

# Exploring the Effectiveness of Loop Perforation for Quality of Service Profiling

by

Sasa Misailovic

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

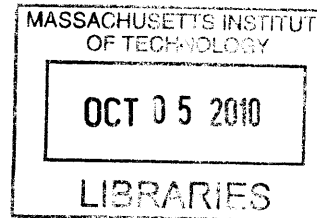
Master of Science in Computer Science and Engineering

at the


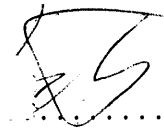
MASSACHUSETTS INSTITUTE OF TECHNOLOGY


September 2010

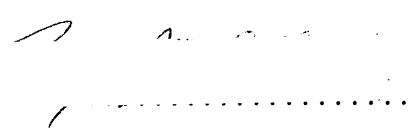
**ARCHIVES**



© Massachusetts Institute of Technology 2010. All rights reserved.

Author .....  .....  .....  
Department of Electrical Engineering and Computer Science  
September 3, 2010

Certified by .....  .....  
Martin C. Rinard  
Professor  
Thesis Supervisor

Accepted by .....  .....  
Terry P. Orlando  
Chairman, Department Committee on Graduate Students



# Exploring the Effectiveness of Loop Perforation for Quality of Service Profiling

by  
Sasa Misailovic

Submitted to the Department of Electrical Engineering and Computer Science  
on September 3, 2010, in partial fulfillment of the  
requirements for the degree of  
Master of Science in Computer Science and Engineering

## 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, 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.

Thesis Supervisor: Martin C. Rinard

Title: Professor

## Acknowledgements

First, I am grateful to my advisor Martin Rinard. It has been a pleasure to work with Martin during the past two years. Martin's enthusiasm and support are a source of great inspiration. I appreciate and absorb, as much as possible, the knowledge that he so often shares.

I am grateful to Hank Hoffmann and Stelios Sidiroglou with whom I worked on loop perforation-related projects. They contributed significantly to this thesis with their ideas, discussions and advice.

I would also like to thank all the members of the PAC CSAIL group, Michael Carbin, Vijay Ganesh, Deokhwan Kim, Jeff Perkins, and Karen Zee for all their individual help and support.

A special thanks to all my friends at MIT: Eunsuk Kang, Neha Gupta, Sachithra Hemachandra, Jacqueline Lee, Aleksandar Milicević, Joseph Near, Derek Rayside, Rishabh Singh, Jean Yang and Kuat Yessenov for making graduate student life an enjoyable experience.

I would like to thank Mary McDavitt and Janet Fischer for their help during the final stages of thesis writing.

Finally, I would like to thank my parents for providing continuous and unconditional love and support throughout my life.

# Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
1.1	Performance Profiling . . . . .	9
1.2	Quality of Service Profiling . . . . .	11
1.3	Contributions . . . . .	14
<b>2</b>	<b>Quality of Service Profiler</b>	<b>15</b>
2.1	Performance Profiling . . . . .	16
2.2	Loop Perforation . . . . .	16
2.2.1	Induction variables . . . . .	17
2.2.2	Sampling Perforation . . . . .	17
2.2.3	Truncation Perforation . . . . .	18
2.2.4	Randomized Perforation . . . . .	19
2.3	Acceptability Model . . . . .	20
2.3.1	Output Abstraction . . . . .	20
2.3.2	Quality of Service Loss Measure . . . . .	21
2.4	Optimization Candidate Selection . . . . .	22
2.4.1	Amenable Computation Identification . . . . .	24
2.4.2	Individual Optimization Candidate Discovery . . . . .	24
2.4.3	Cumulative Optimization Candidate Discovery . . . . .	26
2.5	Profiler Reports . . . . .	27
<b>3</b>	<b>Experimental Evaluation</b>	<b>29</b>
3.1	Benchmark Applications . . . . .	29
3.1.1	x264 . . . . .	30
3.1.2	bodytrack . . . . .	31
3.1.3	swaptions . . . . .	31
3.1.4	blackscholes . . . . .	31
3.1.5	canneal . . . . .	32

3.1.6	Representative Inputs . . . . .	32
3.2	Experimental Setup . . . . .	33
3.3	Loop Perforation Results . . . . .	33
3.3.1	Individual Loop Perforation . . . . .	34
3.3.2	Cumulative Loop Perforation . . . . .	36
3.4	Benchmark Application Results . . . . .	38
3.4.1	Profiling Reports . . . . .	38
3.4.2	Profiling Results for x264 . . . . .	39
3.4.3	Profiling Results for Bodytrack . . . . .	43
3.4.4	Profiling Results for Swaptions . . . . .	46
3.4.5	Profiling Results for Blackscholes . . . . .	48
3.4.6	Profiling Results for Canneal . . . . .	50
3.4.7	Discussion . . . . .	51
<b>4</b>	<b>Related Work</b>	<b>58</b>
4.1	Performance Profiling . . . . .	58
4.2	Performance vs Quality of Service Trade-Offs . . . . .	58
4.3	Trade-Off Management . . . . .	59
<b>5</b>	<b>Conclusion</b>	<b>61</b>

# List of Figures

1-1	Compiler Framework Overview . . . . .	11
2-1	Loop selection algorithm pseudocode . . . . .	23
2-2	perforateLoopSet pseudocode . . . . .	24
2-3	selectLoopSet pseudocode . . . . .	27
3-1	Perforatable Loop Count . . . . .	34
3-2	Performance Comparison of Different Perforation Strategies . . . . .	37

# List of Tables

3.1	Basic Application Statistics . . . . .	30
3.2	Cumulative Perforation Scores . . . . .	36
3.3	Individual Loop Perforation Results for x264 . . . . .	41
3.4	Individual Loop Perforation Results for bodytrack . . . . .	44
3.5	Individual Loop Perforation Results for swaptions (with bias) . . . . .	47
3.7	Individual Loop Perforation Results for canneal . . . . .	49
3.6	Individual Loop Perforation Results for blackscholes . . . . .	49



# Chapter 1

## Introduction

Modern software is designed with multiple objectives in mind. The development of functionally correct software which produces acceptable results is one of the most important goals of software engineering. Besides functional correctness, the software often needs to satisfy various additional requirements aimed at user satisfaction. The set of desired program properties includes usability, performance, responsiveness, energy efficiency, etc. Common engineering practice divides the task of delivering these properties into two phases. In the first phase, the developers focus on the correctness of the computation. In the second phase, known as *optimization*, the developers improve secondary properties, such as performance.

A typical approach to performance optimization is to find and reimplement subcomputations that are good candidates for optimization and that would make the program run faster. But reimplementing computations may require a significant development effort. An appropriate prioritization of optimization candidates helps the developer focus on the most promising candidates, resulting in significant time and effort savings. Selection of the most profitable optimization candidates becomes an essential part of the optimization process.

### 1.1 Performance Profiling

Developers often use *profiling* to help identify good performance optimization candidates. The developer monitors the execution of the program using a profiler tool. A profiler instruments the program, executes the program on one or more representative inputs, and collects and summarizes the execution time of subcomputations. A profiler report typically presents subcomputations ordered by the time they consumed while processing the input. The developer uses this information to prioritize

which subcomputations to reimplement, often starting with the most time-consuming subcomputations.

Many traditional applications such as compilers, databases, payroll or account management applications have strict correctness requirements — for any input there is exactly one correct result. The standard approach to optimizing these applications is to produce alternative (ideally faster) subcomputations, that always produce the same result as the original subcomputation. The performance profiler report often serves as a good starting point for identifying which subcomputations to optimize in such applications.

However, there exists an emerging class of applications that can produce multiple different acceptable results. Examples of applications from this class include programs that process media formats (audio, video or images), machine learning applications, heuristic searches, scientific simulations and financial simulations. Some of the results produced by such applications may be more accurate, precise or subjectively desirable than the others. These properties of the result are captured by the *Quality of Service* (QoS) measure. In this thesis Quality of Service refers to the accuracy or quality of the result that the application produces, not the timing with which it produces or delivers this result.

Adopting the more flexible notion of QoS instead of requiring the optimized computation to produce the identical result allows for a much broader range of potential optimizations. Developers have the option of implementing a range of subcomputations that can operate with different accuracy and performance characteristics. In many instances a developer may be happy to trade a small amount of QoS for considerable performance gains.

The inherent differences between the trade offs offered by various subcomputations complicate the choice of which computation to attempt to optimize — the developer needs to select the computation that can offer the best opportunity to make a profitable trade off. Traditional profilers provide only part of the relevant information in this scenario — they can only identify where the program spends the time, but not tell which subcomputations are the best optimization candidates. In addition to traditional profiling, a new profiling approach that can identify subcomputations which are the best candidates for trading off Quality of Service for performance would make optimization easier.

## 1.2 Quality of Service Profiling

In this thesis we present *Quality of Service Profiling*, a novel profiling technique for detecting subcomputations that may offer profitable trade offs between performance and Quality of Service. These computations are typically good optimization candidates. Quality of Service Profiling explores the space of potential optimization candidates by automatically generating and evaluating potential optimized subcomputations.

Figure 1-1 presents an overview of our Quality of Service profiler algorithm. Given the source code of a program and a representative input, the profiler first performs performance profiling to find the time-consuming subcomputations. The profiler also collects the execution results of the original, unmodified program.

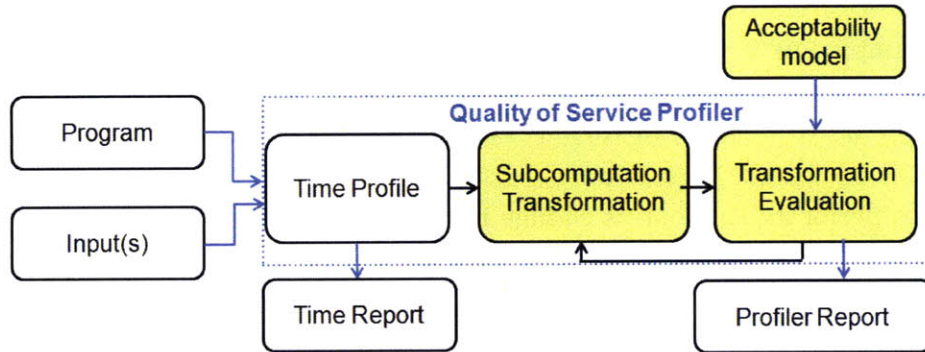


Figure 1-1: Compiler Framework Overview

In the following steps, the Quality of Service profiler applies predefined transformations on the original subcomputations to generate alternative versions of individual time-consuming subcomputations. The goal of the transformations is to cause the transformed subcomputations to execute less work. The transformed computation is allowed to generate a result that is different from the result of the original computation as long as it still produces an acceptable result. In each step the profiler applies a single transformation. The profiler evaluates the effects of the transformation on the performance and QoS by executing the modified program on the representative inputs and comparing with the results of the original program. In addition to the evaluation of individual subcomputations, the profiler also evaluates groups of subcomputations to find potential synergies between the subcomputations.

The Quality of Service Profiler uses a developer-specified *acceptability model* to extract and summarize parts of the program output relevant to the QoS, calculate the difference between the result summaries of the original and the transformed program

to determine the level of *Quality of Service loss* (QoS loss), and specify the greatest acceptable level of QoS loss. The profiler considers the computations that, when transformed, produce results with an acceptable QoS loss to be possible optimization candidates. The computations whose transformations cause unacceptable results or unexpected program termination are considered critical subcomputations.

## Loop Perforation

The key element of Quality of Service Profiling is the choice of transformations used to create alternative subcomputations. In this thesis we evaluate the effectiveness of a class of transformations called *loop perforation*, which applies to `for`-style loops. Loop perforation transforms amenable loops to execute fewer iterations, by changing the rate of the loop increment value or the exit conditions.

For example, given the following loop:

```
for( i = 0; i < max; i+=1 ) { /* ... */ }
```

one way to perforate the loop is change the value of the increment to force the loop execute only half of the iterations:

```
for( i = 0; i < max; i+=2 ) { /* ... */ }
```

Loop perforation comprises multiple variants. In this thesis we compare three variants of loop perforation. The first variant, called *sampling perforation*, skips 1 out of every  $k$  iterations (as in previous example with  $k = 2$ ). The second variant, called *truncation perforation*, skips  $k$  iterations at the beginning or at the end of the loop. The third variant, called *randomized perforation*, randomly skips  $k$  iterations.

## Rationale

The rationale for transformation-based approach to Quality of Service profiling is that good optimization candidates perform *partially redundant* computations. The redundancy manifests itself in the form of additional computation, not proportional to the increase of the quality of the result. Removing parts of such subcomputations will typically reduce the work performed by the subcomputation and increase the performance of the application. Quality of Service Profiling is based on the following observation — if the original subcomputation performs redundant work, then the transformed subcomputation typically produces a result that is close to the original

result, but will require less time than the original subcomputation. The evaluation results that we present support the well-foundedness of this observation.

The rationale for using loop perforation is that many partially redundant computations are implemented as loops, where the iterations of the loop have different contribution to the result. The evaluation results provide evidence that loop perforation can be an effective transformation that for finding partially redundant computations.

## Implementation

We implemented a prototype Quality of Service profiler, *Qosprof*. *Qosprof* uses loop perforation to find good optimization candidates. We perform an experimental evaluation of the effectiveness of *Qosprof* to find optimization candidates on five applications from the PARSEC benchmark suite [6].

Our experimental results indicate that loop perforation can successfully transform a number of partially redundant subcomputations to produce new subcomputations with improved performance and small QoS losses. An inspection of the source code of the application shows that most of the computations that the profiler identifies are in fact good optimization candidates.

## Reports

The Quality of Service Profiler generates the reports based on the execution of the profiling algorithm. The developer can use these reports to order the subcomputations and focus his or her effort on the most promising subcomputations first. The Quality of Service Profiler produces two reports based on the evaluation of automatically transformed programs:

1. Individual transformation report — contains the performance increase and quality of service loss results of subcomputations that were transformed individually. The developer can use this report to find the subcomputations that offer the most profitable performance/QoS trade offs.
2. Cumulative transformation report — contains the performance increase and quality of service loss results of multiple subcomputations transformed together. The developer can use this report to find potential synergies between the subcomputations that are good optimization candidates.

## 1.3 Contributions

This thesis makes the following contributions:

- **Quality of Service Profiling:** This thesis proposes a technique for exploring a space of optimization candidates for applications that profit from trade offs between performance and quality of service.
- **Transformation:** This thesis proposes loop perforation as a transformation that can be used by Quality of Service Profiling to identify subcomputations that offer profitable trade offs between performance and QoS.
- **Evaluation:** The thesis presents an experimental evaluation of the effectiveness of different loop perforation variants in automatically generating subcomputations with desired properties. We analyze the perforated computations, summarize the results and provide reasons why loop perforation works.

Chapter 2 gives an overview of the profiler design, presents the details of loop perforation, describes the algorithms for the optimization candidate search, and describes the generated reports. Chapter 3 provides an experimental evaluation. Chapter 4 discusses the related work.

# Chapter 2

## Quality of Service Profiler

Quality of Service profiling explores the space of subcomputations that can be optimized by automatically generating potential program optimizations. As a first step, the Quality of Service Profiler performs standard time profiling to find time-consuming subcomputations. While it can in principle work with any existing profiler, we implemented our own profiler that identifies the loops that perform most of the work. We describe the profiler in Section 2.1.

In the following steps the profiler uses a set of predefined transformations to automatically generate alternative versions of the original subcomputations. This thesis explores *loop perforation*, a class of applicable computation transformations. We describe the details of the loop perforation transformations in Section 2.2. To compare the effects of program transformations on the final result, the developer specifies the procedure to extract and summarize the information relevant to QoS from a program output and a procedure to compare two QoS values obtained from execution of different versions of the program on the same input as parts of *acceptability model*. We describe the details of acceptability model in Section 2.3.

Given the application source code, a set of representative inputs, and acceptability model, the Quality of Service Profiler performs a set of steps that insert instrumentation to perform program profiling, identify loops that are good optimization candidates, and select of loops that, when perforated together, deliver a result within QoS bounds. We describe the steps of the profiling algorithm in Section 2.4. From the information gathered during these steps, the profiler generates a profiling report described in Section 2.5.

## 2.1 Performance Profiling

The Qosprof performance profiler produces an instrumented version of the original program that, when it executes, counts the number of times each basic block executes. The instrumentation also maintains a stack of active nested loops and counts the number of (LLVM bit code) instructions executed in each loop, propagating the instruction counts up the stack of active nested loops so that outermost loops are credited with instructions executed in nested loops.

The performance profiler produces two outputs. The first is a count of the number of instructions executed in each loop during the loop profiling execution. The second is a directed graph that captures the dynamic nesting relationships between different loops (note that the loops may potentially be in different procedures).

## 2.2 Loop Perforation

Given a loop to perforate, our loop perforation transformation takes as input a percentage of iterations to skip during the execution of the loop and a perforation strategy. Loop perforation can be applied to the loops that behave as `for` loops. These loops have an *induction variable*, whose value is changed linearly in each iterations. A transformation pass modifies the calculation of the loop exit condition (including the induction variable) to manipulate the number of iterations that a loop executes. The pass conceptually performs the following loop transformation:

```
for( i = 0; i < max; i+=1 ) { /* ... */ }
```

to

```
for( i = 0; i < max; i+=1 ) {  
    if (doPerforate(i, environment)) continue;  
    //...  
}
```

The percentage of non-executed iterations is called the *perforation rate* ( $pr$ ). Depending on the selected perforation rate a different performance/QoS loss trade-off can be made. For example for a perforation rate  $pr = 0.5$ , half of the iterations are skipped, for  $pr = 0.25$ , one quarter of the iterations are skipped, while for  $pr = 0.75$ ,



three quarters of the iterations are skipped, i.e. only one quarter of the initial work is carried out.

The compiler supports a range of perforation options, including sampling, or modulo, perforation (which skips or executes every  $n$ th iteration), truncation perforation (which skips either an initial or final block of iterations), and random perforation (which skips randomly selected iterations at a mean given rate). The actual generated code exploits the characteristics of each specific loop perforation option to generate optimized code for that option.

### 2.2.1 Induction variables

Qosprof perforation operations manipulate loops whose induction variables are in canonical form [17]. Qosprof uses the LLVM built-in pass `loopinfo` to identify loops. Before perforating the loop, Qosprof uses the built-in LLVM passes `loopsimplify` and `indvars` to canonicalizes the loop induction variable. As a consequence of these initial bitcode transformations, it is possible to successfully perforate the loops that have different increment values, or reversed minimum and maximum value. Moreover, some of the syntactic `while` loops can be perforated if bitcode analysis finds that they are controlled by an induction variable. After the transformations the induction variable `i` has an initial value of 0 and is incremented by 1 in every iteration until `maxvalue` is reached:

```
for ( i = 0; i < maxvalue; i++ ){ /* ... */ }
```

### 2.2.2 Sampling Perforation

Sampling perforation skips every  $n$ -th iteration, *or* executes every  $n$ -th iteration. The percentage of skipped iterations is determined by the perforation rate,  $pr$ , which is determined using the following formula:

$$pr = \begin{cases} \frac{1}{n} & \text{if every } n\text{-th iteration is skipped} \\ 1 - \frac{1}{n} & \text{if every } n\text{-th iteration is executed} \end{cases}$$

The implementation of sampling perforation considers three cases: 1) large, 2) small, and 3) small where  $n$  is a power of 2. In the following paragraphs, the implementation for the case when  $pr \geq 0.5$  will be referred to as *large* perforation, while the case when  $pr < 0.5$  will be referred to as *small* perforation. Additionally, for

small perforation, if  $n$  is power of 2, a more efficient implementation is available for some computer architectures. The following examples describe each transformation.

**Large Perforation:** For *large* perforation the increment of the induction variable is changed from 1 to  $n$ :

```
for (i = 0; i < maxvalue; i += n) { /* ... */ }
```

**Small Perforation:** *small* perforation is implemented by adding a new term to the induction variable increment. The goal is to increment the value of the induction variable by 2 when the iteration is to be skipped. The value of the induction variable is incremented by 2 if the remainder of  $i$  divided by  $n$  is equal to some constant value  $k$ ,  $0 \leq k < n$ :

```
for (i = 0; i < maxvalue;
      i = i + 1 + ( i % n == k ? 1:0 ) ) {
    //.....
}
```

**Small Perforation when  $n$  is Power of 2:** When  $n$  is power of 2 ( $n = 2^m$ ), small perforation uses faster bitwise *and* operations to calculate the remainder of  $i$  divided by  $n$ , which in this case are lowest  $m$  bits of  $i$ :

```
for (i = 0; i < maxvalue;
      i = i + 1 + ( i & (n-1) == k ? 1:0 ) ) {
    //.....
}
```

### 2.2.3 Truncation Perforation

Truncation perforation skips iterations at the beginning or at the end of the loop execution. The iteration count of the perforated loop is equal to  $(1 - pr) \cdot maxvalue$ . Discarding iterations at the beginning of the loop involves initialization of the induction variable  $i$  to  $i = pr * maxvalue$  where  $maxvalue$  is known before the loop invocation and is not changed during the loop's execution. The example of the loop is:

```
for (i = pr * maxvalue; i < maxvalue; i++) { /* ... */ }
```

Perforating iterations at the end of the loop accomplishes an earlier exit from the loop. This perforation is implemented by decreasing the loop condition bound. The new condition becomes `i < (1 - pr) * maxvalue`:

```
for (i = 0; i < (1 - pr) * maxvalue; i++) { /* ... */ }
```

If `maxvalue` is not modified from within the loop body, the new condition can be precomputed. Otherwise, it needs to be checked in every iteration. Instead of performing floating point multiplication, which may be expensive on some architectures, it is possible to represent the rational number  $1 - pr$  as a fraction of the form  $p/q$ , where  $p$  and  $q$  are natural numbers. Then, the condition can be represented as `q * i < p * maxvalue`. If  $p$  or  $q$  are powers of 2, shifting may be used instead of multiplication.

## 2.2.4 Randomized Perforation

Randomized loop perforation skips individual iterations at random, based on a user-specified distribution with mean  $pr$ :

```
for( i = 0; i < maxvalue; i++ ) {  
    if (skipIteration(i, pr)) continue;  
    //...  
}
```

This type of perforation is the most flexible, but introduces the greatest overhead. It allows the runtime to dynamically control the perforation during the execution of the loop body and change the underlying perforation distribution in the course of loop execution. The decision to skip an iteration is always made at the run-time, unlike the previous perforation types.

The implementation of the `skipIteration()` function can be arbitrary complex, but the function may become a performance bottleneck, due to its frequent execution. We provide a simple implementation of the `skipIteration()` function that generates a pseudo-random number from an uniform distribution using standard library function `rand()`. It is preferable to apply this technique on loops that perform more work per iteration or perform a larger number of iterations, which can offset the overhead of the call. The call to `skipIteration()` is, in most cases, inlined by the compiler to reduce the call overhead.

## 2.3 Acceptability Model

To measure the effect of loop perforation, Qosprof requires the user to provide an acceptability model for the program output. This model consists of three components: a) an output abstraction — a procedure for extracting or summarizing relevant parts of the output, b) a Quality of Service loss measure – a procedure for calculating the difference between QoS values of two program executions, and c) a bound on the maximum QoS loss that the user is willing to tolerate.

### 2.3.1 Output Abstraction

The output abstraction maps the result of the program execution to a numerical value or tuple of values that represent relevant parts of the execution. In addition to the parts of the concrete output, the output abstraction may also use information from the input or the environment. Formally, output abstraction accepts three inputs — concrete output, the input, and relevant parts of the environment. The output abstraction produces a tuple  $(o_1, \dots, o_m)$ , where each component  $o_i$  is a numerical value.

The form of the output abstraction is strongly tied to the program domain. In many cases defining output abstractions is an intuitive process. For many application domains there exists an extensive body of work for evaluating the quality of the result and often the important parts of the result are directly available. The procedures for selecting and expressing the quality of the output often exist already as parts of the software testing framework. For example, a financial simulation may produce a prediction of prices of the securities. The price of each security represents the the value of interest for quality of service that the application provides. As an another example, the quality of an encoded video can be measured by considering the accuracy of the encoding, using e.g. peak signal-to-noise ratio and measuring the size of the encoded video.

Our experience with the benchmark applications used to evaluate the profiler was that creating a program output abstraction is a straightforward process for users with basic knowledge of an application. Without prior knowledge of the PARSEC benchmark applications, we were able to produce output abstractions for each examined application in a short time. We describe the output abstractions for the set of benchmark applications in Section 3.1.

### 2.3.2 Quality of Service Loss Measure

The Quality of Service loss (QoS loss) measures the effect of the program transformation on the program outputs, with respect to the original program. A small value of QoS loss indicates that the transformed program produced a result that is similar to the original program, while large QoS loss indicates a dissimilarity between the two results. We compare two program transformations by calculating the QoS losses of the two transformed programs on the same input with respect to the (same) original program. Additionally, QoS losses obtained from the execution of the same transformed program with respect to the original program on different inputs should be comparable. This property makes it possible to compare the effects of a single transformation on both smaller and larger inputs.

In this thesis we calculate QoS loss based on the relative scaled difference between selected outputs from the original and perforated executions. Given an output (concrete, or results of output abstraction function)  $o_1, \dots, o_m$  from an unmodified execution and an output  $\hat{o}_1, \dots, \hat{o}_m$  from a perforated execution, the following quantity  $d$ , which we call the *distortion*, 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. Because each difference is scaled by the corresponding output component, distortions from different executions and inputs can be compared. By default the distortion equation weights each component equally, but it is possible to modify the equation to weigh some components more heavily. Also note that because of the scaling of individual components, it is possible to meaningfully compare the distortions obtained from executions on different inputs. More detailed discussion on distortion is available in [20].

#### Bias Definition and Use

The distortion measures the absolute error induced by loop perforation. It is also sometimes useful to consider whether there is any systematic direction to the error. To measure any systematic error introduced through loop perforation we use the *bias* [20] metric:

$$b = \frac{1}{m} \sum_{i=1}^m \frac{o_i - \hat{o}_i}{o_i}$$

Note that this is the same formula as the distortion with the exception that it preserves the sign of the summands. Errors with different signs may therefore cancel each other out in the computation of the bias instead of accumulating as for the distortion.

If there is a systematic bias, it may be possible to compensate for the bias to obtain a more accurate result. Consider, for example, the special case of a program with a single output component  $o$ . If we know that bias at a certain is  $b$ , we can simply divide the observed output  $\hat{o}$  by  $(1 - b)$  to obtain an estimate of the correct output whose expected distortion is 0.

## 2.4 Optimization Candidate Selection

The goal of the optimization candidate selection algorithm is to find the set of sub-computations that can be transformed to produce significant performance increases at the cost of the QoS losses which are within the acceptable bound. The algorithm transforms each subcomputation, executes the transformed program, collects the execution time, uses the acceptability model to calculate the QoS loss.

The algorithm consists of two parts, individual and cumulative optimization candidate selection. Figure 2-1 presents the pseudocode of the selection algorithm. The algorithm performs the following steps:

- **Identification.** It identifies the subcomputations (loops) on which loop perforation can be applied (Section 2.4.1).
- **Individual computation discovery.** Discovery of optimization candidates by transforming individual subcomputations and measuring performance and QoS loss for each candidate loop (Section 2.4.2).
- **Cumulative computation discovery.** The discovery of the set of loops that maximizes performance for a specified Quality of Service loss bound (Section 2.4.3).

In the current implementation of Qosprof, the user selects the perforation strategy and the perforation rate before the search. The algorithm works with one transfor-

```

LoopSelection (program, inputs, maxLoss)
  candidateLoops = {}
  scores = {}

  for i in inputs
    candidateLoops[i] = performProfiling(program, i)
    for each l in candidateLoops[i]
      scores[i][l] = assignInitialLoopScore(l)
    filterProfiledLoops(candidateLoops[i])

    for l in candidateLoops
      spdup, QoSloss = perforateLoopSet(program, {l}, i)
      analysisRes = performAnalysis(program, {l}, i)
      scores[i][l] = updateScore(spdup, QoSloss, analysisRes)
    filterSingleExampleLoops(candidateLoops[i], scores[i], maxLoss)

  candidateLoops, scores =
    mergeLoops(candidateLoops[*], scores[*])

  if size(candidateLoops) == 0
    return {}

  candidateLoopSets = {}
  for i in inputs
    candidateLoopSets[i] =
      selectLoopSet(program, candidateLoops, scores, i, maxLoss)

  return loopsToPerforate

```

Figure 2-1: Loop selection algorithm pseudocode

mation at a time. It is straightforward to extend the algorithm to perform multiple transformations on multiple types of subcomputations in a single run.

### 2.4.1 Amenable Computation Identification

Initially, all loops are candidates for perforation. The algorithm invokes the profile-instrumented program, described in Section 2.1 on all profiling inputs to find candidate loops for perforation. The algorithm assigns a score for each loop. The score is proportional to the amount of cumulative work that the loop performs. The algorithm removes from the list the loops that have only a minor contribution to the work that the program executes, an unsatisfactory number of iterations/invocations, or that cannot be instrumented.

### 2.4.2 Individual Optimization Candidate Discovery

The algorithm perforates each candidate loop in isolation and observes the effect of the perforation on the speedup and quality of service. The loop is statically perforated with a predefined perforation rate. Figure 2-2 shows the pseudocode of the algorithm. After the execution of the instrumented program Qosprof calculates the quality of service loss.

```
perforateLoopSet(program, loopSet, input)
    program' = instrumentLoops(loopSets)

    time, output = execute(program, input)
    time', output' = execute(program', input)

    abstrOut = abstractOutput(output)
    abstrOut = abstractOutput(output')

    speedup = calculateSpeedup(time, time')
    QoSloss = calculateQoSloss(abstrOut, abstrOut')

    return speedup, QoSloss
```

Figure 2-2: perforateLoopSet pseudocode

The score for the loops contains the measured speedup and quality of service loss from the performed executions. Based on the score, the effects of the loop perforation can be grouped into following categories:



- **Unexpected Termination** – perforating the loop caused the program to crash (e.g. due to segmentation fault) or hang during execution.
- **High QoS loss** – perforating the loop caused the program to produce a result with high QoS loss.
- **Small performance gains** – perforating the loop caused the program to execute more slowly.
- **Latent errors** – perforating the loop produced an acceptable result, but the dynamic memory error detector revealed latent errors in program state that could cause the program to crash.
- **Perforatable** – executing the program with perforated loop produced an acceptable result, faster than the original program; perforation did not introduce latent errors.

A program with perforated loops may terminate unexpectedly or hang during loop evaluation. In these cases, the Quality of Service loss is set to 1.0, disqualifying the loop from further consideration. If the program is not responsive for a time greater than the execution of the reference version, it is terminated, and the speedup set to 0, also disqualifying the loop. Loops that do not increase the performance and loops that cause Quality of Service loss greater than the maximum bound specified by the user are also removed from the candidates list.

The profiler performs an analysis of the program to ensure that loop perforation does not cause latent memory errors. We use Valgrind Memcheck dynamic analyzer [18] to analyze the program for latent memory errors.

The algorithm can perform profiling on one or multiple inputs in parallel. If multiple inputs are tried in parallel, Qosprof merges the results for individual loops from all inputs. The loop scores of perforatable loops are averaged over all inputs. The loops that are not perforatable for at least one input are considered non-perforatable.

## Ordering Perforated Loops

After the individual loop profiling, the profiler orders the loop according to the trade-off between the performance gains and the quality of service loss. The developer may have different optimization objectives, according to which he or she would like the results to be ordered. For example, the developer may prefer the highest performance gains given that they satisfy the QoS bound. On the other extreme, the developer may

prefer the computations that produce smallest QoS loss, despite small performance gains.

The developer may also choose to show the computations that present a balance between the performance and QoS loss. The developer can select the function that would average the individual trade-offs. Common average functions include arithmetic mean ( $\frac{a+b}{2}$ ), geometric mean ( $\sqrt{ab}$ ) and harmonic mean ( $\frac{2ab}{a+b}$ ). Additionally, the user may choose a weighted version of the means to perform a fine-grain ordering. These functions operate on the terms derived from the performance  $p$  and the quality of service loss  $q$  — the performance increase, calculated as  $a = p - 1$ , and the result similarity, calculated as  $b = 1 - \frac{q}{q_{max}}$  ( $q_{max}$  is maximum acceptable QoS loss). If the value of either of these terms is below or equal 0, the average is not calculated, but the total score is set to 0, indicating an unacceptable result.

The means are different in the way they treat small performance gains and quality of service losses close to the maximum loss bound. Harmonic and to some extent geometric mean additionally penalize these trade-offs, unlike arithmetic mean. Harmonic mean may give a small score to the computation that delivers high performance, but also high QoS loss, favoring instead the computations that provide balanced trade-offs. Arithmetic mean, on the other hand will rate each trade-off uniformly and will not penalize borderline trade-offs.

Quality of Service Profiler allows the user to choose ahead of time the strategy for the computation trade-off ordering. The ordering of computations is also important for the cumulative loop profiling, described in Section 2.4.3.

### 2.4.3 Cumulative Optimization Candidate Discovery

The next step is to combine loops with high individual perforation scores on all profiling inputs and observe their joint influence on program execution. Note that the QoS losses and speedups of programs with multiple perforated loops may not be linear in terms of the individual perforated loop results (because of potentially complex interactions between loops). This step is executed separately for each input. Figure 2-3 presents the pseudocode for cumulative loop selection.

The algorithm maintains a set of loops that can be perforated without exceeding the maximum acceptable Quality of Service loss bound (`maxLoss`) selected by the user. The algorithm orders loop according to the developer-provided scoring function described in Section 2.4.2. In this thesis we use the harmonic mean between the performance gain and the distortion to order the individual loops. At each step, the

algorithm tries the loop with the highest individual score, and executes the program where all the loops from the set and the new loop are perforated. If the performance increases, and the Quality of Service loss is smaller than the maximum allowable, the loop is added to the set of perforated loops.

```
selectLoopSet(program, candidateLoops, scores, input, maxLoss)

    loopQueue = sortLoopsByScore(candidateLoops, scores)

    LoopSet = {}
    cummulativeSpeedup = 1
    while loopQueue is not empty
        tryLoop = loopQueue.remove()
        trySet = LoopSet U {tryLoop}
        speedup, QoSloss = runPerforation(trySet, input)

        if speedup > cummulativeSpeedup and QoSloss < maxLoss
            loopSet = trySet
            cummulativeSpeedup = speedup

    return LoopSet
```

Figure 2-3: selectLoopSet pseudocode

## 2.5 Profiler Reports

Qosprof produces a profiling report that summarizes the information obtained during the profiler algorithm run. The report contains two sections — individual transformation results and cumulative transformation results. The results are gathered during the executions of profiling algorithms from Sections 2.4.2 and 2.4.3.

The individual Quality of Service Profiling report presents the results of transforming individual subcomputations. The developer can use this information to select the individual subcomputations that show the greatest potential for optimization. For each transformed loop the report presents the source code identifier (function name, source code file name and line number), the amount of work that the computation

performs, as reported by the performance profiling phase, and the effect of the transformation on the program's execution time and the QoS — the report shows the speedup over the sequential version and the QoS loss. The report highlights loops that are good optimization candidates.

The cumulative Quality of Service Profiling report helps the developer identify positive or negative interactions between loops that are transformed together. For example, if the loops are nested, the profiling results show the level of potential optimization of subcomputation – the ability to have a larger part of a subcomputation discarded with small QoS penalty makes the subcomputation potentially more profitable optimization target. On the other hand, if the perforated loops belong to different loop nests, the profiling results identify the existence of a positive interaction between the subcomputations, which would lower the QoS loss or greater speedup, or a negative interaction between the subcomputations which would cause the program to produce unacceptable results or execute slower.

The profiling report for cumulative profiling contains the source code identifiers of transformed computations, the effect of adding the loop to the set of already perforated loops – the corresponding performance and QoS loss results.

# Chapter 3

## Experimental Evaluation

This chapter provides experimental evidence that loop perforation can generate alternative subcomputations that execute faster than the original subcomputations while still producing acceptable results. We empirically observe that many loops that generate acceptable results with loop perforation correspond to good optimization candidates that trade off QoS for performance, making loop perforation a suitable transformation for Quality of Service profiling. We perform experiments with five benchmarks from the PARSEC benchmark suite which we describe in Section 3.1. We describe the details of the experimental setup in Section 3.2.

Section 3.3 contains the results of the experimental evaluation. Sections 3.3.1 and 3.3.2 present an overview of the individual and the cumulative loop perforation results. We explain the reasons why loop perforation produces the observed results for our benchmark applications in Section 3.4. We provide the summary and the conclusion of the experimental results in Section 3.4.7.

### 3.1 Benchmark Applications

We evaluate the effectiveness of loop perforation using five applications from the PARSEC suite of benchmark applications [6]. The PARSEC suite contains applications from diverse areas, which, together with the provided inputs, represent emerging workloads for the modern multicore processors.

Table 3.1 presents summary and size of each application. The second column (LOC) shows the number of lines of code for each application. The third column (Loops) shows the number of loops that exist in each of the applications, while the fourth column (for loops) shows the number of loops that have induction variables,

Benchmark	LOC	Loops	for loops	Language	Domain
x264	31527	884	700	C	Video Encoding
bodytrack	6709	377	358	C++	Machine Vision
swaptions	1568	97	90	C++	Financial Analysis
blackscholes	289	8	2	C	Financial Analysis
canneal	2431	76	42	C++	Engineering

Table 3.1: Basic Application Statistics

i.e. the loops that are potential optimization candidates. The fifth column (Language) shows the language which was used to implement the application. The sixth column (Domain) presents the domain to which the application belongs.

All applications, except blackscholes, have moderate size, over one thousand lines of code. The largest application, x264, has over 30 000 lines of code. The number of for-style loops indicates that the loop perforation is suitable transformation for a large number of subcomputations. Note that the number of loops reported in this table is somewhat larger than the number of loops in the source code since LLVM counts the loops belonging to the C++ standard template library. In addition, x264 defines macros containing loops that are expanded during the compilation.

### 3.1.1 x264

x264 is a lossy video encoder for the popular H.264 standard. It takes as input a raw video stream and generates a compressed video file. The encoder applies various heuristic algorithms to find parts of the video stream that can be best compressed, such that the final video has acceptable good quality and a small size. This application performs a heuristic computation over large set of data, which makes it appropriate for Quality of Service profiling.

The output abstraction for this application includes a) the quality of the encoded video, numerically represented by the *peak signal to noise ratio* (PSNR), a standard measure of objective video quality [14], and b) the size of the encoded video. The PSNR value is calculated from the original unencoded video and the encoded video produced by the application. In this experiment we weight both parameters equally but can easily support alternative configurations, which would place higher emphasis on file size or video quality.

### 3.1.2 bodytrack

Bodytrack is a machine vision application. It identifies a human body in an image and traces the parts of the body across the stream of images captured by a number of security cameras. Bodytrack uses an annealed particle filter, which consists of an on-line Markov Chain Monte Carlo (MCMC) simulation, combined with simulated annealing to discover the parts of the body on the screen. The heuristic nature of the computation, together with the potentially large volume of data that this application processes makes it an appropriate benchmark for this experiment.

The output abstraction for this application includes the final positions of the vectors of body parts. The application outputs the values of vectors as a textual file.

### 3.1.3 swaptions

Swaptions is a financial application that calculates the pricing of swaption financial instruments<sup>1</sup>. Swaptions uses Monte Carlo simulation within the Heath-Jarrow-Morton framework to model the financial market. The approximate computation that this application performs makes it an appropriate benchmark for this experiment.

We use the final price of the individual swaptions as the output abstraction. The application prints the prices of swaptions to the standard output.

### 3.1.4 blackscholes

Blackscholes is a financial application that calculates the prices of European-style stock options<sup>2</sup>. The value of stock options is calculated as a numerical solution of partial differential equations, belonging to the Black-Scholes method for modeling the financial market. The approximate computation of option prices makes the benchmark appropriate for this experiment.

We use the final price of the individual options as the output abstraction. We augmented the applications to print the list of final prices to the textual file.

---

<sup>1</sup>A swaption is a financial instrument granting its owner the right to exchange other financial instruments, such as bonds, with an other party

<sup>2</sup>A European-style option is a financial instrument, that gives its owner either the ability to buy or sell an asset, which can be exercised only at one specific date

### 3.1.5 canneal

Canneal is an engineering application that calculates the optimal placement of the logic gates and minimal routing of wiring on an electronic chip. Canneal uses simulated annealing as a heuristic mechanism for the discovery of the optimal placement. In each step the calculation randomly exchanges the positions of the logic gates and evaluates the modified chip based on the total length of wires between the logic gates. The fact that the computation uses heuristic search makes this benchmark appropriate for the experiment.

We use the price of routing as the output abstraction. The application calculates and prints this value to the standard output.

### 3.1.6 Representative Inputs

The benchmark applications from the PARSEC suite include the representative inputs. Typically, for each benchmark, the benchmark developers provided four inputs. The inputs are chosen such that they require a different amount of time to process. We use larger versions of the inputs to reduce errors attributed to execution noise. These inputs are denoted as `simlarge` and `native` by the benchmark developers.

For some applications we augmented or replaced inputs from the benchmark suite with additional inputs to improve the representativeness of the inputs. The modification to the original working sets include the following:

1. The benchmark inputs for `x264` represent the same animated movie in different resolutions. Instead, we used 2 high-definition videos (the same resolution as the `native` input) from Xiph.org foundation<sup>3</sup>. We use 20 frames from each video.
2. The `simlarge` input for `bodytrack` is a shortened version of the `native` data set. We used another sequence provided to us by the benchmark developers. We also changed the length of image sequences: while the `simlarge` input has 4 frames, and `native` has 260 frames, the new input consists of the first 60 frames of the `native` input. The additional input contains 100 frames.
3. All financial options in the `swaptions` benchmark had the same initial and final price. We instead changed the initial prices of the swaptions to ensure that

---

<sup>3</sup><http://media.xiph.org/video/derf/>



each swaption has an unique price. For the first input, we used the `simlarge` input, which calculates 64 options. For the second input we used the input that calculates 128 options, as in the `native` input but with a number of Monte Carlo simulations similar to the `simlarge`.

## 3.2 Experimental Setup

We performed the experiments reported in this thesis on a dual 4-core Intel Xeon E5520 workstation, running Ubuntu Linux 10.04, with Linux kernel version 2.6.32. We use LLVM version 2.7 to compile programs. We use Valgrind version 3.6 for the dynamic analysis of the perforated programs.

We perform the Quality of Service profiling experiments independently on two inputs from each application. We set the maximum acceptable Quality of Service loss to 0.1, which typically corresponds to 10% difference between the outputs from the original and the perforated programs.

We perform Quality of Service profiling with the three perforation strategies described in Chapter 2: a) sampling perforation, which skips every  $k$ -th loop iteration, b) truncation perforation, which skips iterations at the end of the loop, and c) randomized perforation, which makes a dynamic decision to skip random iterations of the loop. For each strategy we set the perforation rate to 0.5. In the rest of the thesis we will refer to the combination of the benchmark, input and the perforation strategy as the experimental *configuration*.

During Quality of Service profiling, we considered only loops that execute more than 1% of the total number of instructions, as these loops are most likely to have a significant effect on the execution time. We compiled the reference version of the program and each perforated version of the program with standard optimizations at level `-O3`. For each input Qosprof executes every program 3 times and averages the execution times. We did not observe a major variance between the execution times of the same configurations. We calculated the speedup as the mean execution time of the original program divided by the mean execution time of the perforated program.

## 3.3 Loop Perforation Results

This section presents the results of the experiments. We evaluate three properties of the loop perforation variants:

- **Transformation potential** — the number and the influence of the loops that loop perforation is able to perforate. Table 3.1 presented the number of `for`-type loops as the first indication that loop perforation can transform a large number of subcomputations in the benchmark applications. The experimental results in Section 3.3.1 show that experimental results that each benchmark has perforatable loops.
- **Performance improvement potential** — the amount of performance increase when perforating subcomputations. In Section 3.3.2 we use the cumulative perforation results to characterize the amount of performance that can be gained by performing QoS-related optimizations on the benchmark programs.
- **Optimization candidates** — the correlation between perforatable loops and heuristic computations in the benchmark applications. In Section 3.4 we provide a detailed description of the benchmark application results, and relate the effect of loop perforation to the semantics of the application. We explain the reason why loop perforation was able to produce acceptable results. In Section 3.4.7 we summarize the findings from the individual applications.

### 3.3.1 Individual Loop Perforation

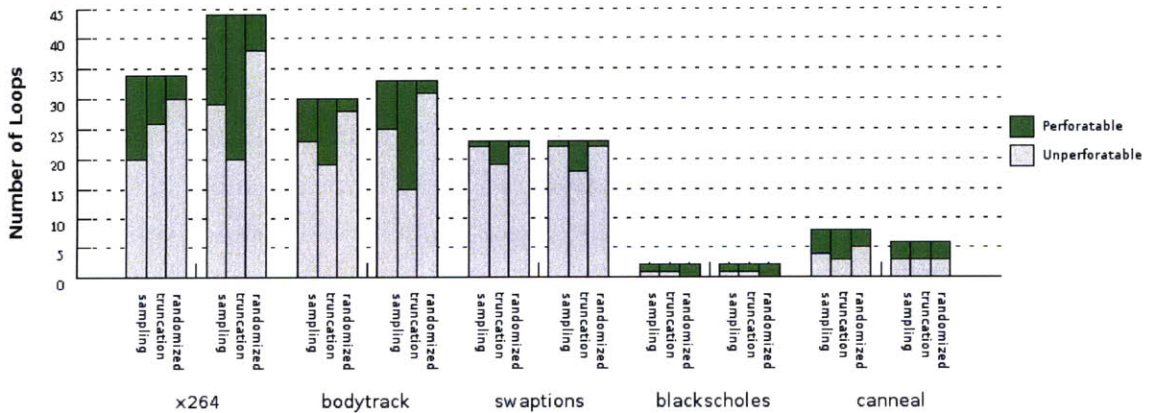


Figure 3-1: Perforatable Loop Count

Figure 3-1 shows the number of perforatable loops that the Quality of Service Profiler found. Each bar on the plot represents one configuration (which consists of the benchmark application, input and the perforation strategy).

The plot shows on the Y-axis the number of perforatable loops (top part of the bar) and the number of non-perforatable loops (bottom part of the bar) — the loops

that caused the program to crash, produce an unacceptable output, run slower than the original program or introduce latent memory errors. The sum of perforatable and non-perforatable loops is equal to the total number of loops that were above the bound of 1% of the performed work.

For each configuration, loop perforation was able to find at least one perforatable loop. These results provide an initial indication that loop perforation is able to produce alternative versions of some subcomputations that caused the program to execute faster while still producing acceptably accurate results. Moreover, the number of perforated and total loops indicates that loop perforation is a transformation that is applicable to a wide range of subcomputations. The detailed results, which we present in Section 3.4, show that in many configurations perforated loops have a significant effect on the execution time. Furthermore, we provide evidence in Section 3.4 that transforming perforatable loops often has meaningful domain-specific interpretation, which strongly correlates these loops with good QoS optimization targets.

The choice of perforation strategy also influences the number of perforatable loops. This behavior is consistent across all benchmarks and inputs. We attribute this to the nature of the computation — some computations deliver smaller QoS loss results if a contiguous part of the computation is skipped at the end, while in contrast, other subcomputations deliver smaller QoS loss results if a small part of the computation is skipped regularly during the computation. Overall, for most configurations the truncation perforation strategy selected the largest number of loops (for 6 out of 10 configurations), followed by the randomized strategy (for 2 configurations, both for the same program) and the sampling strategy (for 1 configuration), while for one configuration all three strategies perforated the same number of loops. The number of perforated loops per strategy indicates the potential of a particular strategy to be successfully applied to a variety of subcomputations. It does not, however, reflect the impact of each of the perforated subcomputations, because it does not consider the influence of the computation on performance and accuracy. In Section 3.3.2 we develop the comparison metric for the effectiveness of the strategy with the regard to the impact of the perforated loops.

Figure 3-1 also shows that the profiler considers a different number of loops for two inputs for x264, bodytrack and canneal benchmarks. It also found a different number of perforatable loops for these applications. We attribute these differences to the diversity of the execution profiles for different inputs. The profiler was set to filter out the loops that on an input execute for less than 1% of the total work. Some of the borderline loops performed more than 1% of the work for one input and less for

the other. For sampling and truncation perforation, the loops that are not common to both inputs contribute less than 2% of the work, and do not have potential to be strong optimization targets. On the other hand, the profiling results for both inputs contain all perforatable loops that have a major influence on the execution. The dependence of profiling on the representative input, inherent to all execution-based performance-profiling techniques, is the reason for slight differences between profiling results.

### 3.3.2 Cumulative Loop Perforation

Benchmark	Input	<i>Sampling</i>			<i>Truncation</i>			<i>Random</i>		
		Loops	QoS Loss	Speedup	Loops	QoS Loss	Speedup	Loops	QoS Loss	Speedup
x264	tractor	10	0.056	2.860	8	0.086	2.570	3	0.013	1.347
	blue_sky	13	0.073	2.363	15	0.072	2.146	6	0.041	1.324
bodytrack	seqA	6	0.076	2.923	9	0.077	2.879	2	0.034	1.965
	seqB	8	0.074	2.438	10	0.023	2.460	1	0.039	1.678
swaptions	input_1	1	0.029	1.953	3	0.029	2.426	1	0.039	1.869
	input_2	1	0.015	1.994	4	0.013	2.423	1	0.022	1.902
blackscholes	simlarge	1	0.000	1.970	1	0.000	2.060	2	0.000	3.742
	native	1	0.000	1.950	1	0.000	1.960	2	0.000	3.107
canneal	simlarge	2	0.040	1.270	1	0.074	1.344	2	0.086	1.388
	native	1	0.009	1.162	2	0.018	1.302	2	0.009	1.239

Table 3.2: Cumulative Perforation Scores

Table 3.2 summarizes the results of the cumulative loop perforation. The rows are grouped by the benchmark and the corresponding input. For each of the three perforation strategies, the table presents the total number of perforated loops (column Loops), the total QoS loss (column QoS Loss) and the performance increase (column Speedup) that the perforation of the loop causes. Note that for all of the benchmarks, the cumulative QoS loss is smaller than the user specified bound (0.1 in this experiment). The results indicate that loop perforation can be effective in finding computations that can trade off QoS for performance. Perforation was able to offer a significant performance increase with QoS loss which in many cases was substantially below the acceptable bound.

We will use the maximum performance gain of the perforated application as the *optimization potential* of the application exposed by loop perforation. As discussed in Section 2.4.2, this metric emphasizes the amount of time savings that could be expected from the developer’s optimization, while the amount of imprecision introduced by the optimization is below the bound that the developer selected as acceptable.

Figure 3-2 shows the optimization potential that was revealed by each perforation strategy. Each bar corresponds to one configuration (consisting of the application,

input, and the perforation strategy). The Y-axis shows the optimization potential. This information corresponds to the Speedup column from the Table 3.2.

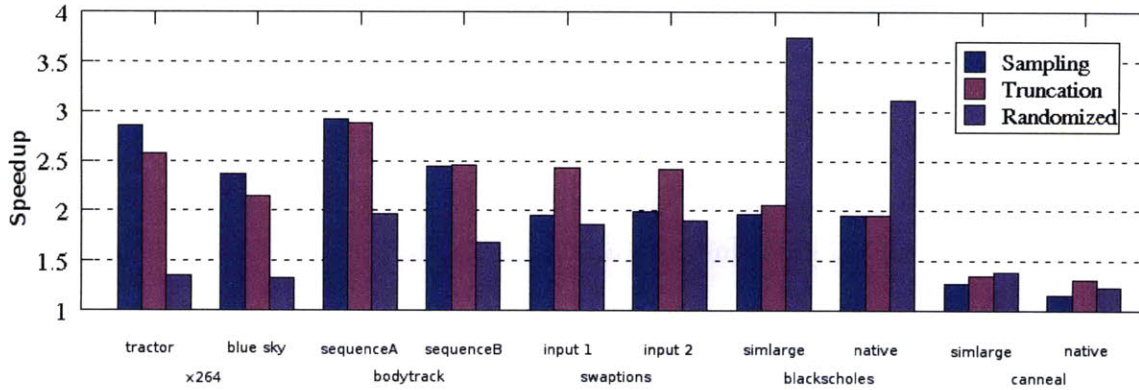


Figure 3-2: Performance Comparison of Different Perforation Strategies

The results in Figure 3-2 show that loop perforation can increase the application performance from 15% to almost 300%, while still producing a result that is within the 0.1 acceptability bound (and sometimes significantly below the bound). The number of loops perforated per application shows that all applications (for at least one perforation strategy) can tolerate multiple perturbed subcomputations, indicating that the amount of exploiting the redundancy in the benchmark applications goes across multiple loops, including potential synergies between subcomputations.

The selection and the impact of the perforatable loops depend on the perforation strategy and the type of the computation that is performed. The sampling and the truncation perforation strategies provide similar overall potential for all benchmarks, with the cumulative performance gains close. Sampling strategy offers better optimization potential for x264, while truncation offers better potential for swaptions. For bodytrack, blackscholes, and canneal the potentials were almost identical. Randomized perforation provided the best potential for blackscholes, which is the best overall score, but on larger benchmarks it performs worse than the other strategies — it has lower scores for x264 and bodytrack, while for swaptions its score was comparable to sampling. Canneal is the only benchmark in which a single perforation strategy was not able to reveal the greatest optimization potential for both inputs. For canneal, on `simlarge` input randomized perforation had somewhat better performance gains than truncation perforation, which in turn was somewhat better on `native` input.

## 3.4 Benchmark Application Results

### 3.4.1 Profiling Reports

Tables 3.3, 3.4, 3.5, 3.6, and 3.7 present Quality of Service profiling results for individual loops in the benchmark applications. The rows of the tables are grouped by the corresponding input. The rows are sorted by the number of instructions executed in the loop in the original unperforated application. For space reasons, the tables contain only the top 10 loops for each application. The columns of the table include the following:

**Function:** The first column contains the name of the function of the perforated loop.

**Location:** The second column contains the location of the loop, including the name of the source code file and the line numbers of the beginning and end of loop source code.

**Instruction %:** The third column contains the percentage of dynamically executed instructions within the loop in the original unperforated program. The percentage of work in each loop is aggregated from its dynamically (intraprocedural or interprocedural) nested loops.

**QoS Loss:** These columns contain the quality of service loss metric, which expresses the effect of the loop perforation on the quality of the result. The value 0 of the QoS loss indicates that loop perforation did not influence the final result. A dash (-) indicates that the perforated execution terminated unexpectedly. We present the values of QoS loss for each perforation strategy individually.

**Speedup:** The fifth column contains the speedup of the perforated application – the execution time of the unperforated program divided by the time of the perforated program. Speedups greater than 1 indicate that the perforated application runs faster than the original, while speedups less than 1 indicate that the perforated program is slower than the original program. We present the values of speedup for each perforation strategy individually.

The tables show the results for different perforation strategies. Each strategy has its own set of results consisting of the QoS loss and performance increase. We use special sign ( $\emptyset$ ) next to certain results to denote the loops for which Valgrind identified latent memory errors caused by loop perforation. Note that we mark the executions for each perforation strategy independently.

Tables 3.3, 3.4, 3.5, 3.6, and 3.7 also present part of the cumulative perforation scores. We write each pair of the individual perforation results for a single perforation

strategy in bold numerals if these results caused the loop to be accepted by the cumulative loop perforation. For example, the loop in `refine_subpel` from Table 3.3 (input tractor), was selected in cumulative profiling results for all three perforation strategies. In the same table, the outer loop in `pixel_satd_wxh` was selected when using the sampling and truncation perforation strategies (and is thus written in bold numerals) but not selected when using the randomized perforation strategy.

The tables show which loops in a program support profitable trade offs between Quality of Service and performance. The developer can then identify candidate sub-computations associated with perforated loops and focus his or her optimization effort on those subcomputations. Good candidates for the Quality of Service related optimizations are the computations that contain loops with the following properties: a) the application spends most of the execution time in the loop, b) when the loop is perforated, the performance of the application is significantly improved, and c) perforating the loop causes small quality of service losses with a meaningful interpretation in the application domain. This section shows the relation between perforatable loops and good Quality of Service optimization targets.

For each application we first describe the perforation results with the sampling strategy. We describe the main computations that it identified as good optimization candidates. We especially focus on those loops that were selected in both the individual and the cumulative results, as these loops show the greatest promise of being good optimization candidates. We also consider other perforatable loops that contribute non-trivially to the performance increase. For these loops we provide a rationale, based on the analysis of the semantics of the application, why the transformations deliver acceptable results. Finally, we compare the profiling results using the sampling perforation strategy against the profiling results using truncation and randomized loop perforation strategies.

### 3.4.2 Profiling Results for x264

#### Sampling Perforation Strategy

The individual loop profiling results (Table 3.3) for the sampling perforation strategy identify a number of loops that provide acceptable trade-offs between the Quality of Service and performance. In particular, the two loops in the function `pixel_satd_wxh` are promising optimization candidates. Perforating these loops causes a considerable speedup of over 30%, with acceptable distortion, the majority of which is due to the increased size of the video. Perforating the loop in the function `refine_subpel` can

also produce a favorable trade-off between the Quality of Service and the performance, with smaller performance increases, but also smaller QoS loss than the previous loops. The results are consistent across both inputs.

The profiling also identifies subcomputations that are poor optimization candidates. Although the loop in the function `encode` performs most of the work, perforating this loop causes an unacceptable QoS loss. This loop encodes individual video frames. The perforation of the loop causes a significant drop in the PSNR, going down by about 40% for both inputs. Perforating the loops in `pixel_sub_wxh_2013` produces an unacceptable Quality of Service loss (in the case of the loop on the line 181) or a latent memory error (in the case of the loop on the line 183). Perforating the loop in function `x264_mb_analyse_inter_p8x8` causes the program to crash due to segmentation fault.

The cumulative profiling results highlight the distinctions between good and bad optimization candidates. The set of loops that were perforated together includes, for both inputs, the profitable loops from `pixel_satd_wxh` and `refine_subpel`. The cumulative results also contain additional loops, including loops in `x264_sad_16x16` (shown in Table 3.3 for `blue_sky` input). These loops generally have a smaller influence on both the performance and on the quality of the result. Both inputs shared 8 loops, which perform almost 80% of the total work.

An analysis of the cumulative loop perforation reveals that all of the discovered loops are parts of the computation called *motion estimation* [14]. Motion estimation is a central part of the video compressing process. It performs a local, heuristic search over regions of neighboring frames, looking for similar blocks in these frames. In most cases, the blocks represent parts of the objects whose position changes between frames either due to the motion of the objects or due to the motion of the camera.

The function `pixel_satd_wxh`, identified by the Quality of Service Profiler as the best optimization candidate, calculates the similarity metric between two blocks. Perforating either of the loops in this loop nest causes the encoder to sample fewer points when calculating the similarity metric. The amount of QoS loss accumulated in the program after perforating both loops is 0.048 for the `tractor` input and 0.073 for the `blue_sky` input, with respective speedups of 1.80 and 1.5. The majority of the QoS loss comes from the increased video size. Successful perforation of both loops in the same loop nest indicates that there is a positive interaction between the loops, and possibly an additional amount of redundancy in computation, which makes the loop nest even more palatable as an optimization candidate.

The loop from `refine_subpel`, which was also selected in the cumulative per-



Input	Function	Location	Instruction %	sampling		truncation		random	
				QoS loss	Speedup	QoS loss	Speedup	QoS loss	Speedup
tractor	Encode	x264.c, 839	100.00%	0.382	1.921	0.227	2.070	0.498	2.830
	refine_subpel	me.c, 739	40.10%	<b>0.001</b>	<b>1.100</b>	<b>0.001</b>	<b>1.093</b>	<b>0.002</b>	<b>1.169</b>
	pixel_satd_wxh	pixel.c, 208	37.30%	<b>0.025</b>	<b>1.582</b>	<b>0.038</b>	<b>1.543</b>	0.166	1.223
	pixel_satd_wxh	pixel.c, 211	36.80%	<b>0.021</b>	<b>1.586</b>	<b>0.035</b>	<b>1.538</b>	0.234	0.971
	x264_mb_analyse_inter_p16x16	analyse.c, 984	23.10%	0.000	0.980	0.000	0.974	0.014	1.265
	x264_mb_analyse_inter_p8x8	analyse.c, 1130	21.00%	-	-	-	-	-	-
	pixel_sub_wxh2013	pixel.c, 181	19.70%	0.757	0.830	0.799	0.825	0.048	0.655
	pixel_sub_wxh2013	pixel.c, 183	17.60%	0.789	0.829	0.168	1.018	0.042	0.326
	x264_mb_analyse_inter_p8x16	analyse.c, 1225	15.90%	-	-	-	-	-	-
x264_mb_analyse_inter_p8x16	analyse.c, 1236	15.80%	-	-	-	0.976	-	-	
blue_sky	Encode	x264.c, 839	100.00%	0.375	1.847	0.213	2.096	0.471	2.719
	pixel_satd_wxh	pixel.c, 208	36.30%	<b>0.034</b>	<b>1.402</b>	0.059	1.362	0.333	0.977
	pixel_satd_wxh	pixel.c, 211	35.90%	<b>0.037</b>	<b>1.390</b>	<b>0.054</b>	<b>1.384</b>	0.235	0.794
	refine_subpel	me.c, 739	35.20%	<b>0.001</b>	<b>1.067</b>	<b>0.000</b>	<b>1.004</b>	<b>0.000</b>	<b>1.150</b>
	x264_mb_analyse_inter_p16x16	analyse.c, 984	33.00%	0.000	0.993	<b>0.000</b>	<b>1.003</b>	0.028	1.098
	x264_mb_analyse_inter_p8x8	analyse.c, 1130	21.30%	-	-	-	-	-	-
	pixel_sub_wxh2013	pixel.c, 181	19.20%	1.672	0.749	1.708	0.747	0.055	0.571
	pixel_sub_wxh2013	pixel.c, 183	17.10%	0.007 <sup>⊗</sup>	1.079	0.014 <sup>⊗</sup>	1.092	0.043	0.281
	x264_pixel_sad_16x16	pixel.c, 97	13.60%	<b>0.002</b>	<b>1.034</b>	<b>0.002</b>	<b>1.044</b>	0.018	0.814
x264_pixel_sad_16x16	pixel.c, 97	13.10%	<b>0.001</b>	<b>1.054</b>	<b>0.002</b>	<b>1.046</b>	0.008	0.469	

<sup>⊗</sup> Dynamic latent memory check failed for this loop

Table 3.3: Individual Loop Perforation Results for x264

foration results, performs a sub-pixel refinement computation<sup>4</sup>. The perforatable loop calls the similarity metric computation in function `pixel_satd_wxh` during the search for good matches. Perforating this loop causes the application to compare fewer nearby blocks during the comparison. The cumulative score shows that this loop can be perforated together with the loops that calculate the similarity metric between blocks, but when combined they provide a small additional performance increase, suggesting that the similarity metric computation is a primary optimization candidate, while sub-pixel refinement should be optimized after the similarity metric computation.

The other perforatable loops in `x264` compute similarity metrics for different sizes of blocks. The implementation of these metrics is simpler than the one in `pixel_satd_wxh` function. Perforating these loops also decreases the number of compared points.

### Truncation Perforation Strategy

In comparison to Quality of Service profiling with the sampling strategy, the individual profiling results for perforation with the truncation strategy (Table 3.3) show that this strategy is also able to find most of the loops, although in general with somewhat larger QoS loss and smaller speedup than the sampling perforation. The loops that were deemed unacceptable by the sampling Quality of Service Profiling, are also unacceptable after performing the truncation Quality of Service Profiling.

The cumulative loop perforation results show that although for one of the inputs (`blue_sky`) more loops can be perforated using the truncation strategy, the resulting program in both cases produces a result with somewhat larger QoS loss and smaller speedup than the program perforated with the sampling strategy. A more detailed inspection of the results shows that the two main loops that the sampling QoS Profiling discovered (`pixel_satd_wxh`, lines 208 and 211) for input `blue_sky` cause the program to produce the result with unacceptable Quality of Service loss. Thus, the Profiler selects only one of these loops (the one on line 211, with smaller QoS loss). Because it selected only one loop in this loop nest, Qosprof was able to add more small loops while preserving acceptable output.

---

<sup>4</sup>Subpixel refinement computation shifts block across the frames for non-integer increment. The new pixel values are calculated by interpolating the existing pixels.

## Randomized Perforation Strategy

The randomized strategy for QoS profiling does not perform as well as the other strategies. It was only able to identify the loop in `refine_subpel`, but not the loops in `pixel_satd_wxh` (Table 3.3). It also failed to recognize many of the other smaller loops in the motion estimation computation. As a result, the cumulative loop profiling step contains fewer, and less significant loops. The total optimization potential is as low as 1.3, although the QoS loss is also smaller than with the other perforation strategies. The overhead of dynamically deciding whether to perforate the loop at every iteration causes many of the perforated subcomputations to execute slower than the original subcomputations.

### 3.4.3 Profiling Results for Bodytrack

#### Sampling Perforation Strategy

Table 3.4 presents the results for the individual loop profiling using the sampling perforation strategy. Five out of the top ten time consuming loops show the promise of acceptable perforation. The loop in the function `ParticleFilter::Update` is the main computation for each frame. The computation rooted at this loop refines the probabilistic estimation model of the human body position by performing particle filtering. For each frame the algorithm performs several annealing steps. In each step, the algorithm stochastically disperses a number of points according to the previous body position model, and compares the number of points that cover the body. The algorithm uses a fitness function to compare the new model with the previous one. Perforating this loop causes only half of the refinement steps to execute. The obtained performance of the perforated loop is over 40%, while the QoS loss in this case is acceptable, only around 0.02. The loops belonging to the `ImageMeasurements` class calculate the fitness function value — many of these functions are perforatable.

The loop in the function `MultiCameraProjectedBody::ImageProjection` does not belong to the group of top ten loops that perform most of the work its computation executes only around 8% of the total work. However, because it provides a significant speedup of over 50%, with an acceptable QoS loss of 0.07, we appended the results for this loop to the results in Table 3.4. A source code inspection shows that this function unifies the projections from individual cameras into a single model. Perforating this loop causes the computation to consider only inputs from half of the cameras, making the input from other cameras unnecessary. The functions from the

Input	Function	Location	Instruction %	sampling		truncation		random	
				QoS loss	Speedup	QoS loss	Speedup	QoS loss	Speedup
sequenceA	mainPthreads	main.cpp, 219	99.70%	1.000	1.954	1.000	1.968	1.000	2.536
	ParticleFilter::Update	ParticleFilter.h, 220	80.50%	<b>0.014</b>	<b>1.508</b>	<b>0.014</b>	<b>1.503</b>	<b>0.042</b>	<b>1.788</b>
	ImageMeasurements::ImageErrorInside	ImageMeasurements.cpp, 140	34.40%	-	-	<b>0.018</b>	<b>1.181</b>	-	-
	ImageMeasurements::ImageErrorInside	ImageMeasurements.cpp, 142	34.30%	<b>0.021</b>	<b>1.158</b>	<b>0.024</b>	<b>1.103</b>	-	-
	ImageMeasurements::ImageErrorEdge	ImageMeasurements.cpp, 126	28.40%	-	-	<b>0.028</b>	<b>1.107</b>	-	-
	ImageMeasurements::ImageErrorEdge	ImageMeasurements.cpp, 128	28.40%	<b>0.031</b>	<b>1.089</b>	<b>0.022</b>	<b>1.070</b>	<b>0.018</b>	<b>1.045</b>
	ImageMeasurements::InsideError	ImageMeasurements.cpp, 110	26.60%	0.027 <sup>⊗</sup>	1.077	<b>0.024</b>	<b>1.083</b>	0.021	0.925
	ImageMeasurements::InsideError	ImageMeasurements.cpp, 114	22.60%	<b>0.038</b>	<b>1.138</b>	<b>0.026</b>	<b>1.050</b>	0.018	0.665
	TrackingModel::GetObservation	TrackingModel.cpp, 125	13.70%	0.070 <sup>⊗</sup>	1.197	0.056 <sup>⊗</sup>	1.229	0.022 <sup>⊗</sup>	1.039
	ImageMeasurements::EdgeError	ImageMeasurements.cpp, 65	13.60%	0.021	1.016	0.022	1.016	0.016	0.808
MultiCameraProjectedBody::ImageProjection	ImageProjection.cpp, 86	8.71% <sup>‡</sup>	<b>0.070</b>	<b>1.547</b>	<b>0.055</b>	<b>1.562</b>	<b>0.021</b>	<b>1.156</b>	
sequenceB	mainPthreads	main.cpp, 219	99.40%	1.000	1.907	1.000	1.969	1.000	2.763
	ParticleFilter::Update	ParticleFilter.h, 220	64.70%	<b>0.021</b>	<b>1.395</b>	<b>0.021</b>	<b>1.390</b>	<b>0.039</b>	<b>1.678</b>
	ImageMeasurements::ImageErrorInside	ImageMeasurements.cpp, 140	27.60%	0.016 <sup>⊗</sup>	1.150	<b>0.013</b>	<b>1.115</b>	-	-
	ImageMeasurements::ImageErrorInside	ImageMeasurements.cpp, 142	27.50%	<b>0.016</b>	<b>1.138</b>	<b>0.011</b>	<b>1.065</b>	-	-
	TrackingModel::GetObservation	TrackingModel.cpp, 125	24.90%	0.053 <sup>⊗</sup>	1.201	0.014 <sup>⊗</sup>	1.187	0.013 <sup>⊗</sup>	1.098
	ImageMeasurements::ImageErrorEdge	ImageMeasurements.cpp, 126	21.90%	0.026 <sup>⊗</sup>	1.073	0.016	1.050	-	-
	ImageMeasurements::ImageErrorEdge	ImageMeasurements.cpp, 128	21.80%	<b>0.028</b>	<b>1.065</b>	0.042	1.054	-	-
	ImageMeasurements::InsideError	ImageMeasurements.cpp, 110	21.20%	0.011 <sup>⊗</sup>	1.054	<b>0.035</b>	<b>1.079</b>	0.013	0.910
	ImageMeasurements::InsideError	ImageMeasurements.cpp, 114	18.00%	<b>0.016</b>	<b>1.086</b>	0.016	1.038	0.012	0.693
	ImageMeasurements::EdgeError	ImageMeasurements.cpp, 65	10.40%	<b>0.024</b>	<b>1.004</b>	0.020	1.010	0.025	0.844
MultiCameraProjectedBody::ImageProjection	ImageProjection.cpp, 86	8.71% <sup>‡</sup>	<b>0.053</b>	<b>1.424</b>	<b>0.014</b>	<b>1.386</b>	<b>0.028</b>	<b>1.117</b>	

<sup>⊗</sup> Dynamic latent memory check failed for this loop

<sup>‡</sup> The loop was appended to the top 10 list due to its notable effect on the computation

Table 3.4: Individual Loop Perforation Results for bodytrack

the class `ImageMeasurement` use this model to calculate the fitness of the provisional body part estimation. A simpler, less precise model causes a smaller amount of work to be performed in related, non-nested subcomputations and eventually results in performance increase that is higher than expected from the work performed in the image projection subcomputation.

Some of the loops that passed the accuracy and performance tests failed on the latent memory error tests. Examples of loops that do not pass latent memory error check include the ones in `getObservation`. The outer loop in `ImageErrorInside` caused the program to crash for the input sequenceA. For the input sequenceB, perforating the loop did not cause the failure of the program, but the dynamic analysis step discovered the control of the program subsequently depended on the uninitialized value. Sometimes, the loops that cause latent memory errors for one input, execute without errors on the other input.

For both inputs, the cumulative loop perforation results show that the majority of the time-consuming perforatable loops can also be perforated together. The application has two sources of redundancy. Part of the redundancy comes from the inputs — all cameras capture the same motion, but from different angles. Although the inputs from a larger number of cameras improve the precision, processing only part of the input can still be acceptable for some uses. The other part of the redundancy comes from the algorithm — performing less refinement steps or having a different number of points dispersed in each step can decrease the fidelity of the recognized motion, but does not cause the computation to fail or completely alter its behavior.

### **Truncation Perforation Strategy**

The profiling results obtained with the truncation perforation strategy show that the truncated loops in general caused the program to behave similarly to the sampling strategy, but have somewhat smaller performance increase and smaller distortion. The loop in `ParticleFilter::Update` function had exactly the same performance in both cases — the influence from the previous iteration is similar with no regard to the order of the iterations. On the other hand the loop in the function `MultiCameraProjectedBody::ImageProjection` caused the program to produce a somewhat different result, which is caused by the choice of the camera from which to read the input. The cumulative loop profiling results for the truncation strategy select additional loops not present in the sampling strategy results, but these loops have mostly a minor contribution to the program performance.

## Randomized Perforation Strategy

The profiling results obtained with the randomized perforation strategy show that both the impact and the number of perforatable loops are significantly smaller in this case — while it identified the loops in functions `ParticleFilter::Update` and `MultiCameraProjectedBody::ImageProjection`, the transformed programs crash on most of the loops that calculate the fitness of the next step. The cumulative loop profiling report contains at most two loops for these inputs, and miss many computations identified by other two strategies.

### 3.4.4 Profiling Results for Swaptions

#### Sampling Perforation Strategy

Table 3.5 presents the individual results for this benchmark. The loop in the function `HJM_Swaption_Blocking` performs the majority of the computation. This loop estimates the price of a single swaption, using Monte Carlo simulation. Perforating this loop using the sampling perforation strategy reduces the number of Monte Carlo trials, resulting in the smaller accuracy of the prices. Because the number of Monte Carlo simulations is divided by the expected number of loop iterations, the final prices of the swaptions are about a half of the price of the swaptions computed by the unperforated program. Qosprof can identify such regularity in the difference between the results and employ the compensation, by calculating a systematic bias between the two results, as described in Section 2.3. The results reported in Table 3.5 are obtained after applying a bias compensation.

The outer loop in the function `worker` performs the computation for each swaption. Perforating this loop causes only half of the results to be calculated. Sampling the computation in function `HJM_SimPath_Forward_Blocking`, which calculates the path of the forward interest rate curve causes an enormous Quality of Service loss. Since other time consuming loops produce similar behavior, the loop from `HJM_Swaption_Blocking` is the only loop selected for the cumulative loop profiling.

#### Truncation Perforation Strategy

The truncation perforation strategy is able to discover some loops that were deemed inadequate for the sampling strategy. The loops that calculate forward interest rate curve rates in function `HJM_SimPath_Forward_Blocking` iterate over matrices, calculating matrix elements which depend on the neighboring elements, calculated in

Input	Function	Location	Instruction %	sampling		truncation		random	
				QoS loss	Speedup	QoS loss	Speedup	QoS loss	Speedup
input.1	worker	HJM_Securities.cpp, 55	100.00%	1.000	1.903	1.000	2.022	1.250	2.580
	HJM_Swaption_Blocking	HJM_Swaption_Blocking.cpp, 164	100.00%	<b>0.029</b>	<b>1.953</b>	<b>0.029</b>	<b>2.016</b>	<b>0.039</b>	<b>1.869</b>
	HJM_SimPath_Forward_Blocking	HJM_SimPath_Forward_Blocking.cpp, 75	45.50%	1.537	0.964	<b>0.000</b>	<b>1.032</b>	1.526	0.781
	HJM_SimPath_Forward_Blocking	HJM_SimPath_Forward_Blocking.cpp, 77	31.20%	0.332	1.071	0.002	0.961	-	-
	HJM_SimPath_Forward_Blocking	HJM_SimPath_Forward_Blocking.cpp, 42	13.70%	1.969	1.057	0.000 <sup>⊗</sup>	1.030	-	-
	Discount_Factors_Blocking	HJM.cpp, 390	13.50%	-	-	0.000	0.986	-	-
	Discount_Factors_Blocking	HJM.cpp, 392	13.40%	1.535	0.953	1.535	0.935	-	-
	HJM_SimPath_Forward_Blocking	HJM_SimPath_Forward_Blocking.cpp, 44	12.10%	-	-	0.000	0.942	-	-
	Discount_Factors_Blocking	HJM.cpp, 394	11.80%	1.161	1.031	1.497	1.047	1.468	0.595
HJM_SimPath_Forward_Blocking	HJM_SimPath_Forward_Blocking.cpp, 64	10.50%	0.332	1.114	0.002 <sup>⊗</sup>	1.163	-	-	
input.2	worker	HJM_Securities.cpp, 55	100.00%	1.000	1.992	1.000	2.013	1.141	2.250
	HJM_Swaption_Blocking	HJM_Swaption_Blocking.cpp, 164	100.00%	<b>0.015</b>	<b>1.994</b>	<b>0.015</b>	<b>2.013</b>	<b>0.022</b>	<b>1.902</b>
	HJM_SimPath_Forward_Blocking	HJM_SimPath_Forward_Blocking.cpp, 75	45.50%	1.512	0.980	<b>0.000</b>	<b>1.116</b>	1.505	0.779
	HJM_SimPath_Forward_Blocking	HJM_SimPath_Forward_Blocking.cpp, 77	31.20%	0.369	1.008	<b>0.002</b>	<b>1.056</b>	-	-
	HJM_SimPath_Forward_Blocking	HJM_SimPath_Forward_Blocking.cpp, 42	13.70%	1.984	1.038	0.000 <sup>⊗</sup>	1.022	1.515	0.965
	Discount_Factors_Blocking	HJM.cpp, 390	13.50%	-	-	0.000	1.021	-	-
	Discount_Factors_Blocking	HJM.cpp, 392	13.40%	1.510	0.990	1.510	1.020	-	-
	HJM_SimPath_Forward_Blocking	HJM_SimPath_Forward_Blocking.cpp, 44	12.10%	-	-	0.000	0.984	-	-
	Discount_Factors_Blocking	HJM.cpp, 394	11.80%	1.156	0.994	1.479	0.988	1.452	0.619
HJM_SimPath_Forward_Blocking	HJM_SimPath_Forward_Blocking.cpp, 64	10.50%	0.369	1.115	0.002 <sup>⊗</sup>	1.130	-	-	

<sup>⊗</sup> Dynamic latent memory check failed for this loop

Table 3.5: Individual Loop Perforation Results for swaptions (with bias)

previous iteration. Unlike the sampling perforation, which skips neighboring rows and columns, the truncated loop calculates these elements, and the computation produces small overall QoS loss (with bias applied). Perforating this loop results in a different, approximate prediction model of the swaption price change.

### **Randomized Perforation Strategy**

Randomized perforation was also able to find the main perforatable loop. The speedup and the QoS loss of this loop are comparable to the loops perforated using sampling and truncation strategies. The effectiveness of this approach to find the perforatable loop can be explained by the substantial amount of work performed in each iteration, which diminishes the overhead of dynamic decision. Additionally the loop executes a large number of iterations of the loop (over 10 million), which ensures that the rate of loop perforation is close to 0.5.

### **3.4.5 Profiling Results for Blackscholes**

This benchmark has the least number of loops that were considered by the Quality of Service Profiler. Most of the loops in this program are `while` loops with no induction variable. In total, two loops that perform almost all of the work in the program are amenable candidates for profiling.

The profiling results for the sampling and the truncation strategy (Tables 3.6) show that the outer loop in function `bs_thread` (line 240) can be perforated with no QoS loss at all. At the same time, the inner loop (line 241), which calculates in each iteration the price of an individual option, causes the program to produce an unacceptable result – the values of only one half of the options are calculated (the rest remains on the initial value 0). An investigation of the source code investigation reveals that the benchmark developers inserted the outer loop in the benchmark application to increase the amount of work that the application performs. This loops performs the same redundant work 100 times. The number of loop iterations is an internal constant, not available to the user of the program.

The profiling results for the randomized strategy show that the Profiler was able to perforate both the outer and the inner loop. While the perforated inner loop produces only half of the results, the amount of redundancy provided by the outer loop, which repeats the entire computation 50 times even when perforated, is sufficient for the perforated program to produce all original results.



Input	Function	Location	Instruction %	sampling		truncation		random	
				QoS loss	Speedup	QoS loss	Speedup	QoS loss	Speedup
simlarge	annealer_thread::Run	annealer_thread.cpp, 102	68.00%	0.074	1.414	<b>0.074</b>	<b>1.344</b>	<b>0.073</b>	<b>1.373</b>
	netlist_elem::swap_cost	netlist_elem.cpp, 81	25.40%	<b>0.018</b>	<b>1.199</b>	0.018	1.112	-	-
	netlist_elem::swap_cost	netlist_elem.cpp, 90	25.40%	-	-	0.021	1.075	-	-
	netlist::netlist	netlist_elem.cpp, 189	25.20%	1.000	1.163	1.000	0.891	1.000	1.100
	MTRand::randInt	MersenneTwister.h, 309	2.70%	0.271	1.606	<b>0.035</b>	<b>1.026</b>	-	-
	netlist_elem::routing_cost_given_loc	netlist_elem.cpp, 63	1.76%	1.000	1.364	0.225	0.990	-	-
	netlist_elem::routing_cost_given_loc	netlist_elem.cpp, 57	1.76%	0.200	1.183	0.200	0.997	-	-
MTRand::randInt	MersenneTwister.h, 306	1.55%	<b>0.023</b>	<b>1.222</b>	0.022	1.022	<b>0.009</b>	<b>1.099</b>	
native	annealer_thread::Run	annealer_thread.cpp, 102	91.40%	0.004	1.146	<b>0.004</b>	<b>1.136</b>	<b>0.002</b>	<b>1.053</b>
	netlist_elem::swap_cost	netlist_elem.cpp, 90	36.30%	-	-	0.009	0.916	-	-
	netlist_elem::swap_cost	netlist_elem.cpp, 81	36.30%	0.010	0.957	0.010	0.943	-	-
	netlist::netlist	netlist_elem.cpp, 189	7.03%	1.000	1.061	1.000	1.062	1.000	1.020
	MTRand::randInt	MersenneTwister.h, 132	2.84%	0.353	4.397	<b>0.012</b>	<b>1.171</b>	-	-
	MTRand::randInt	MersenneTwister.h, 132	1.63%	<b>0.009</b>	<b>1.162</b>	0.001	0.985	<b>0.003</b>	<b>1.134</b>

Table 3.7: Individual Loop Perforation Results for canneal

Input	Function	Location	Instruction %	sampling		truncation		random	
				QoS loss	Speedup	QoS loss	Speedup	QoS loss	Speedup
simlarge	bs_thread	blackscholes.c, 240	99.00%	<b>0.000</b>	<b>1.970</b>	<b>0.000</b>	<b>2.060</b>	<b>0.000</b>	<b>2.541</b>
	bs_thread	blackscholes.c, 241	99.00%	0.481	1.885	0.488	2.103	<b>0.000</b>	<b>1.825</b>
native	bs_thread	blackscholes.c, 240	99.00%	<b>0.000</b>	<b>1.950</b>	<b>0.000</b>	<b>1.964</b>	<b>0.000</b>	<b>2.519</b>
	bs_thread	blackscholes.c, 241	99.00%	0.481	1.890	0.488	1.973	<b>0.000</b>	<b>1.741</b>

Table 3.6: Individual Loop Perforation Results for blackscholes

This benchmark shows the ability of QoS profiling to discover completely redundant computations.

### 3.4.6 Profiling Results for Canneal

#### Sampling Perforation Strategy

Table 3.7 presents the results for the individual loop profiling with the sampling strategy. The loop that performs the major computation in this benchmark is in the function `annealer_thread::Run`. This function uses simulating annealing to search for the optimal placement of the logic gates on a digital chip. In each annealing step, the gates randomly exchange position until there are more "good" exchanges than "bad" exchanges according to the fitness metric. The loop discovered by the Quality of Service Profiler exchanges a number of logic gates, one swap per iteration, before reevaluating number of "bad" and "good" swaps. After perforating this loop, fewer gates are swapped in one step, leading to a more frequent evaluation of the number of "bad" or "good" swaps. This subcomputation provides another interesting pattern: the loop is contained inside a while loop which calculates the exit condition for a step. Perforating the inner loop causes the outer while loop to execute more work (by invoking inner loop more times). This results in somewhat lower performance gains than expected from the amount of work that the application performs. The two inputs show somewhat different behavior: the QoS loss and the speedup are greater for the smaller input, and more conservative for the larger input.

Qosprof finds two more perforatable loops. One of them calculates a heuristic for the cost of performing the swap (`netlist_elem::swap_cost`). This loop speeds up the computation for one input, but slows it down for the other input. The other loop, which belongs to the class `MTRand`, uses the Mersenne Twister algorithm to calculate the sequence of pseudo-random numbers. Perforating of this loop would impact the strength of the pseudo-random number generator, which excludes this loop from the list of good optimization candidates. However, by inspecting the places from where this subcomputation is called, the developer can find a subcomputation that can trade off QoS for performance: the only use of the computation from the class `MTRand` is in the perforatable `annealer_thread::Run` function.

### Truncation Perforation Strategy

The individual results for the truncation strategy are similar to the results of the sampling strategy. The cumulative loop perforation selects the loop from the function `annealer_thread::Run`.

### Randomized Perforation Strategy

The randomized perforation strategy was able to identify the main computation loop, and its results were comparable to the other two strategies. This loop performs a larger number of iterations (over 10000) and each iteration performs a considerable amount of work, which partially compensates the overhead of dynamic perforation decision.

### 3.4.7 Discussion

The results of the individual and the cumulative loop profiling show that for a range of computations loop perforation can automatically perturb the computation to profitably trade QoS for performance. Our analysis of individual perforated computation in Section 3.4 shows that, in many cases, perforating the subcomputations has a meaningful interpretation in the application's domain.

In this section we will further describe the correlation between the perforatable computations and good QoS optimization targets. Both sets of computations share the property that the perforation of the computation results in significant performance gains and small QoS losses. Additionally, perforating good QoS optimization candidates most often has a meaningful interpretation in the application domain. The evaluation provided a strong correlation between the perforatable loops and one class of good QoS optimization targets — those computations that are implemented using `for` loops.

Qosprof found good optimization candidates in all 5 benchmarks. For 4 out of 5 applications loop perforation discovered subcomputations with favorable trade-offs, that have a sound interpretation in the application domain. For the fifth benchmark, `blackscholes`, loop perforation identified a completely redundant computation inserted by the benchmark developers to increase the amount of work that the application performs.

The ability to trade QoS for performance for some of the computations discovered by Qosprof was previously recognized by the benchmark developers and explicitly

exposed to the user. However, Qosprof also discovered a number of computations for which the trade offs were not exposed. Qosprof also discovered computations which were known to be profitable optimization targets, but the specific performance/QoS operation point identified by loop perforation was not made available to the user. In general, the ability of Qosprof to identify both exposed and previously unexposed optimization candidates increases our confidence in the effectiveness of Quality of Service profiling.

The subcomputations whose trade offs were recognized by developers are usually exposed as command line parameters. The user of the application can choose appropriate values of the parameters for every run. For some of the exposed computations, the effects of changing command line parameters were analogous to the effects of loop perforation. The command line parameters for these subcomputations control the number of iterations of the loops. Examples of such computations include the two computations in bodytrack (`ParticleFilter::Update` and `MultiCameraProjectedBody::ImageProjection`), the computation performing Monte Carlo simulations in swaptions (`HJM_Swaption_Blocking`) and the computation that swaps logic gates in canneal (`annealer_thread::Run`). For these three applications, loop perforation identified all subcomputations that depend on the command line parameters.

Qosprof also discovered a number of optimization candidates that were not exposed by the developer. For these computations, typically only a single implementation in the benchmark source code exists, and the execution time of this computation does not depend on user-controllable parameters. The examples of such computations include `refine_subpel` and smaller computation that calculate the block similarity cost in x264. Other applications that provide non-exposed trade offs are swaption pricing computation (`HJM_Simpath_Forward_Blocking`) in swaptions, the redundant blackscholes computation (`bs_thread`), and swap cost calculation (`netlist_elem::swap_cost`) in canneal.

For some subcomputations, the application developers provided multiple implementations with different performance/QoS trade-offs. The choice of the implementation is determined indirectly from the command line parameters. An example of such computation is the block similarity computation performed in function `pixel_satd_wxh` in x264, which has two different implementations. Loop perforation creates a set of alternative subcomputations that provide new, previously not covered trade offs between QoS and performance. Specifically, the QoS loss and performance offered by loop perforation lies between the QoS loss and performance offered by the

built-in subcomputation versions.

Quality of Service profiling provides new, valuable information that is not provided by standard performance profiler, but that can help the developer prioritize his or her optimization effort. The detailed perforation results have shown that the execution time of the loop is not always equivalent, and in some cases not proportional to the impact of that loop on the QoS. The loops that are selected by the Qosprof as the good optimization candidates are in most cases not the top-most loops, although most of them execute a significant amount of time. A notable example of the optimization candidates is the loop in function `MultiCameraProjectedBody::ImageProjection` in `bodytrack` benchmark, which executes less than 10% of the work, but its perforation contributes to 50% performance increase with small QoS loss.

Additionally, the perforatable loops which perform a similar amount of work may offer different trade offs. The loops that perform parts of the motion estimation computation are good examples of this behavior. The functions `refine_subpel` and `pixel_satd_wxh` consume almost the same amount of time, but trade off points are different — while the function `pixel_satd_wxh` offers a much larger performance benefits, the function `refine_subpel` offers much smaller QoS loss. The Quality of Service profiling provides the developer with an option to decide which computation to optimize first.

## Manual Optimization

By manually optimizing the application the developer may expect performance gains comparable to and potentially greater than those available via loop perforation, while the accuracy drop remains within the acceptable range. For the subcomputations from two of the benchmark applications (`x264` and `bodytrack`) that the Quality of Service profiler identified, we were able to devise manual optimizations that provide more favorable performance/QoS trade-offs than perforation alone. The three manual optimizations for `x264` and `bodytrack` delivered results that are comparable to the results of loop perforation.

The first manual transformation for `x264` modifies the computation in `pixel_satd_wxh` to exclude the Hadamard transformation. The results of executing this program on the `tractor` input show a performance increase of 1.41, and a QoS loss of 0.0039. The results of executing this program on the `blue_sky` input show a performance increase of 1.41 and a QoS loss 0.0004. These results are better than the results obtained from running alternative versions of the motion estimation computation settable from the

command line, using `subme` parameter (the value of this parameter used for the original runs is 5, alternative versions have smaller value of the parameter).

The second manual transformation for `x264` subsampled the points in the calculation from `pixel_satd_wxh`, in addition to skipping the Hadamard transformation. The result of executing the manually transformed program on `tractor` input showed the performance increase of 1.90 with a QoS loss 0.054. The manually transformed program on `blue_sky` input produced performance increase of 1.77 and a QoS loss of 0.068. The performance increase of this transformation is somewhat larger than the result of the built-in version of the motion estimation when the `subme` parameter has value 2. The QoS loss that the manually optimized program generates is higher than the QoS loss of the built-in versions, but is still within the acceptable range.

The manual transformation for `bodytrack` dynamically changes the number of annealing layers for each frame. This optimization increases the performance of the application by 1.36, with the QoS loss of 0.025 for input sequenceA and the performance of 1.30 and the QoS loss of 0.011 for sequenceB. The results are comparable to the results of changing the number of annealing layers from the command line.

Alternatively, if the developer is not able to devise a more profitable manual optimization, he or she can apply loop perforation and expose the number of iterations of the perforated loop as another user-settable parameter. Loop perforation alone exposed a significant potential for optimization of the benchmark applications. As presented in Section 3.3.2, the perforation can increase the performance of the applications from 15% to almost 300%, while still producing a result that is within the 0.1 acceptability bound.

## Work Reduction Patterns

We attribute the ability of loop perforation to successfully transform program computations to two general work reduction patterns — processing less data from the input, or performing less computation steps on a single set of data. We base the classification on a manual inspection of the source code.

**Data perforable loops.** These loops, when perforated, skip some parts of the inputs during the computation. These computations effectively calculate the result from a sampled subset of the input. Examples of this class of loops are all loops identified in the `x264` benchmark (including `pixel_satd_wxh` and `refine_subpel`), the loops in `bodytrack` that compute the model of the body from different cameras (`MultiCameraProjectedBody::ImageProjection`) and error calculation loops

(loops from the class `ImageMeasurement`), the forward path pricing loop in swaption (`HJM_Simpath_Forward_Blocking`) and the two computations in canneal benchmark: the main computation (`annealer_thread::Run`) and cost calculation from `netlist_elem::swap_cost`. For these loops, the result obtained on subset of data is a good approximation of the result that would be obtained from the whole input data.

**Computation perforatable loops.** These loops, when perforated, consider the entire data set, but skip some of the computation for all input elements. These computations often execute a number of refinement steps on data and stop when the acceptable computation error bound is satisfied. The example of these loops include the particle filter computation in bodytrack (`ParticleFilter::Update`), the Monte Carlo simulation step computation (`HJM_Swaption_Blocking`) in swaptions, and redundant computation in blackscholes (`bs_thread`). For these loops, partially processed data still provides a good approximation of fully processed data.

## Comparison between Perforation Strategies

In Section 3.3.2 we analyzed the optimization potential that each perforation strategy was able to uncover. For most of the subcomputations in the benchmark applications the sampling and truncation strategies identify a greater number of potential optimization targets than the randomized strategy. For some of the computations one of the strategies provided better results than the other. The randomized perforation strategy, on the other hand excelled at discovering massively redundant subcomputations that were beyond the reach of other strategies. In this section we further discuss the differences between the perforation strategies, inspired by the detailed benchmark results.

The sampling perforation strategy gives the best final result for the x264 benchmark, in which it was able to identify the loops that give the greatest optimization potential. Furthermore, it was able to deliver the smallest QoS loss. It was also the best performing perforation strategy for the bodytrack benchmark. The sampling perforation is especially suitable for a computation pattern where the processing of one loop iteration does not depend on the processing performed in adjacent loop iterations. An example of such a computation is the main block similarity computation (`pixel_satd_wxh`) in the x264 benchmark. In comparison with the randomized perforation strategy, which also samples the elements of the input space over the entire input domain, the sampling strategy has much less overhead, because the order of

discarding loop iteration is regular, and has virtually no overhead in the runtime. It also executes an expected number of iterations for loops with a small number of the original iterations, which is not guaranteed for the randomized strategy.

The truncation perforation strategy gives the best final result for the swaptions and canneal benchmarks. It was the only strategy able to identify the loop for calculating the forward option price path in the swaption benchmark. That computation is an example of a computation for which the results of adjacent loop iterations significantly influence the computation of that iteration's result. The sampling strategy, in contrast, does not handle well such subcomputations as it discards the calculation of adjacent loop iterations. Similarly, the randomized strategy does not guarantee that the results of adjacent iterations will be computed.

The randomized perforation strategy provided best results for blackscholes benchmark. It identified both loops as perforatable. As discussed in the application results, the amount of redundancy in this particular benchmark caused the inner computation to calculate values of all return variables. Randomized perforation also performed well in canneal and swaptions benchmarks, in which the number of iterations of perforated and the amount of work per iteration was large enough to successfully amortize the costs of this perforation strategy.

While sampling and truncation perforation discover almost the same perforatable loops, some of the perforatable loops in the benchmark applications were identified by only one of the strategies. Additionally, for some subcomputations, a version perforated using one strategy had significantly lower QoS loss than the version perforated using the other strategy. In general, a Quality of Service Profiling algorithm would benefit from the combination of these two perforation strategies during the program exploration, which would help the profiler identify and present the developer a more complete report of palatable optimization opportunities.

## **Limitations**

This thesis makes two main conclusions. First, Quality of Service Profiling presents useful information to a developer who wants to perform optimizations that trade off accuracy for performance. Second, loop perforation is an effective transformation for finding optimization candidates. We identify several threats to the validity of these conclusions and limitations of our approach.

The ability of Quality of Service profiling using loop perforation to find computations that offer favorable trade offs depends on the characteristics of the application.



Based on our experimental results, we believe that Quality of Service profiling in this setting will work well for applications where good optimization candidates are implemented as `for`-style loops. Computations that are implemented using other programming language constructs may be missed. This limitation can be partially overcome by including additional transformations that would create acceptable alternative subcomputations out of other kinds of subcomputations.

Second, the choice of the input used for profiling has a substantial influence on the final results. Some subcomputations may be wrongly rejected — e.g. because of no significant work was performed by the computation or due to small performance gains of perforated version, or wrongly accepted — e.g. the perforated subcomputation may produce acceptable result only for a given input, but not in general. Our experimental results showed that for the benchmark applications, profiling with both inputs produced very similar results, with the main optimization candidates being discovered in both cases. Note that the existence of a well-chosen representative input is a necessary requirement of any performance profiling technique. It is even more important for Quality of Service profiling because of the influence of the representative input on the quality of the result. Selecting a proper representative input is a responsibility of a developer or a domain expert.

Third, the developer is responsible for providing an acceptability model that truly represent the the envisioned use of the optimized program. If the output abstraction or the maximum acceptable QoS bound are not defined according to the true needs, the profiler may select a set loops which do not represent good optimization candidates for the intended program use.

Fourth, the choice of a single perforation strategy may limit the number of computations that are discovered during profiling. While our experimental results show that the sampling strategy and the truncation strategy perform equally well on the set of benchmark applications, this may not generalize to other applications. As noted previously, a larger set of transformations would help identify more promising subcomputations.

Finally, the performance may depend on the environment in which the program executes, including hardware, compiler and the operating system. Computations that execute faster in one environment may fail to do so in other environments. This is a general limitation of any execution-based performance profiling technique. To obtain more reliable performance results, the developer should profile the program in multiple execution environments.

# Chapter 4

## Related Work

### 4.1 Performance Profiling

Profiling a system to understand where it spends its time is an essential component of modern software engineering. Examples of profiling tools include prof [1], gprof [10], DCPI [3], jprof [19], and VTune [2]. Standard profiling mechanisms include code instrumentation (compiler-assisted [10], binary translation [23], runtime instrumentation [15], runtime injection [8]), sampling (procedure executions [10], instruction execution [1], and hardware counters [3]). The potential impact of memory system effects on the performance has motivated the development of profilers that are designed specifically to identify memory system bottlenecks [13, 16].

In comparison with standard profilers, a quality of service profiler adds the extra dimension by providing developers with information about the quality of service implications of changing the implementation of specific subcomputations. This additional information can enhance developer productivity because it can enable developers to avoid computations that may consume a significant amount of time but cannot be optimized without unacceptable quality of service losses. The quality of service information enables developers to instead focus on more promising subcomputations that loop perforation has already shown can be optimized with acceptable quality of service losses.

### 4.2 Performance vs Quality of Service Trade-Offs

Trading accuracy of the computation for other favorable program properties is a well-known technique. At the first place, many algorithms are designed to address the

performance/accuracy trade-offs as one of their inherent properties. A prominent example including lossy compression algorithms, including image [25], audio [7] or video encoding algorithms [14]. Quality of Service profiling points to the subcomputations that can be optimized in a particular context such that they result in results that are appropriate to the given context.

The researchers have also explored automatically trading off accuracy for performance [20], robustness [20], energy consumption [9, 24, 20] and fault tolerance [20] both at software and hardware levels. Most of the automated approaches provide lower level transformations and are either applied automatically in production system under unexpected environment conditions, or are opaque to the software developer. Quality of Service profiling operates on high level programming language constructs and provides the information to the developer which computations are good optimization candidates. In Section 4.3 we will provide a scenario on how Quality of Service profiling can be used to help system operate under unexpected environment changes.

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 [20, 21]. These approaches require explicit division of the work in tasks, for which the specialized language Jade was used. Quality of Service profiling operates on arbitrary program written in standard programming languages, effectively treating individual loops as the tasks.

### 4.3 Trade-Off Management

Recently a few frameworks for managing performance and quality of service trade-offs have emerged. Our framework SpeedGuard [11] can be used to dynamically adapt the performance or energy consumption of the program in response to the potentially disruptive environment changes like clock frequency changes, core failures or load fluctuations. SpeedPress uses QoS profiling for automatic identification of subcomputations that can trade-off accuracy for performance, and subsequently uses loop perforation as a mechanism for obtaining alternative subcomputation implementations. The system uses runtime control system which uses the facilities provided by the dynamic perforation instrumentation described in this thesis to turn the perforation of specific loops on or off during the program execution.

We have explored additional points in the performance/QoS trade off space by tuning the program command line parameters. Our PowerDial system [12] automatically introduces adaptability into the existing applications by changing the values

of appropriate command line parameters. It relies on the performance/QoS trade offs that were explicitly exposed by the program developers, both by providing alternative computation implementations and user settable parameter. In contrast, loop perforation does not rely on the explicitly exposed trade-off parameters. Instead it automatically transforms the program code and generates alternative implementations whose trade offs may be different from manually exposed trade offs.

An alternative 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 [22, 4, 5]. These systems inherently rely on the developer to identify the optimization opportunities and provide multiple subcomputation implementations. In contrast, Quality of Service profiling automatically identifies optimizable subcomputations, while loop perforation automatically generates one alternative version of the subcomputation. Based on the information from the profiling report, the developer may either accept loop perforation or manually write alternative program implementations.

# Chapter 5

## 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 identifies promising optimization opportunities by replacing original subcomputations with the automatically generated subcomputations that perform less work. The profiling results can help developers focus their optimization effort on the subcomputations that have demonstrated the potential for significant performance improvements with acceptably small quality of service losses.

We also presented loop perforation as a class of transformations that can generate alternative subcomputations which execute fewer loop iterations than the original subcomputations. Experimental results from our set of benchmark applications show that our Quality of Service profiler can use loop perforation to effectively separate subcomputations that are promising optimization opportunities from the subcomputations that show less promise. Furthermore, the experimental results showed that loop perforation can effectively reduce the execution time in multiple computations with acceptably small QoS losses.

# Bibliography

- [1] prof. Digital Unix man page.
- [2] VTune Performance Analyser, Intel Corp.
- [3] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.T.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, 1997.
- [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*.
- [5] Woongki Baek and Trishul Chilimbi. Green: A system for supporting energy-conscious programming using principled approximation. Technical Report TR-2009-089, Microsoft Research, August 2009.
- [6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT '08*.
- [7] K. Brandenburg. MP3 and AAC explained. In *AES 17th International Conference on High-Quality Audio Coding*. Citeseer, 1999.
- [8] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, Winter 2000.
- [9] L.N.B. Chakrapani, K.K. Muntimadugu, A. Lingamneni, J. George, and K.V. Palem. Highly energy and performance efficient embedded computing through approximately correct arithmetic: A mathematical foundation and preliminary experimental validation. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, 2008.

- [10] S.L. Graham, P.B. Kessler, and M.K. Mckusick. Gprof: A call graph execution profiler. In *SCC '82*.
- [11] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures . Technical Report MIT-CSAIL-TR-2009-042, MIT, September 2009.
- [12] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Power-aware computing with dynamic knobs. Technical Report MIT-CSAIL-TR-2010-027, Computer Science and Artificial Intelligence Laboratory, MIT, May 2010.
- [13] M. Itzkowitz, B.J.N. Wylie, C. Aoki, and N. Kosche. Memory profiling using hardware counters. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing, 2003*.
- [14] D. Le Gall. MPEG: A video compression standard for multimedia applications. *CACM*, April 1991.
- [15] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, June 2005.
- [16] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: analyzing memory system bottlenecks in programs. In *SIGMETRICS*, 1992.
- [17] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman Publishers, 1997.
- [18] N. Nethercote and J. Seward. Valgrind A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44–66, 2003.
- [19] G. Pennington and R. Watson. jProf – a JVMPI based profiler, 2000.
- [20] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS '06*.
- [21] Martin Rinard. Using early phase termination to eliminate load imbalance at barrier synchronization points. In *OOPSLA '07*.

- [22] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: a language and runtime system for perpetual systems. In *SenSys '07*.
- [23] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, 1994.
- [24] P. Stanley-Marbell, D. Dolech, A. Eindhoven, and D. Marculescu. Deviation-Tolerant Computation in Concurrent Failure-Prone Hardware. 2008.
- [25] G.K. Wallace. The JPEG still picture compression standard. 1991.