Quality of Service Profiling

Sasa Misailovic
MIT CSAIL
misailo@csail.mit.edu

Stelios Sidiroglou
MIT CSAIL
stelios@csail.mit.edu

Henry Hoffmann
MIT CSAIL
hank@csail.mit.edu

Martin Rinard
MIT CSAIL
rinard@csail.mit.edu

ABSTRACT

Many computations exhibit a trade off between execution time and quality of service. A video encoder, for example, can often encode frames more quickly if it is given the freedom to produce slightly lower quality video. A developer attempting to optimize such computations must navigate a complex trade-off space to find optimizations that appropriately balance quality of service and performance.

We present a new *quality of service profiler* that is designed to help developers identify promising optimization opportunities in such computations. In contrast to standard profilers, which simply identify time-consuming parts of the computation, a quality of service profiler is designed to identify subcomputations that can be replaced with new (and potentially less accurate) subcomputations that deliver significantly increased performance in return for acceptably small quality of service losses.

Our quality of service profiler uses *loop perforation* (which transforms loops to perform fewer iterations than the original loop) to obtain implementations that occupy different points in the performance/quality of service trade-off space. The rationale is that optimizable computations often contain loops that perform extra iterations, and that removing iterations, then observing the resulting effect on the quality of service, is an effective way to identify such optimizable subcomputations. Our experimental results from applying our implemented quality of service profiler to a challenging set of benchmark applications show that it can enable developers to identify promising optimization opportunities and deliver successful optimizations that substantially increase the performance with only small quality of service losses.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – performance

General Terms

Performance, Experimentation, Measurement

ICSE '10, May 2-8 2010, Cape Town, South Africa

Keywords

Profiling, Loop Perforation, Quality of Service

1. INTRODUCTION

Performance optimization has been an important software engineering activity for decades. The standard approach typically involves the use of profilers (for example, `gprof` [13]), to obtain insight into where the program spends its time. Armed with this insight, developers can then focus their optimization efforts on the parts of the program that offer the most potential for execution time reductions.

In recent years a new dimension in performance optimization has emerged — many modern computations (such as information retrieval computations and computations that manipulate sensory data such as video, images, and audio) exhibit a trade off between execution time and quality of service. Lossy video encoders, for example, can often produce encoded video frames faster if the constraints on the quality of the encoded video are relaxed [16]. Developers of such computations must navigate a more complex optimization space. Instead of simply focusing on replacing inefficient subcomputations with more efficient subcomputations that produce the same result, they must instead consider (a potentially much broader range of) subcomputations that 1) increase performance, 2) may produce a different result, but 3) still maintain acceptable quality of service.

For such developers, existing profilers address only one side (the performance side) of the trade off. They provide no insight into the quality of service implications of modifying computationally expensive subcomputations. To effectively develop computations with an appropriate trade off between performance and quality of service, developers need new profilers that can provide insight into both sides of the trade off.

1.1 Quality of Service Profiling

The goal of quality of service profiling is to provide information that can help developers identify *forgiving subcomputations* — i.e., subcomputations that can be replaced with different (and potentially less accurate) subcomputations that deliver increased performance in return for acceptable quality of service losses. Note that a necessary prerequisite for quality of service profiling is the availability of multiple implementations (with different performance and quality of service characteristics) of the same subcomputation.

We present a technique, *loop perforation*, and an associated profiler that is designed to give developers insight into

the performance versus quality of service trade-off space of a given application. Loop perforation automatically generates multiple implementations (with different performance and quality of service characteristics) of a given subcomputation. The profiling tool leverages the availability of these multiple implementations to explore a range of implementation strategies and generate tables that summarize the results of this exploration. The goal is to help developers focus their optimization efforts on subcomputations that do not just consume a significant amount of the computation time but also offer demonstrated potential for significant performance increases in combination with acceptable quality of service losses.

1.1.1 Loop Perforation

Given a loop, loop perforation transforms the loop so that it performs fewer iterations (for example, the perforated loop may simply execute every other iteration of the original loop). Because the perforated loop executes fewer iterations, it typically performs less computational work, which may in turn improve the overall performance of the computation. However, the perforated loop may also produce a different result than the original loop, which may, in turn, reduce the computation's quality of service. An appropriate analysis of the effect of loop perforation can therefore help the developer identify subcomputations that can be replaced with less computationally expensive (and potentially less accurate) subcomputations while still preserving an acceptable quality of service.

The rationale behind the use of loop perforation to identify promising optimization opportunities is that many successful optimizations target partially redundant subcomputations. Because this partial redundancy often manifests itself as extra loop iterations, one way to find such subcomputations is to find loops that the profiler can perforate with only small quality of service losses. Our experimental results (see Sections 4 and 5) support this rationale — many of the optimization opportunities available in our set of benchmark applications correspond to partially redundant subcomputations realized as loops in heuristic searches of complex search spaces. Our results indicate that quality of service profiling with loop perforation is an effective way to uncover these promising optimization targets.

1.1.2 Quality of Service Metrics and Requirements

To quantify the quality of service effects, the profiler works with a developer-provided quality of service metric. This metric takes an output from the original application, a corresponding output from perforated application run on the same input, and returns a non-negative number that measures how much the output from the perforated application differs from the output from the original application. Zero indicates no difference, with larger numbers indicating correspondingly larger differences. One typical quality of service metric uses the scaled difference of selected output fields. Another uses the scaled difference of output quality metrics such as peak signal to noise ratio.

A quality of service requirement is simply a bound on the quality of service metric. For example, if a 10% difference from the output of the original program is acceptable, the appropriate quality of service requirement for a scaled difference metric is 0.1.

1.1.3 Individual Loop Profiling

The profiler starts by characterizing the impact of perforating individual loops on the performance and quality of service. For each loop in the program that consumes a significant amount of the processing time, it generates a version of the program that perforates the loop. It then runs this perforated version, recording the resulting execution time and quality of service metric. It is possible for the perforation to cause the program to crash or generate clearly unacceptable output. In this case the profiler identifies the loop as a *critical loop*. Loops that are not critical are *perforatable*.

The developer can use the individual loop profiling results (as presented in the generated profiling tables, see Section 4) to obtain insight into potentially fruitful parts of the program to explore for optimizations. The profiling tables can also identify parts of the program that show little potential as optimization targets.

1.1.4 Combined Loop Profiling

The profiler next explores potential interactions between perforated loops. Given a quality of service requirement, the profiler searches the loop perforation space to find the point that maximizes performance while satisfying the quality of service requirement. The current algorithm uses a cost/benefit analysis of the performance and quality of service trade off to order the loops. It then successively and cumulatively perforates loops in this order, discarding perforations that exceed the quality of service requirement. The result is a set of loops that, when perforated together, deliver the required quality of service.

The developer can use the combined loop profiling results (as presented in the generated tables, see Section 4) to better understand how the interactions (both positive and negative) between different perforated loops affect the performance and quality of service. Positive interactions can identify opportunities to optimize multiple subcomputations with more performance and less quality of service loss than the individual loop perforation results might indicate. Conversely, negative interactions may identify subcomputations that cannot be optimized simultaneously without larger than expected quality of service losses.

1.2 Results and Scope

We have applied our techniques to a set of applications from the PARSEC benchmark suite [9]. This benchmark suite is designed to contain applications representative of modern workloads for the emerging class of multicore computing systems. Our results show that, in general, loop perforation can increase the performance of these applications on a single benchmark input by a factor of between two or three (the perforated applications run between two and three times faster than the original applications) while incurring quality of service losses of less than 10%. Our results also indicate that developers can use the profiling results to develop new, alternative subcomputations that deliver significantly increased performance with acceptably small quality of service losses.

Separation of Optimization Targets: Our results show that quality of service profiling produces a clear separation of optimization targets. Some perforated loops deliver significant performance improvements with little quality of service loss. The subcomputations in which these loops appear are good optimization targets — at least one optimization at-

tempt (loop perforation) succeeded, and the application has demonstrated that it can successfully tolerate the perturbations associated with this optimization attempt. Other, potentially more targeted, optimization attempts may therefore also succeed.

Other perforated loops, on the other hand, either fail to improve the performance or produce large quality of service losses. These loops are much less compelling optimization targets because the first optimization attempt failed, indicating that the application may be unable to tolerate perturbations associated with other optimization attempts.

Application Properties: Guided by the profiling results, we analyzed the relevant subcomputations to understand how loop perforation was able to obtain such significant performance improvements with such small quality of service losses. In general, we found that the successfully perforated loops typically perform (at least partially) redundant subcomputations in heuristic searches over a complicated search space. For five out of seven of our applications, the results show that it is possible to obtain a significantly more efficient search algorithm that finds a result that is close in quality to the result that the original search algorithm finds. This analysis clearly indicates that, for our set of benchmark applications, loops that perform partially redundant computations in heuristic search algorithms are an appropriate optimization target (and an optimization target that our profiler can enable developers to quickly and easily find).

Manual Optimization Results: We also used the profiling results to identify optimization opportunities in two PARSEC applications (x264, a video encoder, and bodytrack, a computer vision application). Guided by the profiling results, we manually developed alternate implementations of the identified subcomputations. These alternate implementations deliver significantly increased performance with small quality of service losses.

Scope: We note that our overall approach is appropriate for computations with a range of acceptable outputs. Computations (such as compilers or databases) with hard logical correctness requirements and long dependence chains that run through the entire computation may not be appropriate targets for optimizations that may change the output that the program produces.

1.3 Contributions

This paper makes the following contributions:

- **Quality of Service Profiling:** It introduces the concept of using a quality of service profiler to help the developer obtain insight into the performance and quality of service implications of optimizing different subcomputations.
- **Loop Perforation:** Any quality of service profiler needs a mechanism to obtain alternative subcomputations with a range of performance and quality of service characteristics. This paper identifies loop perforation as an effective mechanism for automatically obtaining alternative subcomputations for this purpose.
- **Rationale:** It explains why loop perforation is an effective mechanism for identifying promising optimization opportunities. Many optimizable subcomputations manifest themselves as loops that perform extra iterations. Eliminating loop iterations, in combination with measuring quality of service effects, is an effective way to find such subcomputations.
- **Experimental Results:** It presents experimental results that characterize the effectiveness of quality of service profiling. These results use a set of benchmark applications chosen to represent modern workloads for emerging multicore computing platforms.
  - **Loop Perforation Results:** It presents individual and cumulative loop perforation tables for our benchmark applications. The information in these tables provides a good separation between optimization targets, enabling developers to identify and focus on promising targets while placing a lower priority on less promising targets.
  - **Manual Optimization Results:** It presents results that illustrate how we were able to use the profiling results in two manual optimization efforts. These efforts produced optimized versions of two benchmarks. These versions combine significantly improved performance with small quality of service losses.
- **Unsound Program Transformations:** Over the last several years the field has developed a range of unsound program transformations (which may change the semantics of the original program in principled ways) [23, 8, 20, 10, 11, 18, 17, 14]. This paper presents yet another useful application of an unsound transformation (loop perforation), further demonstrating the advantages of this approach.

2. LOOP PERFORATION

We implemented the loop perforation transformation as an LLVM compiler pass [15]. Our evaluation focuses on applications written in C and C++, but because the perforator operates at the level of the LLVM bitcode, it can perforate applications written in any language (or combination of languages) for which an LLVM front end exists.

The perforator works with any loop that the existing LLVM loop canonicalization passes, `loopsimplify` and `indvars`, can convert into the following form:

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

In this form, there is an 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 or decrement the induction variable by an arbitrary constant value on each iteration, and terminate when the induction variable exceeds an arbitrary bound.

The *perforation rate* (r) determines the percentage of iterations that the perforated loop skips — the loop perforation transformation takes r as a parameter and transforms the loop to skip iterations at that rate. Modulo perforation transforms the loop to perform every `n`th iteration (here the perforation rate is $r = \texttt{n-1/n}$):

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

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). The loop

perforator takes as input a specification of which loops to perforate, what kind of perforation to apply (modulo, truncation, or random), and the perforation rate r. It produces as output the corresponding perforated program. All of the experimental results in this paper use modulo perforation with an `n` of 2 and a perforation rate of $\frac{1}{2}$ (i.e., the perforated loop skips half the iterations of the original loop).

3. QUALITY OF SERVICE METRIC

In general, it is possible to use any quality of service metric that, given two outputs (typically one from the original application and another from the perforated application), produces a measure of the difference between the outputs.

The quality of service metric for our benchmark applications all use a program output abstraction to obtain numbers from the output to compare. These abstractions typically select important output components or compute a measure (such as peak signal to noise ratio) of the quality of the output. The quality of service metric simply computes the relative scaled difference between the produced numbers. Specifically, we assume the output abstraction produces a sequence of numbers $o_1, \ldots, o_m$. Given numbers $o_1, \ldots, o_m$ from an unmodified execution and numbers $\hat{o}_1, \ldots, \hat{o}_m$ from a perforated execution, the following quantity d, which we call the *distortion* [21], measures the accuracy of the output from the perforated execution:

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

The closer the distortion d is to zero, the less the perforated execution distorts the output. By default the distortion equation weighs each component equally, but it is possible to modify the equation to weigh some components more heavily than others.

The distortion measures the absolute error that loop perforation (or, for that matter, any other transformation) induces. It is also sometimes useful to consider whether there is any systematic direction to the error. We use the *bias* [21] metric to measure any such systematic bias. If there is a systematic bias, it may be possible to compensate for the bias to obtain a more accurate result.

4. PROFILING RESULTS

To evaluate the effectiveness of our approach, we apply quality of service profiling to seven benchmarks chosen from the PARSEC benchmark suite [9]. Unless otherwise noted, the input for the profiling runs is the `simlarge` input provided as part of the benchmark suite.

- **x264.** This media application performs H.264 encoding on a video stream. The quality of service metric includes the mean distortion of the peak signal-to-noise ratio (PSNR) (as measured by the H.264 reference decoder) and the bitrate of the encoded video. We ran this application on the tractor input from xiph.org (available at `http://media.xiph.org/video/derf/`).
- **streamcluster.** This data mining application solves the online clustering problem. The quality of service metric uses the *BCubed* (B^3) clustering quality metric [4]. This clustering metric calculates the homogeneity and completeness of the clustering generated by the application, based on external class labels for data points. The value of the metric ranges from 0 (bad clustering) to 1 (excellent clustering). The quality of service metric itself is calculated as the difference between the clustering quality of the perforated and original application. It is possible for the perforated program to perform better than the unmodified version, in which case the quality of service metric is zero.
  For this application, the `simlarge` input contains a uniformly distributed set of points with no clusters. This input is therefore not representative of production data. We instead ran streamcluster on the `covtype` input from the UCI machine learning repository [2].
- **swaptions.** This financial analysis application uses Monte-Carlo simulation to solve a partial differential equation and price a portfolio of swaptions. The quality of service metric uses the scaled difference of the swaption prices from the original and perforated applications. With the `simlarge` input from the PARSEC benchmark suite, all swaptions have the same value. To obtain a more realistic computation, we changed the input to vary the interest rates of the swaptions.
- **canneal.** This engineering application uses simulated annealing to minimize the routing cost of microchip design. The quality of service metric is the scaled difference between the routing costs from the perforated and original versions.
- **blackscholes.** This financial analysis application solves a partial differential equation to compute the price of a portfolio of European options. The quality of service metric is the scaled difference of the option prices.
- **bodytrack.** This computer vision application uses an annealed particle filter to track the movement of a human through a scene. The quality of service metric uses the relative mean squared error of the series of vectors that the perforated application produces to represent the changing configurations of the tracked body (as compared with the corresponding series of vectors from the original application). The metric divides the relative mean squared errors by the magnitudes of the corresponding vectors from the original application, then computes the mean error for all vectors as the final quality of service metric. The `simlarge` input from the PARSEC benchmark suite has only four frames. We therefore used the first sixty frames from the PARSEC `native` input. These sixty frames contain the four frames from the `simlarge` input.
- **ferret.** This search application performs content-based similarity search on an image database. It returns a list of images present in the database whose content is similar to an input image. The quality of service metric is based on the intersection of the sets returned by the original and perforated versions. Specifically, the quality of service metric is 1 minus the number of images in the intersection divided by the number of images returned by the original version.

In addition to these benchmarks, the PARSEC benchmark suite 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. Because we were unable to decipher the meaning of these files given the time available to us for this purpose, we were unable to develop meaningful

quality of service metrics. We do not include facesim because it does not produce any output at all (except timing information).

4.1 Single Loop Profiling Results

Table 1 presents the results of the single loop profiling runs. The rows of the table are grouped by application. Each row of the table presents the results of the profiling run for a single perforated loop. The rows are sorted according to the number of instructions executed in the loop in the original unperforated application.[1] Our profiling runs perforated all loops that account for at least 1% of the executed instructions. For space reasons, we present at most the top eight loops for each application in Table 1.

First Column (Function): The first column contains the name of the function that contains the loop, optionally augmented with additional information to identify the specific loop within the function. Many functions contain a single loop nest with an outer and inner nested loop; the loops in such functions are distinguished with the outer and inner labels. Other functions contain multiple loops; the loops in such functions are distinguished by presenting the line number of the loop in the file containing the function.

Second Column (Instruction %): The second column presents the percentage of the (dynamically executed) instructions in the loop for the profiling run. Note that because of both interprocedural and intraprocedural loop nesting, instructions may be counted in the execution of multiple loops. The percentages may therefore sum to over 100%.

Third Column (Quality of Service): The third column presents the quality of service metric for the run. Recall that a quality of service metric of zero indicates no quality of service loss. A quality of service metric of 0.5 typically indicates a 50% difference in the output of the perforated application as compared with the output of the original application. A dash (-) indicates that the perforated execution either crashed or produced a clearly unacceptable output.

Fourth Column (Speedup): The fourth and final column presents the speedup of the perforated application — i.e., the execution time of the perforated application divided by the execution time of the original application. Numbers greater than 1 indicate that the perforated application runs faster than the original; numbers less than 1 indicate that the perforated application runs slower than the original.

Interpreting the Results: Good candidate subcomputations for manual optimization contain loops with the following three properties: 1) the application spends a significant amount of time in the loop, 2) perforating the loop significantly increases the performance, and 3) perforating the loop causes small quality of service losses. The developer can easily identify such loops by scanning the data in Table 1. For example (and as discussed further in Section 5), the `pixel_satd_wxh` loops in x264 are promising candidates for optimization because they have all three properties. The success of the loop perforation optimization provides evidence that the computation is amenable to optimization and that other optimization attempts may also succeed.

[1] Our technique is designed to work with any execution time profiler. Our profiling runs use an execution time profiler that counts the number of times each basic block executes. This execution time profiler uses an LLVM compiler pass to augment the program with the instrumentation required to count basic block executions.

Perhaps more importantly, the developer can easily reject otherwise promising candidates that fail to satisfy one of the properties. For example, the `pixel_sub_wxh2018`, outer loop in x264 would be a reasonable candidate for optimization except for the large quality of service loss that the application suffers when the loop is perforated. In comparison with a standard profiler, which only provides information about where the application spends its execution time, the additional information present in a quality of service profile can help the developer focus on promising optimization opportunities with demonstrated potential while placing a lower priority on opportunities with more potential obstacles.

4.2 Multiple Loop Profiling Results

Table 2 presents the profiling results for the multiple loop profiling runs. The quality of service requirement for each application is set to 0.10 (which corresponds to a 10% difference between the outputs from the original and perforated applications). The rows are grouped by application, with each row presenting the profiling results from augmenting the existing set of perforated loops with the next loop. The first column (function) identifies the loop added to set of perforated loops from the preceding rows. The next two columns repeat the profiling results for that loop from the single loop profiling runs as presented in Table 1. The final two columns present the quality of service metric and speedup from the corresponding cumulative profiling run. This run perforates all loops from all rows up to and including the current row.

Interpreting the Results: The multiple loop profiling results identify a group of loops that have demonstrated potential when optimized together. The lack of negative interactions between the perforated loops in the group indicates that other optimization attempts that target all of the corresponding computations as a group may also succeed.

4.3 Individual Application Results

Profiling Results for x264: The single loop profiling results for x264 (Table 1) indicate that the the top two loops in the table (the outer and inner loops from `pixel_satd_wxh`) are promising optimization candidates — these loops account for a significant percentage of the executed instructions and perforation delivers a significant performance improvement with more than acceptable quality of service loss. Guided by these profiling results, we were able to develop an optimized implementation of the corresponding functionality with even more performance and less quality of service loss than the perforated version (see Section 5).

The profiling results also identify subcomputations that are poor optimization candidates. Perforating the loops in `pixel_sub_wxh2018`, for example, produces a large quality of service loss for relatively little performance improvement. Perforating the loop in `x264_mb_analyse_inter_p8x8` (as well as several other loops) causes x264 to crash.

The cumulative profiling results in Table 2 highlight these distinctions. The final perforated collection of loops contains all of the promising optimization candidates from Table 1 and none of the poor optimization candidates. The cumulative profiling results also include loops that were not in the top 8 loops from Table 2. Because perforating these loops adds relatively little performance but also relatively little quality of service loss, they made it into the cumulative set of perforated loops.

x264			
Function	Instruction %	Quality of Service	Speedup
pixel_satd_wxh, outer	41.70%	0.0365	1.459
pixel_satd_wxh, inner	40.90%	0.0466	1.452
refine_subpel	40.90%	0.0005	1.098
pixel_sub_wxh2018, outer	20.50%	–	–
x264_mb_analyse_inter_p8x8	20.10%	–	–
pixel_sub_wxh2018, inner	16.20%	0.2787	1.210
pixel_avg, outer	12.90%	0.0158	1.396
x264_mb_analyse_inter_p8x16, outer	12.30%	–	–
streamcluster			
Function	Instruction %	Quality of Service	Speedup
pFL, inner	99.10%	0.0065	1.137
pgain	95.30%	0.2989	0.865
dist	92.00%	0.5865	1.249
swaptions (with bias correction)			
Function	Instruction %	Quality of Service	Speedup
worker	100.00%	1.0000	1.976
HJM_Swaption_Blocking, outer	100.00%	0.0285	1.979
HJM_SimPath_Forward_Blocking, L.74	45.80%	1.5329	1.146
HJM_SimPath_Forward_Blocking, L.75	45.80%	1.5365	1.167
HJM_SimPath_Forward_Blocking, L.77	43.70%	1.5365	0.982
HJM_SimPath_Forward_Blocking, L.80	31.00%	0.3317	1.011
HJM_SimPath_Forward_Blocking, L.41	15.10%	0.0490	1.008
HJM_SimPath_Forward_Blocking, L.42	15.00%	1.9680	1.005
canneal			
Function	Instruction %	Quality of Service	Speedup
annealer_thread::run	68.40%	0.0014	0.992
netlist_elem::swap_cost, L.80	26.20%	0.0048	0.981
netlist_elem::swap_cost, L.89	26.20%	–	–
netlist::netlist	25.30%	0.0000	0.989
MTRand::reload, L.307	2.79%	0.0414	1.079
netlist_elem::routing_cost_given_loc, L.56	1.84%	0.2000	1.002
netlist_elem::routing_cost_given_loc, L.62	1.79%	–	–
MTRand::reload, L.309	1.42%	0.0254	1.079
blackscholes			
Function	Instruction %	Quality of Service	Speedup
main, outer	98.70%	0.0000	1.960
main, inner	98.70%	0.4810	1.885
bodytrack			
Function	Instruction %	Quality of Service	Speedup
mainPthreads	99.40%	1.00000	1.992
ParticleFilter::Update	74.70%	0.00050	1.542
ImageMeasurements::ImageErrorInside, outer	36.00%	0.00038	1.201
ImageMeasurements::ImageErrorInside, inner	35.90%	0.00028	1.165
ImageMeasurements::InsideError, outer	28.10%	0.00036	1.121
ImageMeasurements::ImageErrorEdge, outer	27.30%	0.00062	1.112
ImageMeasurements::ImageErrorEdge, inner	27.30%	0.00146	1.033
ImageMeasurements::InsideError, inner	24.00%	0.00023	1.134
ferret			
Function	Instruction %	Quality of Service	Speedup
raw_query, L.168	62.30%	–	–
emd, L.134	37.60%	0.0020	1.020
__LSH_query, L.569	27.70%	0.4270	1.60
LHS_query_bootstrap, L.243	27.10%	–	–
LHS_query_bootstrap, L.247	26.80%	0.2523	1.162
emd, L.418	19.10%	0.8391	6.663
emd, L.419	18.70%	–	–
LSH_query_bootstrap, L.254	15.30%	–	–

Table 1: Single loop profiling results. For Quality of Service, smaller is better. For Speedup, larger is better.

x264				
	Individual		Cumulative	
Function	Quality of Service	Speedup	Quality of Service	Speedup
pixel_satd_wxh, outer	0.0365	1.459	0.0365	1.459
pixel_satd_wxh, inner	0.0466	1.452	0.0967	1.672
refine_subpel	0.0005	1.098	0.0983	1.789
pixel_sad_8x8, outer	0.0001	1.067	0.0996	1.929
pixel_sad_8x8, inner	0.0003	1.060	0.0994	1.986
x264_me_search_ref	0.0021	1.014	0.0951	2.058
streamcluster				
	Individual		Cumulative	
Function	Quality of Service	Speedup	Quality of Service	Speedup
pFL, inner	0.0065	1.137	0.0065	1.137
swaptions (with bias correction)				
	Individual		Cumulative	
Function	Quality of Service	Speedup	Quality of Service	Speedup
HJM_Swaption_Blocking, outer	0.0285	1.979	0.0285	1.979
HJM_Swaption_Blocking, middle	0.0111	1.015	0.0245	2.044
HJM_Swaption_Blocking, inner	0.0131	1.009	0.0255	2.068
HJM_SimPath_Forward_Blocking, L.41	0.0490	1.008	0.0799	2.117
canneal				
	Individual		Cumulative	
Function	Quality of Service	Speedup	Quality of Service	Speedup
MTRand::reload, L.307	0.0254	1.079	0.0254	1.079
blackscholes				
	Individual		Cumulative	
Function	Quality of Service	Speedup	Quality of Service	Speedup
main, outer	0.0000	1.960	0.0000	1.960
bodytrack				
	Individual		Cumulative	
Function	Quality of Service	Speedup	Quality of Service	Speedup
ParticleFilter::Update	0.00050	1.542	0.00050	1.542
ImageMeasurements::ImageErrorInside, outer	0.00038	1.201	0.00032	1.809
ImageMeasurements::ImageErrorInside, inner	0.00028	1.165	0.00040	1.960
TrackingModel::GetObservation	0.00371	1.200	0.00309	2.282
ImageMeasurements::InsideError, inner	0.00023	1.134	0.00573	2.376
ImageMeasurements::InsideError, outer	0.00036	1.121	0.00684	2.420
ImageMeasurements::ImageErrorEdge, inner	0.00146	1.033	0.00684	2.627
ImageMeasurements::EdgeError, L.71	0.00099	1.049	0.00500	2.742
ImageMeasurements::EdgeError, L.64	0.00056	1.042	0.00564	2.827
ImageMeasurements::ImageErrorEdge, outer	0.00062	1.112	0.01260	2.984
ferret				
	Individual		Cumulative	
Function	Quality of Service	Speedup	Quality of Service	Speedup
emd, L.134	0.0020	1.020	0.0020	1.020
LSH_query_bootstrap, L.257	0.0066	1.021	0.0066	1.021

Table 2: Multiple loop profiling results using 0.1 quality of service bound. For Quality of Service, smaller is better. For Speedup, larger is better.

An analysis of x264 shows that the vast majority of successfully perforated loops perform computations that are part of *motion estimation* [12]. Motion estimation performs a heuristic search for similar regions of different frames. The profiling results indicate that perforation somewhat degrades the quality of this heuristic search, but by no means disables it. Indeed, eliminating motion estimation entirely makes the encoder run more than six times faster than the original version but, unfortunately, increases the size of the encoded video file by more than a factor of three. The cumulatively perforated version, on the other hand, runs more than a factor of two faster than the original program, but increases the size of the encoded video file by less than 18% (which indicates that motion estimation is still working well even after perforation). This result (which is directly reflected in the profiling tables) shows that there is a significant amount of redundancy in the motion estimation computation, which makes this computation a promising target for optimizations that trade small quality of service losses in return for substantial performance increases.

Profiling Results for streamcluster: Perforating the loop in `pgain` actually decreases the performance. Upon examination, the reason for this performance decrease becomes clear — this loop is embedded in a larger computation that executes until it satisfies its own internally calculated result quality metric. Perforating this loop causes the computation to take longer to converge, which decreases the performance.

Perforating the inner loop in `pFL`, on the other hand, does produce a performance improvement. This loop controls the number of centers considered when generating a new clustering. Perforation causes the computation to relax the constraint on the actual number of centers. In practice this modification increases performance but in this case does not harm quality of service — there is enough redundancy in the default set of potential new centers that discarding other new centers does not harm the overall quality of service.

The loop in `dist` computes the Euclidean distance between two points. Perforating this loop causes the computation to compute the distance in a projected space with half the dimensions of the original space, which in turn causes a significant drop in the quality of service.

Profiling Results for swaptions: The single loop profiling results for swaptions indicate that there is only one promising optimization candidate: the outer loop in the function `HJM_Swaption_Blocking`. Further examination reveals that perforating this loop reduces the number of Monte-Carlo trials. The result is a significant performance increase in combination with a reduction in the swaption prices proportional to the percentage of dropped trials. The bias correction mechanism (see Section 3) corrects this reduction and significantly decreases the drop in quality of service.

Perforating the other loops either crashes the application or produces an unacceptable quality of service loss. The cumulative profiling results reflect this fact by including `HJM_Swaption_Blocking` from the single loop profiling runs plus several other loops that give a minor performance increase while preserving acceptable quality of service.

Profiling Results for canneal: The profiling results for canneal identify no promising optimization candidates — perforating the loops typically provides little quality of service loss, but also little or no performance gain.

Profiling Results for blackscholes: The profiling results for blackscholes indicate that there are only two loops of interest. Perforating the first loop (the outer loop in main) produces a significant speedup with no quality of service loss whatsoever. Further investigation reveals that this loop was apparently added to artificially increase the computational load to make the benchmark run longer. While this loop is therefore not interesting in a production context, these profiling results show that our technique is able to identify completely redundant computation. Perforating the other loop (in `main,inner`) produces unacceptable quality of service loss coupled with significant performance improvement.

Profiling Results for bodytrack: The profiling results for bodytrack indicate that almost all of the loops are promising optimization candidates. Only one of the loops (the top loop, `mainPthreads`) has substantial quality of service loss when perforated. The others all have very small quality of service losses. And indeed, the cumulative profiling results show that it is possible to perforate all of these loops (and more, the cumulative profiling results in Table 2 present only the first eight perforated loops) while keeping the quality of service losses within more than acceptable bounds.

We attribute this almost uniformly good quality of service even after perforation to two sources of redundancy in the computation. First, bodytrack uses a Monte Carlo approach to sample points in the captured images. It subsequently processes these points to recognize important image features such as illumination gradients. Sampling fewer points may reduce the accuracy of the subsequent image processing computation, but will not cause the computation to fail or otherwise dramatically alter its behavior. Second, bodytrack does not simply perform one sampling phase. It instead performs a sequence of sampling phases, with the results of one sampling phase used to drive the selection of points during the next sampling phase. Once again, performing fewer sampling phases may reduce the accuracy of the overall computation, but will not cause the computation to fail or dramatically alter its behavior.

Optimizations that target such sources of redundancy can often deliver significant performance gains without corresponding reductions in the quality of service. This application shows how quality of service profiling can help the developer identify such sources of redundancy. See Section 5 for a discussion of how we used quality of service profiling to develop a manually optimized version of this application.

Profiling Results for ferret: For each query image, ferret performs two processing phases. In the first phase ferret divides the image into segments. In the second phase ferret uses the image segments to find similar images in its database. Our profiling results show that the most time consuming loops are found in the database query phase. For most of these loops, perforation produces unacceptable output. For two loops the output distortion is acceptable, but perforation does not significantly improve the performance.

5. CASE STUDIES

We next present two case studies that illustrate how we used the profiling information to guide application optimization while preserving acceptable quality of service.

x264: The top two perforated loops in the x264 profile (see Tables 1 and 2) both occur in the `pixel_satd_wxh()` function. The profiling results indicate that perforating these loops delivers a significant performance improvement with acceptable quality of service loss, which makes the `pixel_satd_wxh()` subcomputation a promising candidate for optimization.

A manual examination of the `pixel_satd_wxh()` function indicates that it implements part of the temporal redundancy computation in x264, which finds similar regions of different frames for motion estimation [12]. The function `pixel_satd_wxh()` takes the difference of two regions of pixels, performs several Hadamard transforms on 4×4 subregions, then computes the sum of the absolute values of the transform coefficients (a Hadamard transform is a frequency transform which has properties similar to the Discrete Cosine Transform).

One obvious alternative to the Hadamard-based approach is to eliminate the Hadamard transforms and simply return the sum of absolute differences between the pixel regions without computing the transform. Replacing the original `pixel_satd_wxh()` function with this simpler implementation delivers a speedup of 1.47 with a quality of service loss of 0.8%. This loss is due to a 1.4 % increase in the size of the encoded video and a .1 dB loss in PSNR. As Table 2 indicates, the automatically perforated version of `pixel_satd_wxh()` delivers speedup of 1.67 with a quality of service loss of 9.67%

— a larger performance improvement than the manually optimized version, but also a larger quality of service loss.

Our next optimization subsamples the sum of absolute differences computation by discarding every other value in each row of the subregion. This optimization produces a speedup of 1.68 and a quality of service loss of 0.4 %. This loss results from a 0.3 dB loss in PSNR with the size of the encoded video remaining the same. Note that this PSNR loss remains well below the accepted 0.5 dB perceptability threshold. These results show that, guided by the profiling information, we were able to identify and develop an optimized alternative to an important x264 subcomputation while preserving acceptable quality of service.

bodytrack: Bodytrack processes video streams from four coordinated cameras to identify and track major body components (torso, head, arms, and legs) of a subject moving through the field of view (see Figure 1). It uses a particle-based Monte-Carlo technique — it randomly samples a collection of pixels (each sampled pixel corresponds to a particle), then processes the samples to identify visual features (such as sharp illumination gradients) and map the visual features to body components. The application itself performs an annealing computation with several phases, or layers, of samples. Each layer uses the image processing results from the previous layer to guide the selection of the particles during its sampling process.

The top loop in the individual loop profile table is the `mainPthreads` loop. But the results show that perforating this loop causes an unacceptable quality of service loss. We therefore focus our optimization efforts on the next loop in the profile (`ParticleFilter::Update`), which can be perforated with significant performance improvement and very little quality of service loss. This loop performs the core of bodytrack's computation in which it randomly samples and processes particles (using the annealed particle filter algorithm) to perform edge and foreground silhouette detection. The loop itself iterates through the annealing layers performing directed particle filtering at each layer.

Guided by the profiling results, we developed an optimization that reduces the number of particles sampled at each layer. Specifically, the optimization reduces the number of sampled particles from 4000 to 2000. This optimization produces a speedup of 1.86 with a quality of service loss of 0.01% — a larger performance increase than the automatically perforated version of this loop with a smaller quality of service loss. Figure 2 presents the output of this optimized version. This figure shows that this optimized version of bodytrack successfully identifies all of the major body components (with the exception of one forearm). Each figure presents four frames, one from each of the cameras.

6. RELATED WORK

Performance Profiling. Profiling a system to understand where it spends its time is an essential component of modern software engineering. Standard profilers simply identify the amount of time spent in each subcomputation [1, 3, 5, 13, 19, 26]. A quality of service profiler, in contrast, adds the extra dimension of providing developers with information about the quality of service implications of changing the implementation of specific subcomputations. This additional information can enhance developer productivity by enabling the developer to focus on promising subcomputations that loop perforation has already shown can be optimized with acceptable quality of service losses while avoiding less promising subcomputations for which one optimization attempt has already failed.

Automatic Generation and Management of Performance versus Quality of Service Trade Offs. We have also used loop perforation to automatically enhance applications with the ability to execute at a variety of different points in the underlying performance versus quality of service trade-off space [14]. We have demonstrated how an application can use this capability to automatically increase its performance or (in combination with an appropriate monitoring and control system) dynamically vary its perforation policy to adapt to clock frequency changes, core failures, load fluctuations, and other disruptive events in the computing substrate [14].

Rinard has developed techniques for automatically deriving empirical probabilistic quality of service and timing models that characterize the trade-off space generated by discarding tasks [21, 22]. Loop perforation operates on applications written in standard languages without the need for the developer to identify task boundaries.

An alternate approach to managing performance/quality of service trade offs enables developers to provide alternate implementations for different pieces of functionality, with the system choosing implementations that are appropriate for a given operating context [6, 7]. Loop perforation, in contrast, automatically identifies appropriately optimizable subcomputations.

Unsound Program Transformations. We note that loop perforation is an instance of an emerging class of unsound program transformations. In contrast to traditional sound transformations (which operate under the restrictive constraint of preserving the semantics of the original program), unsound transformations have the freedom to change the behavior of the program in principled ways. Unsound transformations have been shown to enable applications to productively survive memory errors [23, 8, 24], code injection attacks [23, 20, 25, 24], data structure corruption errors [10, 11], memory leaks [18], and infinite loops [18]. They have also been shown to be effective for automatically improving application performance [21, 22, 14] and parallelizing sequential programs [17]. The success of loop perforation in enabling quality of service profiling provides even more evidence for the value of this class of program transformations.

7. CONCLUSION

To effectively optimize computations with complex performance/quality of service trade offs, developers need tools that can help them locate promising optimization opportunities. Our quality of service profiler uses loop perforation to identify promising subcomputations. Developers can then leverage the generated profiling tables to focus their optimization efforts on optimization targets that have already demonstrated the potential for significant performance improvements with acceptably small quality of service losses.

Experimental results from our set of benchmark applications show that our quality of service profiler can effectively separate promising optimization opportunities from opportunities with less promise. And our case studies provide concrete examples that illustrate how developers can use the profiling information to guide successful efforts to develop new, alternate, and effectively optimized implementations of important subcomputations.

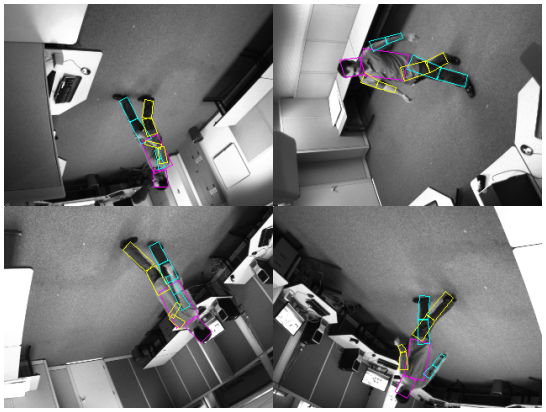

Figure 1: bodytrack reference output.

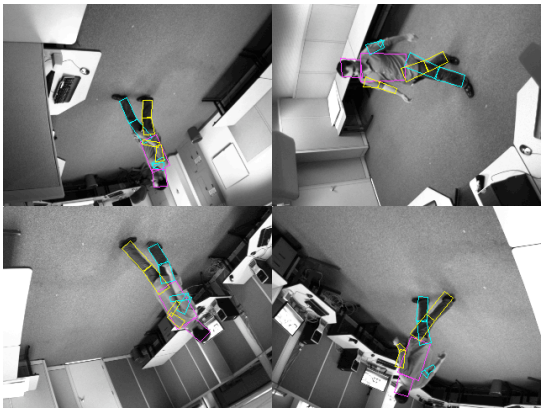

Figure 2: bodytrack output after manual optimization.

8. ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation under Grant Nos. CNS-0509415, CCF-0811397 and IIS-0835652, DARPA under Grant No. FA8750-06-2-0189 and Massachusetts Institute of Technology. We would like to thank Derek Rayside for his input on earlier drafts of this work.

9. REFERENCES

[1] prof. Digital Unix man page.

[2] UCI Machine Learning Repository. http://archive.ics.uci.edu/ml/.

[3] VTune Performance Analyser, Intel Corp.

[4] 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.

[5] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, R. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, 1997.

[6] J. Ansel, C. Chan, Y. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: a language and compiler for algorithmic choice. In *PLDI '09*.

[7] W. Baek and T. Chilimbi. Green: A system for supporting energy-conscious programming using principled approximation. Technical Report TR-2009-089, Microsoft Research, Aug. 2009.

[8] E. Berger and B. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *PLDI*, June 2006.

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

[10] B. Demsky, M. Ernst, P. Guo, S. McCamant, J. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA '06*.

[11] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *ICSE '05*, 2005.

[12] B. Furht, J. Greenberg, and R. Westwater. *Motion Estimation Algorithms for Video Compression*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[13] S. Graham, P. Kessler, and M. Mckusick. Gprof: A call graph execution profiler. In *SCC '82*.

[14] 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.

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

[16] D. Le Gall. MPEG: A video compression standard for multimedia applications. *CACM*, Apr. 1991.

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

[18] H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *ISMM '07*.

[19] G. Pennington and R. Watson. jProf – a JVMPI based profiler, 2000.

[20] J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, , W. Wong, Y. Zibin, M. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP '09*.

[21] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS '06*.

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

[23] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI '04*.

[24] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: automatic software self-healing using rescue points. In *ASPLOS '09*.

[25] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *USENIX '05*.

[26] D. Viswanathan and S. Liang. Java virtual machine profiler interface. *IBM Systems Journal*, 39(1), 2000.