BROAD: BOLD AND RELIABLE ONLINE APPROXIMATE COMPUTING FRAMEWORK FOR DIVERSE APPLICATIONS

BY

LIULIN ZHONG

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Associate Professor Rakesh Kumar
Approximate computing is an emerging computing paradigm that leverages the inherent resilience of applications while designing energy-efficient computing systems. Approximate computing systems must satisfy user-provided requirements for quality of service (QoS), a quantitative criterion imposed on the output of an application such that the output is qualitatively useful. Previous software frameworks for approximate computing rely on the assumptions that approximation errors do not propagate through applications and that occasional QoS violations are acceptable. In this thesis, we explore the application of software approximations to applications for which these assumptions do not hold. We also observe that to avoid unacceptable QoS degradation (i.e., degradation beyond the QoS requirement), previous frameworks had to include a static approximation level guardband, which reduces the benefits to energy-efficiency. We propose BROAD, a Bold and Reliable Online Approximate Computing Framework for Diverse Applications. BROAD explicitly provides a checkpoint/rollback mechanism to allow applications to recover from QoS violations and error accumulation. The checkpoint/rollback mechanism further obviates BROAD from having a static approximation level guardband by allowing BROAD to operate near the QoS requirement without concern for permanent QoS degradation.
To my parents, for their love and support.
I would like to thank my advisor, Professor Rakesh Kumar, for his thoughtful guidance, encouragement, patience, and wisdom through my graduate school. I would like to thank Gilles Pokam from Intel Lab and Henry Duwe from my research group for their invaluable feedback on this work. I would also like to thank all of my groupmates for their encouragement and friendship.
# TABLE OF CONTENTS

LIST OF ABBREVIATIONS ........................................ vi

CHAPTER 1  INTRODUCTION ....................................... 1

CHAPTER 2  BACKGROUND ......................................... 3

CHAPTER 3  MOTIVATION .......................................... 5
  3.1 Motivation for Dynamically Adjusting Approximation Level . 5
  3.2 Motivation for Checkpoint/Rollback ......................... 8

CHAPTER 4  APPROXIMATION-CENTRIC CHECKPOINT AND
  ROLLBACK ....................................................... 13

CHAPTER 5  EXAMPLE APPLICATIONS .............................. 15
  5.1 Category 1: No Error Propagation, Single Checkpoint Kept
          at a Time .............................................. 15
  5.2 Category 2: With Error Propagation and Accumulation,
          Multiple Checkpoints Kept at a Time ................. 19
  5.3 Category 3: No Checkpoint/Rollback ...................... 21

CHAPTER 6  BROAD: BOLD AND RELIABLE ONLINE AP-
  PROXIMATE COMPUTING FRAMEWORK FOR DIVERSE
  APPLICATIONS ............................................... 23
  6.1 BROAD Framework ......................................... 23
  6.2 Parameters in BROAD ..................................... 27
  6.3 Selection of Candidate Approximation Levels ............ 28

CHAPTER 7  EXPERIMENTS ......................................... 29
  7.1 Application Performance with BROAD .................... 29
  7.2 Checkpoint Overhead ..................................... 36
  7.3 Exploration of Interval Sizes ............................ 37

CHAPTER 8  CONCLUSIONS ........................................ 41

REFERENCES ..................................................... 42
## LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>QOS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>BROAD</td>
<td>Bold and Reliable Online Approximate Computing Framework for Diverse Applications</td>
</tr>
<tr>
<td>BNCR</td>
<td>BROAD with No Checkpointing and Rollback</td>
</tr>
<tr>
<td>PSNR</td>
<td>Peak Signal-to-Noise Ratio</td>
</tr>
<tr>
<td>SSQ</td>
<td>Sum of Squared Distance</td>
</tr>
<tr>
<td>THR</td>
<td>Threshold</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
</tbody>
</table>
Approximate computing relaxes computation accuracy to achieve better performance and power efficiency. It is different from conventional computing, which frequently requires significant effort (performance and power) to guarantee the last bits of accuracy [1, 2, 3, 4, 5, 6, 7]. Approximate computing is being touted as a promising new computing paradigm for late-CMOS [8] and post-CMOS devices [9], server and HPC applications where the amount of data is scaling faster than the amount of computational resources [6, 10], and where many applications show inherent cognitive and algorithmic error resilience [11], e.g., multimedia applications, Monte Carlo applications, machine learning applications, etc. [12, 13].

In spite of the purported promise, however, since approximate computing delivers potentially inaccurate results throughout an applications execution, several critical challenges need to be addressed before such computing becomes mainstream. One challenge is determining the appropriate degree of approximation, also called approximation level. Approximation level affects both the quality of service (QoS) and performance, and there generally exists a trade-off between QoS and performance. Higher approximation level results in better performance, but lower quality; lower approximation level results in worse performance, but higher quality. However, there can be significant variation in the performance-QoS trade-off between different runtime inputs (Section 3.1). In addition, QoS variation exists even within an application for the same input (Section 3.1). These variations make it difficult to choose approximation levels at runtime. Another challenge is to provide dependable end-to-end QoS. While approximate computing does not provide a precise level of QoS as traditional computing does, it should provide a minimum reliable QoS in order to be widely useful. Yet another challenge is limiting the propagation of approximation errors through an application. Approximation errors, even small ones, at the beginning of an application
may cause unacceptably large QoS degradation in the later phases of the application (Section 3.2). This is a significant challenge since the effect of error propagation is hard to predict.

Previous software frameworks for approximate computing [5, 6, 7] do not meet many of these challenges. For example, they rely on the assumption that approximation errors do not propagate through applications. They also assume that occasional QoS violations are acceptable. In this research, we explore the application of software approximations to applications for which these assumptions do not hold. We also observe that to avoid unacceptable QoS degradation (i.e., degradation beyond the QoS requirement), previous frameworks had to include a static approximation level guardband, which reduces the benefits to energy-efficiency. We propose BROAD, a Bold and Reliable Online Approximate Computing Framework for Diverse Applications. BROAD explicitly provides a checkpoint/rollback mechanism to allow applications to recover from QoS violations and error accumulation. The checkpoint/rollback mechanism further enables BROAD to operate near the QoS requirement without concern for permanent QoS degradation.

This research makes the following contributions:

• BROAD is the first framework that explores checkpoint/rollback in the context of approximate computing. Unlike conventional checkpoint/rollback, the goal of checkpoint/rollback in approximate computing is to fix unacceptable QoS violations, reduce QoS variation, and to eliminate the error propagation problem to meet a QoS requirement.

• Provided with a fallback mechanism, BROAD aggressively adapts the approximation level during runtime while satisfying QoS requirement in every interval of execution.

• Finally, we evaluate BROAD in terms of performance and QoS over a diverse set of approximable applications. We demonstrate that BROAD is flexible enough to be applied to applications requiring different checkpoint/rollback configurations and as a basis for future approximation-centric checkpoint/rollback studies.
CHAPTER 2

BACKGROUND

Three frameworks have been recently proposed for software approximate computing—Green [5], PowerDial [6], and Sage [7]—that allow some form of dynamic approximation and can be applied in different contexts.

Green [5] allows application programmers to identify alternate, approximate version of functions which trade off QoS for performance or to identify loops which can have their iteration counts truncated to trade off QoS for performance. By performing extensive profiling, Green selects the appropriate level of approximation for an application in order to provide a statistical guarantee that it completes with acceptable QoS. Unlike BROAD, Green suffers from an approximation level guardband (Section 3.1). Furthermore, Green cannot provide an end-to-end QoS guarantee (i.e., the output of each sample interval is acceptable). Also, Green provides no mechanism (e.g., checkpoint/rollback) to recover from catastrophic errors or error accumulation.

PowerDial [6] identifies a set of dynamic knobs (i.e., parameters) within applications that allow a trade-off between QoS and performance. Based on performance feedback from automatically inserted instrumentation, PowerDial uses a control scheme to adjust the approximation level to meet performance goals at the expense of QoS. PowerDial++ [14] extends this work to integrate system-level QoS vs. performance trade-offs as well as adapting to QoS feedback. However, a statistical QoS guarantee is provided only over many sample intervals since a QoS violation in one interval can only be compensated for by decreasing the approximation level in the next interval (i.e., individual intervals and their corresponding output may have QoS violations). Also, PowerDial++ does not provide a method (e.g., checkpoint/rollback) to recover from a catastrophic degradation in QoS.

Sage [7] uses automatic compiler techniques to generate parameterized approximate versions of GPU kernels. At the start of execution, the pre-
processing necessary for each software approximation is performed by the CPU. An initial tuning phase is used to select an appropriate starting approximation level (i.e., set of approximations and parameters). The tuning phase employs a greedy tree algorithm to efficiently select an approximation level from among many approximation levels. During execution, QoS is monitored and the approximation level is dynamically changed by the CPU based on backtracking through the tree of approximation level used during tuning. Sage provides a statistical guarantee over intervals of execution (i.e., most intervals will have acceptable QoS and thus it is likely that overall QoS will be met). However, Sage does not provide an end-to-end guarantee that each interval meets QoS. It also does not provide a method (e.g., checkpoint/rollback) to recover from a catastrophic degradation in QoS.

There are some other related works on QoS profiling, software checkpointing and probabilistic assertions. Hoffman et al. [12, 13] study the trade-off between execution time and quality of service. They use quality of service profiling to identify the potential of loop perforation. They show that by perforating selected loops, applications can gain significant improvement in performance. Relax [15] is a framework for software recovery of hardware faults, which allows hardware faults to be exposed to the software and saves software checkpoints for the system to recover from. Sampsons et al. [16] propose probabilistic assertions for approximation applications that have probabilistic outcomes. The programmer can express probabilistic assertions in the applications to check the correctness of the computation. Some work [17] explores off-line profiling and searching of approximation configurations. The programmer provides the initial annotations for approximable software, and the tool heuristically searches the design space of precise and approximate decompositions of the program and returns the best configuration. EnerJ [1] provides data type annotations to declare data that can be mapped to approximate computation hardware. It isolates the parts in software that must be precise from the parts that are approximable. Truffle [2] is a hardware design that supports ISA extensions so that annotated approximable software can be mapped to truffle and save energy.
CHAPTER 3
MOTIVATION

BROAD is a software approximate computing framework which has two major features.

1. BROAD allows aggressive, better-than-worst-case dynamic approximation.

2. BROAD provides a checkpoint/rollback scheme to correct QoS violations, allowing end-to-end dependable QoS.

3.1 Motivation for Dynamically Adjusting Approximation Level

Several prior software approximate computing frameworks such as the baseline Green [5] proposals are profiling-based and do not change approximation based on QoS monitoring. Therefore, such frameworks have an implicit approximation level guardband, meaning that the approximation level chosen for execution based on profiling results is conservative and thus does not achieve the best possible performance. Green, for example, profiles the trade-off between QoS and performance based on a large data input set, ranging from 30 to 200K data inputs. Green selects an initial approximation level guardband such that the approximation level would satisfy QoS for every profiled input. For applications which have high variation in QoS trade-off across inputs or within an input, this approximation level guardband may represent a significant loss in performance. Note that the approximation level guardband is determined by the worst-case input.

We demonstrate the existence of an approximation level guardband using x264. x264 is a lossy video encoding application and has many parameters that affect both the QoS and encoding time (performance). We call such parameters that can trade off QoS for performance approximation knobs,
Figure 3.1: Normalized QoS and Performance of x264 for various approximation levels. Due to the variation of QoS between different inputs, a static approximation level unavoidably incorporates a guardband to achieve dependable QoS for all inputs.

as PowerDial does. We discuss more details about the application and the approximation knobs in Chapter 5. We select three approximation knobs; each knob has a range of values such that there are 60 different configurations (i.e., approximation levels). Figure 3.1a shows the QoS for different inputs for the 60 configurations on the $x$ axis, sorted according to the average QoS. Each line represents the QoS for a different input. A higher QoS is better,
with the highest accuracy encoding having a QoS of 1.0. We calculate QoS using Equation 5.1 in Chapter 5.

We observe that the QoS of x264 varies significantly between inputs. If a framework chooses the approximation level statically based on profiling results (a la Green [5]), the framework can only choose the configurations that satisfy the QoS requirement (0.8 in this case) even for the worst-case input. Among the 60 approximation levels in Figure 3.1a, only ten approximation levels allow all the inputs to satisfy the QoS requirement. Figure 3.1b shows the normalized runtime for the ten approximation levels. The runtimes are normalized to the static oracle approximation level, which is the approximation level that provides the best performance for a given input while still satisfying the QoS requirement for that input. All 10 approximation levels that meet the QoS requirement for all inputs have an average runtime of at least twice that of the static oracle. For certain inputs, the runtimes are over five times worse than for the static oracle. Therefore, choosing the approximation level based on profiling results as some prior work (e.g., Green) unavoidably does introduces an approximation level guardband. A dynamic scheme to pick the approximation level based on the input is needed instead. BROAD intends to be such a scheme.

Even with an approximation level guardband on the final QoS, QoS is unpredictable for inputs that have not been observed during profiling. Unseen inputs may have a very different QoS response from the inputs we use in
profiling (see Figure 3.1a). Thus, even if QoS meets the requirement in the profiling phase, the runtime QoS may still fail to meet the QoS requirement. A dynamic scheme such as BROAD is needed instead that can compute the appropriate approximation level for the unseen input.

We also observe that any static approximation level guardband is needlessly conservative, even for a single input. Figure 3.2 shows the QoS of x264 during execution at four different approximation levels. During execution at a single approximation level the QoS may vary by over 16%. If the QoS requirement was 0.6, the static oracle for this input would require execution at approximation level five. However, the QoS requirement could be met with execution starting at approximation level one, moving to approximation level three at interval 3, then to approximation level five at interval 6, and finishing at approximation level three for the final three intervals. Such a dynamic sequence of approximation levels could result in a significant increase in performance over the static oracle. BROAD is aimed at supporting such dynamic switches in approximation levels.

3.2 Motivation for Checkpoint/Rollback

BROAD provides the mechanism of checkpoint/rollback for several reasons—fixing intermediate QoS violations, correcting poor prediction of approximation levels, and protecting against error propagation and accumulation.

**Fixing QoS Violation** In many applications, even those which can tolerate approximation errors due to users’ perceptual abilities, an intermediate QoS violation may be unacceptable since such a violation may affect a user’s experience. For example, in streaming video services which use x264, QoS violations lower users’ perception of quality of the received videos. Previous work that investigates the impact of video quality on user behavior [18] has shown that a user who experiences poor video quality is less likely to revisit the same site within a week than a similar viewer who did not experience the poor quality. Since previous software approximation frameworks do not have a fallback correction mechanism such as checkpoint/rollback, the only way to compensate the QoS loss is to increase the QoS for the future intervals, which increases the QoS variation within the application.

**Correcting Poor Prediction** Typical approximate computing frame-
works predict the next approximation level based on incomplete information. For example, Green and PowerDial++ base their next approximation level decision on the QoS loss in the previous interval and the profiled results, while Sage bases its next approximation level on an instantaneous sampling of the QoS of nearby approximation levels. These prediction mechanisms can work well if there is no variation between application inputs or within adjacent intervals. However, application inputs and intervals within applications can vary significantly as shown in Section 3.1. Previous works do not handle the case where the QoS is violated due to imperfect prediction. A scheme such as BROAD which can fix occasional QoS violations can execute at an aggressive approximation level, close to the QoS requirement, since for every QoS violation detected, the QoS violation is fixed rather than accepted.

**Protecting Against Error Propagation/Accumulation**

Propagation and accumulation of approximation errors has always been a big concern for approximate computing. Previous approximate computing frameworks, including PowerDial, Green, and Sage, are based on the assumption that approximation errors do not propagate (and accumulate) through the application and that a decrease in the approximation level will recover the application’s final QoS. An example application where these assumptions break down is bodytrack, a human motion tracking kernel. Bodytrack is a key task in many applications including Kinect box, sport science, movie industry, and medical diagnostics [19]. In these applications, losing track of the body qualitatively denotes an unacceptable QoS. Figure 3.3 shows a qualitative violation of the QoS requirement in bodytrack where the entire torso of the body vector has lost the body images (left two cameras) and all the appendages are no longer tracking the appropriate body parts (all cameras).

In Figure 3.4, we use bodytrack to demonstrate the existence of error propagation problem. In this section, to explain the problem, we get the distance of the approximated body vector (as shown by the color boxes) and the accurate body vector. The body vector is represented by three translational elements $x_i$ and $n$ angular elements $\theta_i$. The distance is calculated by Equation 3.1, in which $\theta_i$ and $x_i$ are from the accurate body vector while $\hat{\theta}_i$ and $\hat{x}_i$ are from the approximated body vector.
\[ QoS = \frac{1}{n} \sum_{i=1}^{n} \text{weight}_i \bigg| \frac{\theta_i - \hat{\theta}_i}{2\pi} \bigg| + \alpha \sum_{i=1}^{3} \frac{|x_i - \hat{x}_i|}{x_i} \]  \quad (3.1)

The distance is regarded as the QoS; a larger distance means the QoS is worse (note that the QoS monitor in BROAD uses a different metric which is significantly light weight, see Section 5.2.1). We calculate the QoS for every interval of the bodytrack application. The \( x \) axis is the interval index and each interval is 30 frames. The \( y \) axis shows QoS of the frame at the end of each interval, where a lower QoS is better (i.e., a QoS of 0 is defined as accurate). Based on qualitative observation (Figure 3.3), we select a QoS requirement of 15. QoS below 15 results in the body vector being attached to the torso and generally correctly oriented. QoS beyond 15 means that the body vector loses track of the body or the orientation of the body vector is incorrect (e.g., right and left legs are switched). In the first interval, we use an aggressive approximation level (1,400—i.e., 1 layer and 400 particles—see Section 5.2.1). When the QoS requirement is violated (in iteration 2 where the QoS is higher than 15), we switch the approximation level to the most accurate level available (5,4000—i.e., 5 layers and 4000 particles—see Section 5.2.1), and continue execution at this level. However, due to approximation errors within the first two iterations, the application cannot meet the QoS requirement even using the most accurate approximation level in the last six intervals of execution.

Figure 3.4a shows the qualitative results of the process. Image 0 shows the QoS of Frame 0 (the initial input), the body vector accurately represents the body pose and position. Image 1 shows the QoS of at the end of Interval 1 (Frame 30). The body vector is still on the torso and QoS is below the requirement of 15. Image 2 shows the QoS after Interval 2. The body vector loses track of the torso and the QoS requirement is violated. After Interval 2, the application switches to the accurate level. As Image 7 shows, the position of the body vector gets aligned with the body. However, a portion of the body vector is reversed by 180 degrees (the lower yellow box represents the right leg and the lower teal box represents the left leg; the body vector is twisted, while the actual human body is not). Therefore, the QoS fails to meet the requirement, even after several iterations of accurate execution.
Checkpoint/rollback coupled with adjusting the approximation level allows the correction of a catastrophic propagated error and accumulated errors.

Figure 3.3: Failure of meeting quality of service from the view of four cameras.
(a) Body pose changes though frames from a single camera

(b) Even using the accurate algorithm cannot meet the QoS requirement due to error accumulation

Figure 3.4: Error Propagation.
A common and well-researched reliability mechanism, checkpoint/rollback, can be used to recover from error propagation and accumulation. If error propagation or accumulation causes a QoS violation, an approximation framework could restart execution at a lower approximation level (i.e., a more accurate approximation level) from a point before a catastrophic error or set of errors. Although approximation-centric checkpoint/rollback shares some basic mechanisms (e.g., process checkpointing) with reliability checkpoint/rollback, approximation requires several additional considerations. The main consideration is locating the catastrophic error or set of errors whose effects need to be negated from among many benign errors. Once such a catastrophic error is located, an approximation-centric checkpoint/rollback framework must select an appropriate approximation level which will mitigate the catastrophic error(s), while still maximizing performance. The following are the specific questions that such a framework must either answer or allow an application developer to explore:

- **When to checkpoint?** In a software approximation-centric checkpoint/rollback framework, there frequently exist specific points in the code or during execution where checkpoints may naturally be taken—for example, before any approximation is applied or when an approximation level is changed.

- **How often to checkpoint?** More frequent checkpointing allows a finer-grained catastrophic error localization to reduce the amount of re-computation; however, more frequent checkpointing can have a significant performance penalty due to an increased volume of disk writes.

- **How many checkpoints should be kept?** Limited disk space allocated to a checkpoint/rollback scheme may require that checkpoints be overwritten during execution. Fewer checkpoints mean that more
re-computation (and thus longer execution times due to larger rollbacks) may need to be performed. This is especially true if multiple iterative rollbacks are required because the catastrophic error(s) cannot be accurately located or the wrong approximation level for restart is selected.

- **Which checkpoints should be overwritten first?** When only a limited number of checkpoints may be kept during execution, selecting which checkpoints to save becomes important. Keeping the most recent checkpoints is beneficial as it reduces the amount of re-computation, yet older checkpoints may be required to protect against error accumulation and propagation.

- **Which checkpoint should be used during a rollback?** An approximation centric checkpoint/rollback framework must locate the interval in which a catastrophic error or set of errors occurred in order to successfully roll back. We note that approximate computing execution frequently has many errors in each interval of execution, so locating a catastrophic error is non-trivial.

- **How to determine an approximation level for the next iteration?** A checkpoint/rollback scheme allows dynamic approximation to run closer to the QoS requirement because any QoS violation can be fixed, yet frequent rollbacks can degrade performance benefits. Therefore, approximation levels must be carefully chosen during execution to maximize total benefits.

- **How to determine the new approximation level after rollback?** Once a catastrophic error has been detected and localized, an approximation-centric checkpoint/rollback framework must select an approximation level at which to restart execution, which will not incur the same catastrophic error(s) while still providing the maximum performance.
In order to explore the approximation-centric checkpoint/rollback questions described in Chapter 4, we studied applications from different areas. We categorize the applications into three basic categories based on their behavior with respect to approximation-centric checkpoint/rollback. The applications are summarized in Table 5.1.

For each application, we show how to speed up the application by simply using software approximation techniques, i.e., not changing any underlying hardware. We identify approximation knobs that trade off QoS for performance, a QoS metric that can be used to quantify the QoS of the final/total application output, a QoS monitor that can efficiently estimate the QoS of an interval of execution, and how each application responds to checkpoint/rollback.

5.1 Category 1: No Error Propagation, Single Checkpoint Kept at a Time

For applications in this category, the computation in each interval does not strongly rely on the computation of previous intervals. Therefore, these applications only need a basic checkpoint/rollback scheme to provide dependable QoS under approximation.

5.1.1 X264

X264 is a lossy video encoding application. Given a stream of frames, x264 will look for spatial and temporal redundancy in the frames to perform compression. Each frame is composed of macroblocks while each macroblock is composed of pixels. For encoding each macroblock in a frame, the analy-
sis/motion estimation block searches the previously encoded frame for temporal redundancy and decides the best encoding mode. Quantization maps a continuous range of signal to a small discrete set of signals. In the last step, the frame store saves the reconstructed frame as the reference frame for encoding the following frames [20].

**Approximation Knobs:** x264 has a number of parameters that can change the encoding quality and time. We choose three knobs to adjust x264’s approximation level. They are `qp`, `subme`, and `ref`. `qp` is a knob for rate control, which directly decides the video quality (bitrate). A smaller `qp` results in higher quality video. `subme` changes the subpixel estimation complexity. It has a range of values from 1 to 11 (larger subme results in higher quality). `ref` sets the maximum number of previous frames that can be used by `P-frames` as reference frames. It ranges from 1 to 16 and larger value results in better quality.

**QoS Metric:** We use two standard QoS metrics in the signal processing field to measure the quality of video: PSNR and bitrate. PSNR is peak signal-to-noise ratio and bitrate is the data throughput in a given amount of time. We use Equation 5.1 to calculate QoS value, in which `PSNR_high` and `bitrate_high` represent the QoS of an accurate specification.

$$QoS = 0.5 \times \frac{PSNR}{PSNR_{high}} + 0.5 \times \frac{bitrate}{bitrate_{high}} \quad (5.1)$$

<table>
<thead>
<tr>
<th>Application</th>
<th>Approximation Knob</th>
<th>QoS Metric</th>
<th>QoS Monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>x264</td>
<td><code>qp</code>, <code>subme</code>, <code>ref</code></td>
<td>PSNR and bitrate</td>
<td>built-in function for PSNR and bitrate computation</td>
</tr>
<tr>
<td>streamcluster</td>
<td>loop perforation kmin and kmax</td>
<td>SSQ (Eq. 5.2)</td>
<td>the SSQ for each interval</td>
</tr>
<tr>
<td>ferret</td>
<td>loop perforation</td>
<td>intersection ratio (Eq. 5.4)</td>
<td>the average intersection ratio for the last n queries</td>
</tr>
<tr>
<td>bodytrack</td>
<td>number of particles and layers</td>
<td>distance of body vectors (Eq. 5.5)</td>
<td>distance of the last two frames in the interval</td>
</tr>
<tr>
<td>pagerank</td>
<td>loop perforation</td>
<td>convergence</td>
<td>convergence and max number of iterations</td>
</tr>
<tr>
<td>srad</td>
<td>loop perforation</td>
<td>mean &amp; variance in the pixel intensities</td>
<td>mean and variance in pixel intensities and max number of iterations</td>
</tr>
</tbody>
</table>
**QoS Monitor:** We use the built-in functions in x264 that calculate PSNR and bitrate. The PSNR and bitrate are calculated cumulatively over all the frames that are encoded.

**Checkpointing:** QoS monitoring and checkpointing are performed after \( N \) frames are encoded, where \( N \) may be as small as 1. A checkpoint is needed for two reasons: (1) as a streaming application, there is no guarantee when the final output is required (e.g., taking raw video input from a camera as input, x264 will not know when the input will cease) and (2) the output of intermediate intervals may be used (e.g., when encoding video for video chat, the quality of x264’s intermediate outputs is still important, not just the final cumulative QoS for the entire chat), so each interval should meet the QoS requirement. Only a single checkpoint is kept because error propagation and accumulation are minimal (i.e., when reference frames are fully encoded, error propagation and accumulation are averted or reduced depending on approximation level). Determining approximation levels for each interval of execution and after a rollback is still a challenge.

### 5.1.2 Streamcluster

Streamcluster is a data mining application in PARSEC suite [21]. It uses K-means clustering algorithm to group a large number of data points into clusters. Given a stream of data, it will assign each data point to its nearest center.

**Approximation Knobs:** BROAD uses three approximation knobs including \( k_{\text{min}} \), \( k_{\text{max}} \) and loop perforation. \( k_{\text{min}} \) is the minimum number of centers while \( k_{\text{max}} \) is the maximum number of centers. We also applied loop perforation to the loop called \( pFL \). Perforation yields large performance improvement with small QoS degradation.

**QoS Metric:** The QoS metric is SSQ, which is the sum of squared distance from all the points to their own centers, as shown in Equation 5.2. \( T \) is the set of centers while \( C_z \) is the cluster around center \( z \). The SSQ calculation is a part of the original streamcluster algorithm and we utilize the result of SSQ as the QoS metric.

\[
SSQ = \sum_{z \in T} \sum_{x \in C_z} ||x - z||^2
\]  

(5.2)
QoS Monitor: We allow the application user to provide a target QoS for the streamcluster, which sets the upperbound for the looseness of all the data points in each interval. At the end of each interval when the application has processed a chunk of data, the QoS monitor compares the SSQ of the data points with user provided target QoS after each chunk of data is processed (Equation 5.3).

\[
QoS_{\text{metric}} = \frac{\text{target QoS}}{\text{SSQ for the interval}} \tag{5.3}
\]

Checkpointing: QoS monitoring and checkpointing are performed at the end of each interval when \( N \) points are grouped as different clusters. Other checkpointing characteristics and decisions mirror x264 (see Section 5.1.1).

5.1.3 Ferret

Ferret is an image similarity search application. Provided with a number of image queries, ferret searches the image database and returns the images that are similar to the query.

Approximation Knobs: The approximation knob for ferret is loop perforation of the loop in function \_LSH\_query.

QoS Monitor: There are \( N \) image queries in each interval. They are processed using an approximation level predicted by BROAD. For the last \( n \) image queries, the monitor runs them again using the accurate level and computes the number of intersection images for each query between the accurate results and approximation results. The monitor gets this ratio for all the \( n \) queries and computes the average. The goal is to have \( n/N \) to be a small number so that monitoring is not a big overhead to the overall performance, and \( n \) is large enough to represent the quality of the entire interval. The runtime of BROAD for ferret in Section 7.1.4 incorporates the overhead of the QoS monitor.

\[
QoS_{\text{metric}} = AVG(\frac{\text{number of intersection images}}{N}) \tag{5.4}
\]

Checkpointing: QoS monitoring and checkpointing are performed at
the end of each interval when \( N \) queries have been served.

5.2 Category 2: With Error Propagation and Accumulation, Multiple Checkpoints Kept at a Time

Applications in this category have strong dependencies between intervals. In each interval, there may be a difference between the approximate result and the accurate result. We call this difference approximation error. Since each interval uses the result from the previous interval which contains the approximation error, approximate earlier intervals can significantly degrade the QoS of later intervals, even those which are executed accurately many intervals later. Therefore, multiple checkpoints may be required to achieve performance improvement from approximation, while still meeting the final QoS requirements.

5.2.1 Bodytrack

Bodytrack [21] is a computer vision application that tracks a human body through image sequences from multiple cameras. It recovers the body poses from every frame of image.

**Approximation Knobs:** Bodytrack uses a particle filter to track the body poses. The accuracy of the particle filter is decided by the number of layers and the number of particles. The number of layers can vary from 1 to 5 and the number of particles varies from 400 to 4000. Increasing the number of layers and particles increases the execution time and improves the QoS.

**QoS metric:** The output of the application is a set of body vectors corresponding to each frame. Our QoS metric is to calculate the difference between the last two frames of each interval. Our QoS metric is described in Equation 5.5. A body vector contains angle elements \( \theta_i \) and translational elements \( x_i \). \( \hat{\theta}_i \) and \( \hat{x}_i \) are from the approximate body vector. \( \theta_i \) and \( x_i \) are from the accurate body vector. The angle elements represent the joint angles for different parts of the body and we weight them according to the length of the body part, so that a larger body part has bigger effect on the QoS metric.
We scaled the translational vector elements with a factor of $\alpha$ which allows the angle elements and translational elements to show equivalent influence on the QoS.

$$QoS = \frac{1}{n} \sum_{i=1}^{n} \text{weight}_i \left| \frac{\theta_i - \hat{\theta}_i}{2\pi} \right| + \alpha \sum_{i=1}^{3} \left| \frac{x_i - \hat{x}_i}{x_i} \right|$$  \hspace{1cm} (5.5)

**QoS Monitor:** The QoS monitor computes the QoS at the end of each interval where $N - 1$ frames have been processed using a certain approximation level. BROAD will increase approximation level to allow the $N$th frame to run at the most accurate level. Then the QoS monitor computes the difference between body vectors from the $(N - 1)$th frame and the $N$th frame based on Equation 5.5. The insight is that the model in bodytrack application is always trying to align the vectors with the body. By using an accurate level to process a frame, it will move the body vector in the correct direction towards the body (although the vector might still be not aligned with the body). If the $(N - 1)$th frame and the $N$th frame have a relatively large difference, this directly indicates that the current body vector is too far away from the actual body so that the accurate frame adjusts the body vector by a large step. If the $(N - 1)$th frame and the $N$th frame have a relatively small difference, this indicates that the body vector is close enough to the actual body. In the ideal case, the difference of the last two frames is 0 and means that the body is being accurately tracked.

**Checkpointing:** QoS monitoring and checkpointing are performed after $N$ frames are processed. bodytrack also requires checkpointing for the same reasons as Category 1 applications. Multiple checkpoints can be required because a tolerable error such as losing track of single appendages during one interval may lead to an unrecoverable twisting of orientation during later intervals (Figure 3.4a). Multiple checkpoints allow BROAD to rollback several intervals to correct catastrophic errors even if the individual intervals do not violate their QoS requirement. Identification of intervals containing catastrophic errors (such as unrecoverable twisting of orientation) is difficult and we explore practical algorithms to identify critical checkpoints (i.e., checkpoints immediately preceding intervals containing catastrophic errors—see Section 6.2). Determining approximation levels for each interval of execu-
tion and after a rollback is still a challenge.

5.3 Category 3: No Checkpoint/Rollback

Not all applications amenable to approximation require checkpoint/rollback to meet QoS requirements. These applications have a “self-control” ability to perform just enough computation. The structures of these applications are usually similar. The maximum number of iterations, or the difference between two adjacent iterations for convergence, or another metric can serve as the signs for enough computation. If the number of iterations is smaller than the maximum number of iterations and the difference between the last two iterations is larger than the convergence criterion, more iterations will be executed. Examples of such applications include srad [22], pagerank [23] and swaptions [21].

5.3.1 Srad

Srad (speckle reducing anisotropic diffusion) is an application that reduces speckles and other noise in images from medical ultrasound and radar applications. The goal of srad is to remove speckles without destroying important features of the image. It performs image extraction, continuous iterations over the image and image compression [22].

**Approximation Knobs:** srad improves the result by iterating a large loop in the main function. A way to approximate this application is by perforating this large loop.

**QoS Metric and Monitor:** The goal of this application is to iteratively process the image and reduce the variance as the speckles diffuse. Further iteration can always help to improve the QoS of ferret. The QoS metric is the variance in the pixel intensities in the image. The QoS monitor computes the variance of pixel intensities and checks whether it has met the requirement.

**Checkpointing:** We can turn off checkpointing for srad since the application is self-controllable, which means that enough work is done when the termination condition is met. The termination condition can be either the maximum number of iterations or the threshold of variance of pixel intensities.
5.3.2 Pagerank

Pagerank is an application used by Google to assess the importance of web pages. The core computation is matrix multiplication that updates the page importance vector in every iteration. The application will stop when the convergence criterion is met or when the number of iterations has reached a threshold. Each iteration the pagerank, $I$, is updated according to $I = H \times I$, where $H$ is a matrix representing the correlations between pages. The more iterations, the closer $I$ will be to the real importance vector.

**Approximation Knobs:** The approximation knobs are convergence value and maximum number of iterations. These knobs determine the amount of computation. There is a diminishing return for the amount of computation performed. Smaller value for both parameters results in improved performance and degraded QoS.

**QoS Metric and Monitor:** The QoS monitor can use the simple metric “convergence,” which is defined as the difference between the result of the previous interval and the current interval. If the value is smaller than the QoS requirement, the application will stop.

**Checkpointing:** No checkpointing is required because the intermediate interval outputs are never used and simply adding further intervals reduces the approximation error. Determining the approximation level dynamically is straightforward—if QoS is not met, more computation is required.
CHAPTER 6

BROAD: BOLD AND RELIABLE ONLINE APPROXIMATE COMPUTING FRAMEWORK FOR DIVERSE APPLICATIONS

6.1 BROAD Framework

We propose BROAD, a bold and reliable online approximate computing framework. Fundamentally, BROAD treats approximate computing as speculation by taking system-level checkpoints before intervals of approximate execution. BROAD can then select approximation levels aggressively, relying on the rollback to fix violations due to approximation. If QoS fails to satisfy the requirement, BROAD will roll back to the last checkpoint, change the approximation level, and restart execution until QoS is satisfied. Otherwise, BROAD will take a checkpoint, increase the approximation level depending on how close to the QoS requirement the last interval was, and continue execution. In BROAD, we use the Berkeley Lab Checkpoint/Restart library [24]. This checkpoint/rollback enables dependable end-to-end QoS throughout the entire execution of an application depending on the quality monitor provided by the application programmer. BROAD applies a control scheme to pick approximation levels dynamically. The control scheme is based on Equation 6.1. The amount of change in the approximation level is proportional to the value of $\text{difference}_{\text{QoS}}$, which is the difference between current QoS and QoS requirement. $\alpha$ is a scaling factor which is determined by the significance of effect on QoS by changing the approximation level.

BROAD requires some domain-specific knowledge from the user or application developer. First, BROAD requires the application developer to identify approximation knobs within the application. Approximation knobs are algorithmic techniques that provide the trade-off in QoS and performance (i.e., improving QoS results in longer runtime while shorter runtime results in degraded QoS). An approximation knob can be an application parameter that changes the application configuration or can be another software approxima-
tion technique such as loop perforation that drops iterations within loops. Second, BROAD requires a QoS metric, which is used to accurately evaluate the quality of the output. The QoS metric is used during profiling to order the candidate approximation levels (see Section 6.3). Third, BROAD requires a QoS monitor to estimate or evaluate the quality of intermediate application outputs online (i.e., throughout the execution). The QoS monitor must have low performance overhead and operates on intermediate outputs, so the method used by the QoS monitor may need to be approximate. Intermediate QoS is checked by the QoS monitor after every interval of the application. An interval is defined by the application programmer’s insertion of API calls to the QoS monitor.

Figure 6.1: Modification of applications in BROAD.

Figure 6.1 shows the modification of an example application to work in BROAD. The overview of functions provided by the application developer and BROAD library are shown in Figure 6.2. A detailed description of functions in BROAD is provided in Figure 6.3. The function \texttt{initApprox} is used to read in profiling data containing a set of approximation levels sorted in the order of increasing QoS. \texttt{initApprox} maps each approximation level into a configuration of knobs. The application programmer provides the \texttt{sample} function that determines when the QoS should be checked (an interval end may be based on an iteration count or any arbitrary event). At the end of each interval, the application calls \texttt{qosMonitor} function to get the current QoS. Since BROAD uses the BLCR library which is system level checkpoint/restart, it needs to store the QoS and approximation
level to files \((\text{store}QoS)\) before taking a checkpoint or rollback. The stored QoS is used to retain a record of the QoS of the approximation level before performing a process-level rollback. After rolling back, the framework will read the previous QoS and adjust the approximation level accordingly. The function \(\text{checkpointRollback}\) calls the \text{cr\_checkpoint} function from BLCR to take a checkpoint or the \text{cr\_restart} function to rollback. If \(\text{numCheckpoint}\) provided by the application programmer equals 0, the application simply requires more computation without checkpoint/rollback (a la Green). If \(\text{numCheckpoint}\) equals to 1, the application rolls back to the last checkpoint using \(\text{BLCR}\) library. If \(\text{numCheckpoint}\) is larger than 1, the application has the option to be rolled back to a further checkpoint (critical checkpoint) to recover from devastating errors.

At the heart of BROAD’s dynamic selection of approximation level, function \(\text{changeApproxLevel}\) selects the approximation level for the following interval. \(\text{changeApproxLevel}\) reads the latest QoS from the QoS file and adapts approximation level based on the difference between the latest QoS and target QoS. The approximation level is changed based on the linear control scheme described by Equation 6.1. \(\text{upperQoS}\) and \(\text{lowerQoS}\) denote the thresholds of switching approximation levels. If QoS is between \(\text{upperQoS}\) and \(\text{lowerQoS}\), the approximation level is not changed. If the QoS is above \(\text{upperQoS}\), \(\text{changeApproxLevel}\) decreases the approximation level according to the control scheme while \(\text{changeApproxLevel}\) increases the

---

**BROAD library:**

```c
#define BROADLIBRARY_API __declspec(dllexport)

BROADLIBRARY_API void initApproxF(File* profileFile, int** knobs, startApproxLevel, numApproxLevel);
BROADLIBRARY_API void storeQoS(currentQoS, currentApproxLevel);
BROADLIBRARY_API void checkpointRollback(numCheckpoint, targetQoS, currentQoS, numApproxLevel);
BROADLIBRARY_API void changeApproxLevel(alpha, targetQoS, upperQoS, lowerQoS, numCheckpoint, );
```

**Application Developer/User:**

```c
bool sample (void *)
float qosMonitor(int argc, char** argv)
```

Figure 6.2: Overview of functions provided by the BROAD library and application user.
Figure 6.3: Functions provided by BROAD.

BROAD library:
restart.sh
restart_critical.sh
checkpoint.sh
broad.h

void initApproxF(File* profileFile, int** knobs, startApproxLevel, numApproxLevel){
    // read the profileFile and create an array that contains the
    configuration of knobs for each approximation level
}
void storeQoS(currentQoS, currentApproxLevel){
    // write current QoS to a file (File A)
    // write currentApproxLevel to a file (File B)
}
void checkpointRollback (numCheckpoint, targetQoS, currentQoS, numApproxLevel) {
    if(numCheckpoint==1){
        if(QoS meets requirement or flag==0){
            // take a checkpoint, flag=0 indicates taking the initial
            checkpoint before execution
        }else{
            // roll back to the last checkpoint
        }
    }else{
        // roll back to the last checkpoint
    }
}
void changeApproxLevel (alpha, targetQoS, upperQoS, lowerQoS, numApproxLevel, numCheckpoint){
    // open File A to get currentQoS and File B to get
    currentApproxLevel
    readApproxLevel=currentApproxLevel;
    if(currentQoS>upperQoS and currentQoS<lowerQoS)
        currentApproxLevel=lowerApproxLevel where alpha=(targetQoS-currentQoS);
    if(currentApproxLevel==readApproxLevel)
        currentApproxLevel++;
    if(currentApproxLevel>numApproxLevel-1)
        currentApproxLevel=numApproxLevel-1;
}
approximation level if the QoS is below lowerQoS. numCheckpoint equals to 0 means that the application is self-control, as discussed in Chapter 5. For such application, the approximation level can stay the same, only more computation is needed. Finally, changeApproxLevel maps the approximation levels to a configuration of knobs. We note that other implementations of this function are possible and allow future research on approximation-centric checkpoint/rollback.

\[
approximation\_level = current\_approximation\_level + \text{Int}(\alpha \times difference\_QoS)
\] (6.1)

6.2 Parameters in BROAD

As described in Chapter 5, different applications require different configurations of BROAD. In this section, we talk about the three parameters that an application programmer can change in BROAD to accommodate the three application categories.

**Parameter 1: Number of Checkpoints** For applications which exhibit the error propagation problem, saving only the last checkpoint may not be sufficient to provide an end-to-end dependable QoS, even if an accurate algorithm is used after restarting from the checkpoint. Saving more than one checkpoint allows BROAD to trace back to an earlier checkpoint and restart from there. The problem is to find the checkpoint that contains the “catastrophic error.” We call such checkpoint *critical checkpoint*. There are many possible algorithms to identify a critical checkpoint. BROAD currently identifies critical checkpoints using the following properties: (1) the approximation level increases after the critical checkpoint and (2) the QoS worsens during the interval after critical checkpoint. This parameter allows an application developer to explore the disk-space verses performance trade-off for a particular application.

**Parameter 2: QoS Threshold** For certain applications, BROAD only needs to change the approximation level when the QoS requirement is violated. However, other applications benefit from BROAD proactively changing the approximation level before the QoS requirement is violated, thus
reducing the number of rollbacks. Application programmers can set a QoS threshold, which is lower than the QoS requirement (if lower is better). If the QoS is less than the lowerQoS threshold, BROAD will increase the approximation level. If QoS is larger than the upperQoS threshold, BROAD will decrease the approximation level. By adding a QoS threshold, BROAD allows the application programmer to explore the approximation-centric checkpoint/rollback characteristics for a particular application by balancing the frequency of rollbacks with the aggressiveness of approximation.

Parameter 3: Starting Approximation Level There exists a trade-off in the starting approximation level. Choosing an aggressive level in the beginning can achieve better performance but can rapidly accumulate errors leading to frequent rollbacks. Choosing to start with the accurate level can prevent accumulating errors in the beginning phase of the application, but it may take longer to stabilize on the ideal approximation level. The starting approximation level parameter allows the application developer to determine a good starting approximation level for each application.

6.3 Selection of Candidate Approximation Levels

BROAD requires a set of approximation levels that have different QoS and performance trade-offs as input, which means that for these approximation levels, increasing QoS results in performance degradation. Given an application with n knobs, we can use a range of values for each knob. If $Q_i$ is the number of configurations for $Knob_i$, there are $\prod_{i=1}^{n} Q_i$ approximation levels in total. However, some of the approximation levels do not lie on the Pareto optimal QoS-performance curve since there exist other levels that provide better QoS and performance. We eliminate all such non-Pareto-optimal approximation levels and select a set of candidate approximation levels for BROAD in the following way. BROAD first obtains the QoS and performance for each approximation level across a set of profiling inputs. Then BROAD marks all the levels as effective. Each approximation level is compared with all other approximation levels. If $level_A$ has better QoS and smaller runtime than $level_B$, we will mark $level_B$ as inefficient. In the end, the approximation levels that are marked as efficient are selected as the candidate approximation levels.
CHAPTER 7

EXPERIMENTS

In this section we demonstrate that BROAD achieves a significant performance improvement for each application and that it provides more dependable QoS relative to prior works due to its support of checkpoint/rollback. We also present an exploration of possible answers to the approximation-centric checkpoint/rollback questions posed in Chapter 4.

For our experiments, the runtime of BROAD for each application has incorporated all the overheads including checkpoint/restart overhead, QoS monitor and adapting approximation level overhead. The baseline for each application is the original algorithm running at the default setting (which does not use checkpointing/restart).

7.1 Application Performance with BROAD

7.1.1 Experiments with x264

We use three large raw films as the inputs to x264, which are *big buck bunny (480p24)* [25], *elephants dream (480p24)* [25] and the *native* input from the PARSEC benchmark suite [21]. We also set two different QoS requirements: 0.9 and 0.8, which means that the QoS value (calculated with Equation 5.1) is 90% or 80% of the QoS from the accurate specification. Since a higher QoS is better, 0.9 is a stricter requirement than 0.8. After using the algorithm described in Section 6.3, there are 9 approximation levels left, which are ranked from Level 0 to Level 8 with increasing QoS and runtime.

BROAD offers user the flexibility to choose any starting approximation level. To evaluate the performance of different starting approximation levels, we set three different starting approximation levels, which are Level 0, Level 4 and Level 8. Level 0 is the most aggressive approximation level with the
least computation accuracy. Level 4 is an approximation level in the middle, while Level 8 is the most accurate approximation level. Note that these are starting approximation levels used in the first interval. BROAD changes the approximation levels dynamically for the following computation based on the QoS. The two baselines of x264 are static oracle and framework with guardband. For each input, the static oracle approximation level is achieved by profiling all the approximation levels on each individual input and choosing the level that satisfies the QoS requirement with the least execution time. The results are shown in Figure 7.1, in which “broad start $x$ ($y$)” represents starting at approximation level $x$ and setting $y$ as the QoS requirement for BROAD. The runtime of BROAD is normalized to the runtime of using Level 8 throughout execution.

The second baseline is a framework with a static approximation level guardband. We use the same set of inputs from Figure 3.1 as the profiling inputs. We choose the approximation level that meets the QoS requirement with the least average execution time for all the profiling inputs. A framework with a guardband has even lower performance than the static oracle since it incorporates a guardband to work for the worst case profiling input.

We also plot the number of restarts for each configuration in Figure 7.2. Starting with Level 4 (note that only the first interval is using Level 4 while the following intervals are using levels selected by BROAD) turns out to be an efficient configuration for x264, since it has smaller runtime and no restarts. It reduces runtime by 75% compared to using only Level 8. Although most of the intervals use Level 0, the approximation level is adjusted accurately based on the QoS. BROAD has better performance than the static oracle configuration since there exist QoS variations within the application for each input. For x264, different frames of the same video may need very different approximation levels to satisfy the QoS requirement. With the support of checkpoint/rollback, BROAD picks aggressive approximation levels dynamically for any given input. Since increasing the approximation level by a small amount can result in much better performance, BROAD has overall speedup even though it may have to restart executions occasionally.
7.1.2 Experiments with streamcluster

The performance of streamcluster in BROAD is shown in Figure 7.3. We evaluate the performance of BROAD for different configurations \((\text{startLevel}, \text{targetQoS})\). \text{startLevel} is the starting approximation level for the application. \text{targetQoS} is the QoS requirement/specification which is provided by the application user. We have three different QoS requirements as shown in the Figure, T1, T2 and T3. T1 represents the target SSQ cost is \(3.5 \times 10^{13}\); T2 represents the target SSQ cost is \(2.8 \times 10^{13}\); T3 represents the target SSQ cost is \(2.1 \times 10^{13}\). A lower QoS is better for streamcluster, T3 is more strict than T2 and T2 is more strict than T1. As Figure 7.3 shows, when the QoS requirement is decreased, the runtime increases towards that of a fully
accurate execution.

Figure 7.4 shows the performance of BROAD with no checkpoint/rollback (BNCR). BNCR dynamically adapts the approximation level based on the QoS; however, like previous frameworks in approximate computing, it has no mechanism to fix QoS violations. The normalized runtime is shown for different configurations \((\text{startLevel}, \text{targetQoS})\). We also show the QoS violation rates, which are the fractions of intervals that violate the QoS requirement. The QoS violation rate can be as high as 56%. Such high violation rates suggest that for frameworks, such as BROAD, which perform approximation level speculation and eliminate the approximation level guardband, it is important to support a checkpoint/rollback scheme to fix QoS violations. Such high violation rates in the context of streaming applications also imply that designing a system that provides a dependable end-to-end QoS with optimized performance may not be feasible since it is hard to decide how large the guardband should be to guarantee a low QoS violation rate. Besides, even if a guardband works for an interval in the application, since the inputs for an application vary from time to time, a guardband that works for one interval may not work on other intervals. Therefore, a framework like BROAD is necessary, which performs aggressive approximation to improve performance of applications while providing even interval level QoS guarantees with a checkpoint/rollback mechanism.

![Figure 7.3: Performance of streamcluster using BROAD.](image-url)
Figure 7.4: Performance and QoS violation rates of BNCR applied to streamcluster.

7.1.3 Experiments with bodytrack

For bodytrack, we evaluate BROAD using different configurations (\textit{startApproxLevel}, \textit{alpha}, \textit{threshold}), where \textit{startApproxLevel} is the starting approximation level; \textit{alpha} is the scaling factor in changing approximation levels; and \textit{threshold} is the QoS value that triggers the change in approximation level, i.e., when the QoS is lower than the \textit{threshold}, the approximation level will be decreased. The value of \textit{alpha} has an effect on the performance.

The benefit of a smaller \textit{alpha} is that it changes the approximation level more conservatively. Therefore, it is less likely to jump to a level that provides much higher QoS and runtime than the target. The drawback is that only changing the approximation level once may not be sufficient to get to the correct approximation level that meets the QoS requirement. It may hurt the performance by rolling back and changing the approximation level several times. A larger \textit{alpha} enables the approximation level to change faster in both directions. But it may hurt performance when the prediction is not accurate. Figure 7.5 shows the normalized runtime of bodytrack, where approximation Level 7 (most accurate) is the baseline. If BROAD starts at the most aggressive approximation level (Level 0), the runtime is larger than starting at either Level 1, Level 5, or Level 6. Starting at Level 0 causes significant numbers of rollbacks.

In Figure 7.6, we show the performance of BROAD without checkpoint/rollback (BNCR). \textit{BNCR start n} means setting Level \textit{n} as the starting approx-
imation level. \textit{targetQoS} is set to 15 based on the qualitative manifestation of QoS described in Chapter 3. Figure 7.6 also shows the fraction of intervals which violate QoS. In streaming applications like bodytrack where the application does not know a priori which intervals’ outputs will be used by subsequent applications, high violation rates (all executions had violation rates higher than 0.5) prohibit any statistical QoS guarantee (i.e., the final QoS is more likely to be unacceptable than acceptable).

![Figure 7.5: Performance of bodytrack using BROAD.](image)

7.1.4 Experiments with ferret

We use the native input from the PARSEC benchmark suite as the input data for the experiments. For each image query, ferret returns 50 similar queries. The QoS monitor runs the last ten images in each interval using the accurate level and computes the average intersection ratio. We perforate the loop in \textit{LSH\_query} by different degrees ranging from omitting \(2/3, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8, 1/9\) of total work. BROAD will switch between these approximation levels during runtime. Figure 7.7 shows the normalized runtime of BROAD with different QoS requirements. The \textit{x} axis shows dif-
Figure 7.6: Performance and QoS violation rates of BNCR applied to bodytrack.

Different QoS requirements. A QoS requirement of 0.5 means that at least 50% of returned images from the accurate result are found in the returned images from approximated result. The baseline is the original algorithm of ferret without any modification. BROAD’s runtime has included all the overheads, including QoS monitoring overhead, checkpointing overhead and recovering overhead.

Figure 7.7: Performance of ferret using BROAD.
7.1.5 Experiments with srad

The normalized runtime of the approximated version of srad is shown in Figure 7.8. We performed two kinds of evaluations. In the first evaluation, we set the QoS (variance of pixel intensities) as the termination condition. The application will be terminated once the QoS meets the QoS requirement. In the second evaluation, we set the maximum number of iterations as the termination condition. Therefore, the application will be terminated when the num_iteration in the main loop meets the maximum value. For the approximated version of the algorithm, we use loop perforation to skip 80% of the iterations. The runtime of the approximated version of srad is normalized to the original version of srad. As shown in Figure 7.8, approximation does not help improve performance of srad if the termination constraint is the QoS. In the end, the approximated version of the application and the original version of the application perform about the same amount of work to achieve the same QoS. If the QoS is not a strict requirement, the approximated version of srad can reduce the amount of computation by loop perforation.

Figure 7.8: Runtime of srad.

7.2 Checkpoint Overhead

BROAD provides checkpointing-rollback scheme to achieve dependable end-to-end QoS under aggressive approximation speculation. As discussed above,
BROAD’s runtime incorporates the overhead of checkpointing. Although we are not using the most lightweight checkpointing scheme, BROAD still demonstrates significant performance improvement. In this section, we will examine the additional overhead of checkpointing/rollback on the performance of applications. We measure two different kinds of checkpointing performance overhead. The first is updating the checkpoint when the QoS requirement is met. The second is killing the process and restarting from the previous checkpoint when the QoS requirement is not met. The performance and storage overhead of checkpointing for different applications are shown in Table 7.1. Only applications in Category 1 and Category 2, which require checkpoints, incur the overhead of checkpointing. The applications in Category 3 have no overhead from checkpointing (as shown by the last two rows in Table 7.1).

The overhead of updating a checkpoint is relatively small, which is between 10 and 20 ms for x264, streamcluster and bodytrack and 410 ms for ferret. The overhead of restarting from a checkpoint is relatively large, which is approximately 1 second for the four applications. These overheads are not a big concern considering the application runtime and the not-so-often restarts. The storage overhead is different for different applications. Streamcluster, as we discussed previously, processes around 1 million data points in each interval and this has an effect on the size the checkpoint.

<table>
<thead>
<tr>
<th>Application</th>
<th>Update Checkpoint</th>
<th>Restart from Checkpoint</th>
<th>average % of overall execution time</th>
<th>Storage Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>x264</td>
<td>11.5ms</td>
<td>972ms</td>
<td>0.88%</td>
<td>39MB</td>
</tr>
<tr>
<td>streamcluster</td>
<td>18.6ms</td>
<td>986ms</td>
<td>2.44%</td>
<td>246MB</td>
</tr>
<tr>
<td>bodytrack</td>
<td>10.2ms</td>
<td>994ms</td>
<td>1.69%</td>
<td>34MB</td>
</tr>
<tr>
<td>ferret</td>
<td>410ms</td>
<td>990ms</td>
<td>3.31%</td>
<td>104MB</td>
</tr>
<tr>
<td>srad</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pagerank</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

7.3 Exploration of Interval Sizes

There are several factors that determine how interval sizes affect the performance of applications:
1. With a smaller interval size, there is more randomness in the QoS at the end of each interval. Therefore, there are likely to be more QoS requirement violations and more restarts that decrease the performance.

2. With a smaller interval size, the overhead of checkpointing/restart and computing QoS is relatively large compared to the runtime of the interval.

3. With a large interval size, each interval applies a certain approximation level for a long time. If the QoS requirement is met, it is possible that the application is stuck with a higher level (with worse performance) for too long. If the QoS requirement is not met, the recovery overhead of the application is higher.

4. The most significant factor that affects the performance is the number of restarts. The number of restarts can be difficult to estimate since it is decided by whether the QoS is met at the end of each interval and how far away the QoS is from the target (which affects BROAD’s selection of approximation level).

We explored the performance of different applications at different interval sizes. Figure 7.9 shows the performance of x264 with different sizes varying from 100 frames to 14000 frames. The input we are using is big buck bunny [25] which contains 14000 frames in total. We normalize the runtime to the oracle runtime. The oracle is the ideal case that has preselected optimized approximation levels for each interval in such a way that the application meets the end-to-end QoS and has the best performance. Note that the oracle does not have the overhead of checkpointing and rollback. In Figure 7.9, for each interval size, we show the number of changes in the approximation levels, the total number of restarts in runtime, and the runtime normalized to the oracle case. With a larger interval size (for example 2500), the application can avoid restarts by adapting the approximation levels. This takes advantage of BROAD’s control mechanism to carefully predict an approximation level for the next interval. However, when the interval size increases further, the benefit is reduced since the application may stick to one approximation level for too long. On the other hand, with a smaller interval size, BROAD encounters more restarts and adapts the approximation level more frequently.
Figure 7.10 shows the performance of streamcluster with interval sizes ranging from 0.2 million to 5 million data points. The interval size corresponds to chunksize in streamcluster. All the data in an interval is considered as a chunk during computation. The runtime is normalized to the oracle case’s runtime. The oracle is based on interval size of 1 million data points and it applies the most efficient combination of approximation levels which achieves the best performance. From Figure 7.10, we observe that as the interval size grows, the runtime increases. This is because with a larger interval size, the computation time for each interval is much longer and does not grow linearly with the interval size. Therefore, the performance decreases with a larger interval size.

Figure 7.11 shows the runtime of bodytrack with different interval sizes. We use the native input from the PARSEC benchmark suite, which contains 260 frames. For bodytrack, the runtime for interval size equal to 10 is much higher than the other configurations. This is because the cost of the QoS monitor is higher and the number of restarts is larger. Similar to the above two applications, the runtime is normalized to the oracle case where the most efficient combination of approximation levels is applied.

For the potential of BROAD’s performance: although BROAD improves the performance by approximately 50% on average compared to the original version of the algorithm (which applies the default settings and has no checkpoint/restart overhead), it still has a performance degradation compared to
The most important factor determining the performance against the oracle is whether the speculation on the approximation level is globally best. Speculating conservatively can result in a higher than the optimal approximation level that does not have the best global performance. Speculating aggressively can result in violation of the QoS and requires a restart. The secondary factors that determine performance against the oracle include the overheads of the QoS check, checkpointing, and restarts.
CHAPTER 8

CONCLUSIONS

Approximate computing systems must satisfy user-provided requirements for quality of service (QoS), a quantitative criterion imposed on the output of an application such that the output is qualitatively useful. We propose BROAD, a Bold and Reliable Online Approximate Computing Framework for Diverse Applications. BROAD explicitly provides a checkpoint/rollback mechanism to allow applications to recover from QoS violations and error accumulation. The checkpoint/rollback mechanism further saves BROAD from having a static approximation level guardband by allowing BROAD to operate near the QoS requirement without concern for permanent QoS degradation. In the context of several applications with differing responses to checkpoint/rollback, we demonstrate that BROAD can provide dependable end-to-end QoS corresponding to qualitatively acceptable outputs, while increasing performance by at least 4x over accurate execution.
REFERENCES


