Loop Perforation for Approximate Computing

Joseph Lee and Naman Jain

April 30, 2014

1 Introduction

1.1 Problem

Computer systems should to be correct at all costs. Though error-free execution is beneficial, the tradeoff is high execution time. For many applications, perfect correctness is unnecessary and todays over-provisioned systems waste time and energy providing precision they dont need. For example, while trying to smooth an image, the exact output value of pixel can vary. If the output quality is acceptable for the user or the quality degradation is not perceivable, approximation can be employed to improve the performance. Many image, audio, and video processing algorithms today use approximation techniques to compress and encode multimedia data to various degrees that provide trade-offs between size and correctness, such as lossy compression techniques [SM13].

1.2 Approach

We implemented loop perforation, that automatically helps improving the performance while trading-off some accuracy, within the user acceptable level. Our optimization identifies parts of the computation, specifically loop iterations, that can be skipped without violating the user specified Target Output Quality (TOQ). The result is a computation that performs less work (and therefore consumes fewer computational resources) while still producing acceptable output.

1.3 Related Work

Trading accuracy for performance is quite popular technique. Our work is inspired by Sidiroglou et al. at CSAIL, MIT [SDS13]. They proposed a new technique code perforation, for automatically augmenting existing computations with the capability of trading off accuracy in return for performance. In contrast to existing approaches, which typically require the manual development of new algorithms, their implemented SpeedPress compiler can automatically apply code perforation to existing computations with no developer intervention. The result is a transformed computation that can respond almost immediately

to a range of increased performance demands while keeping any resulting output distortion within acceptable user-defined bounds.

1.4 Contributions

We implemented two of the loop perforation optimizations described by Sidiroglou et al, including the primary optimization using LLVM. We also have a script that can be used to discover which loops in a program are eligible for perforation at a certain stride.

2 Implementation Details

We first implemented two types of perforation passes. One is the traditional perforation optimization described by Sidiroglou et al, and the other is an alternate pass based on modulo rather than stride.

The traditional optimization first requires that loops have canonical induction variables, i.e. an induction variable that increments by one each iteration. In order to maximize the number of canonical loops in a program, we first run the llvm canonicalize induction variables pass (`-indvars`). This will transform some non-canonical induction variables into induction variables.

The basic loop perforation optimization turns loops of the form

```
for(int i = 0; i < bound; i++) { ... }
```

into loops of the form

```
for(int i = 0; i < bound; i+=stride) { ... }
```

where the stride value is determined beforehand. In order to select the stride value, we do analysis on the program and loop, which we will describe later.

The modulo loop perforation optimization takes the idea of dropping every Nth iteration of a loop. This pass transforms loops of the form

```
for(int i = 0; i < bound; i++) { body }
```

into loops of the form

```
for(int i = 0; i < bound; i++) { if(i % n != 0) { body } }
```

where like the stride value, n is determined through analysis. In LLVM this pass requires the creation of two new basic blocks in the loop – one for checking the induction variable modulo n and one for jumping back to the program head. In order to do a correct transformation, our pass also had to recompute phi nodes in the loop header.

With these optimizations, the difficult part of the optimization lies in the selection of the loops. The candidate loop search attempts to perforate each loop in loops L with strides K. Loop perforation has four possible results:

1. The perforation is successful, and we see speedup of the program while meeting the bounds set by the TOQ.
2. The perforation is successful, but we do not meet the bounds set by the TOQ.
3. The perforation causes a crash in the program.
4. The perforation causes the program to diverge (e.g. a stride of 8 causes the program to infinite loop).

With this in mind, we implemented the search in python, which runs different configurations of the loop perforation pass and tests to see that the new program does not diverge or crash. Unfortunately, TOQ bounds cannot be easily computed by a general program like our python script due to the unlimited different types of output that can be generated. A unique python script must be written for each new program that we wish to optimize.

3 Experimental Setup

We evaluated loop perforation on a set of benchmark applications from the PARSEC 3.0 benchmark suite [C11]. These applications are representative of modern and emerging workloads and all the evaluations were compiled with the standard clang and LLVM compiler settings.

We verified the idea of loop perforation on x264, blackscholes, streamcluster, swaptions and bodytrack applications which represents a broad range of applications ranging from media processing, computer vision, data mining to financial analysis.

A brief description of the benchmarks:

x264 This is a media application that performs H.264 encoding on raw video data. It outputs a file encoded according to the H.264 standard. The abstracted output after encoding has information on frames per second (fps) and bitrate (kbps) of the video. We observed that fps varied on every execution, therefore accuracy was calculated using bitrate information.

Bodytrack It is a computer vision application that uses an annealed particle filter to track the movement of a human body. It generates vectors representing the positions of the body over time and also corresponding bmp image. We used differences in position vectors to compute accuracy for this benchmark.

Streamcluster It is a data-mining application which solves the online clustering problem. It generates set of cluster centers found for the input passed. We used these values for computing our accuracy.

Blackscholes It is financial analysis application that solves a partial differential equation to compute the price of a portfolio. It generates prices in a file which was used for computing accuracy for this application.

Swaptions It uses Monte-Carlo simulations to solve a partial differential equation. It generates swaption prices, which was used for computing accuracy.

4 Experimental Evaluation

For each benchmark suite, PARSEC has provided simdev inputs which causes code execution comparable to a typical input for this programme. We used these inputs for all our analysis. A number of benchmarks like x264, facesim, dedup, vips of PARSEC benchmark suite did not successfully compile with LLVM, therefore we verified a few of the benchmarks using gcc.

For each loop in the program we ran different configurations of the loop perforation pass and tests to see that the new program does not diverge or crash. Based on this analysis we selected candidate loops and applied loop perforation on them.

Table 1 shows results obtained after perforating selected loops with different perforation ratios. 100% perforation ratio means that no perforation is performed, i.e. stride is 1. 50% perforation ratio means that every other iteration was skipped, i.e. stride is 2 and similarly, perforation ratio 25% means only 1 of 4 iterations are implemented.

Accuracy was calculated based on the abstracted outputs as mentioned in previous section.

Accuracy Metric Used: Difference in output abstraction components and original output divided by original output component.

$$acc = \frac{|o - \hat{o}|}{|o|}$$

where o is the original output and $\hat{o}$ is the obtained output after perforation.

To measure speedup execution time of the program was first calculated using our python script, but the precision of the results were too low to come to a conclusion, therefore we added built-in C/C++ functions in the main function of the benchmarks to calculate time differences. Execution time and their respective speedup is demonstrated in Table 1. Figure 1 shows speedup obtained due to different perforation ratios.

4.1 Analysis

x264 `Encode` function in x264.c divides each frame into blocks and performs encoding for individual blocks. Most of the execution time is spent to find a block from previously encoded frame that is similar to block that x264 is currently attempting to encode. Therefore, perforating hot loop in the function causes x264 to search for fewer reference frames and thereby, trading a bit of accuracy.

It is interesting to note that we get less speedup when stride is 4 compared to the result when stride was 2. It could be the case as skipping 3 iterations for this small video leads to large differences between previously encoded

Applications	Perforation Rate (%)	Accuracy Loss (%)	Time Taken (μs)	Speedup
x264	100	-	3047	1
	50	0.47	1652	1.8443
	25	61.52	1967	1.549
bodytrack	100	-	28.2	1
	50	4.35	15.6	1.8076
	25	4.55	12.2	2.3114
streamcluster	100	-	891	1
	50	0	649	1.3728
	25	0	502	1.7749
blackscholes	100	-	8243	1
	50	0	7691	1.0717
	25	75.1	1411	1.5386
swaptions	100	-	2171	1
	50	49.8	1525	1.4236
	25	75.1	1411	1.5386

Table 1: Accuracy and speedup obtained due to different perforation rates

frame, and therefore more computation would be required. As expected accuracy was reduced drastically when changing stride to 4 for this small video application. It was also noticed that skipping 1st iteration leads to high accuracy penalties. Thus, a special check should be added for such applications.

Bodytrack The candidate loops try to compute a metric that characterizes body pose. This requires set of points that represent body position. On perforation, loops check for a subset of those points, which does not lead to a lot of accuracy loss.

Streamcluster It is a data-mining application which solves online clustering problem. It generates set of cluster centers found for the input passed. On perforating candidate loops, f ewer attempts to improve clustering are made. Though we see in the Table 1 that accuracy loss is 0, but for certain strides, excessive clusters are reported and for some a subset of clusters are reported, which thereby does make perforation very helpful in this application.

Blackscholes It was interesting to note that on perforating loops, output did not change at all. Accuracy was 100% and was not breaking as was in streamcluster.

Swaptions It was noted that perforating candidate loops leads to computation of swaption prices with fewer Monte-Carlo simulations and thus accuracy tends to reduce proportionally with respect to perforation ratio.

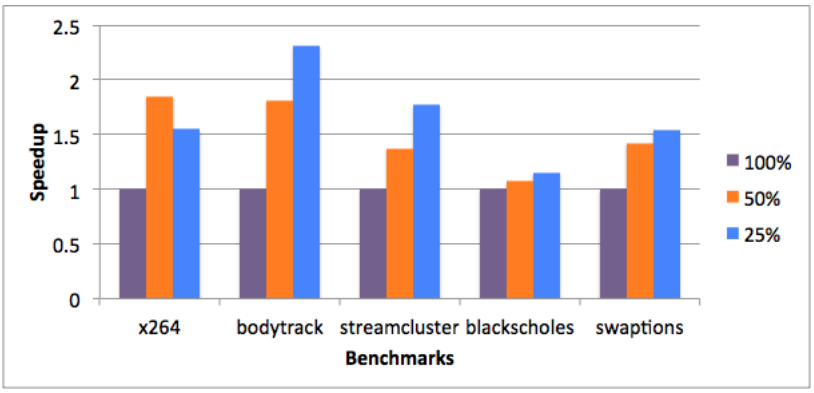

Figure 1: Speedup due to loop perforation on different benchmarks

5 Surprises and Lessons Learned

One of our surprises is how many programs are tolerant of loop perforation optimizations. Many programs in our benchmark did not crash after perforation of either variety.

Secondly, it was quite surprising to find that generally results do not change when skipping iterations of a loop. It is probably due to the fact that a lot of applications try to repeatedly improve an approximate result to obtain a more accurate result.

One lesson learned is that Phi-Nodes can make CFG rewriting tricky. Even a conceptually simple pass like the modulo perforation pass can become an exercise in Phi-Node bookkeeping.

6 Conclusions and Future Works

Loop perforation can be considered a rather fruitful approximation computing optimization in many program domains. A simple optimization like loop perforation can, as demonstrated by our experiments, potentially give desirable speedups at reasonable cost to program accuracy. Our program for loop selection for perforation presents a very accessible way to analyze programs for potential perforation points.

There are some directions one can take from here. One can consider perforating multiple loops in the program at further cost to the accuracy of the output. We considered an algorithmic solution to this problem. The problem can be formulated as: Given the set of valid loops and strides, can you find a subset that maximizes speedup while still meeting the TOQ bounds? Sidiroglou et al proposed a greedy algorithm for selecting multiple loops, but it stands to reason that such a greedy algorithm may not produce the most optimal result with respect to maximizing speedup.

Another potential extension is the implementation of auto-memoization of loop iterations. Rather than skipping iterations like perforation, one can simply try to use results of previous iterations to construct an approximation of the current iteration. The trickiness of this optimization is closely related to alias analysis and how induction variables are being used in the bodies of loops.

7 Distribution of Total Credit

Joseph implemented the optimizations while Naman did the experimentation and benchmarking. As such we think the credit should be divided equally among us, 50-50.

References

[SM13] Samadi, Mehrzad, et al. "SAGE: self-tuning approximation for graphic engines". Proceeding of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2013.

[SDS13] 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.

[C11] Christian Bienia, Benchmarking Modern Multiprocessors Ph.D. Thesis. Princeton University, January 2011.

[ARSMH] Agarwal, Rinard, Sidiroglou, Misailovic, Hoffman. "Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures". http://dspace.mit.edu/handle/1721.1/46709