Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation

Shikai Li
University of Michigan, Ann Arbor
shikaili@umich.edu

Sunghyun Park
University of Michigan, Ann Arbor
sunggg@umich.edu

Scott Mahlke
University of Michigan, Ann Arbor
mahlke@umich.edu

ABSTRACT

Loop perforation is one of the most well known software techniques in approximate computing. It transforms loops to periodically skip subsets of their iterations. It is general, simple, and effective. However, during analysis, it only considers the number of instructions to skip, but does not consider the differences between instructions and loop iterations. Based on our observation, these differences have considerable influence on performance and accuracy. To improve traditional perforation, we introduce *selective dynamic loop perforation*, a general approximation technique that automatically transforms loops to skip selected instructions in selected iterations. It provides the flexibility to craft approximation strategies at the dynamic instruction level. The main challenges in selective dynamic loop perforation are how to capture the characteristics of instructions, optimize perforation strategies based on these characteristics, and minimize additional runtime overhead. In this paper, we propose several compiler optimizations to resolve these challenges, including optimized instruction-level, load based and store based selective perforation, and self-directed dynamic perforation with a dynamic start and dynamic perforation rates. Across a range of 8 applications from various domains, selective dynamic loop perforation achieves average speedups of 2.89x and 4.07x with 5% and 10% error budgets, while traditional loop perforation achieves 1.47x and 1.93x, respectively, for the same error budgets.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; Runtime environments; • **General and reference** → *Performance*;

KEYWORDS

Approximate Computing, Compiler, Runtime System

ACM Reference Format:
Shikai Li, Sunghyun Park, and Scott Mahlke. 2018. Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation. In *ICS '18: 2018 International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3205289.3205317

ICS '18, June 12–15, 2018, Beijing, China

ACM ISBN 978-1-4503-5783-8/18/06…$15.00
https://doi.org/10.1145/3205289.3205317

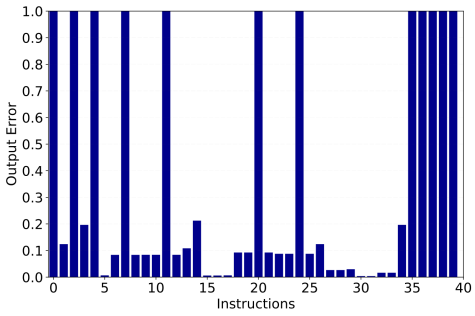
1 INTRODUCTION

Emerging compute-intensive applications in popular domains, such as machine learning, computer vision, and data mining, create a rapid increase in computation demands. However, with the diminishing benefits of CMOS scaling [11], it gets harder and harder for the computation capacity to keep up with the demands. Approximate computing provides a practical solution to this problem through trading off output accuracy for performance improvement or energy reduction. Prior works have shown that, through software [5, 31, 33, 35] and hardware [12, 13, 20, 26] techniques, approximate computing could achieve significant performance speedup with moderate accuracy lost, which is acceptable or even not noticeable to end users in these domains.

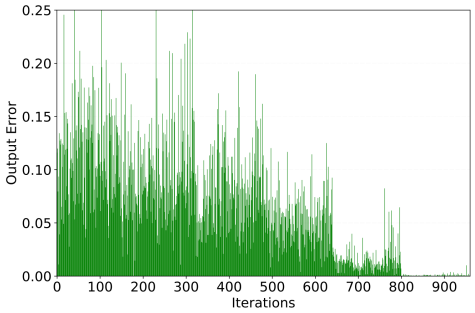
Task skipping [28] is a crucial idea in approximate computing, in which a part of computational work is skipped to reduce the corresponding execution time or energy consumption while compromising output quality. Loop perforation [35] is one of the most widely applied task skipping techniques. It transforms loops to periodically skip subsets of their iterations. It takes a parameter named loop perforation rate, representing the percentage of iterations to skip, and only executes the first iteration among every n iterations. Loop perforation has shown to be a simple, general, and effective task skipping technique. However, it still has limitations in discovering potential approximation opportunities.

Loop perforation provides a coarse-grained and inflexible solution of *where* and *when* to conduct skipping. It is limited to skip iterations *entirely* with all internal instructions and *periodically* with a constant rate during runtime. In practice, different instructions and iterations present varying characteristics at runtime. Ignoring these differences and blindly skipping iterations entirely and periodically can lose considerable approximation opportunities.

We conduct an in-depth investigation of applications in PARSEC [6] and Rodina [10]. Based on the investigation, we find out that skipping different instructions and different iterations of the same loop have different impacts on performance and accuracy. Figure 1a and Figure 1b present two examples of this phenomena. Figure 1a shows final output errors when different instructions of a selected loop in *Hotspot* from Rodinia [10] benchmark suite are skipped every other iteration. As shown in the figure, being skipped at the same rate, some instructions hurt output quality significantly, while other instructions hurt output quality negligibly. Figure 1b shows final output errors when different iterations of a selected loop in *Bodytrack* from the PARSEC [6] benchmark suite are skipped during program execution. One can see that the output errors incurred by skipping different iterations vary substantially during the execution. In loop perforation, these differences are ignored. Instructions are skipped together, iterations are skipped periodically, regardless of output errors. Therefore, a better solution is desired. We need to



(a) Skipping Different Instructions in *Hotspot*



(b) Skipping Different Iterations in *Bodytrack*

Figure 1: Output Error of Skipping Different Content

skip work intelligently, skipping instructions and iterations that are unimportant for output accuracy, while executing instruction and iterations that are crucial for output accuracy.

To solve these problems, we propose *selective dynamic loop perforation*, which gives a fine-grained, highly-flexible, and intelligent solution of *where* and *when* to conduct skipping and provides the possibility to optimize skipping strategies at the dynamic instruction level. Instead of skipping iterations entirely and periodically, selective dynamic loop perforation intelligently selects instructions as *where* to skip and dynamically selects iterations as *when* to skip, based on program behaviors derived through a combination of offline analysis and online observation.

This paper makes the following contributions:

- We propose selective dynamic loop perforation which transforms programs to skip intelligently selected instructions at dynamically selected iterations, aiming to achieve best performance and accuracy trade-off.
- Selective loop perforation is introduced to intelligently select a subset of instructions to skip. Instruction-level, load based, and store based selective loop perforation are proposed to balance selection flexibility and optimization complexity.
- Dynamic loop perforation is introduced to dynamically skip iterations with dynamic start and perforation rate. Call-based and iteration-based dynamic rates are proposed to change the degree of approximation aggressiveness at runtime.
- We compare selective dynamic loop perforation to traditional loop perforation across 8 applications from PARSEC [6] and Rodina [10]. Selective dynamic loop perforation achieves 2.89x and 4.07x speedup with 5% and 10% error budgets. In comparison, traditional loop perforation achieves 1.47x and 1.93x speedup, respectively, for the same error budgets.

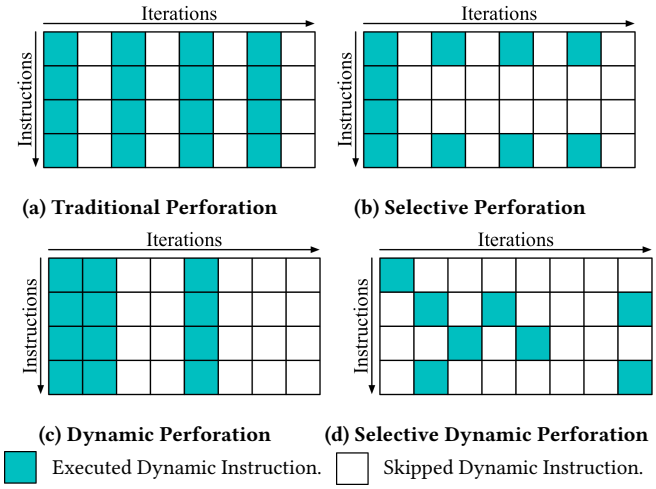


Figure 2: Illustration of Different Perforation Techniques

2 OVERVIEW

2.1 Approach

Loop perforation (referred as perforation) transforms loops to execute a constant subset of their iterations. At the instruction level, it performs regular skipping of all dynamic instructions in periodically selected iterations.

However, as discussed in Section 1, we discover that considerable skipping opportunities are hidden from approximation due to the coarse granularity. As a result, we expect a fine-grained dynamic instruction skipping technique with low runtime overhead to expose those opportunities. Here, we propose selective dynamic perforation, a new technique to irregularly skip specific instructions at specific iterations. It aims to optimize skipping strategies for all dynamic instructions to achieve best performance accuracy trade-off.

Figure 2 shows a simple illustration. It shows different skipping strategies of different perforation techniques at runtime. A grid represents all iterations in original execution. Each column represents a single iteration. Each cell represents a dynamic instruction. Shaded cells represent executed dynamic instructions and blank cells represent skipped dynamic instructions.

In **traditional perforation**, loops are transformed to skip a constant subset of their iterations. We are limited to skip iterations entirely with all dynamic instructions and periodically with a constant skip rate. As shown in Figure 2a, with these limitations, we could only skip 50% dynamic instructions under the error budget.

In **selective perforation**, loops are transformed to skip a flexible subset of their instructions. Compared to traditional perforation, we don't have to skip iterations entirely with all instructions. We can skip a subset of instructions that is unimportant to output accuracy and keep executing instructions that are important to output accuracy. As shown in Figure 2b, with this relaxation, we could skip 68.75% dynamic instructions under the error budget.

In **dynamic perforation**, loops are transformed to skip a flexible subset of their iterations. Compared to traditional perforation,

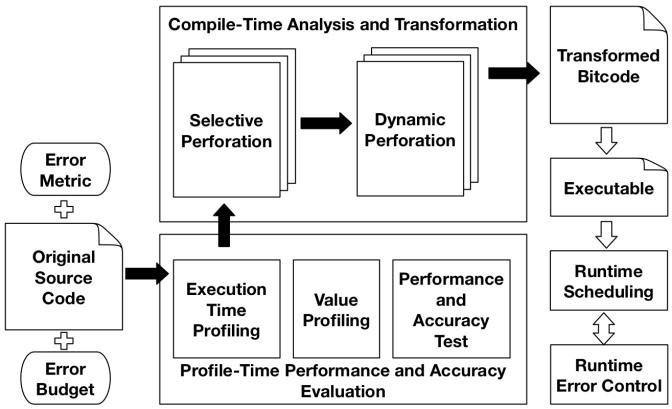


Figure 3: Sculptor Overview

we don't have to skip iterations periodically with a constant skip rate. We can skip a subset of iterations that is unimportant to output accuracy, while keeping iterations that are important to output accuracy. As shown in Figure 2c, with this relaxation, we could skip 62.5% dynamic instructions under error budget.

In **selective dynamic perforation**, loops are transformed to skip a dynamic subset of their instructions in a dynamic subset of their iterations. It relaxes all the constraints presented in traditional perforation and achieves high flexibility. Theoretically, we could choose to skip every single dynamic instruction in the reverse order of their importance to output accuracy. As shown in Figure 2d, with the strong flexibility, we could skip 75% dynamic instructions under the error budget.

In this paper, we introduce methodologies to design selective perforation and dynamic perforation strategies, and optimizations to reduce both online and offline overhead.

2.2 Compilation and Profiling System

Figure 3 presents a high-level overview of the compilation and profiling system of Sculptor. It consists of two parts. One conducts analysis and transformation at compile-time. Another conducts performance and accuracy evaluation at profile-time. During perforation strategy design and integration, the analysis, transformation and evaluation process are interleaved to compare different optional strategies.

In this system, besides application source code, programmers only need to provide an end-to-end error metric, an error budget, and test inputs for performance and accuracy evaluation. Afterwards, the system is able to conduct analysis, transformation and evaluation automatically, and generate the approximated executable with built-in dynamic perforation schedulers. The dynamic perforation scheduling will be performed by built-in schedulers at runtime. Compared to other approximation techniques, it is widely applicable and requires no hardware modification.

In Figure 3, the black arrows show analysis and transformation process as a primary process. There are five steps in the primary process. Firstly, the system performs execution time profiling to find target loops to perforate. Loops consuming a high ratio of total execution time are selected as target loops. Secondly, the system compiles the source code into compiler IR and designs selective perforation strategies (as the analysis process introduced in Section 3) for each target loop. Thirdly, the system designs dynamic perforation strategies (as the analysis process introduced in Section 4) for each target loop. Fourthly, the system searches for a near optimal approximation strategy for the entire program through integrating different selective and dynamic perforation strategies. Finally, the system transforms the program and inserts built-in dynamic perforation schedulers (as the transformation processes introduced in Section 3 and Section 4).

Along with the primary process, performance and accuracy evaluation process is repeatedly conducted as the secondary process. In general, this process helps the primary process to compare different optional perforation strategies and ultimately find a near optimal approximation perforation strategy for the program.

After the compilation and profiling process, an error management mechanism is designed to further adjust performance and accuracy trade-off at runtime (introduced in Section 5).

3 SELECTIVE PERFORATION

In selective perforation, loops are transformed to skip a subset of their iterations during program execution.

In this paper, three selective perforation methods, *instruction level selective perforation*, *load based selective perforation* and *store based selective perforation*, are introduced for different balances of selection flexibility and optimization complexity.

In the analysis process, through data flow analysis and profiling, the system can identify instructions to skip (discussed in Section 3.1 and Section 3.2) or instructions to execute (discussed in Section 3.3).

In the transformation process, through customized compiler optimizations, the system can transform the loop to skip instructions with a small runtime overhead (discussed in Section 3.1.3 and Section 3.3).

For error management, Sculptor can execute selectively perforated loops at varying perforation rates or execute unperforated loops to adjust approximation aggressiveness at runtime.

3.1 Instruction Level Selective Perforation

In instruction level selective perforation, the system gets a subset of unimportant instructions through individual performance and accuracy evaluation (discussed in Section 3.1.1) and expands this subset through data flow analysis (discussed in Section 3.1.2). Instruction level selective perforation is performed in three stages, selection stage, expansion stage and transformation stage.

3.1.1 Selection Stage. In the selection stage, the system selects instructions that have a large impact on performance and a small impact on accuracy as target instructions to be perforated.

Firstly, the system selects instructions based on their estimated impact on performance. Inside the target loop, if there are function calls which directly live in the loop and consume most execution time of the loop, only these function calls are selected as potential perforation targets. Otherwise, all instructions are selected as potential perforation targets. The execution time of the target loops and inside function calls are derived through profiling.

Secondly, the system filters out part of selected instructions that might cause program corruption or catastrophic errors based on static analysis. Among these selected instructions, all branch

```
1   load r2, r1
2 for.body:
3   load r4, r3
4   load r5, r4
5   mul r6, r2, r5
6   add r7, r6, r5
```

Code 1: Code Segment Example for Expansion Stage

instructions and address calculation instructions, together with their producers, are filtered out.

Thirdly, the system again filters out part of selected instructions that might cause the output error to exceed the error budget based on profiling. It uses *local error estimation* to filter out ALU operations, load instructions and pure function calls. Besides, it uses *global error estimation* to filter out store instructions and other function calls.

For local error estimation, it detects temporal data similarity among consecutive executions of instruction and uses it to estimate the accuracy lost when the instruction is skipped. Temporal data similarity describes a phenomenon in which an instruction tends to have similar results among consecutive executions. A value profiling methodology is used to detect it. During profiling, a sliding window (as FIFO) is maintained for each selected instruction. It records the result of last m executions of the instruction. Each time the instruction is executed, the sliding window will be updated. At sampling, the relative deviation from it will be calculated and accumulated. After profiling, the average relative deviation of the sliding windows will present as a measurement of the temporal data similarity of the instruction. Instructions with low temporal data similarity are filtered out. As their results fluctuate greatly during program execution and might cause large output error when they are skipped. The algorithm could be described using Equation 1. n represents the number of sampled sliding windows, m represents the size of the sliding window, $x_{i,j}$ represents results of instruction executions.

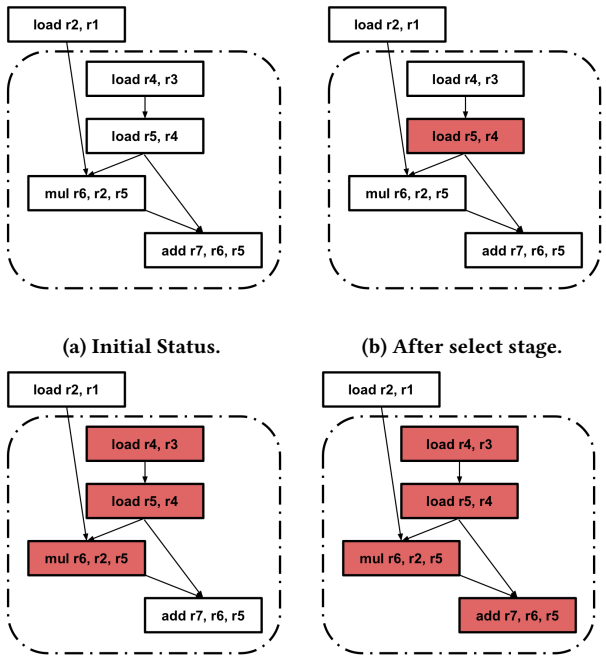
$$E[\frac{\sigma^2}{\mu^2}] = \frac{1}{n}\sum_{i=1}^{n}\frac{(\frac{1}{m}\sum_{j=1}^{m}x_{i,j}^2 - (\frac{1}{m}\sum_{i=1}^{m}x_{i,j})^2)}{(\frac{1}{m}\sum_{i=1}^{m}x_{i,j})^2} \quad (1)$$

For global error based estimation, it perforates a single instruction with basic perforation rate, runs the transformed program, and calculates the end-to-end output error for each selected instruction. Instructions that cause the output error to exceed the error budget are filtered out.

3.1.2 Expansion Stage. After the selection stage, the system gets a set of target instructions to be perforated. In the expansion stage, it iteratively expands this set through data flow analysis, adding more instructions without additional accuracy lost.

There are two rules in the update process: 1) All instructions which only use results of target instructions or loop invariant values can be selected as target instructions. 2) All instructions whose results are only used by other target instructions can be selected as target instructions as well.

Take Code 1 and Figure 4 as an example. In Figure 4, the dashed box marks the loop and the shaded rectangles mark the selected instructions in each step. Assume only the load instruction at line 4 (referred as load.4) is selected in the selection stage. In the first iterative update, as the multiply instruction only uses the result of



(a) Initial Status. **(b) After select stage.**

(c) Round 1. Apply rule 1 and 2. **(d) Round 2. Apply rule 2.**

Figure 4: Illustration of Expansion Stage

load.4 and loop invariant value, it is selected based on rule 1; As the load instruction at line 3 (referred as load.3) generates the result only used by load.4, it is selected based on rule 2. In the second iterative update, as the add instruction only uses the result of load.4 and multiply, it is selected based on rule 2. As iterative updates afterward will not further expand the set of target instructions, the expansion stage terminates.

As shown in the example, the expansion stage is similar to *loop invariant code motion*. Actually, when an instruction is perforated, the result of the instruction will remain the same during skipped iterations. It creates *temporal invariant* values that don't change among consecutive iterations, which is similar to *loop invariant* values, and brings the optimization opportunities. Rule 1 is justified because the instruction uses constant values as operands in the skipped iterations. Rule 2 is also justified because the instruction which only has target instructions as users do not have any impact on the program if these target instructions are skipped. Therefore, instructions which satisfy two suggested rules can be skipped as target instructions.

3.1.3 Transformation Stage. After the expansion stage, the system gets a set of target instructions to perforate. In the transformation stage, it conducts several compiler optimizations to perforate these target instructions with low runtime overhead. For simplicity, the system uses the same perforation rate for all the target instructions in the same loop.

In traditional perforation, the transformation process only needs to change the increment of the induction variable. However, in selective perforation, scattering target instructions inside loop body make it more complicated.

Intuitive Solution: An intuitive solution is splitting the basic block and inserting a branch instruction before each consecutive target instruction chain. However, it brings too much control divergence overhead and may cause performance downgrade at sometimes. Compiler optimizations could be performed to reduce this overhead, including instruction reordering, loop unswitching, loop unrolling, etc.

Unswitching Optimization: The compiler creates two versions of the loop body, one as original, another as perforated. It inserts one branch instruction to select one version of the loop body to execute instead of redundant branch instructions to skip different instructions at runtime.

Unrolling Optimization: However, for small loops, even one inserted branch instruction might incur non-negligible performance overhead. Based on unswitching optimization, the compiler further unrolls the perforated version of the loop body multiple times to reduce the overhead.

3.2 Load Based Selective Perforation

Load based selective perforation is a variation of instruction level selective perforation. The only difference lies in the selection stage. It only considers load instructions as target instructions. The main purpose of this variation is to reduce the profiling overhead in the selection stage. Because of the iterative updates in the expansion stage, chains of instructions will be selected and perforated in the end. Similar performance and accuracy trade-off will be achieved compared to instruction level selective perforation.

3.3 Store Based Selective Perforation

Algorithm 1 Store based selective perforation algorithm

```
1:  kernel_insns ← ∅
2:  for all (store in loop) do
3:    if ∀get_alias_loads(store) ∉ loop then
4:      kernel_insns.insert(store)
5:      kernel_insns.insert(get_addr_calc_insns(store))
6:      if relative_error_estimate(store) > δ then
7:        kernel_insns.insert(get_data_calc_insns(store))
8:      end if
9:    end if
10: end for
11: perforated_loop ← loop
12: for all (regions in perforated_loop) do
13:   if ∀kernel_insns ∉ region then
14:     delete region
15:   else
16:     delete region \ (kernel_insns ∪ branches)
17:   end if
18: end for
```

In store based selective perforation, the system gets a subset of important store instructions through accuracy evaluation and expands this subset through data flow analysis. Algorithm 1 shows the procedures of store based selective perforation.

Firstly, the system identifies store instructions that might alias with load instructions outside the loop or cross loop iterations through alias analysis. If all these output stores directly live in the loop, it continues to next step. Otherwise, it terminates.

Secondly, all output stores and its address calculations are identified as instructions necessary to be kept. Besides, the system performs global error estimation on output stores as described in Section 3.1.1. For output stores that cause output error to exceed error budget, their data calculations are also identified as instructions necessary to be kept.

Thirdly, based on the unswitching optimization described in Section 3.1.3, it creates an original and perforated version of the loop body. In the perforated version, for all regions without any necessary instructions, it deletes all basic blocks inside the region and connects two branches towards and outwards the region.

Store based selective perforation captures the results of iterations and directly approximates them if applicable. It avoids the complexity of analyzing every instruction and the runtime overhead to go through the unnecessary control flow divergences. At certain cases, even entire loop entries can be perforated.

4 DYNAMIC PERFORATION

In dynamic loop perforation, loops are transformed to skip a flexible subset of their iterations under the scheduling of built-in schedulers during program execution.

As discussed in Section 1, different iterations have different impacts on output accuracy. However, it is almost impossible to precisely predict each iteration's accuracy impact at runtime. Fortunately, we discover that iterations' accuracy impacts can be clustered according to the program execution circumstances that they are executed in. The accuracy impact fluctuates negligibly inside the same circumstance but significantly across different circumstances. This inspires the self-directed dynamic perforation technique to use different perforation strategy under different circumstances.

In the analysis process, the system identifies optimal skipping strategies of the loop under different circumstances through the control flow analysis and profiling. In the transformation process, the system transforms the loop and inserts a built-in scheduler. At runtime, the built-in scheduler observes the current circumstance and applies the corresponding strategy designed offline.

4.1 Dynamic Rate

In dynamic rate, a built-in scheduler is used to change perforation rate according to different circumstances at runtime.

In traditional perforation, the perforation rate is constant. However, as loop behavior changes during program execution, skipping the same amount of iterations at different circumstances have different accuracy impacts. It is understandable that using more adapted degrees of approximation aggressiveness during program execution will deliver better performance and accuracy trade-off.

Here, we introduce two methods, *active function call based dynamic rate* and *active loop iteration based dynamic rate*, to capture regular loop behavior variations and change perforation rate dynamically at different circumstances.

For error management, Sculptor can scale the pre-tuned dynamic perforation rates (referred as the basic perforation rates) to adjust approximation aggressiveness at runtime.

```
1  int kernel(DataType data){
2    iterative_updates(data.primary);
3    iterative_updates(data.secondary);
4    return combine(data);
5  }
6  void iterative_updates(int* k){
7    for(int itr=0; itr<100; itr++)
8      single_update(k);
9  }
10 void single_update(int* k){
11   for(int idx=0; idx<100; idx++)
12     k[idx] = compute(k[idx]);
13 }
```

Code 2: Code Example for Active Function Call Based Dynamic Rate

4.1.1 Active Function Call Based Dynamic Rate. **Idea:** In this method, the scheduler chooses a perforation rate based on the observed active function calls at each loop entry. An active function call is defined as a function call that has been called but has not been returned at the moment. It is designed based on the phenomena that executions of a loop tend to have different accuracy impacts during different function calls.

Example: In Code 2, it is clear that, at each program execution, *compute* function will be executed 20000 times. However, the first 10000 executions and the second 10000 executions of *compute* are operated on different parts of data. Through profiling, if *data.secondary* is observed to be highly tolerant of errors but *data.primary* is observed to be less tolerant to errors, it is better to approximate the first 10000 executions of *compute* more aggressively but approximate the second 10000 executions of *compute* less aggressively at runtime. This motivates the idea of active function call based dynamic rate.

Algorithm 2 Active function call based dynamic rate algorithm

```
1:  procedure Analysis
2:    signature_funcs ← ∅
3:    for all (func_call in program) do
4:      if (prof_time(func_call) ≥ total_time × θ) and \
5:       (is_reachable(func_call, loop)) then
6:        signature_funcs.insert(func_call)
7:      end if
8:    end for
9:  end procedure
10: procedure Profiling(test_rates)
11:   for all loop_entries do
12:     dyn_rate ← test_rates[active_func]
13:   end for
14:   estimate_error[test_rates] ← calc_relative_error()
15: end procedure
16: procedure Runtime(base_rates)
17:   for all loop_entries do
18:     dyn_rate ← base_rates[active_func] × scale
19:   end for
20:   for all check_points do
21:     sample_error ← cal_relative_error()
22:     scale ← get_scale_ratio(sample_error)
23:   end for
24: end procedure
```

Implementation: As shown in Algorithm 2, active function call based dynamic rate is implemented in three steps.

Firstly, through call graph analysis, the system identifies possible active function calls at all loop entries. The system transforms the loop and instruments instructions. Some instrumentations are used to store active function call information. Other instrumentations are used to read this information and load the corresponding pretuned perforation rate. They work together as a lightweight built-in scheduler.

Secondly, through profiling, it samples and perforates loop entries and calculates output errors at different observed active function calls. If the output errors are uniform, the perforation rate will be assigned as 1 when the scheduler observes the same active function call as the case that produces highest output error. If the output errors are diverse, the perforation rate will be tuned using the steepest ascent hill climbing algorithm [36], changing the perforation rates until no significant improvements can be achieved.

Thirdly, at runtime, the scheduler will change the basic perforation rates based on the observed active function calls. Furthermore, it will also scale the rates with the help of error management mechanisms.

4.1.2 Active Loop Iteration Based Dynamic Rate. **Idea:** In this method, the scheduler chooses perforation rates based on the observed active loop iterations at runtime. An active loop iteration is defined as a loop iteration that is being executed or waiting for return values at the moment. It is designed based on the phenomena that executions of a loop tend to have different accuracy impacts at different iterations of its outer loops.

To further explain the method, we give following definitions. At runtime, if loop A finishes an entry during the time loop B finishes an iteration, we call loop A(B) as a *super-inner(outer) loop* of loop B(A). With these definitions, we can further explain that executions of a loop tend to have different accuracy impacts at different iterations of its super-outer loops, not limited to its outer loops in the same function.

Example: In Code 2, for the first 10000 executions of *compute*, the 1st-100th executions is performed during the first iteration of the loop inside *iterative_updates*, the 101st-200th executions is performed during the second iteration of the loop inside *iterative_updates*, etc. Through profiling, if we observe that the accuracy loss of skipping *iterative_updates* is monotonously diminishing, it might better to approximate *compute* executions during latter iterations more aggressively but approximate *compute* executions during former iterations less aggressively. This motivates the idea of *active loop iteration based dynamic rate*.

Implementation: In the analysis process, through call graph analysis and control flow analysis, the system identifies super-outer loops of the target loop. Through profiling, the system identifies one super-outer loop such that the target loop has most significant accuracy impact fluctuations within its entries. Furthermore, super-outer loop iterations of a single entry are partitioned into multiple successive phases. The system tunes the perforation rates similar to Section 4.1.1. These phases can be further partitioned with retuned rates until no significant improvement can be achieved. The transformation process and the runtime scheduling is also similar to Section 4.1.1 and will not be discussed in detail.

```
1  void kmeans(float** data, float** center){
2    while(delta != 0){
3      assign_cluster(data, center);
4      delta = update_center(center);}
5  }
6  void assign_cluster(float** data, float** center){
7    for(int i=0; i<1000000; i++)
8      assign_cluster_single(data[i], center);
9  }
10 int update_center(float** center){
11   for(int i=0; i<100; i++)
12     delta += update_center_single(center[i]);
13   return delta;
14 }
```

Code 3: Kmeans Algorithm

4.2 Dynamic Start

In the dynamic start, the scheduler changes iteration start points at different loop entries during runtime. It is designed to reduce output errors by providing better execution fairness and coverage.

This method is inspired by an interesting phenomenon that the non-uniform distribution of executed iterations might incur much higher output errors compared to the uniform distribution. One common scenario is array access. Most loops access arrays through reading/writing. In traditional perforation, only a fixed subset of elements will be read from or written to. As a result, considerable accuracy might be lost by not accessing other elements in arrays.

Take the kmeans algorithm in Code 3 as an example. If *assign_cluster* is perforated with a fixed start and a fixed rate, it will create two problems. Firstly, only a fixed subset of data points will be read. It influences the accuracy of cluster centers as it is calculated and updated based on a small subset of data. It shows perforation needs *fairness*, as all iterations should have the same probability to be executed, especially for iterations that read arrays. Secondly, only a fixed subset of data points will be assigned. It incurs unacceptable output with unassigned data points. It shows perforation needs *coverage*, as all iterations should be at least executed once, especially for iterations that write arrays.

To address above problem, active loop iteration based dynamic start is implemented. At runtime, the dynamic perforation scheduler will change iteration start points at different loop entries according to the iteration count of a super-outer loop. It manages to schedule different subsets of iteration to be executed in a round-robin fashion, providing better execution fairness and coverage.

5 RUN-TIME ERROR MANAGEMENT

Sculptor uses a calibration-based aggressiveness adjustment mechanism to perform error management at runtime. Figure 5 shows an example execution segment from one of the benchmarks. The dashed red line stands for real-time error, the solid blue line stands for real-time performance, and the approximation and calibration phases are divided by dashed black lines. It is similar to the error control mechanisms in previous works [4, 5, 30, 31]. The basic algorithm is presented in Algorithm 3.

Sculptor performs both accurate and approximate execution during calibration phases, and calculates relative errors at regular intervals during runtime. At offline, Sculptor performs code transformation for selective and dynamic perforation and sets initial perforation rates through profiling. Selective and dynamic perforation are traded as black boxes for the runtime error management. No

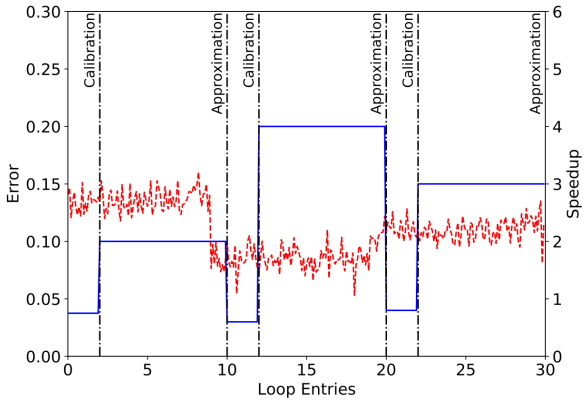


Figure 5: Runtime Error Management Example

Algorithm 3 Error management algorithm

while (*program_execution*) **do**
  if $entry\_cnt++ \neq check\_pt$ **then**
    $approximate\_execution(skip\_rate)$
  else
    $res_{accu} \leftarrow accurate\_execution()$
    $res_{approx} \leftarrow approximate\_execution()$
    $rel\_err \leftarrow rel\_err\_calc(res_{accu}, res_{approx})$
    if $rel\_err \leq \delta_{lower}$ **then**
      $scale \leftarrow max(max\_scale, scale \times \beta_{up})$
    else if $rel\_err \geq \delta_{upper}$ **then**
      $scale \leftarrow scale \times \beta_{down}$
    end if
    $skip\_rate \leftarrow floor(basic\_rate \times scale)$
    $check\_pt \leftarrow check\_pt + floor(basic\_stride \times scale)$
  end if
end while

expensive code transformation or behavior analysis is conducted online. At online, it only scales the initial perforation rates with the guidance of runtime error management. Based on the measured error and the target, the system adjusts the skip rate. The skip rate can range from 0, in which case the loop is not being perforated at all, to $basic\_rate \times max\_scale$, in which case the loop is being perforated at the maximum rate allowed. It achieves a good dynamic balance of accuracy and performance.

Sculptor can also work with other error management mechanisms [5, 19, 31]. It can set the basic perforation rates or generate different versions of approximated functions offline. With the help of the error management module, it can be more or less aggressive to approximate through scaling the basic perforation rates or choosing an appropriate version to execute online. It can replace traditional loop perforation in existing approximation solutions with a better performance and accuracy trade-off.

6 EVALUATION

6.1 Methodology

Software Tools and Hardware Configuration Sculptor's compilation phase is implemented in Clang 4.0 version. Its analysis

Application	Domain	Train Size	Test Size	Error Metric
Canneal	Engineering	400K	2.5M	ARE
Streamcluster	Data Mining	100K	500K	NMI Score [8]
Blackscholes	Financial	64K	10M	ARE
Bodytrack	Computer Vision	100 Frames	261 Frames	ARE
Swaptions	Financial	6M	13M	ARE
X264	Media	128 Frames	512 Frames	SSIM Index
Hotspot	Physics	512x512	1024x1024	ARE
KMeans	Data Mining	100K	500K	NMI Score

Note: ARE: Average Relative Error

Table 1: Evaluation Benchmarks

and transformation phases are implemented in LLVM 4.0 version. It uses an execution time profiling tool and a value profiling tool developed in LLVM 4.0 version. Evaluation results are collected through running serial code on a workstation with 3.40GHz Intel Core i7-6700 CPU under Ubuntu 16.04 version.

Applications and Error Metric We evaluate our methodology using 6 applications from PARSEC benchmark suite [6] and 2 additional applications from Rodina benchmark suite [10] with default data sets, as shown in Table 1. These 8 benchmarks cover various important domains including data mining, financial analysis, media processing, computer vision, physics simulation, etc.

In this paper, most error metrics are based on average relative error, which is defined as:

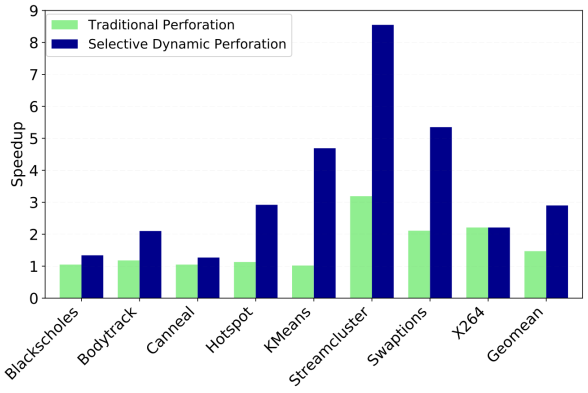
$$Average\ Relative\ Error = \frac{1}{N}[\sum_{i=1}^{N}\frac{|y_i^* - y_i|}{|y_i|}] \quad (2)$$

In Equation 2, N represents the number of output data, y_i represents accurate output data, y_i^* represents approximate output data. y_i and y_i^* could either be scalar or vector.

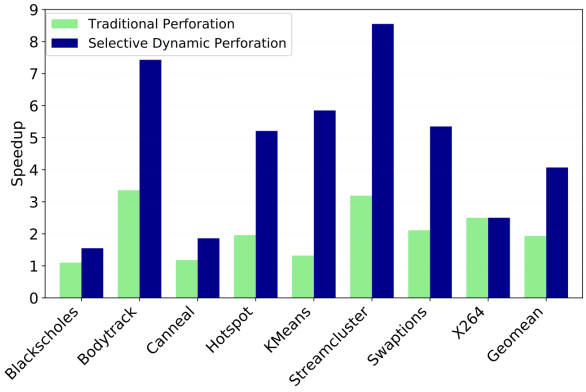
6.2 Performance Improvement

Overall Improvement Figure 6a and Figure 6b shows the results for selective dynamic loop perforation and traditional loop perforation [35] with 5% and 10% error budgets respectively. Speedups are compared to original programs. As shown in the figures, traditional loop perforation achieves 1.47x and 1.93x geometric mean speedup with 5% and 10% error budgets, while selective dynamic loop perforation achieves 2.89x and 4.07x geometric mean speedup respectively. Selective dynamic perforation presents a conspicuous improvement over traditional perforation through discovering latent approximation opportunities. *KMeans* shows the most significant improvement through discovering program dynamic characteristics. While *X264* shows the least significant improvement as the application is already crafted to trade off computation overhead with video quality loss and compression rate, and traditional loop perforation works well.

Take selective dynamic loop perforation with 10% error budget as an example. Table 2 shows numbers of loops transformed with



(a) 5% Error Budget



(b) 10% Error Budget

Figure 6: Selective Dynamic Perforation Performance Speedup with Different Error Budgets

Application	TLP	SLP	DLP	SDLP
Canneal	0	0	2(100%)	0
Streamcluster	0	0	3(100%)	0
Blackscholes	0	0	0	1(100%)
Bodytrack	2(33.3%)	0	3(50%)	1(16.7%)
Swaptions	2(28.6%)	3(42.8%)	2(28.6%)	0
X264	5(100%)	0	0	0
Hotspot	1(50.0%)	1(50.0%)	0	0
KMeans	0	0	1(100%)	0
Average	23.54%	10.31%	53.18%	12.97%

Note: TLP: Traditional Loop Perforation; SLP: Selective Loop Perforation; DLP: Dynamic Loop Perforation; SDLP: Combination of Selective and Dynamic Loop Perforation.

Table 2: Applied Perforation Techniques

different perforation techniques in different applications. As shown in the table, different perforation techniques have various contributions across applications. *Bodytrack* and *Swaptions* show a mixed contribution of different perforation techniques; *Hostspot* shows dominating contribution of selective loop perforation and traditional loop perforation; *Cannel*, *Streamcluster* and *KMeans* show dominating contribution of dynamic loop perforation; *Blackscholes*

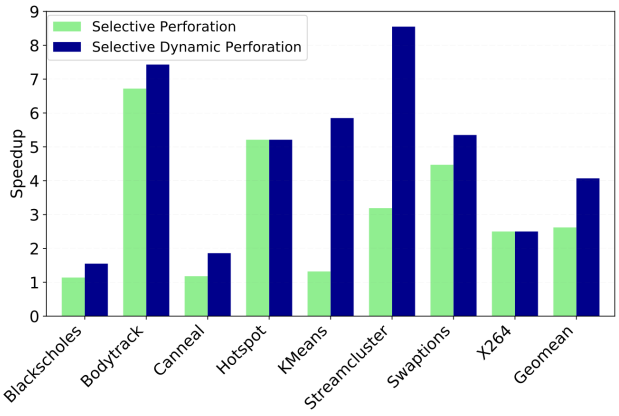


Figure 7: Selective Perforation Performance Speedup with 10% Error Budget

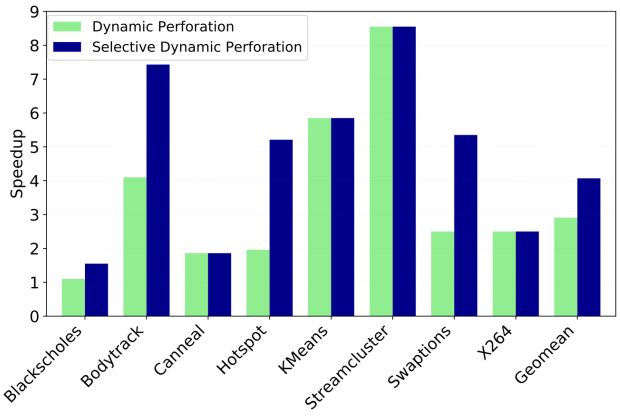


Figure 8: Dynamic Perforation Performance Speedup with 10% Error Budget

shows dominating contribution of selective dynamic loop perforation. In the following context, we further discuss how different perforation techniques contribute.

Selective Loop Perforation Improvement In Figure 7, dark bars represent performance speedup achieved by selective perforation only, light bars represent performance speedup achieved by selective dynamic perforation. As shown in the figure, *Bodytrack*, *Swaptions* and *Hostspot* benefit most from selective perforation.

In *Bodytrack*, the conditional loop inside *CalcWeights* is used to calculate particle weights and find the highest likelihood particle. Traditional perforation cannot perforate the conditional loop. However, selective loop perforation selectively skips the instructions for weight calculation but keeps the remaining instructions which update the data structure. This approximation leads to updating the data structure with previous calculation results. It delivers acceptable approximate weights based on the similarity of consecutive particles.

In *Swaptions*, the loops inside *HJM_SimPath_Forward_Blocking* update path of forward interest rate and the loops inside *Discount_Factor_Blocking* update discount factors to compute price. Part of these computation results are highly tolerant to approximation. Traditional perforation perforates these loops without updating data in skipped iterations. However, selective loop perforation selectively skips corresponding calculation instructions but keeps the data structure update instructions. Similar to *Bodytrack*, this approximation leads to updating the data structure with previous calculation results. which turns out to be better than not updating at all.

In *Hotspot*, the second loop inside *single_iteration* updates temperatures based on previous phase temperature. The calculation loads previous phase temperature and adds it to current phase temperature variation. Traditional perforation perforates the loop without storing temperatures in skipped iterations. However, selective loop perforation selectively skips temperature variation calculation instructions and keeps previous phase temperature load instructions as well as final result store instruction. This approximation leads to calculating temperature based on temperature variation of nearby locations.

Dynamic Loop Performance Improvement In Figure 8, dark bars represent performance speedup achieved by dynamic perforation only, light bars represent performance speedup achieved by selective dynamic perforation. As shown in the figure, different benchmarks benefit differently from dynamic perforation.

In *Canneal*, loops inside *reload* traverse the state vector machine in Mersenne Twister random number generator, which directs the pseudo-random element swap test. In traditional perforation, perforated code constantly skips updates of fixed part of state vector machine, leaving those part of state vector machine unchanged during program execution, which finally improves cache locality. With dynamic starts and dynamic rates, it has better coverage of updates of state vector machine, balancing cache locality benefits and element swap benefits.

In *Streamcluster*, the conditional loop inside *pkmedian* iteratively improves local clustering result. Loops inside *pFL* and *pgain* improve local clustering result based on feasible cluster centers and corresponding data points respectively. With dynamic starts, it has better coverage of all feasible clusters and all data points. Moreover, all above loops are executed during *localSearch* activation time. *localSearch* is called at the two different sites. One site performs clustering of local centers, and the other site performs clustering of all local centers to get global centers. With dynamic rates, all perforations are turned off at the second call site, which is crucial to final clustering accuracy but consumes a small portion of execution time.

In *Bodytracks*, loops inside *ImageErrorInside*, *ImageErrorEdge*, *InsidedError* compute information to locate particles of body pose based on image data sampling. With dynamic starts, it has better fairness of image data sampling. With dynamic rates, it tunes down perforation rate referred to the iteration count of the loop in *Update*, which actually indicates the annealing layer of the execution, as the approximation in later layers has a higher influence on final output quality.

In *KMeans*, as introduced in Section 4.2, traditional loop perforation will miss a fixed subset of centers or data points or features during execution and result in large errors. With dynamic starts, the kernel loop in *kmeans_clustering* is perforated, which finds and assigns centers for all data points and updates new cluster centers.

This solves the fairness and coverage problem which are discussed previously. It updates centers during a single traversal of all data points and makes clustering much faster. Besides, with dynamic rates referred to the outer loop, it decreases perforation rate after all data points have been assigned, as ever since then the loop could terminate early when not all data points are well assigned.

7 RELATED WORK

Trading output accuracy for performance improvement or energy efficiency has become a well-known concept. Lots of ideas have been proposed to achieve it: skipping computations [1, 28, 35], relaxing constraints for efficient parallel execution [9, 23], rough value estimation for expensive instructions [21, 41, 42], etc. Also, there have been continuous researches to build cost-efficient resilient system by applying approximate computing. Provided that the application can tolerant a certain degree of error, excessive protection could be avoided in a delicate manner and cost-efficient detection/-correction system could be built [18, 29, 39]. Different techniques to implement these ideas have been proposed ranging from hardware level to software level: algorithms [1, 24, 28, 35] , runtime systems [5, 9, 16, 19, 37] , programming languages [3, 7, 22, 32, 33] , middleware [2, 14] , compiler [7, 9, 14, 18, 30, 31] , and hardware [12, 20, 21, 26, 27, 27, 38, 40, 41].

Hardware Approximation Techniques Neural network accelerators [13, 26, 38, 40] has been intensively investigated to approximate general-purpose computing. Approximate value prediction is proposed and discussed in [21, 41, 42]. Cache and memory system designs are optimized to explore data similarity and redundancy in [17, 20, 34].

Software Approximation Techniques Most software approximation techniques utilize compiler to analyze and transform programs [5, 9, 23, 30, 31, 33]. The idea of task skipping is presented in [28]. Inspired by this idea, loop perforation is introduced and discussed in [1, 24, 35]. The idea of dynamic approximation has been presented in previous works [15]. Input responsive approximation introduced in [19] changes approximate strategies using canary inputs at each execution. Its dynamism is input-to-input based. It is dynamic between different executions, but still static during the same execution. Phase-aware optimization introduced in [25] changes approximate strategies at different phases. However, it doesn't provide automatic methodologies to identifies different phases. Besides, the idea of instruction level approximation is presented in [39]. It analyzes the possibilities to approximate different instructions through error injection.

8 CONCLUSION

In this paper, we propose selective dynamic loop perforation to capture the differences between instructions and iterations, and design a highly optimized approximation strategy based on these differences. It is compatible with most prior approximation systems and can be used to replace traditional loop perforation to achieve better performance improvements under the same error budgets. In selective dynamic loop perforation, loops are transformed to skip a selected subset of instructions at a selected subset of iterations. It breaks the integrity and periodicity constraints in traditional loop perforation, providing the flexibility to explore underneath approximation opportunities in two orthogonal dimensions. The insight of this paper is to customize approximation strategies for different dynamic instructions. The specific approximation technique is not limited to perforation, other techniques could also be applied based on the methods proposed in this paper. Across evaluated applications, selective dynamic loop perforation achieves an average speedup of 2.89x and 4.07x with less than 5% and 10% accuracy loss.

ACKNOWLEDGEMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), under Award Number DE-SC0014134. Support was also provided by the National Science Foundation (NSF) grant XPS-1438996.

REFERENCES

[1] Anant Agarwal, Martin Rinard, Stelios Sidiroglou, Sasa Misailovic, and Henry Hoffmann. 2009. *Using code perforation to improve performance, reduce energy consumption, and respond to failures.* Technical Report. Technical report, MIT.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 29–42.

[3] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 85–96.

[4] Matthew Arnold, Michael Hind, and Barbara G Ryder. 2002. Online feedback-directed optimization of Java. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 111–129.

[5] Woongki Baek and Trishul M Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, Vol. 45. ACM, 198–209.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 72–81.

[7] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. 2015. Probability type inference for flexible approximate programming. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 470–487.

[8] Gerlof Bouma. 2009. Normalized (pointwise) mutual information in collocation extraction. In *Proceedings of the Biennial GSCL Conference*, Vol. 156.

[9] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. 2015. Helix-up: Relaxing program semantics to unleash parallelization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 235–245.

[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 44–54.

[11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 365–376.

[12] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Architecture support for disciplined approximate programming. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 301–312.

[13] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 449–460.

[14] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. 2015. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 383–397.

[15] Henry Hoffmann. 2015. JouleGuard: energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 198–214.

[16] Henry Hoffmann, Jonathan Eastep, Marco D Santambrogio, Jason E Miller, and Anant Agarwal. 2010. Application heartbeats for software performance and health. *ACM Sigplan Notices* 45, 5 (2010), 347–348.

[17] Animesh Jain, Parker Hill, Shih-Chieh Lin, Muneeb Khan, Md E Haque, Michael A Laurenzano, Scott Mahlke, Lingjia Tang, and Jason Mars. 2016. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–13.

[18] Daya Shanker Khudia and Scott Mahlke. 2014. Harnessing soft computations for low-budget fault tolerance. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 319–330.

[19] Michael A Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. 2016. Input responsiveness: using canary inputs to dynamically steer approximation. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 161–176.

[20] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. 2015. Doppelgänger: a cache for approximate computing. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 50–61.

[21] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load value approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 127–139.

[22] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. 2014. Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 309–328.

[23] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. 2013. Parallelizing sequential programs with statistical accuracy tests. *ACM Transactions on Embedded Computing Systems (TECS)* 12, 2s (2013), 88.

[24] Sasa Misailovic, Daniel M Roy, and Martin C Rinard. 2011. Probabilistically accurate program transformations. In *International Static Analysis Symposium*. Springer, 316–333.

[25] Subrata Mitra, Manish K Gupta, Sasa Misailovic, and Saurabh Bagchi. 2017. Phase-aware optimization in approximate computing. In *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*. IEEE, 185–196.

[26] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. 2015. SNNAP: Approximate computing on programmable socs via neural acceleration. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 603–614.

[27] Sriram Narayanan, John Sartori, Rakesh Kumar, and Douglas L Jones. 2010. Scalable stochastic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 335–338.

[28] Martin Rinard. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*. ACM, 324–334.

[29] Mohamed M Sabry, Georgios Karakonstantis, David Atienza, and Andreas Burg. 2012. Design of energy efficient and dependable health monitoring systems under unreliable nanometer technologies. In *Proceedings of the 7th International Conference on Body Area Networks*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 52–58.

[30] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based approximation for data parallel applications. In *ACM SIGARCH Computer Architecture News*, Vol. 42. ACM, 35–50.

[31] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 13–24.

[32] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. 2015. Accept: A programmer-guided compiler framework for practical approximate computing. *U. Washington, Tech. Rep. UW-CSE-15-01-01* (2015).

[33] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 164–174.

[34] Joshua San Miguel, Jorge Albericio, Natalie Enright Jerger, and Aamer Jaleel. 2016. The Bunker Cache for spatio-value approximation. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[35] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. 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*. ACM, 124–134.

[36] Steven S Skiena. 1998. *The algorithm design manual: Text*. Vol. 1. Springer Science & Business Media.

[37] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D Corner, and Emery D Berger. 2007. Eon: a language and runtime system for perpetual systems. In *Proceedings of the 5th international conference on Embedded networked sensor systems*. ACM, 161–174.

[38] Renée St Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. 2014. General-purpose code acceleration with limited-precision analog computation. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 505–516.

[39] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V Adve. 2016. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–14.

[40] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. 2015. Neural acceleration for gpu throughput processors. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 482–493.

[41] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C Mowry. 2016. RFVP: Rollback-free value prediction with safe-to-approximate loads. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 4 (2016), 62.

[42] Amir Yazdanbakhsh, Bradley Thwaites, Hadi Esmaeilzadeh, Gennady Pekhimenko, Onur Mutlu, and Todd C Mowry. 2016. Mitigating the memory bottleneck with approximate load value prediction. *IEEE Design & Test* 33, 1 (2016), 32–42.