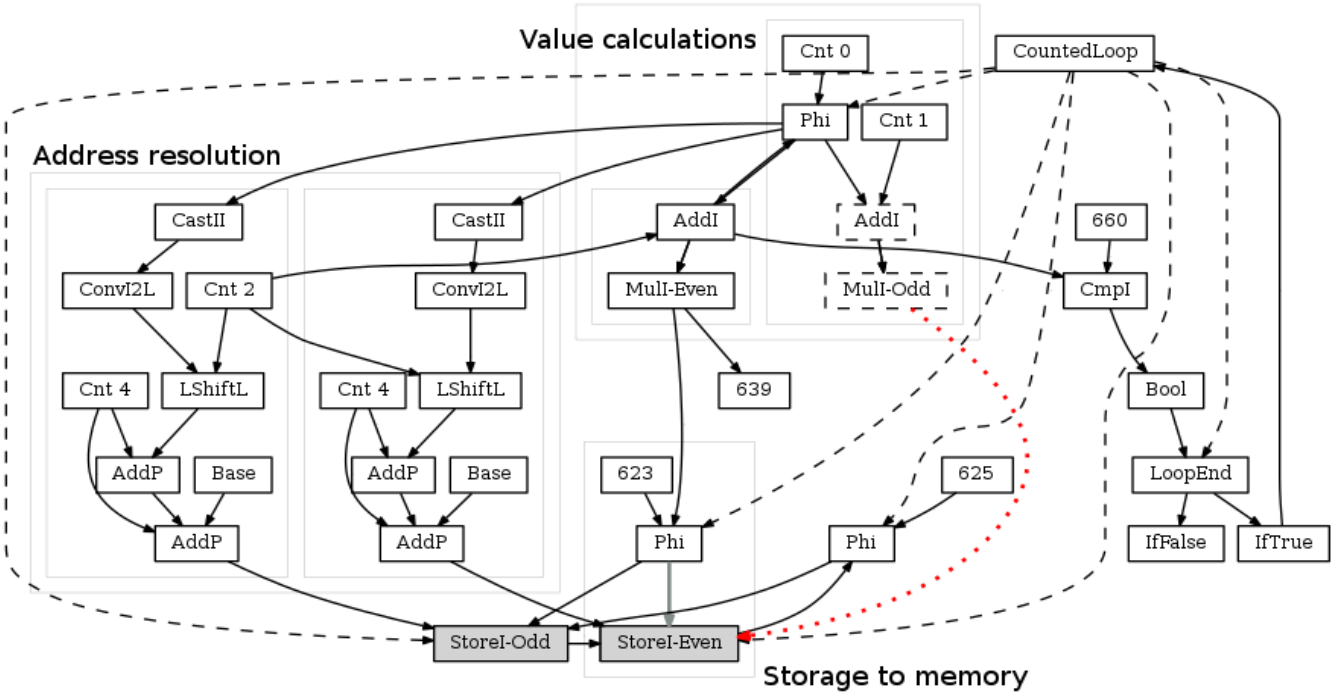

Figure 3: The ideal graph for the unrolled loop of listing 3.

optimization kicks in, the compiler have already recognized the counted loops, knows the instruction incrementing the counter loop variable and knows the instructions belonging to the loop's body using the work of Click[2].

As the IG contains information on data dependencies, by construction an `Store` node writing to an array indexed by the loop's counter variable must be connected to the increment instruction by data edges. Therefore, to find an array indexed by the counter variable, it suffices to search the instructions of the counted loop's body, looking for `Store` nodes operating over arrays. When one of such node is found, we use its address input as starting point to perform a Depth First Search in the IG, traversing the data edges, trying to reach the loop's increment instruction. If we can reach the instruction, then we have found an array indexed by the loop counter variable.

Approximating the loop.

Our implementation of **A-Roll** piggybacks on the Loop Unrolling optimization provided by the C2 compiler, since **A-Roll** takes as input the unrolled loop.

While unrolling, the compiler clones all the instructions of the loop's body, creating also clones for all existing `Store` nodes. Due to C2's design, the cloned nodes belongs to the even iteration of the loop.

Once the loop has become unrolled, **A-Roll** reshapes the graph to achieve the interpolated step. To perform nearest neighborhood interpolation, **A-Roll** connects the input value of the odd `Store` nodes to the input of their respective clones in the even iteration. The input node of the even `Store` are deleted if they becomes dead (i.e. it has no more nodes using it as input). This deletion process is recursively performed in all nodes becoming dead as the result of the first removal.

Figure 3 shows the graph after the loop has been unrolled by the C2 compiler. Solid black edges are data edges, while dashed edges are control edges. The loop had initially only one array storage: `Store-Odd` and the unrolling process added yet another: `Store-Even`. In the picture, the dotted edge is the value input to `Store-Even` which is removed by **A-Roll**. The gray edge is added by **A-Roll**, since the `Phi` node was the input to the odd `Store`. Once the dotted edge is removed, the nodes with dashed borders becomes dead (`MulI-Even` and the `AddI`) and are eliminated from the graph.

Listing 4 shows the code generated for the loop in the example without using **A-Roll**, while listing 5 shows the code generated for the same loop using our approximate optimization. In the example, the compiler has unrolled the loop twice: notice the four storages to memory. In the non approximate code, we may find four multiplication instructions (imull), while in the approximate code we can only find two.

Indicating that a loop can approximated.

Sidiroglou [5] presented the notion of *critical* and *tunable* loops. Critical loops were those that did not admit any inexactitude. Every Java application have this kind of loops (i.e. in the class loaders).

We have incorporated the `@Approximated` method annotation and modified the C2 so it recognizes this annotation. Only loops in methods annotated with the `@Approximated` annotations are approximated by **A-Roll**.

```
B8: #
movl  [RCX + #16 + R9 << #2], R8      # int
movl  R9, R10 # spill
addl  R9, #3  # int
imull R9, R9  # int
movl  R8, R10 # spill
incl  R8      # int
imull R8, R8  # int
movl  [RCX + #20 + R10 << #2], R8     # int
movl  RDI, R10        # spill
addl  RDI, #2 # int
imull RDI, RDI        # int
movl  [RCX + #24 + R10 << #2], RDI    # int
movl  [RCX + #28 + R10 << #2], R9     # int
addl  R10, #4 # int
movl  R8, R10 # spill
imull R8, R10 # int
cmpl  R10, R11
jl,s  7
```

Listing 4: Assembler code generated for the example loop without using A-Roll

```
B7: #     B8 <- B8  top-of-loop Freq: 986889
movl    RBX, R8 # spill
B8: #
movl    [R11 + #16 + RBX << #2], RCX    # int
movl    [R11 + #20 + R8 << #2], RCX     # int
movl    RBX, R8 # spill
addl    RBX, #2 # int
imull   RBX, RBX        # int
movl    [R11 + #24 + R8 << #2], RBX     # int
movl    [R11 + #28 + R8 << #2], RBX     # int
addl    R8, #4  # int
movl    RCX, R8 # spill
imull   RCX, R8 # int
cmpl    R8, R9
jl,s    B7
```

Listing 5: Assembler code generated for the example loop using A-Roll. Notice is a much more shorter code with only half of the arithmetic operations

4. ROADMAP

In order to effectively implement **A-Roll**, there are several challenges to be addressed.

Detect loops that can be approximated using **A-Roll** .

In this status report, we mentioned some constraints to the shape of the loops that our approximative optimization can work on. In our immediate research path lies to incorporate the last of these constraints (having a locally smooth function) into our implementation. The challenge here is that detecting such form of function is costly. Perhaps a fast way of determining it is by comparing the differences in values between two iterations in the loop. If the differences are too high it would mean the data is not smooth and therefore cannot be interpolated.

We will also work on determine the domains on which **A-Roll** be exploited successfully to obtain good results in terms of accuracy lost, speed gain and energy savings. At this point we have obtained good experimental results with signal processing, but our intuition tell us that the technique can be applied in other fields as well.

Interpolation's performance.

Another challenge to be addressed is that the interpolated iteration must perform always better that the exact one in terms of speed an energy consumption. Several interpolation strategies can be used, nearest neighbor, linear, polynomial. The challenge will be to select one that performs better that the exact computations and yet produces acceptable results.

Using static analysis we can achieve some estimations that can be less or more accurate depending on the ability to determine execution paths. Since we aim at implementing our optimization in the C2 compiler, we can also have some runtime information and we must came up with strategies to get the most out of the data the compiler has to offer.

Evaluation.

Determining whether there is an effective gain in terms of performance is a challenging task that requires great deal of care and expertise due to the many sources of non-determinism that may arise in the CPU, the compiler and operative system [1]. The experiments to evaluate our technique would require a thorough planing and then, the results should be validated through careful analysis

5. CONCLUSIONS

In this report we have described **A-Roll**, an approximate optimization. We enumerate the challenges that must be tackled in order to effectively achieve this optimization. Finally, we report our progress to implement the optimization in the Hotspot JVM.

6. REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] R. Tarjan. Testing flow graph reducibility.

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