Code Perforation with LLVM
EECE 571p: Program Analysis and Optimization

Ken Mansfield
kmansfield@ece.ubc.ca
Department of Electrical and Computer Engineering
University of British Columbia

Abstract—Approximate computing describes a class of algorithms that are designed to tolerate some loss of quality or accuracy in return for increased performance. Modern computations, such as multimedia encoding and machine learning, can be adjusted to meet the requirements of the user. Such algorithms generally require domain-specific techniques developed specifically to fine tune the computation's performance.

Loop perforation is a technique for improving the performance of approximate computations by reducing the number of iterations executed. An optimization algorithm identifies tunable loops that respond well to perforation and automatically generates variants of approximate computations. The result is a set of ideal perforation rates for each of the tunable loops that return significant speedups while maintaining sufficient accuracy. Perforation is performed by an LLVM compiler pass which modifies the method in which the induction variable is incremented in loops such that entire iterations are skipped. The implementation is evaluated by applying code perforation to applications contained within the PARSEC benchmark suite. Results show that significant speedups in code execution time are achievable while maintaining reasonable accuracy.

Index Terms—Compilers, Approximate Computing, LLVM, EECE 571p, Program Analysis, Optimization.

1 Introduction

TRADITIONALLY, algorithm designers have strived for increasing accuracy following the notion that every computation should be executed correctly. Approximate computing aims to trade the accuracy of computation for gains in the domains of energy and performance [5]. This paper presents a technique for automatically improving the performance of such existing applications by performing less computations to improve speed while maintaining reasonable accuracy.

This technique, loop perforation, improves performance by skipping entire iterations within loops. Reducing the number of iterations that are executed in turn reduces the amount of computational work that is done, along with the amount of energy. Modifying loop structures within an application does have an effect on the correctness of an application, so the goal is to find loops that can be perforated without producing errors or crashing. Approximate computations can generally tolerate perforation but this often results in reduced accuracy.

• *K. Mansfield is a Masters student in the Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC.*

Manuscript received April 27, 2016.

1.1 Approximate Computing

Approximate computing is not a new idea, many areas already use it such as multimedia which uses lossy compression algorithms which trades accuracy for faster performance and smaller file sizes [3]. JPEG compression, for example, samples an image in the frequency domain and filters out high frequency information that may not even be perceivable by the eye to produce a file with a smaller size.

Approximate computing is particularly essential for a classification of computations called NP-Complete, whereby optimal solutions cannot be found within a reasonable amount of time. One example of such an algorithm is graph partitioning.

Finding an optimal solution to graph partitioning for a system with many thousands of nodes may take years for a normal computer to process. To work around this, many heuristic search algorithms have been created that can be shown to find a good solution (quasi-optimal) in a reasonable amount of time, such as Kernighan-Lin [7] and Simulated Annealing [6]. Heuristic algorithms often employ a search algorithm which allows sub-optimal solutions to be taken with a certain probability some of the time. This allows the algorithm to explore a greater region of the state-space with the hopes of finding a global minimum rather than getting stuck in local minima. Allowing the algorithm to search more random moves generally improves the accuracy of the solution while increasing the time it takes to compute.

Another fast growing area which depends on approximate computing are Machine Learning algorithms. Reinforcement learning attempts to imitate the way humans and animals learn by rewarding good behavior and punishing bad behavior. In reinforcement learning, the algorithm can only learn from an action if it has taken that action in the past [8]. In most cases, it is impossible to explore the entire state-action space for the whole system as there may be billions or trillions of possible combinations, so approximate results must be computed. Reinforcement learning algorithms generally employ epsilon-greedy search which explores random actions with a probability proportional to epsilon, which decreases over time as a more optimal solution is eventually achieved. The rate at which epsilon is decreased directly affects the amount of time it takes to find a solution. If epsilon is decreased too quickly a less optimal solution is found, however, the time taken to train is proportionally quicker.

2 SOLUTION

2.1 Identify Loops

The first step is to identify all the loops in the test application that have the potential to be perforated. Only loops that can be transformed into a simplified form resembling a standard *for loop* will be considered. The perforator only works with loops that have a constant number of iterations. Traditional loops come in the form of:

$$for(int\, i = 0;\ i < x;\ i{+}{+}) \tag{1}$$

Loops in this form have an induction variable i that is incremented linearly on every iteration terminating when the induction variable exceeds the bound, x.

2.2 Perforate Loops

The main idea of loop perforation is to skip a certain number of iterations in a loop in a manner that will increase performance while maintaining reasonable performance. With a standard loop this can be done by modifying the rate at which the induction variable is incremented. The loop perforation rate can be specified as:

$$r = 1 - \frac{1}{n} \tag{2}$$

This would result in the loop:

$$for(int\, i = 0;\ i < x;\ i \mathrel{+}= n) \tag{3}$$

Where n is the number the induction is incremented in each iteration. Given n = 2, the perforation rate r would be 0.5, signifying half of the iterations would be skipped.

2.3 Feedback Loop

A program can be composed of many different loops which may be perforated or not. Some loops may crash if perforated and some may not respond well to perforation whereas some may respond very well. To explore the set of loops that may be tunable and tune their corresponding perforation rates we need a feedback loop which keeps track of this information and explores the response of the application to different values. Each loop will be tested with perforation one by one to test whether it causes the application to crash, and to determine its performance. To judge whether the perforation maintains reasonable accuracy, a calculation to define accuracy degradation is defined as:

$$AccuracyDegradation = \frac{|A_o - A_p|}{|A_o|} \tag{4}$$

Where A_o is the output accuracy of the original application without perforation and A_p is the accuracy of the perforated application being tested. To calculate performance, speedup is defined as:

$$Speedup = \frac{Sp_p}{Sp_o} \tag{5}$$

2.4 Testing and Optimization Algorithm

1) Run a compiler pass to transform all loops to a simplified form.

2) Run an analysis pass to identify and count the number of all transformable loops.

3) Make an initial run of the program without performing any perforation to get its initial execution time and accuracy.

4) Given the number of loops, modify the perforation rate for one loop at a time for each of the loops and execute the program while storing the results.

5) Given the results from the pass in 2, any loops that have crashed and any loops that did not have an effect on performance can be removed.

6) Continue modifying the perforation rates of each loop while using a sensible algorithm for zeroing in on an optimal set of perforation rates that achieve a suitable accuracy vs performance tradeoff.

Algorithm 1 Testing Phase: Find all Candidate Loop that do not Crash and Respond Well to Perforation

```
1:  procedure TESTINGLOOP
2:      LoopSimplify()
3:      $numLoops \leftarrow$ IdentifyLoops()
4:      InitializePerforationRatesToZero(numLoops)
5:      $originalTime \leftarrow$ Application
6:      $originalAccuracy \leftarrow$ Application
7:      $i \leftarrow$ numLoops
8:  loop:
9:      PerforateLoop(i)
10:     RunApplication()
11:     if Crashes then
12:         RemoveLoopFromCandidates(i)
13:     else
14:         if $execTime > origExecTime$ then
15:             RemoveLoopFromCandidates(i)
16:     $newTime \leftarrow$ Application
17:     $newAccuracy \leftarrow$ Application
18:     $i \leftarrow i + 1$
19:     if $i < numLoops$ then
20:         goto loop.
```

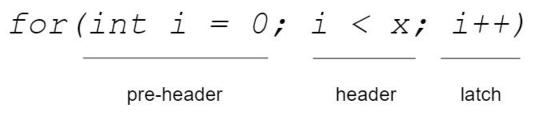
3 IMPLEMENTATION

3.1 Loop Info

LLVM has a transformation pass *loop-simplify* that attempts to transform as many natural loops as possible into the standard form. A natural loop is defined as a loop with a single entry-point and has a back edge whose head dominates its tail. The natural loop is then the smallest set of nodes that includes the head and tail of the back edge and has no predecessors outside the set, except for the predecessors of the header. The *Loop-Info* analysis pass already performs this natural loop detection for us which the loop-simplify pass uses to identify each loop it can potentially simplify, for example, by splitting apart nested loops.

After the loop-simplify pass has been run then the *LoopInfoWrapper* can be used to access each loop, as well as give a count of all the loops that can be potentially perforated. Each time the perforation pass is run, an input file is read to determine the parameters for each pass. For the initial pass, the automation script will set the state of the input file to indicate that the first pass will run with no perforation, and it will indicate to the pass that it wishes to receive information about all the loops contained within the application. This initial pass will create a space seperated file indicating the loop number (in the order it is discovered), as well as its potential to be perforated.

The LLVM runOnLoop is executed for each loop that LLVM has identified. The loop pointer, which is passed in as a variable in the runOnLoop function, can be checked to determine whether it has all the characteristics that we expect for a standard loop: the pre-header, header, and the latch. If the loop has all three of these, and the header ends the loop with a conditional branch, then we indicate that the loop is perforatable in the output file that will be read in by the automation script.



```
for(int i = 0; i < x; i++)
```

Fig. 1. LLVM parts of a loop.

3.2 Implementing Perforation

Using LoopInfoWrapper each loop can be processed individually in the runOnLoop(). The automation script will indicate the perforation rate for each loop in the input file that is read by the perforation pass. The start of a loop begins with initializing the induction variable to zero (or any other constant) in the pre-header. The header contains the branch condition that determines when the loop will end. The latch is generally where the induction variable is incremented on each iteration. To perforate this loop, we search for the add instruction contained in the lath, and change the second operand. For 50% perforation, the 2nd operand can be set to 2 (ie. i += 2). For perforation rates of less than 50%, simply setting changing the second operand of the add operation in the latch is not sufficient. Additional instructions need to be added to the latch to increment the loop on every ith iteration. One way to do this is to add an additional branch instruction that does not branch to the body on every ith iteration. The compare instruction can check the last 2 bits of the induction variable to see if they are 0, for example, to achieve a perforation rate of 25%. The added instructions and the lessened amount of perforation make this approach less desireable.

for(int i = 0; i < x; i += 2)

Results in the IR:

```
%11 = add nsw i32 %10, 2
```

3.3 Automation Script

The feedback loop in the solution will come in the form of a python script. The script will start by running a pass with no perforation to gather information about the number of loops and their potential for perforation. Once all this preliminary processing has occurred, the perforation can begin. The script writes to file a list representing the perforation rates for each loop within the application. For the first round, the perforation rate of one loop at a time is set to 50% in the output file. Next, the pass is performed and the application is compiled and executed. The time taken to execute and the resulting accuracy is recorded.

On execution, perforation of some loops may cause the application to crash. The script will need to monitor applications that terminate unexpectedly. The possibility also exists that computations may diverge resulting in infinite loops, so the script will also need to monitor applications that fail to terminate. This can be done by simply abandoning an application whose execution time exceeds the time of the original, unperforated application. Any perforated application whose execution time is longer than the original application has shown that it has failed to increase performance and will be added to the list of loops that are not suitable for perforation.

3.4 Evaluation

Every approximate computing application produces a different result and must be evaluated seperately. To accomodate this, a seperate Python script is written for each application that can indicate the accuracy degradation by comparing the results of the original application against the perforated application. To do this, the application must output a file with results that the evaluation script can read as input.

4 Experimental Setup

This section describes the methodology used to evaluate the code perforation techniques being proposed in this paper. Code perforation is known to perform significant speedups in computationally expensive applications which spend most of their time in simple loops. Applications used for benchmarking typically fit this description.

4.1 Logistics

The loop perforation code will be written in C++ using LLVM 3.7.1. The automation script is written in Python 2.7. The results will be collected on a dual core Intel Core-M3 @ 1.8GHz running Ubuntu 14.04 in a virtual machine.

4.2 Test Applications

To fully evaluate the system, a set of approximate computation benchmarks will be used from the popular PARSEC 3.0 benchmark suite [4]. These benchmarks represent a diverse set of computational workloads which have been designed to test future generations of CPU architectures. Each one of these will have to be modified to output data for the time taken to complete the computations as well as a metric for evaluating the accuracy.

4.2.1 Canneal

This application uses cache-aware simulated annealing to perform VLSI cell placement with the aim of minimizing the routing cost of a circuit design. Simulated annealing is a probabilistic heuristic that is used to approximate the global optimum given a cost function. The algorithm imitates the metal annealing process by starting at a high temperature, T, which allows the cost function to search for solutions by allowing many random moves. As T is reduced, less random moves are allowed resulting in the cost function converging to an optimal solution. Accuracy will be determined by the difference in routing cost between the perforated and non-perforated versions.

4.2.2 Blackscholes

This application uses the popular Black-Scholes partial differential equation to calculate the price for a portfolio of stock options. Accuracy is calculated by comparing the price of the options determined by the perforated and non-perforated versions.

4.2.3 x264

This application performs H.264 encoding on video data and outputs a file. Accuracy can be determined by comparing the bitrate information between the original and perforated versions.

4.2.4 Bodytrack

This is a computer vision application that uses an annealed particle filter to track the movement of a human over time. Vectors representing the positions of the body over time are generated as well as corresponding bitmap images. Accuracy can be determined by comparing differences in position vectors.

4.2.5 Swaptions

This is a financial analysis application used to price a portfolio of swaptions using Monte Carlo simulations to solve a partial differential equation. Accuracy can be determined by calculating the price of the perforated swaption and comparing it to the calculated version from the original.

5 Related Work

This paper is inspired by the work performed by Hoffman et al in the paper [1].

5.1 Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures

[1] proposed the technique known as code perforation for automatically augmenting existing algorithms with the ability of trading off accuracy in return for performance. They have implemented their own compiler, SpeedPress, which automatically applies code perforation to existing algorithms without the need for developer intervention. The resulting application, post compilation, is a transformed computation with increased performance while keeping accuracy degradation within acceptable user-defined bounds. They also present the SpeedGuard system that uses code perforation to allow applications to automatically adjust the amount of code perforation in response to changes in the application environment.

The compiler profiles the application to determine the performance and accuracy tradeoffs that are possible by removing loop iterations from the program. It does this by attempting to perforate each loop. If perforating a loop does not cause a speedup, or causes unacceptable accuracy degradation or causes the program to crash, then that loop is removed from the list of loops that can be perforated. When this process is completed, the compiler has a complete list of speedup and accuracy degradation numbers for each candidate loop. Then they order the candidate loops by their speedup (the execution time before speedup divided by the execution time after perforation).

5.2 Managing Performance vs. Accuracy Tradeoffs With Loop Perforation

The same group in their paper [2] extended their work to search for Pareto-optimal perforation policies from the set of tunable loops that were discovered by SpeedPress. Their technique presents exhaustive and greedy algorithms for exploring the perforation space to find a set of Pareto-optimal perforations.

Their exhaustive exploration algorithm starts with the set of tunable loops and explores all combinations of loops with different perforation rates. They also run each combination under Valgrind to detect memory errors. Their evaluation of the exhaustive exploration algorithm shows that it is extremely processor intensive for anything more than simple applications due to the number of combinations executed as well as the use of Valgrind. Using a cluster of processors with 64 cores in total, the exhaustive exploration took 65 hours to find the solution for Parsec streamcluster benchmark.

6 Experimental Results

Large speedups are easily achieved by perforating loops within the test applications. Evaluating the resulting accuracy degradations is much harder and requires good understanding of the application. For the blackscholes parsec benchmark, for example, using the proposed evaluation metrics described by [9], speedups of 261% are achievable while only inducing an accuracy degradation 16.1%. Upon further examination of the resulting output data, it is evident that the perforation speedup is achieved by simply not computing the value on every perforated loop, resulting in a 0.0 stock option price in the output file. In the finance world, this result would be completely unnacceptable. If the evaluation is instead changed to one which determines the error as:

$$error = \frac{|Original_i - Perforated_i|}{|Original_i|} \tag{6}$$

then the accuracy degradation reaches 50% for a perforation rate of 50%. This is inline with the findings in [2] which shows that they were unable to find any speedup with an acceptable error margin.

The optimization algorithm tests each individual loop with a perforation rate of 50% and 66%. This gives us data showing the speedup and accuracies achieved by each loop at different perforation rates, however, an exhaustive search of different combinations of perforations for different loops was not performed due to time constraints. It should be fairly trivial to test multiple different sets of loops that have shown to respond well to perforation, and return the set with the best results.

Benchmark	Accuracy Degradation	Execution Time (s)	Speedup Factor
Canneal	3.2%	5.23s	4.5%
Blackscholes	16.1%	2.1	261%
fluidanimate	2.6e-08	0.124s	32%
Bodytrack	-	-	-
Swaptions	-	-	-
x264	-	-	-

TABLE 1
Sample of experimental results after perforating each benchmark. Note 1. Blackscholes result using evaluation proposed by [9]. If changed to a strict evaluation where no 0 values are accepted, then there is no acceptable perforation. 2. fluidanimate accuracy evaluations come from [9] appear to be order of magnitudes off. 3. Perforation of Bodytrack, Swaptions, and x264 not completed due to time constraints.

Benchmark	Time to optimize
Canneal	29.336s
Blackscholes	506.712s
fluidanimate	342.526s

TABLE 2
Total time required to optimize each application. The number of loops, as well as the size of the test dataset used, greatly influences the time require to optimize.

7 Discussion

7.1 Strengths and Weaknesses

Not all applications are suitable for perforation. Most applications compute an exact result and will not respond well to missing iterations. For this reason the test applications are restricted to approximate computations. Applications that compute approximate results generally take in parameters that determine the number of iterations that computed. Simulated annealing algorithms [6], for example, have an annealing schedule which define the start temperature, and the number of iterations run per temperature step the outer and inner loops respectively. The annealing schedule is typically carefully tuned by a software developer to produce the accuracy they require within their required performance specification. This raises the question of whether it is preferable to tune the application variables or use perforation to increase perforation.

It is likely that a competent developer would be able to tune the application with finer grain using the application variables rather than by perforation. The benefit of perforation arises in situations where the user does not have knowledge about how the application works and would rather use the code perforation to automatically tune the application. Perforation may be able to find efficiencies that the developer may overlooked, or it may fail entirely by causing the application to crash or diverge. This goes against normal compiler optimization design which aims to optimize the performance of applications without changing the way the underlying application functions.

Perforation may have profoundly negative impact on the way an application behaves. One cannot optimize a slow running word processing application by simply removing random words from the document. This means that a developer may need to selectively tag loops that perform approximate computations that are suitable for perforation, effectively defeating the purpose of performing automated optimization. The use of code perforation will most likely be limited to very few, academic examples.

After perforation has been completed, it is likely that resulting application would only work well with the exact same inputs. A video encoding application that has been perforated with a specific video being used as the input will not produce the same results if a different video is being used. It is possible that the perforated application will not provide optimized results with the new input, and possibly output incorrect results, or crash entirely.

7.2 Automation

The ultimate goal of having an automatic optimizing compiler is that it may take a poorly performing application and make it run faster with minimal accuracy degradation while requiring very little user input. In a perfect scenario, the user would not need know the inner workings of the application, or may not need to be a knowledeable software developer at all. However, it is clear that this is not the case and that user interaction is neccessary. In particular, the evaluation script used to compare perforated applications against the original application must be handwritten for each new application. Writing each evaluation script requires knowledge about how the application works and what the output should look like. The blackscholes parsec application, for example, requires the user to know that the outfile describes prices and that the perforated output should have values that do not differ by too much. In other applications, for example, canneal, no file was output at all, and required the code to be edited such that the resulting value would be written to an output file.

A new Makefile specifically written for each application is necessary to run the perforation pass and compile the perforated application. However, it is conceivable that with enough time a more generic Makefile could have been created that would work with different codebases.

7.3 Time Required to Optimize

In the paper [2], the authors performed exhaustive optimization to find the set of parameters that would produce the best perforation results for a given application. They also decided to use Valgrind in an attempt to find all memory errors in the resulting perforated application. The combination of these two efforts resulted in extremely long optimization times requiring a large cluster of computers which may not be available to the average user. On the other hand, if automatic perforation optimization is performed successfully, it would only need to be done once.

For the purposes of this paper, Valgrind will not be used. If the application is able to perform accurate computations without crashing, then it does not matter if memory errors occur. To speed up the optimization process, a less exhaustive search will be performed as well.

7.4 Future Work

Due to time constraints, only 3 benchmarks were tested. The perforation pass, as well as the optimizing script, is able to adapt to a large number of applications (possibly any application), however, a significant amount of time must be spent to write both a Makefile, and an Evaluation script for each application. Many of the applications did not output a file with results that could be evaluated. Others had more complex directory structures, mixed .c and .cpp files, and required additional included libraries which increased the complexities of their required Makefiles.

The testing was done on a Linux Virtual Machine. On each run of the optimizations, it was found that the execution time varied slightly. This was likely due to background processes running on the operating systems. This may lead to small inaccuracies in the recorded speedups. This slight difference in execution could cause the optimization script to incorrectly identify the perforation of a candidate loop as incurring a larger execution time than the unperforated version, leading to it being added to the list of loops unsuitable for perforation. In the future, the tests should be done on a system with absolutely no background processes running to ensure accuracy.

Parallelization would allow the optimization process to occur much faster if a machine with multiple cores, or a cluster is used. Since each compilation and execution of the program is run independently, it would be fairly trivial to execute each test on a seperate thread or process. For the purpose of this paper, however, a low performance dual core processor was used and would unlikely benefit from parallelization. With a faster multi-core machine, or cluster, a much more exhaustive optimization search can be completed in a more reasonable amount of time.

References

[1] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard. *Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures*. Technical Report MIT-CSAIL-TR-2009-042, MIT, Sept. 2009.

[2] Sidiroglou-Douskos, Stelios, et al. *Managing performance vs. accuracy trade-offs with loop perforation*. Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, 2011.

[3] Samadi, Mehrzad, et al. *SAGE: self-tuning approximation for graphic engines*. Proceeding of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2013.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. *The PARSEC benchmark suite: Characterization and architectural implications*. PACT '08, Oct 2008.

[5] J. Han, M. Orshansky, *Approximate computing: An emerging paradigm for energy-efficient design*. 18th IEEE European Test Symposium. ETS, 2013

[6] D.S. Johnson, C. Aragon, l. McGeoch, and C. Schevon, *Optimization by Simulated Annealing: An Experimental Evaluatoin, Par 1, Graph Partitioning*. Operations Research, vol. 37, pp. 865-892, 1989.

[7] B. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*. Bell Systems Technical Journal, vol. 49, pp. 291-307, 1970

[8] Sutton, R.S. *Learning to predict by the methods of temporal differences*. Machine Learning 3: 9-44, erratum p. 377. 1988

[9] Adrian Sampson, Andre Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. *ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing.* Technical Report UW-CSE-15-01-01. University of Washington. 2015

Ken Mansfield M.Eng candidate, B.A.Sc.'06 Electrical Engineering, University of British Columbia