MIT Open Access Articles

Managing performance vs. accuracy trade-offs with loop perforation

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation	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 (ESEC/FSE '11). ACM, New York, NY, USA, 124-134.
As Published	http://dx.doi.org/10.1145/2025113.2025133
Publisher	Association for Computing Machinery (ACM)
Version	Author's final manuscript
Accessed	Mon Mar 25 15:44:56 EDT 2019
Citable Link	http://hdl.handle.net/1721.1/72440
Terms of Use	Creative Commons Attribution-Noncommercial-Share Alike 3.0
Detailed Terms	http://creativecommons.org/licenses/by-nc-sa/3.0/

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

DSpace@MIT

Managing Performance vs. Accuracy Trade-offs With Loop Perforation

Stelios Sidiroglou Sasa Misailovic Henry Hoffmann
Martin Rinard
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{stelios,misailo,hank,rinard}@csail.mit.edu

ABSTRACT

Many modern computations (such as video and audio encoders, Monte Carlo simulations, and machine learning algorithms) are designed to trade off accuracy in return for increased performance. To date, such computations typically use ad-hoc, domain-specific techniques developed specifically for the computation at hand.

Loop perforation provides a general technique to trade accuracy for performance by transforming loops to execute a subset of their iterations. A criticality testing phase filters out *critical loops* (whose perforation produces unacceptable behavior) to identify *tunable loops* (whose perforation produces more efficient and still acceptably accurate computations). A perforation space exploration algorithm perforates combinations of tunable loops to find Pareto-optimal perforation policies. Our results indicate that, for a range of applications, this approach typically delivers performance increases of over a factor of two (and up to a factor of seven) while changing the result that the application produces by less than 10%.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Reliability, availability, and serviceability

General Terms

Performance, Reliability, Experimentation

Keywords

Profiling, Loop Perforation, Quality of Service

1. INTRODUCTION

Many computations are designed to produce approximate results. Lossy video encoders, for example, are designed to give up perfect fidelity in return for faster encoding and smaller encoded videos [38]. Machine learning algorithms are designed to produce probabilistic models that capture some, but not all, aspects of phenomena that are difficult (if not impossible) to model with complete accuracy [15]. Monte-Carlo computations use random simulation to deliver inherently approximate solutions to complex systems of equations that are, in many cases, computationally infeasible to solve exactly [18].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.
Copyright 2011 ACM 978-1-4503-0443-6/11/09 ...$10.00.

This paper presents and evaluates a technique, *loop perforation*, for generating new variants of computations that produce approximate results. These variants occupy different points in the underlying performance vs. accuracy tradeoff space. Our results show that perforating appropriately selected loops can produce significant performance gains (up to a factor of seven reduction in overall execution time) in return for small (less than 10%) accuracy losses. Our results also show that the generated variants occupy a broad region of points within the tradeoff space, giving users and systems significant flexibility in choosing a variant that best satisfies their needs in the current usage context.

1.1 Loop Perforation

Loop perforation transforms loops to execute a subset of their iterations. The goal is to reduce the amount of computational work (and therefore the amount of time and/or other resources such as power) that the computation requires to produce its result. Of course, perforation may (and in our experience, almost always does) cause the computation to produce a different result. But approximate computations can often tolerate such changes as long as they do not unacceptably reduce the accuracy. Our implemented system uses the following techniques to find effective perforations:

- **Criticality Testing:** It perforates each loop in turn, executing the perforated computation on representative inputs to filter out *critical* loops whose perforation causes the computation to produce unacceptable results, crash, increase its execution time, or execute with a memory error. Filtering out such critical loops enables the subsequent perforation space exploration algorithm to focus on the remaining *tunable* loops that respond well to loop perforation.
- **Perforation Space Exploration:** It explores the space of variants generated by perforating combinations of tunable loops together. Our implemented system supports both exhaustive and greedy search algorithms. The result of the exploration is a set of Pareto-optimal variants, each of which maximizes performance for a specific accuracy loss bound when run on representative inputs.

1.2 Experimental Results

We evaluate our implemented system on applications from the PARSEC benchmark suite [6]. Our results show that the perforation space exploration algorithm can find perforations that deliver significant performance increases for five of the seven applications. Specifically, our performance measurements show that the perforated applications can run as much as seven times faster than the original applications while producing outputs that differ by less than 10% from the corresponding outputs of the original applications. Our results also show that the perforation space explo-

ration algorithm finds effective perforations — our evaluation of the source code of the applications indicates that the final perforations are appropriate for a wide range of inputs (and not just the training and production inputs used in the experimental evaluation).

1.3 Scope

Applications that interact well with loop perforation have some flexibility to change the result that they produce (subject, of course, to accuracy requirements). They also contain some underlying source of redundancy that enables them to produce an acceptable result even after discarding parts of their computation. Applications that process sensory data such as video, audio, and images often have both the redundancy and the flexibility required for loop perforation to work well. Other classes of applications include Monte Carlo simulations, information retrieval and machine learning computations, and the wide variety of scientific and economics computations for which the important consideration is producing an output within an acceptable precision range. Our results show that loop perforation can often improve the performance of all of these kinds of applications while preserving acceptable accuracy.

We acknowledge that many computations (for example, compilers, databases, and operating systems) may have hard logical correctness requirements that leave them with little or none of this kind of flexibility. We therefore do not claim that loop perforation is a universally applicable technique. Our experimental results show, however, that when the computation is amenable to loop perforation (and many of our benchmark applications are), loop perforation can be surprisingly effective at improving the performance while maintaining acceptable accuracy.

1.4 Why Loop Perforation Works

We have identified local computational patterns that interact well with loop perforation [24, 32, 25]. Examples include the Sum pattern (which computes the sum of a set of numbers) and the Argmin pattern (which computes the index and value of a minimum array element). An analysis of these patterns delivers a probabilistic guarantee that (under appropriate assumptions) the perforated computation is highly likely to produce a result that is close to the result that the original computation would have produced [24, 25].

While many of the perforated loops in our benchmark applications are instances of these patterns, they are also embedded as components within a larger application. Because the probabilistic analysis does not address how the effect of the perforation propagates through the application as a whole (which may either amplify or dampen it), it provides, at best, only a partial explanation for why it is acceptable to perforate these loops.

In this paper we identify *global computational patterns* that interact well with loop perforation. Analyzing the interaction between these patterns and the local perforated computations, we can understand why loop perforation works well for the application as a whole (and not just for the local computations embedded within the application). We identify the following global patterns (see Section 7 for more details):

- **Search Space Enumeration:** The application iterates over a search space of items. The perforated computation skips some of the items, returning one of the items from the remaining part of the search space.
- **Search Metric:** The application uses a *search metric* to drive a search for the most desirable item from a set of items:
  - **Selection:** A *selection* metric quantifies the desirability of each item encountered during the search.
  - **Filtering:** A *filtering* metric determines if the search should remove the item from a set of active items.
  - **Termination:** A *termination* metric determines if the search should terminate, either because an acceptable item has been found or because the likelihood of finding a more desirable item appears to be small.

  Perforating a search metric computation produces a new, less accurate but more efficiently computable metric. The effect is that the perforated search may return a less desirable but still acceptable item.

  We note that some applications use the same metric for more than one purpose. In this case the metric is a combined selection, filtering, and/or termination metric.
- **Monte-Carlo Simulation:** The application performs a Monte-Carlo simulation. The perforated computation evaluates fewer samples to produce a (potentially) less accurate result more efficiently.
- **Iterative Improvement:** The application repeatedly improves an approximate result to obtain a more accurate result. The perforated computation performs fewer improvement steps to produce a (potentially) less accurate result more efficiently.
- **Data Structure Update:** The application traverses a data structure, updating elements of the data structure with computed values. The perforated computation skips some of the elements, leaving the previous values in the skipped elements.

Successful perforations exploit computations that are partially redundant at both the local (loop) and global (application) level. This redundancy often appears when computations process multiple items to obtain a single result. For example, a computation that searches a set to find the most desirable item is partially redundant when the set contains similar items — searching only a subset of the items is likely to produce an item that is close to the most desirable item in the set.

We note that many of our applications perform inherently approximate computations, for example because they work with approximate models of reality or because exact solutions are too expensive to compute. In such cases perforation may make an already approximate computation even more approximate.

1.5 Uses of Loop Perforation

Potential uses of loop perforation include:

- **Performance Enhancement:** Guided by the parameters of the tradeoff space, a user can select a desired level of performance that provides acceptable accuracy. Or the user can select a desired accuracy, then use loop perforation to maximize the delivered performance.
- **Energy Savings:** Because loop perforation can reduce the computation required to produce an acceptable result, and computation translates directly into energy consumption, loop perforation can reduce the energy required to produce the result. Moreover, for real-time computations, loop perforation may free up the time required to enable additional synergistic energy-saving optimizations such as voltage scaling [28].
- **New Platforms or Contexts:** Applications often come tuned for use in a specific context or computational platform (for example, desktop machines). Loop perforation can support effective redeployment onto a new platform (for example, mobile devices) or into a new context with different performance or accuracy requirements.
- **Dynamic Adaptation:** Our implemented compiler can generate code that can dynamically move between different variants as the computation executes [16]. This capability enables the construction of control systems that use loop perfo-

ration to dynamically adapt application behavior to satisfy real-time deadlines in the face of changes (such as clock speed changes, load variations, or processor loss) in the underlying computational platform [16].
- **Developer Insight:** A developer can examine the perforation to gain insight into where it is possible to trade accuracy for increased performance [26]. The developer may then choose to accept the perforated computation as is, use the perforated computation as a starting point for further optimization, or use the perforation to identify computations that are suitable targets for manual optimization.

1.6 Contributions

This paper makes the following contributions:

- **Loop Perforation**: It presents and evaluates loop perforation as an automatic technique for generating multiple variants of approximate computations, with the different variants occupying different points in the underlying performance vs. accuracy tradeoff space.
- **Criticality Testing:** It presents a criticality testing algorithm that identifies tunable loops that respond well to perforation.
- **Perforation Space Exploration:** It presents and evaluates exhaustive and greedy algorithms for exploring the perforation space to find a set of Pareto-optimal perforations.
- **Experimental Results**: It presents experimental results for applications from the PARSEC benchmark suite. These results show that loop perforation can deliver significant performance improvements (the perforated applications run up to seven times faster than the original applications) while maintaining accurate execution (the results from the perforated applications differ by at most 10% from the original results).
- **Global Patterns:** It identifies and analyzes global computational patterns (for example, loops that combine multiple partially redundant values and searches driven by perforatable metrics) that interact well with loop perforation. These patterns explain why loop perforation works well for our benchmark applications and can serve as a foundation for the broader application of loop perforation across a large range of applications.

We have previously proposed the use of loop perforation for *quality of service profiling* to help developers find suitable manual optimization targets [26]. This paper presents a new criticality testing methodology for identifying perforatable loops, a new methodology for exploring a perforation space with multiple loop perforation rates, experimental results that characterize the induced performance vs. accuracy tradeoff space for a broad range of perforation rates and inputs (including results that characterize how well results from training input generalize to previously unseen production inputs), and an identification of specific global computational patterns that work well with loop perforation, including an explanation of why loop perforation is applicable to these patterns. Together, these results support the use of loop perforation as an optimization in its own right rather than simply as a profiling technique.

2. PERFORATION TRANSFORMATION

We implement loop perforation within the LLVM compiler framework [20]. The perforator works with any loop that the existing LLVM loop canonicalization pass, *loop-simplify*, can convert into the following form:

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

In this form, the loop has an unique induction variable (in the code above, `i`) initialized to `0` and incremented by `1` on every iteration, with the loop terminating when the induction variable `i` exceeds the bound (in the code above, `b`). The class of loops that LLVM can convert into this form includes, for example, `for` loops that initialize an induction variable to an arbitrary initial value, increment the induction variable by an arbitrary constant value on each iteration, and terminate when the induction variable exceeds an arbitrary bound.

The loop perforation transformation takes as a parameter a loop perforation rate r, which represents the expected percentage of loop iterations to skip. Interleaving perforation transforms the loop to perform every `n`-th iteration (here the perforation rate is r = `1-1/n`). Conceptually, the perforated computation looks like (see [16] for a more detailed treatment):

```
for (i = 0; i < b; i += n) { ... }
```

All of the experimental results in this paper use interleaving perforation. Our implemented perforator can also apply truncation perforation (which skips a contiguous sequence of iterations at either the beginning or the end of the loop) or random perforation (which randomly skips loop iterations) [16].

3. ACCURACY METRIC

The accuracy metric measures the difference between an output from the original program and a corresponding output from the perforated program run on the same input. We decompose the metric into two parts: an *output abstraction*, which maps an output to a number or set of numbers, and an *accuracy calculation*, which measures the difference between the output abstractions from the original and perforated executions. The output abstraction typically selects relevant numbers from an output file or files or computes a measure (such as peak signal to noise ratio) of the quality of the output. Many approximate computations come with such quality measures already defined and available (see Section 5).

The accuracy calculation computes the metric acc as a weighted mean scaled difference between the output abstraction components $o_1, \dots, o_m$ from the original program and the output abstraction components $\hat{o}_1, \dots, \hat{o}_m$ from the perforated program [31]:

$$acc = \frac{1}{m} \sum_{i=1}^{m} w_i \left| \frac{o_i - \hat{o}_i}{o_i} \right| \tag{1}$$

Here each weight w_i captures the relative importance of the ith component of the output abstraction. Note that the closer the accuracy metric acc is to zero, the more accurate the perforated program. We often report accuracy metrics as percentages.

4. PERFORATION EXPLORATION

The loop perforation space exploration algorithm takes as input an application, an accuracy metric for that application, a set of training inputs, an accuracy bound b (a maximum acceptable accuracy metric for the application), and a set of perforation rates (in our experiments, 0.25, 0.50, 0.75, and 1 iteration). The algorithm produces a set S of loops to perforate at specified perforation rates.

4.1 Criticality Testing

The criticalility testing algorithm (Algorithm 1) starts with a set of candidate loops L and perforation rates R. L consists of the loops (identified by profiling) that account for at least 1% of the executed instructions (with a cutoff after the top 25 loops). In general, perforating a candidate loop may cause the program to crash, generate unacceptable output, produce an infinite loop, or decrease its

Algorithm 1 (Criticality Testing) Find the set of tunable loops P in A given training inputs T and accuracy bound b

```
Inputs:
  A - an application
  T - a set of representative inputs
  b - an accuracy bound
  L - a set of candidate loops for perforation
  R - a set of perforation rates
Outputs: P - a set of tunable loops and perforation rates for A
           given b and T

P = ∅
for ⟨l, r⟩ ∈ L × R do
  Let A_⟨l,r⟩ be A with l perforated at rate r
  for t ∈ T do
    Run A_⟨l,r⟩ on t, record speedup sp_t and accuracy acc_t
  sp‾ = (∑_{t∈T} sp_t)/||T|| ; acc‾ = (∑_{t∈T} acc_t)/||T||
  if acc‾ < b ∧ sp‾ > 1 then
    for t ∈ T do
      Run A_⟨l,r⟩ using Valgrind to find Err_t (memory errors)
    if ⋃_{t∈T} E_t = ∅ then
      P = P ∪ {⟨l, r⟩}
return P
```

Algorithm 2 (Greedy Exploration) Find a set S of loops to perforate in A given training inputs T and accuracy bound b

```
Inputs:
  A - an application
  T - a set of representative inputs
  b - an accuracy bound
  P - a set of tunable loops generated by Algorithm 1
Outputs: S - a set of loops to perforate in A given b

P′ = { ⟨l, r⟩ | ⟨l, r⟩ ∈ P ∧ ∀p, score_⟨l,r⟩ ≤ score_⟨l,p⟩ }
S = ∅
for ⟨l, r⟩ ∈ P′ in sorted order according to score_⟨l,p⟩ do
  C = S ∪ {⟨l, r⟩}
  Let A_C be A with all loops in C perforated
  for t ∈ T do
    Run A_C on t, record speedup sp_t and accuracy acc_t
  sp‾ = (∑_{t∈T} sp_t)/||T||; acc‾ = (∑_{t∈T} acc_t)/||T||
  if acc‾ < b ∧ sp‾ > 1 then
    for t ∈ T do
      Run A_C using Valgrind to find Err_t (memory errors)
    if ⋃_{t∈T} E_t = ∅ then
      S = C
return S
```

performance. Algorithm 1 is designed to find and remove such *critical* loops from the set of candidate loops. The algorithm perforates each loop in turn at each of the specified perforation rates, then runs the perforated program on the training inputs. It filters out a loop if its perforation 1) fails to improve the performance (as measured by the speedup sp, which is the execution time of the perforated application divided by the execution time of the original unperforated program running on the same input), 2) causes the application to exceed the accuracy bound b or, 3) introduces memory errors (such as out of bounds reads or writes, reads to uninitialized memory, memory leaks, double frees, etc.). If the memory error causes the execution to crash on some input t, its accuracy loss acc_t is ∞. The result of criticality testing is the set of tunable loops $P = \{\langle l_0, r_0\rangle, \ldots, \langle l_n, r_n\rangle\}$, where $\langle l_i, r_i\rangle$ specifies the perforation of loop l_i at rate r_i.

4.2 Perforation Space Exploration Algorithms

We next present two algorithms (exhaustive and greedy) for exploring the performance vs. accuracy tradeoff space of perforated programs.

Exhaustive Exploration Algorithm. The exhaustive perforation space exploration algorithm starts with the set of tunable loops, then exhaustively explores all combinations of tunable loops l at their specified perforation rates r. The algorithm executes all combinations on all training inputs and records the resulting speedup and accuracy. It also runs each combination under Valgrind [27], discarding the combination if Valgrind detects a memory error. We use the results to compute the set of Pareto-optimal perforations in the induced performance vs. accuracy tradeoff space. A perforation is Pareto-optimal if there is no other perforation that provides both better performance and better accuracy.

This approach is feasible for applications (such as those in our set of benchmarks) that spend most of their time in relatively few loops. If exhaustive exploration is infeasible, it is possible to either use the greedy algorithm or drop enough of the least time-consuming tunable loops to make exhaustive exploration feasible. Hybrid approaches are also possible.

Greedy Exploration Algorithm. Algorithm 2 uses a greedy strategy to search the loop perforation space to produce, for a given accuracy bound b, a set of loops and corresponding perforation rates $S = \{\langle l_0, r_0\rangle, \ldots, \langle l_n, r_n\rangle\}$ that maximize performance subject to b. The algorithm uses a heuristic scoring metric to prioritize loop/perforation rate pairs. The scoring metric for a pair $\langle l, r\rangle$ is based on the harmonic mean of terms that estimate the performance increase and accuracy loss of the perforated program:

$$score_{\langle l,r\rangle} = \frac{2}{\dfrac{1}{\overline{sp}_{\langle l,r\rangle} - 1} + \dfrac{1}{1 - \dfrac{\overline{acc}_{\langle l,r\rangle}}{b}}} \quad (2)$$

where $\overline{sp}_{\langle l,r\rangle}$ and $\overline{acc}_{\langle l,r\rangle}$ are the mean speedup and accuracy metric (respectively) for the $\langle l, r\rangle$ perforation over all training inputs and b is the accuracy bound. In comparison to other heuristic functions based on arithmetic mean or geometric mean, this harmonic mean based metric requires a much higher performance increase to select the loop that causes a small increase in accuracy loss.

The algorithm first computes a set of pairs P'. For each tunable loop l, P' contains the pair $\langle l, r\rangle$, where r maximizes $score_{\langle l,r\rangle}$. It then sorts the pairs in P' by $score_{\langle l,r\rangle}$. The algorithm maintains a set S of $\langle l, r\rangle$ pairs that can be perforated together without violating the accuracy bound b. At each step, the algorithm produces a new version of the application with the loops in $S \cup \langle l, r\rangle$ perforated. If the additional perforation of $\langle l, r\rangle$ keeps the overall accuracy within b, the algorithm adds the pair $\langle l, r\rangle$ into S. Note that for each loop l, this algorithm considers only its best perforation rate r (according to $score_{\langle l,r\rangle}$).

5. BENCHMARKS AND INPUTS

We evaluate loop perforation using a set of benchmark applications from the PARSEC 1.0 benchmark suite [6]. These applications were chosen to be representative of modern and emerging workloads for the next generation of processor architectures. We collect all results using a cluster of Intel x86 servers with dual 3.16 GHz Xeon X5460 quad-core processors.

We use the following applications: x264, bodytrack, swaptions, ferret, canneal, blackscholes, and streamcluster. Together these benchmarks represent a broad range of application domains including financial analysis, media processing, computer vision, engineering, data mining, and similarity search. For each benchmark we acquire a set of evaluation inputs, then pseudorandomly par-

Benchmark	Training Inputs	Production Inputs	Source
x264	4 HD videos of 200+ frames	12 HD videos of 200+ frames	PARSEC & xiph.org [1]
bodytrack	sequence of 100 frames	sequence of 261 frames	PARSEC & additional input provided by benchmark authors
swaptions	64 swaptions	512 swaptions	PARSEC & randomly generated swaptions
ferret	256 image queries	3500 image queries	PARSEC
canneal	4 netlists of 2M+ elements	16 netlists of 2M+ elements	PARSEC & additional inputs provided by benchmark authors
blackscholes	64K options	10M options	PARSEC
streamcluster	4 streams of 19K-100K data points	10 streams of 100K data points	UCI Machine Learning Repository [2]

1: Summary of Training and Production Inputs

tition the inputs into training and production inputs. We use the training inputs to drive the loop perforation space exploration algorithm (Section 4) and the production inputs to evaluate how well the resulting perforations generalize to unseen inputs.

Table 1 summarizes the sources of the evaluation inputs. For each application, the PARSEC benchmark suite contains "native" inputs designed to represent the inputs that the application is likely to encounter in practical use. With the exception of streamcluster, we always include these inputs in the set of evaluation inputs. For many of the benchmarks we also include other representative inputs, typically to increase the coverage range of the evaluation set, to promote an effective partition into reasonably-sized training and production sets, or to compensate for deficiencies in the native PARSEC inputs.

The PARSEC benchmark suite also contains the following benchmarks: facesim, dedup, fluidanimate, freqmine, and vips. We do not include freqmine and vips because these benchmarks do not successfully compile with the LLVM compiler. We do not include dedup and fluidanimate because these applications produce complex binary output files. Not having deciphered the meaning of these files, we were unable to develop meaningful accuracy metrics. We do not include facesim because it does not produce any output at all (except timing information).

x264. This media application performs H.264 encoding on raw video data. It outputs a file containing the raw (uncompressed) input video encoded according to the H.264 standard. The output abstraction for the accuracy metric extracts the peak-signal-to-noise ratio (PSNR) (which measures the quality of the encoded video) and the bitrate (which measures the compression achieved by the encoder). The accuracy calculation (see Section 3) weights each component equally (with a weight of one). If the reference decoder fails to parse the encoded video during training, we record an accuracy metric of 100% and reject the perforation. This accuracy metric captures the two most important attributes of a video encoder: the quality of the encoded video and the compression achieved through the encoding. The native PARSEC input contains only a single video. We therefore augment the evaluation input set with additional inputs from xiph.org [1].

Bodytrack. This computer vision application uses an annealed particle filter to track the movement of a human body [12]. It produces two output files: a text file containing a series of vectors representing the positions of the body over time and a series of images graphically depicting the information in the vectors overlaid on the video frames from the cameras. The output abstraction extracts the vectors that represent the position of the body. In the accuracy calculation, the weight of each vector is proportional to its magnitude. Vectors which represent larger body parts (such as the torso) therefore have a larger influence on the accuracy metric than vectors that represent smaller body parts (such as forearms).

The application requires data collected from carefully calibrated cameras. The native PARSEC input contains a single sequence of 261 frames. We augment the evaluation input set with another sequence of 100 frames. We obtained this sequence from the maintainers of the PARSEC benchmark suite.

Swaptions. This financial analysis application uses a Monte Carlo simulation to solve a partial differential equation that prices a portfolio of swaptions. The output abstraction simply extracts the swaption prices. The distortion calculation weights the corrected swaption prices equally (with a weight of one). The resulting accuracy metric directly captures the ability of the perforated application to produce accurate swaption prices.

Each input to this application contains a set of parameters for each swaption. The native PARSEC input simply repeats the same parameters multiple times, causing the application to repeatedly calculate the same swaption price. We therefore augment the evaluation input set with additional randomly generated parameters so that the application computes prices for a range of swaptions.

Ferret. This application performs a content-based similarity search of an image database. For each input query image, ferret outputs a list of similar images found in its database. The accuracy metric computes the intersection of the sets of images returned by the perforated and original versions, divides the size of this set by the size of the set of images returned by the original version, then subtracts this number from one. So if both versions return 10 images, 9 of which are the same, the accuracy is 0.1. We use the 3500 image queries from the PARSEC benchmark suite.

Canneal. This engineering application uses simulated annealing to minimize the routing cost of a microchip design. Canneal prints a number representing the total routing cost of a netlist representing the chip design; we use this cost as the output abstraction. The resulting accuracy metric directly captures the application's ability to reduce routing costs. The PARSEC benchmark contains only one large netlist. We obtained additional input netlists from the maintainers of the PARSEC benchmark suite.

Blackscholes. This financial analysis application solves a partial differential equation to compute the price of a portfolio of European options. The PARSEC application produces no output. We therefore modified the application to print the option prices to a file. The output abstraction extracts these option prices. The distortion calculation weights these prices equally (with a weight of one). The resulting accuracy metric directly captures the ability of the perforated application to compute accurate option prices. The native input from the PARSEC benchmark suite contains 10 million different option parameters. We do not augment this input set with additional inputs.

Streamcluster. This data mining application solves the online clustering problem. The program outputs a file containing the cluster centers found for the input data set. The output abstraction extracts the quality of the clustering as measured by the BCubed metric [3]. The resulting accuracy metric directly captures the ability of the application to solve the clustering problem. The native PARSEC input consists of a randomly generated set of points drawn from a uniform distribution. For this input all clusterings have equal quality and all clustering algorithms (even the trivial algorithm that outputs no clustering at all) have identical accuracy. We therefore use the evaluation input set with more realistic inputs from the UCI Machine Learning Repository [2].

6. EXPERIMENTAL RESULTS

We next present experimental results for the loop perforation space exploration algorithms (Section 4) for our benchmark applications (Section 5). We use the training inputs to drive the exploration; the result is a set of Pareto-optimal perforations in the loop perforation space. We then run selected Pareto-optimal perforations on the production inputs to evaluate how well the training results generalize to previously unseen inputs.

6.1 Criticality Testing Results

Table 2 presents timing results for the criticality testing runs (Algorithm 1). The table contains a row for each application. The second column (Accuracy) presents timing results for executions that measure the speedup and accuracy of different perforations. The third column (Valgrind) presents timing results for executions that use Valgrind to find memory errors. Each entry of the form $X(Y)$ indicates that algorithm considered X different combinations of perforated loops and that the executions took a total of Y minutes to complete. The total execution times range from 6 minutes for blackscholes to approximately 64 hours for streamcluster, with other applications requiring significantly less time. It is, of course, possible to execute different criticality testing runs in parallel on different machines. We performed all of our runs on eight dual quad Intel Xenon 3.1 GHz machines (so the wall clock times required to perform the runs are approximately a factor of eight less than the total execution times reported in Table 2).

Application	Algorithm 1		Total
	Accuracy	Valgrind	Time
x264	500 (108m)	110 (840m)	949m
bodytrack	100 (35m)	47 (1316m)	1351m
swaptions	100 (7m)	16 (108m)	115m
ferret	100 (17m)	40 (53m)	71m
canneal	256 (405m)	60 (540m)	945m
blackscholes	24 (0.5m)	12 (5m)	5.5m
streamcluster	500 (3083m)	17 (782m)	3865m

2: Criticality Testing Statistics for Benchmark Applications

Table 3 summarizes, for each application, the fate of each loop in the criticality testing algorithm from Section 4.1. Each column presents results for a given perforation rate (0.25, 0.5, 0.75 and 1 iteration) for an accuracy bound of 10% . The first row (Candidate) presents the starting number of candidate loops. This number is always 25 unless the application has fewer than 25 loops that account for 1% of the executed instructions (see Section 4).

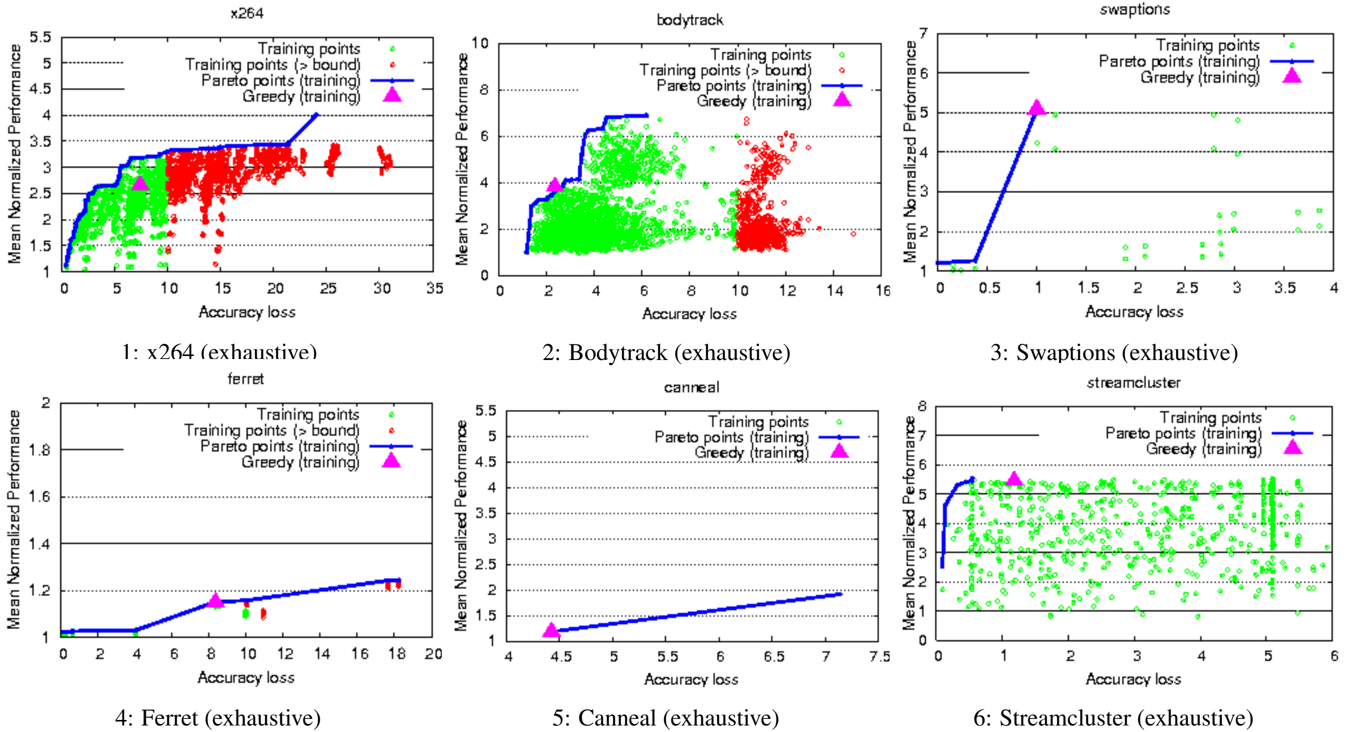
The second row (Crash) presents the number of loops that Algorithm 1 filters out because perforating the loop caused the application to terminate with an error. The third row (Accuracy) presents the number of loops filtered by the algorithm because perforating the loop caused the application to violate the corresponding accuracy bound. The fourth row (Speed) presents the number of remaining loops that Algorithm 1 filters out because perforating the loop does not improve the overall performance (this typically happens for tight loops at a 0.25 perforation rate). The fifth row (Valgrind) presents the number of remaining loops that Algorithm 1 filters out because their perforation introduces a latent memory error detected by the Valgrind memcheck tool [27].

6.2 Perforation Space Exploration Results

Figures 1 through 6 present the results of the exhaustive loop perforation space exploration algorithm. The graphs plot a single point for each explored perforation. The y coordinate of the point is the mean speedup of the perforation (over all training inputs).

x264				
Filter	0.25%	0.50%	0.75%	1 iter
Candidate	25	25	25	25
Crash	1	1	1	1
Accuracy	6	7	7	6
Speed	16	12	10	11
Valgrind	0	0	1	1
Remaining	2	6	6	6
bodytrack				
Filter	0.25%	0.50%	0.75%	1 iter
Candidate	25	25	25	25
Crash	2	5	7	1
Accuracy	1	1	2	2
Speed	12	10	1	1
Valgrind	3	1	6	8
Remaining	7	8	9	13
swaptions				
Filter	0.25%	0.50%	0.75%	1 iter
Candidate	25	25	25	25
Crash	3	6	8	0
Accuracy	13	12	13	16
Speed	5	4	2	2
Valgrind	2	2	1	5
Remaining	2	1	1	2
ferret				
Filter	0.25%	0.50%	0.75%	1 iter
Candidate	25	25	25	25
Crash	8	12	12	6
Accuracy	13	11	11	17
Speed	0	0	0	0
Valgrind	0	0	0	0
Remaining	4	2	2	2
canneal				
Filter	0.25%	0.50%	0.75%	1 iter
Candidate	16	16	16	16
Crash	7	10	10	6
Accuracy	1	1	1	4
Speed	7	4	4	5
Valgrind	0	0	0	0
Remaining	1	1	1	1
blackscholes				
Filter	0.25%	0.50%	0.75%	1 iter
Candidate	6	6	6	6
Accuracy	4	4	4	4
Speed	1	1	1	1
Valgrind	0	0	0	0
Remaining	1	1	1	1
streamcluster				
Filter	0.25%	0.50%	0.75%	1 iter
Candidate	15	15	15	15
Crash	1	1	2	4
Accuracy	0	0	0	0
Speed	13	11	8	7
Valgrind	0	0	0	0
Remaining	1	3	5	4

3: Criticality Testing Results for Individual Loops



1: x264 (exhaustive)

2: Bodytrack (exhaustive)

3: Swaptions (exhaustive)

4: Ferret (exhaustive)

5: Canneal (exhaustive)

6: Streamcluster (exhaustive)

The x coordinate is the corresponding percentage accuracy loss of the perforation. Green points have accuracy losses below 10%; red points have accuracy losses above 10%. The blue line in each graph connects the points from Pareto-optimal perforations (a perforation is Pareto-optimal if there is no other perforation that provides both better performance and better accuracy). The pink triangle identifies the perforation that the greedy algorithm produces.

The graphs show that, for these applications, loop perforation is usually able to increase performance on the training inputs by at least a factor of two (up to a factor of seven for bodytrack) while reducing the accuracy by less than 10% (ferret is the exception). The graphs also illustrate the broad range of points in the performance vs. accuracy tradeoff space that the loop perforation space exploration algorithm is able to find.

The graphs show that the greedy algorithm is able to find points with good combinations of performance and accuracy, but that these points are sometimes less than optimal. We attribute this lack of optimality, in part, to the fact that the greedy algorithm explores only one perforation rate for each loop, specifically the highest priority rate as identified by Equation 2. In some cases this equation conservatively ranks perforations with low speedup and high accuracy over perforations with higher speedup but lower accuracy (even though the lower accuracy is still within the accuracy bound). This is the reason, for example, that the greedy algorithm does not find a better point for canneal (Figure 5).

Table 4 presents the time required to run the exhaustive and greedy exploration algorithms. There is a row for each application and a column for the exhaustive and greedy algorithms. Each entry of the form $X(Y)$ indicates that algorithm considered X different perforated versions of the application and that the runs took a total of Y minutes to complete. As before, the reported times represent the total computation time for execution on a dual quad Intel Xeon running at 3.1 GHz. The exhaustive search times range from 1 minute (for blackscholes) to about 65 hours (for streamcluster). The second column presents the time required to complete the greedy exploration algorithm (Algorithm 2). Note that these times do not include the time required to complete the criticality testing algorithm (Algorithm 1) — Table 2 presents the time required to complete this criticality testing algorithm. The greedy algorithm explores substantially fewer points than and executes in a fraction of the time of the exhaustive algorithm.

Application	Exhaustive	Greedy
x264	3071 (665m)	6 (9m)
bodytrack	5624 (1968m)	14 (33m)
swaptions	32 (9m)	4 (7m)
ferret	255 (43m)	6 (2m)
canneal	2 (12m)	1 (11m)
blackscholes	19 (1m)	3 (0.5m)
streamcluster	639 (3941m)	5 (31m)

4: Exhaustive and Greedy Search Times

6.3 Training and Production Results

Table 5 presents accuracy and speedup results for selected Pareto-optimal perforations in the loop perforation space. There is a row for each application and a group of columns for the training and production inputs. Each group of columns presents results for the Pareto-optimal perforation for four accuracy bounds b: 2.5%, 5%, 7.5%, and 10%. Each entry of the form $X(Y\%)$ presents the corresponding mean speedup X and mean accuracy Y for that combination of application, bound, and input set. With the exception of ferret, all applications show a reasonable correlation between training and production results, indicating that the results from the training inputs generalize well to other inputs.

7. PERFORATION EVALUATION

We next discuss, for each application, our source-code based evaluation of the tunable loops in Pareto-optimal perforations. Table 6 presents data for every loop which is perforated in at least

Application	Training				Production			
	2.5%	5%	7.5%	10%	2.5%	5%	7.5%	10%
x264	2.38 (2.5%)	2.66 (5%)	3.17 (6.53%)	3.25 (9.31%)	2.34 (5.15%)	2.53 (6.08%)	3.12 (8.72%)	3.19 (10.3%)
bodytrack	3.44 (2.23%)	6.32 (4.36%)	6.89 (6.19%)	6.89 (6.19%)	2.70 (4.00%)	4.93 (6.12%)	4.811 (6.58%)	4.811 (6.58%)
swaptions	5.08 (1%)	5.08 (1%)	5.08 (1%)	5.08 (1%)	5.05 (0.2%)	5.05 (0.2%)	5.05 (0.2%)	5.05 (0.2%)
ferret	1.02 (0.2%)	1.03 (4%)	1.03 (4%)	1.16 (10%)	1.002 (0.15%)	1.02 (0.23%)	1.02 (0.23%)	1.07 (7.90%)
canneal	1.14 (4.38%)	1.18 (4.43%)	1.913 (7.14%)	1.913 (7.14%)	1.14 (4.38%)	1.14 (4.38%)	1.46 (7.88%)	1.46 (7.88%)
blackscholes	33 (0.0%)	33 (0.0%)	33 (0.0%)	33 (0.0%)	28.9 (0.0%)	28.9 (0.0%)	28.9 (0.0%)	28.9 (0.0%)
streamcluster	5.51 (0.54%)	5.51 (0.54%)	5.51 (0.54%)	5.51 (0.54%)	4.87 (1.71%)	4.87 (1.71%)	4.87 (1.71%)	4.87 (1.71%)

5: Training and Production Results for Pareto-optimal Perforations for Varying Accuracy Bounds

Function	Time	Type
x264		
x264_mb_analyse_inter_p16x16	64.20%	SSE / Argmin
x264_pixel_sad_16x16, outer	55.80%	SMS+T / Sum
x264_pixel_sad_16x16, inner	54.60%	SMS+T / Sum
x264_me_search_ref	25.00%	SSE / Argmin
pixel_satd_wxh, outer	18.50%	SMS+T / Sum
pixel_satd_wxh, inner	18.30%	SMS+T / Sum
bodytrack		
Update	77.00%	II
ImageErrorInside, inner	37.00%	SME / Ratio
ImageErrorEdge, inner	29.10%	SME / Ratio
InsideError, outer	28.90%	SME / Sum
IntersectingCylinders	1.16%	SMF+SSE
swaptions		
HJM_Swaption_Blocking, outer	100.00%	MC / Mean
HJM_SimPath_Forward_Blocking, outer	45.80%	DSU
HJM_SimPath_Forward_Blocking, inner	31.00%	DSU
Discount_Factors_Blocking	1.97%	DSU
ferret		
emd	37.60%	SMS+II
LSH_query_bootstrap, outer	27.10%	SSE
LSH_query_bootstrap, middle	26.70%	SSE
LSH_query_bootstrap, inner	2.70%	SSE
canneal		
reload	2.38%	DSU
blackscholes		
bs_thread	98.70%	–
streamcluster		
pFL, inner	98.50%	II
pgain	84.00%	SME+T+DSU
dist	69.30%	SME+T / Sum
pgain	5.01%	SME+T

6: Patterns in Pareto-optimal Perforations

one Pareto-optimal variant with an accuracy loss under 10%. The first column presents the function in which the loop appears and (when the loop appears in a loop nest) whether the loop is an inner loop or outer loop of a loop nest. The second column presents the percentage of time spent in each loop (before perforation). The third column presents both the global (as discussed in Section 1) and local (as presented in [25]) computation patterns for the loop. For example, the entry for the first loop in x264 is SSE/Argmin, which indicates that the global pattern for the loop is the Search Space Enumeration pattern, while the local pattern is the Argmin pattern. Some loops are not an instance of any of the identified local patterns. In this case we present the global pattern only.

x264. The x264 encoder divides each frame into *blocks*, then performs the encoding at the granularity of blocks. *Motion estimation* attempts to find a block from a previously encoded frame that is similar to the block that x264 is currently attempting to encode. If it finds such a block, x264 encodes the delta over this previously encoded block. Motion estimation consumes the vast majority of the computation time in x264.

All of the tunable loops in x264 appear in the motion estimation computation. Two of these loops (`x264_mb_analyze_inter_p16x16` and `x264_search_ref`) are instances of the Search Space Enumeration global pattern and the Argmin local pattern (which computes the index and value of the minimum element in an array of elements [25]). The first (`x264_mb_analyze_inter_p16x16`) searches previously encoded reference frames to find a block that is a good match for the block that x264 is currently encoding. Perforating this loop causes x264 to search fewer reference frames. The second (`x264_search_ref`) is given a single reference frame and searches within that frame to find a good match. Perforating this loop causes x264 to consider fewer blocks within the frame.

All of the remaining loops are instances of the Search Metric for both Selection and Termination global pattern and the Sum local pattern (which computes the sum of a set of numbers [25]). These loops all compute a metric that measures how well the current block matches a previously encoded block. Perforating these loops causes x264 to skip pixels when it computes the metric, producing a new, less accurate, but more efficiently computable metric. In addition to using this metric to select the previously encoded block that is the best match, x264 also uses the metric to decide when it should terminate the search (within a frame, x264 terminates the search when it fails to find a new desirable block after a given number of probes). Because the perforated metric makes fewer distinctions between blocks, perforation will typically cause the search to terminate with fewer probes.

All of these perforations may cause x264 to choose a less desirable previously encoded block as a starting point for encoding the current block. But because there is significant redundancy in the set of previously encoded blocks (typically, many previously encoded blocks are a reasonably good match for the current block), the results show that x264 is still usually able to find a good block even after perforation.

Bodytrack. Bodytrack uses an annealed particle filter to track the movement of specific parts of a human body (head, torso, arms, and legs) as a person moves through a scene [12]. At each input video frame bodytrack starts with a probabilistic model of the position of the body from the previous frame. This model consists of a weighted set of *particles*. Each particle consists of a set of angles between body parts that together specify a body pose. Each particle is assigned a weight (a likelihood that it accurately represents the current body pose). The goal of the computation is to compute a new model of the body in the current frame as a weighted average of the particles. Bodytrack starts with the model from the previous frame, then refines the model by iterating through multiple annealing layers. At each annealing layer it processes each particle to create a representation of the body position and location as a set of cylinders, each of which represents a specific body part.

It then compares this representation to image processing data from the current frame, then uses the comparison to update the weight of the corresponding particle.

The `Update` loop performs the annealing steps. Because these steps are designed to improve the accuracy of the model, we identify this loop as an instance of the Iterative Improvement global pattern (even though it is possible for an individual step to produce a slightly less accurate model). Perforating this loop causes bodytrack to perform fewer annealing steps and produce a potentially less accurate model more efficiently.

The `ImageErrorInside`, `ImageErrorEdge`, and `InsideError` loops compute a metric that characterizes how closely the body pose for a given particle matches the observed image data. These loops select a number of sample points along the edges and interiors of the cylinders that represent the body position. We identify these loops as instances of the Search Metric for Selection pattern — they compute values that measure how closely the particle matches the image data. Two of these loops are instances of the Ratio local pattern (which computes the ratio of two sums [25]). The third is an instance of the Sum local pattern. Perforating these loops causes bodytrack to consider only a subset of the sample points when it computes the metric. The result is a more efficiently computable but potentially less accurate metric.

The `IntersectingCylinders` loop iterates over all pairs of cylinders in a given particle to compute if any of the pairs intersect. If so, bodytrack removes the particle from the model (and may potentially replace it with a new particle). We identify this loop as an instance of the Search Metric for Filtering pattern because it computes a simple metric (either the particle is valid or invalid). We also identify this loop as an instance of the Search Space Enumeration pattern because it enumerates all pairs of cylinders. Perforating this loop causes bodytrack to consider only a subset of the pairs of cylinders. The result is that bodytrack may consider a particle to be valid even though it represents an unrealistic body pose. Bodytrack will therefore keep the particle in the model instead of filtering it out. Because these particles represent unrealistic positions, they will typically be given little or no weight in the model and will therefore have little or no effect on the accuracy. Note that this effect may actually decrease the performance (although we observed little or no impact on the performance or accuracy in practice).

Swaptions. Swaptions uses Monte Carlo simulation to price a portfolio of swaptions. Each iteration of the `HJM_Swaption_Blocking` loop computes a single Monte-Carlo sample. Each complete execution of this loop computes the price of a single swaption. Perforating this loop causes swaptions to compute each swaption price with fewer Monte-Carlo samples. Because of the way the computation is coded, the perforation produces prices that are predictably biased by the perforation rate. The system therefore uses extrapolation (as described in [31]) to correct the bias.

The `HJM_SimPath_Forward_Blocking` loop updates the matrix representing the path of the forward interest rate. This matrix is later used to calculate the swaption price. Perforating the computation leaves the skipped matrix elements at their initial value of 0. The `Discount_Factors_Blocking` loop updates a data structure containing discount factors used to compute the price of the swaptions. Perforating the computation leaves the skipped elements at their initial value of 1. Our results show that both of these perforations have minimal impact on the accuracy of the computation.

Ferret. Given a query image, ferret performs a content-based similarity search to return the N images in its image database most similar to the query image. Ferret first decomposes the query image into a set of segments, extracts a feature vector from each segment, indexes its image database to find candidate similar images, ranks the candidate images by measuring the distance between the query image and the candidate images, then returns the highest ranked images. The computation has two main phases. The first phase uses an efficient hash-based technique to retrieve a fixed-length (double the number of requested images) list of likely candidate images from the database. The second phase performs a more accurate comparison between the retrieved images and the query image.

The `LSH_query_bootstrap` loops execute as part of the first phase. The first two loops iterate over lists of images indexed in selected hash buckets to extract the set of candidate images from the database. Perforating these loops causes ferret to skip some of these images so that they are not considered during the subsequent search (these images will therefore not appear in the search result). We identify these loops as instances of the Search Space Enumeration pattern because they iterate over (part of) the search space.

The remaining `LSH_query_bootstrap` loop finds where to insert the current candidate image in the fixed-length sorted list of search results. Perforating this loop may cause the candidate image to appear in the list out of order. Because the second phase further processes the images in this list, the final set of retrieved images is presented to the user in sorted order.

The `emd` loop executes as part of the second phase. This loop computes the *earth mover* distance metric between the query image and the current candidate image from the image database. Because ferret uses this metric to select the most desirable images to return, we identify the loop as an instance of the Search Metric for Selection pattern. This metric is also used for the final sorting of the images according to desirability. Interestingly enough, this search metric is implemented as an instance of the Iterative Improvement pattern – it improves the distance estimate until it obtains an optimal distance measure or exceeds a maximum number of iterations. Perforating this loop creates a new, more efficient, but less accurate metric. As a result, the program may return a different set and/or ordering of images to the user.

Canneal. Canneal uses simulated annealing to place and route an input netlist with the goal of minimizing the total wire length. The reload loop traverses the state vector for the Mersenne twister random number generator to reinitialize the vector. For our set of inputs, the resulting change in the generated sequence of random numbers causes canneal to execute marginally more efficiently.

Blackscholes. The experimental results show that it is possible to perforate the outer loop in `bs_thread` without changing the output at all. Further investigation reveals that this loop simply repeats the same computation multiple times and was apparently added to the benchmark to increase the workload.

Streamcluster. Streamcluster partitions sets of points into clusters, with each cluster centered around one of the points. Each clustering consists of a set of points representing the cluster centers. The goal is to find a set of cluster centers that minimizes the inter- and intra-cluster distances. Streamcluster uses a version of the *facility location* algorithm to find an approximately optimal clustering. The algorithm contains a while loop that executes a sequence of clustering rounds, each of which attempts to improve the clustering from the previous round. The while loop terminates if a round fails to generate a clustering with significantly improved cost.

The `pFL` loop executes once per round. Each iteration of this loop generates (by adding a randomly chosen new candidate cluster center to the current set of cluster centers) and evaluates a new candidate clustering. If this candidate clustering improves on the current clustering, the loop body updates the current clustering (optionally merging some of the clusters). We identify this loop as an instance of both the Search Space Enumeration pattern (because it iterates

over a part of the search space of candidate clusterings) and the Iterative Improvement pattern (because it uses the current clustering to generate improved clusterings). Perforating the `pFL` loop therefore causes streamcluster to consider fewer candidate clusterings per round. The result is that streamcluster performs fewer attempts to improve the clustering before the next round termination check, which may in turn cause streamcluster to produce a less desirable clustering more efficiently.

We note that the following comment appears in the source code above the definition of the constant (`ITER`) that controls the number of iterations of the pFL loop:

```
/* higher ITER --> more likely to get correct number of centers */
/* higher ITER also scales the running time almost linearly */
```

This comment reflects the fact that the number of iterations of the pFL loop controls a performance vs. accuracy tradeoff (which is not exposed to the user of the application). In effect, the perforation space exploration algorithm rediscovers this tradeoff.

The first `pgain` loop calculates the partial cost of a candidate clustering by computing sums of distances between data points and the new cluster center. It also marks the data points that would be assigned to the new cluster center. The second `pgain` loop uses the computed partial sums to compute the total cost of the candidate clustering. We identify these loops as instances of the Search Metrics for Selection (because the computed costs are used to select desirable clusterings) and Termination (because the facility location algorithm uses this cost as a measure of progress, and stops if the progress is too small) pattern. Perforating these loops produces a less accurate but more efficiently computable cluster cost metric. We also identify the first `pgain` loop as an instance of the Data Structure Update pattern. Perforating this loop may leave some of the data points assigned to an old cluster, even though these points should be assigned to the newly opened cluster.

The `dist` loop computes the distance between two points. We identify this loop as an instance of the Search Metric for Selection and Termination pattern because it is used to compute candidate clustering costs. It is also an instance of the Sum local pattern. Perforating this loop causes streamcluster to compute the distance between points using a subset of the coordinates of the points. The result is a more efficiently computable but less accurate distance metric.

8. RELATED WORK

Trading accuracy for other benefits is a well-known technique. It has been shown that one can trade accuracy for performance [31, 30, 13, 21, 17], robustness [31, 8], energy consumption [13, 9, 34, 31, 17], fault tolerance [9, 34, 31], and efficient parallel execution [23, 22]. We note that developers have, for years, manually developed algorithms (most prominently, lossy compression algorithms [36, 7]) that are designed to operate at a variety of points in a performance vs. accuracy tradeoff space. In this section we focus on more general techniques that are designed for relatively broad classes of applications.

Loop Perforation and Task Skipping. As we have earlier discussed [25], loop perforation [16, 26] can be seen as a special case of task skipping [31, 30]. The first publication on task skipping used linear regression to obtain empirical statistical models of the time and accuracy effects of skipping tasks and identified the use of these models in purposefully skipping tasks to reduce the amount of resources required to perform the computation while preserving acceptable accuracy [31].

The first publication on loop perforation presented a purely empirical justification of loop perforation with no formal statistical, probabilistic, or discrete logical reasoning used to justify the transformation [16]. The first statistical justification of loop perforation used Monte Carlo simulation to evaluate the effect of perforating local computational patterns [32]. The first probabilistic justification for loop perforation used static analysis of local computational patterns and also presented the use of profiling runs on representative inputs and developer specifications to obtain the required probability distribution information [24, 25].

Chaudhuri et al. present a program analysis for automatically determining whether a function is continuous [10]. The reasoning is deterministic and worst-case. An extension of this research introduces a notion of function robustness, and, under an input locality condition, presents an approximate memoization approach similar to loop perforation [11]. For a special case when the inputs form a Gaussian random walk and the loop body is a robust function, the paper derives a probabilistic bound to provide a justification for applying loop perforation.

These previous analyses (with the exception of the empirical task skipping analysis [31, 30]) all focus on local computations and do not address the effect of loop perforation on global end-to-end application performance and accuracy. The present paper, in contrast, provides empirical results that characterize the global performance and accuracy effects of loop perforation for our set of benchmarks. It also identifies a set of global computational patterns and explains why these patterns interact well with loop perforation. We anticipate that others may be able to identify these or similar patterns in other applications and use these patterns to justify the application of loop perforation to these applications.

Multiple Implementations. Researchers have developed several systems that allow programmers to provide multiple implementations for a given piece of functionality, with different implementations potentially occupying different points in the performance vs. accuracy tradeoff space. Such systems include Petabricks [4], Green [5], Eon [33], and PowerDial [17]. Petabricks is a parallel language and compiler that developers can use to provide alternate implementations of a given piece of functionality. Green also provides constructs that developers can use to specify alternate implementations. The alternatives typically exhibit different performance and accuracy characteristics. Petabricks and Green both contain algorithms that explore the tradeoff space to find points with desirable performance and accuracy characteristics.

Eon [33] is a coordination language for power-aware computing that enables developers to adapt their algorithms to different energy contexts. In a similar vein, energy-aware adaptation for mobile applications [13], adapts to changing system demands by dynamically adjusting application input quality. For example, to save energy the system may switch to a lower quality video input to reduce the computation of the video decoder.

PowerDial exploits multiple implementations without requiring the developer to explicitly modify the application [17]. Instead, PowerDial converts existing command-line parameters into data structures that a control system can manipulate to dynamically adapt to changes (such as load and clock rate) in the underlying computing platform.

In contrast to systems such as Petabricks, Green, Eon, and PowerDial, loop perforation can find accuracy vs. performance tradeoffs even when none are directly exposed in the application. Instead of requiring the developer to provide multiple implementations of the same functionality or find and annotate potential optimization opportunities, our system generates and explores a performance vs. accuracy tradeoff space to find multiple potentially desirable points in the tradeoff space. Other systems have provided mechanisms that are designed to allow developers to identify loops that perform

iterative refinement [5, 4]. Loop perforation, in contrast, can automatically discover instances of a range of computational patterns that include, but are not limited to, iterative refinement.

Autotuners. Autotuners explore a range of equally accurate implementation alternatives to find the alternative or combination of alternatives that deliver the best performance on the current computational platform [37, 39, 14]. Researchers have also developed APIs that an application can use to expose variables for external control (by, for example, the operating system) [29, 19, 35]. Loop perforation, in contrast, operates on applications written in standard languages to find perforations which improve performance while maintaining acceptable, but not necessarily identical, accuracy.

9. CONCLUSION

Standard approaches for obtaining applications that can trade accuracy in return for enhanced performance have focused on the manual development of new algorithms for specific applications. Our results show that loop perforation can effectively augment a range of applications with the ability to operate at various attractive points in the tradeoff space, with perforated applications often able to deliver significant performance improvements (typically around a factor of two reduction in running time) at the cost of a small (typically 5% or less) decrease in the accuracy.

We acknowledge that loop perforation is not appropriate for all applications. But within its target class of applications, results from our implemented loop perforation system show that it can dramatically increase the ability of applications to trade off accuracy in return for other benefits such as increased performance and decreased energy consumption.

10. REFERENCES

[1] Xiph.org.

[2] D. N. A. Asuncion. UCI machine learning repository, 2007.

[3] E. Amigo, J. Gonzalo, and J. Artiles. A comparison of extrinsic clustering evaluation metrics based on formal constraints. In *Information Retrieval Journal*. Springer Netherlands, July 2008.

[4] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *PLDI '09*, Jun 2009.

[5] W. Baek and T. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI '10*, 2010.

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

[7] K. Brandenburg. MP3 and AAC explained. In *AES 17th International Conference on High-Quality Audio Coding*, 1999.

[8] S. Chakradhar, A. Raghunathan, and J. Meng. Best-Effort Parallel Execution Framework for Recognition and Mining Applications. In *IPDPS*, 2009.

[9] L. Chakrapani, K. Muntimadugu, A. Lingamneni, J. George, and K. Palem. Highly energy and performance efficient embedded computing through approximately correct arithmetic: A mathematical foundation and preliminary experimental validation. In *International conference on Compilers, architectures and synthesis for embedded systems*, 2008.

[10] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL '10*.

[11] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving Programs Robust. In *FSE '11*, 2011.

[12] J. Deutscher and I. Reid. Articulated body motion capture by stochastic search. *International Journal of Computer Vision*, 2005.

[13] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *SOSP '99*.

[14] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, pages 1381–1384.

[15] J. Hartigan and M. Wong. A k-means clustering algorithm. *Journal of the Royal Statistical Society C*, 28:100–108, 1979.

[16] 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.

[17] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic Knobs for Responsive Power-Aware Computing. In *ASPLOS '11*, March 2011.

[18] M. Kalos and P. Whitlock. *Monte Carlo Methods*. Wiley-VCH, 2008.

[19] P. Keleher, J. Hollingsworth, and D. Perkovic. Exposing application alternatives. In *ICDCS '99*.

[20] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04*.

[21] J. Meng, A. A. Raghunathan, and S. B. Chakradhar. Exploiting the Forgiving Nature of Applications for Scalable Parallel Execution. In *IPDPS*, 2010.

[22] S. Misailovic, D. Kim, and M. Rinard. Automatic Parallelization with Statistical Accuracy Bounds. Technical Report MIT-CSAIL-TR-2010-007, MIT, 2010.

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

[24] S. Misailovic, D. Roy, and M. Rinard. Probabilistic and Statistical Analysis of Perforated Patterns. Technical Report MIT-CSAIL-TR-2011-003, MIT, 2011.

[25] S. Misailovic, D. Roy, and M. Rinard. Probabilistically Accurate Program Transformations. In *SAS '11*, 2011.

[26] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of Service Profiling. In *ICSE*, 2010.

[27] N. Nethercote and J. Seward. Valgrind A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 2003.

[28] P. Pillai and K. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01*, page 102, 2001.

[29] R. Ribler, J. Vetter, H. Simitci, and D. Reed. Autopilot: adaptive control of distributed applications. In *High Performance Distributed Computing*, July 1998.

[30] M. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *OOPSLA '07*.

[31] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*, 2006.

[32] M. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. In *Onward! '10*.

[33] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: a language and runtime system for perpetual systems. In *SenSys '07*.

[34] P. Stanley-Marbell and D. Marculescu. Deviation-Tolerant Computation in Concurrent Failure-Prone Hardware. Technical Report ESR-2008-01, Eindhoven University of Technology, January 2008.

[35] C. Tapus, I. Chung, and J. Hollingsworth. Active harmony: Towards automated performance tuning. In *Supercomputing, ACM/IEEE 2002 Conference*, 2002.

[36] G. Wallace. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics*, 1991.

[37] R. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *ACM/IEEE conference on Supercomputing (CDROM)*.

[38] x264. http://www.videolan.org/x264.html.

[39] J. Xiong, J. Johnson, R. W. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. In *PLDI '01*.