Green: A System for Supporting Energy-Conscious Programming using Principled Approximation

Woongki Baek
Stanford University
wkbaek@stanford.edu

Trishul M. Chilimbi
Microsoft Research
trishulc@microsoft.com

Abstract

Energy-efficient computing is important in several systems ranging from embedded devices to large scale data centers. Several application domains offer the opportunity to tradeoff quality of service/solution (QoS) for improvements in performance and reduction in energy consumption. Programmers sometimes take advantage of such opportunities, albeit in an ad-hoc manner and often without providing any QoS guarantees.

We propose a system called Green that provides a simple and flexible framework that allows programmers to take advantage of such approximation opportunities in a systematic manner while providing statistical QoS guarantees. Green enables programmers to approximate expensive functions and loops and operates in two phases. In the calibration phase, it builds a model of the QoS loss produced by the approximation. This model is used in the operational phase to make approximation decisions based on the QoS constraints specified by the programmer. The operational phase also includes an adaptation function that occasionally monitors the runtime behavior and changes the approximation decisions and QoS model to provide strong QoS guarantees.

To evaluate the effectiveness of Green, we implemented our system and language extensions using the Phoenix compiler framework. Our experiments using benchmarks from domains such as graphics, machine learning, and signal processing, and a real-world web search application, indicate that Green can produce significant improvements in performance and energy consumption with small and statistically guaranteed QoS degradation.

1 Introduction

Companies such as Amazon, Google, Microsoft, and Yahoo are building several large data centers containing tens of thousands of machines to provide the processing capability necessary to support web services such as search, email, online shopping, etc. [3]. Not surprisingly, power is a large component of the monthly operational costs of running these facilities and companies have attempted to address this by locating them in places where power is cheap [10]. In addition, energy is often a key design constraint in the mobile and embedded devices space given the current limitations of battery technology.

There are several application domains where it is acceptable to provide an approximate answer when the cost and resources required to provide a precise answer are unavailable or not justified. For example, real-time ray tracing is infeasible on current PCs so games employ a variety of techniques to produce realistic looking lighting and shadows while still rendering at 60 frames per second. Content such as images, music, and movies are compressed and encoded to various degrees that provide a tradeoff between size requirements and fidelity. Such approximations typically result in the program performing a smaller amount of processing and consequently consuming less energy while still producing acceptable output.

Programmers often take advantage of such Quality of Service (QoS) tradeoffs by making use of domain-specific characteristics and employing a variety of heuristics. Unfortunately, these techniques are often used in an ad-hoc manner and programmers rarely quantify the impact of these approximations on the application. Even in cases where they attempt to quantify the impact of the heuristics used, these are hard to maintain and keep up-to-date as programs evolve and add new features

and functionality.

To address these issues, this paper proposes Green, which is a system for supporting energy-conscious programming using loop and function approximation. Green provides a simple, yet flexible framework and programming support for approximating expensive loops and functions in programs. It allows programmers to specify a maximal QoS loss that will be tolerated and provides statistical guarantees that the application will meet this QoS target. Programmers must provide (possibly multiple) approximate versions of the function for function approximation. Unless directed otherwise, Green uses the function return value as the QoS measure and computes loss in QoS by comparing against the value returned by the precise function version given the same input. Loops are approximated by running fewer loop iterations. In this case, a programmer must provide a function that computes QoS to enable Green to calculate the loss in QoS that arises from early loop termination.

Green uses this information to construct a "calibration" program version. Green runs this application version with a programmer-provided set of calibration inputs to construct a QoS model that quantifies the loss in QoS that results from using the approximate version of the function or loop and the corresponding improvement in application performance and energy consumption. Green then generates an "approximate" version of the program that uses this QoS model in conjunction with the programmer supplied QoS target to determine when to use the approximate version of the function or terminate the loop early while still meeting the QoS requirement. Since the QoS degradation on actual program inputs may differ from that observed during calibration, Green also samples the QoS loss observed at runtime and updates the approximation decision logic to meet the specified QoS target. In this way, Green attempts to provide statistical guarantees that the specified QoS will be met. Such statistical guarantees are becoming important as cloud-based companies provide web services with Service Level Agreements (SLAs) that typically take the form: "the service will provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second" [7].

Our experimental results indicate that Green can significantly improve performance and reduce energy consumption of several applications with only a small degradation in QoS. In particular, we improved the performance and reduced the energy consumption of `Live Search`, a back-end implementation of a commercial web-search engine by 22.0% and 14.0% respectively with 0.27% of QoS degradation (three queries out of a thousand returned a search result that included at least one different document or the same documents in a different rank order). We also empirically demonstrate that Green can generate a robust QoS model with a relatively small training data-set and that runtime re-calibration can enable applications to meet their QoS targets even if they use imperfect QoS models.

This paper makes the following main contributions.

- Describes the design and implementation of the Green system, which provides simple, yet flexible support for energy-conscious programming using function and loop approximations.
- Experimental evaluation of the Green system that shows significant improvements in performance and energy consumption with little QoS degradation.
- Experimental evidence that indicates that Green's QoS modeling is robust and that its adaptation supports meeting target QoS requirements.

The rest of the paper is organized as follows. Section 2 describes the design of the Green system. Section 3 discusses our implementation of Green. Experimental evaluation of Green is described in Section 4. Section 5 provides a brief overview of related work.

2 Green Design

Figure 1 provides a high-level overview of the Green system that we introduce and discuss in more detail in this section.

2.1 Principled Program Approximation

Expensive functions and loops present attractive targets for program approximation because they are modular and time-consuming portions of programs. We consider function approximations that use an alternative, programmer-supplied approximate version of the function and loop approximations that terminate the loop earlier than the base (precise) version. But we would like to quantify the impact these approximations have on the QoS of the program. First, the programmer must supply code to compute a QoS metric for the application. Next, we require a calibration phase where the program

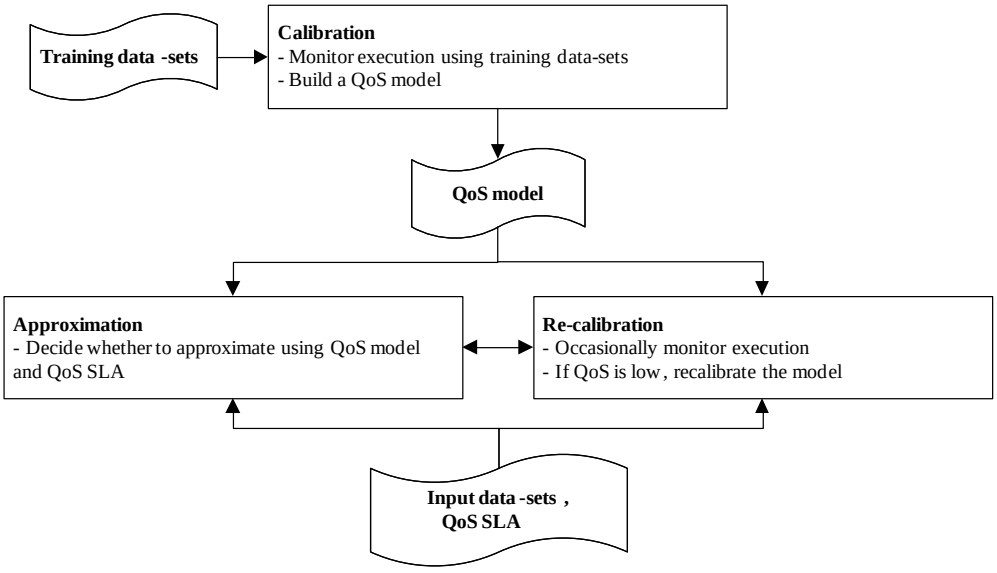

Figure 1. Overview of the Green system.

is run with a collection of training inputs. During these training runs, we monitor and record the impact function or loop approximation has on the program's QoS and its performance and energy consumption. This data is used to build a QoS model that is subsequently used by Green to decide when to approximate and when to use the precise version in order to guarantee user-specified QoS Service Level Agreements (SLAs). Figure 2 illustrates at a high-level how Green approximates loops or function while still attempting to meet required QoS SLAs. The QoS model constructed in the calibration phase is used by QoS_Approx() to decide whether approximation is appropriate in the current situation as determined by the function input or loop iteration count. Since the QoS degradation on the actual program inputs may differ from that observed during the calibration phase, Green provides a mechanism to occasionally measure the program's QoS and update the QoS approximation decisions at runtime.

2.2 Green Mechanisms

As described, Green requires a QoS_Compute() function for computing the program's QoS, a QoS_Approx() function for determining whether to perform an approximation, and a QoS_ReCalibrate() function for revisiting approximation decisions at runtime. In addition, it requires a set of training inputs to construct the QoS model and a QoS_SLA value that must be met. We provide a high-level description of these mechanisms here and leave a detailed discussion to the next section.

2.2.1 QoS Calibration and Modeling

Green's calibration phase collects the data required to build the QoS model. It requires QoS_Compute(), which is application dependent and can range from trivial as in the case of using the approximated function's return value to a complex computation involving pixel values rendered on the screen. Green uses this function along with the set of training inputs to construct a QoS model that relates function inputs in the case of function approximation and loop iteration count in the case of loop approximation to loss in QoS and performance and energy consumption improvements. This QoS model is used in conjunction with a provided target QoS SLA to make approximation decisions.

2.2.2 QoS Approximation

QoS_Approx() comes in two main flavors for loop approximations. In the static variety, the approximation is solely determined by the QoS model constructed in the calibration phase. Here the loop iteration count threshold is determined by the QoS model and the user-specified QoS SLA. Once the loop iteration count exceeds this threshold, the approximation breaks out of the loop. The adaptive variety is based on the law of diminishing returns. Here the approximation uses the QoS model in con-

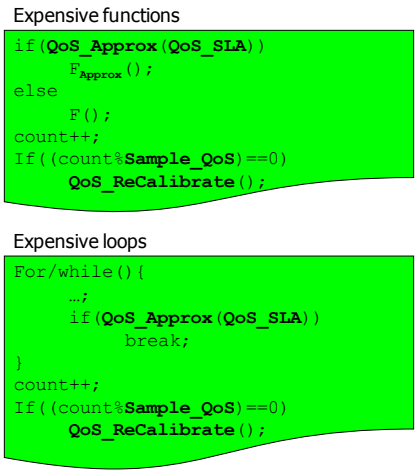

Figure 2. High-level overview of code generation.

junction with the QoS SLA to determine appropriate intervals at which to measure change in QoS and the amount of QoS improvement needed to continue iterating the loop. For function approximation, the QoS model is used in conjunction with the QoS SLA and an optional function input parameter range to determine which approximate version of the function should be used.

2.2.3 QoS Re-Calibration

The program's behavior may occasionally differ from that observed on its training inputs and QoS_ReCalibration() provides a mechanism to detect and correct for this effect. In the case of loop approximation, when used with static approximation re-calibration can update the QoS model and increase the loop iteration count threshold to compensate for higher than expected QoS degradation or decrease this threshold to improve performance and energy consumption when QoS degradation is lower than expected. Similarly, when used with adaptive approximation re-calibration can appropriately change the interval used to measure change in QoS and/or QoS improvement required to continue. For function approximation, re-calibration allows Green to switch the approximate version of the function used to one that is more or less precise as determined by the observed QoS loss.

2.2.4 Discussion

Since, as shown in Section 4, Green's re-calibration mechanism is quite effective, one might underestimate the importance of the calibration phase and attempt to solely rely on the re-calibration mechanism. However, the calibration phase is still important and necessary because it provides (1) faster convergence to a good state, (2) reliable operation even when users choose to avoid or minimize re-calibration to lower overhead, and (3) programmer insight into the application's QoS tradeoff through the QoS model.

3 Green Implementation

This section describes our implementation of the Green system that provides a simple and flexible framework for constructing a wide variety of approximation policies (see Figure 3 for overview). Our goal is to provide a minimal and simple interface that satisfies the requirements of the majority of programmers while providing the hooks that allow expert programmers to craft and implement custom, complex policies. To achieve this, Green comes with a couple of simple, default policies that meet the needs of many applications and enables them to benefit from using principled QoS approximation with minimal effort. At the same time, it allows programmers to override these default policies and supply their own by writing custom versions of QoS_Approx() and QoS_ReCalibrate. We discuss Green's interface and default policies and provide an instance of a customized policy.

3.1 Green Programming Support

3.1.1 Loop Approximation

Green supports loop approximation with a new keyword approx_loop as shown in Figure 4. The programmer uses approx_loop just before the target loop and supplies a pointer to a user-defined QoS_Compute() function, the calibration granularity (calibrate_QoS) for building the loop QoS model, the value of the desired QoS_SLA and indicates whether they want to use static or adaptive approximation. In addition, if the programmer wants to avail of runtime re-calibration he/she must provide the sampling rate (sample_QoS) to perform re-calibration. If a programmer wishes to construct a custom policy, they must also supply pointers to custom QoS_Approx() and/or QoS_ReCalibrate() routines.

3.1.2 Function Approximation

For function approximation, Green introduces an approx_function keyword as shown in Figure 6. The

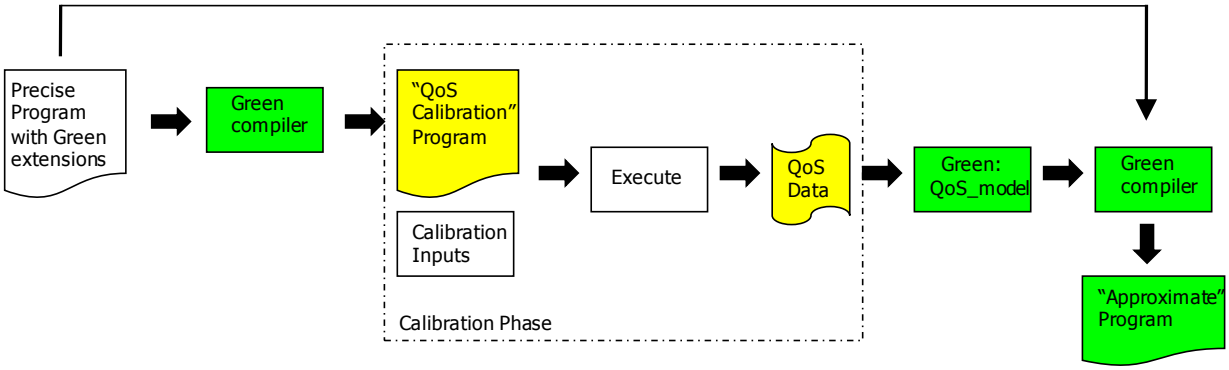

Figure 3. Green implementation overview.

programmer uses approx_function just before the target function implementation and supplies a function pointer array that contains pointers to user-defined approximate versions of that function in increasing order of precision, along with the value of the desired QoS_SLA and a sampling rate (sample_QoS) if re-calibration is required. If the function return value does not provide the desired QoS metric, the programmer must also supply a pointer to a custom QoS_Compute() function. A custom input range for function argument values can also be provided to override the default of <-Inf,+Inf>. If the function takes multiple arguments, Green requires the parameter positions of the arguments that should be used while building the QoS model. As with approx_loop, the rest of the arguments are optional and only needed for custom policies.

3.2 Green System Implementation

Figure 3 provides a high-level overview of the Green system implementation. The Green compiler first generates a "calibration" version of the program that is run with user-provided calibration inputs to generate QoS data needed to build the QoS model. Then the compiler uses this constructed QoS model to generate an "approximate" version of the program that can be run in place of the original. It synthesizes code to implement QoS_Approx() and QoS_ReCalibrate().

3.2.1 Loop Approximation

Figure 4 shows pseudo-code generated by the Green compiler from the original code with the approx_loop keyword. The programmer-supplied QoS_Compute() function is used in the calibration phase to tabulate the loss in QoS resulting from early loop termination at loop iteration counts specified by Calibrate_QoS. The QoS_Compute() function has the following interface: QoS_Compute (return_QoS, loop_count, calibrate, Calibrate_QoS, ...) and the search application's version is shown in Figure 5. Note that when QoS_Compute() is called with return_QoS unset it stores the QoS computed at that point and only returns QoS_loss when this flag is set. Then, it compares the current QoS against the stored QoS to return the QoS loss. When it is called with the calibrate flag set at the end of the loop in calibration mode, it computes and stores the % QoS loss when the loop terminates early at loop iteration counts specified by Calibrate_QoS.

This calibration data is then used by Green's QoS modeling routine that is implemented as a MATLAB program and supports the following interface for loops:

$$M = QoS\_Model\_Loop(QoS\_SLA, static) \quad (1)$$

$$< Period, Target\_Delta >= QoS\_Model\_Loop(QoS\_SLA, adaptive) \quad (2)$$

For static approximation the QoS model supplies the loop iteration count that is used by QoS_Approx() for early loop termination. In the adaptive approximation case, the QoS model determines the period and target QoS improvement required to continue iterating the loop.

The synthesized QoS_Approx() code shown uses the parameters generated by the QoS model to perform approximation. When the approximation is being re-calibrated, QoS_Approx() stores the QoS value that would have been generated with early loop termination and continues running the loop as many times as the original (precise) program version would have in order to compute the QoS loss.

The QoS_ReCalibrate() code generated by Green compares this QoS loss against the target QoS SLA

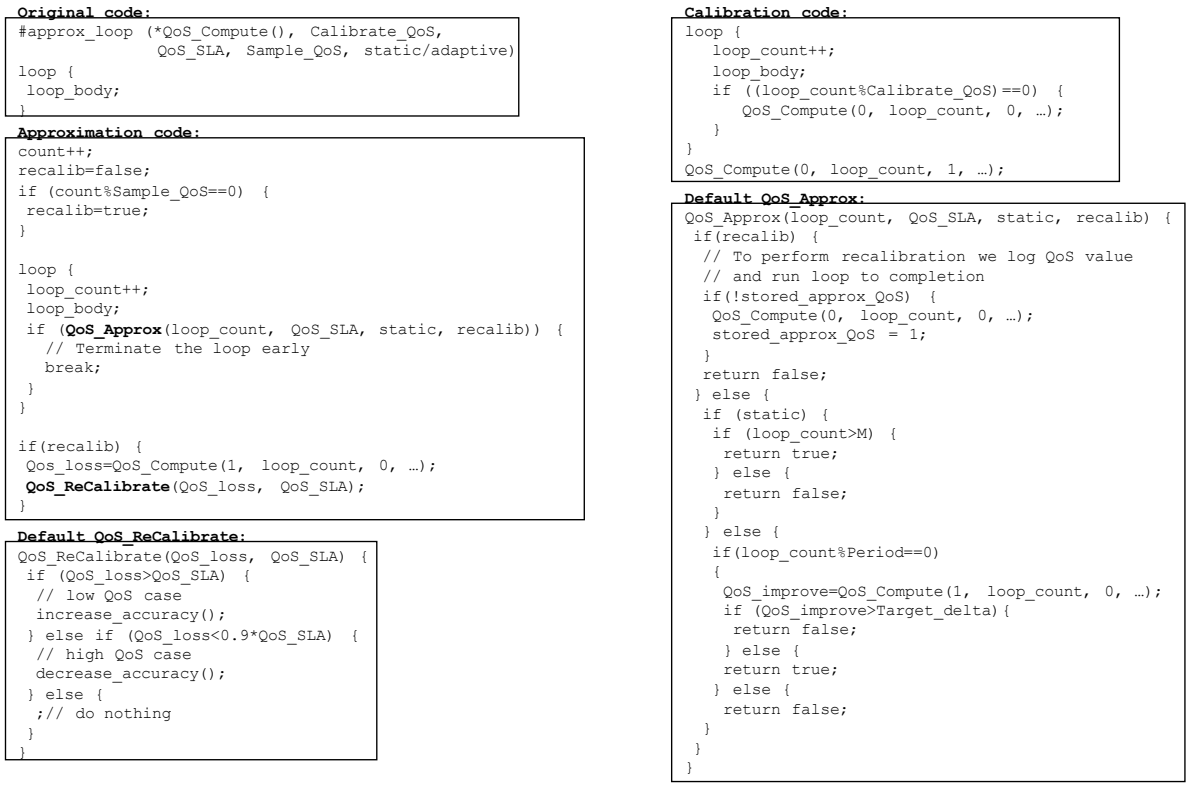

Figure 4. Code generation for loop approximation

and either decreases/increases the approximation by either increasing/decreasing the value M of the early termination loop iteration count (static approximation) or decreasing/increasing the value of Target_Delta (adaptive approximation).

3.2.2 Function Approximation

Figure 6 shows pseudo-code generated by the Green compiler for function approximation specified using the approx_func keyword. By default, Green uses the function return value to compute QoS unless the programmer defines a separate QoS_Compute() function. In the default case, Green also assumes the function is pure without any side-effects. For calibration, Green generates code shown that computes and stores the loss in precision that results from using the family of approximation functions at each call site of the function selected for approximation. We do not currently approximate call sites that use function pointers.

Green's modeling routine uses this data and supports the following interface for functions:

$$< M, approx\_func >= QoS\_Model\_Func(QoS\_SLA, opt\_input\_arg\_range)$$

where it returns the most approximate function version whose worst-case QoS loss satisfies the specified target QoS SLA (within the optional function input argument range, if specified). In the event that none of the approximate function versions meet the QoS requirement, approx_func is set to false.

The generated QoS_Approx() function for the default case is trivial and returns the value of approx_func. Green's QoS_ReCalibrate() function replaces the current approximate function version with a more precise one, if available, to address low QoS, and uses a more approximate version to address higher than necessary QoS.

3.3 Custom Approximation

Green allows programmers to override its default synthesis of QoS_Approx() and QoS_ReCalibrate()

Original code:

```
#approx_function (…(*F_approx[n])(…), QoS_SLA, Sample_QoS)
… F (…) {
 …;
}
```

Approximation code:

```
if (++count%Sample_QoS==0) {
 recalib=true;
}

if (QoS_Approx()) {
 … = (*F_approx[M])(…);
 if(recalib) {
  QoS_funcM = …;
  QoS_precise = F(…);
  QoS_loss = 100*(abs(QoS_precise-QoS_funcM))/QoS_precise);
  QoS_ReCalibrate(QoS_loss, QoS_SLA);
 }
} else {
 … = F(…);
}
```

Default QoS_Approx:

```
QoS_Approx() {
 return approx_func;
}
```

Calibration code:

```
foreach approx func {
 QoS_func[i] = (*F_approx[i])(…);
}
QoS_precise = F(…);
for i = 0..(n-1) {
 store(…, i, 100*(QoS_precise-QoS_func[i])/QoS_precise);
   }
}
```

Default QoS_ReCalibrate:

```
QoS_ReCalibrate(QoS_loss, QoS_SLA){
 if (QoS_loss>QoS_SLA) {
  // low QoS case
  if(++M >= n) {
   M--;
   approx_func = false;
  }
 } else if (QoS_loss<0.9*QoS_SLA) {
  // high QoS case
  if(--M < 0) {
   M = 0;
   approx_func = true;
  }
 } else {
  ;// do nothing
 }
 recalib = false;
}
```

Figure 6. Code generation for function approximation

QoS_Compute for GMY Search:

```
QoS_Compute(return_QoS, loop_count, calibrate,
                                calibrateQoS)
{
 if(!calibrate)
 {
  if(!return_QoS) {
   Store_top_N_docs(loop_count);
   return -1;
  } else {
   if(Same_docs(Get_top_N_docs(loop_count),
              Current_top_N_docs()) {
    return 0;
   } else {
    return 1;
   }
  }
 } else {
  …;
 }
}
```

Figure 5. QoS_Compute for `Live Search`

Customized QoS_ReCalibrate for GMY Search:

```
QoS_ReCalibrate(QoS_loss, QoS_SLA) {
 // n_m: number of monitored queries
 // n_l: number of low QoS queries in monitored queries
 if (n_m==0) {
   // Set Sample_QoS to 1 to trigger QoS_ReCalibrate
   // for the next 100 consecutive queries
   Saved_Sample_QoS=Sample_QoS;
   Sample_QoS=1;
 }
 n_m++;
 if (QoS_loss !=0) {
  n_l++;
 }
 if (n_m==100) {
  QoS_loss=n_l/n_m;
  if(QoS_loss>QoS_SLA) {
   // low QoS case
   increase_accuracy();
  } else if (QoS_loss < 0.9*QoS_SLA) {
   // high QoS case
   decrease_accuracy();
  } else {
   ; // no change
  }
  Sample_QoS=Saved_Sample_QoS;
 }
}
```

Figure 7. Customized QoS_Recalibration for `Live Search`

to implement custom approximation policies. Figure 7 shows an example of a custom QoS_ReCalibrate() that we used for the Search application. For Search, QoS_Compute() returns a QoS loss of 1 (100%) if the top N documents returned by the precise and approximate version do not exactly match and a QoS loss of 0 otherwise. To perform re-calibration and compare against a target QoS SLA that is of the form, "the application returns identical results for 99% of the queries", we need to measure the QoS loss across multiple queries as implemented in the code shown. The calibration and QoS_Approx() code is the same as the default code synthesized by Green.

3.4 Discussion

We believe Green is a well-engineered and novel approach to an important problem that goes a long way towards reducing the programmer burden but

there is still work to be done. Currently, the programmer must first profile the application to identify function and loop-level hotspots. For expensive functions, she must provide one/many approximate versions of the function, calibration test inputs, and an initial QoS_SLA value (the function return value typically serves as the QoS_Compute() function). For expensive loops, she needs to provide a QoS_Compute() function, calibration test inputs and an initial QoS_SLA. The QoS_model data generated by the calibration phase can be used to determine an appropriate QoS_SLA to replace the initial QoS_SLA. Typically, the default QoS_Appox() code can be used along with a user-provided sample_QoS value for re-calibration (using default QoS_Recalibrate()).

While there are still important open research problems related to further reducing programmer burden through more automated program approximation, the Green system provides a framework to pose these questions and serves as a first-step towards the goal of principled approximation. We hope it will generate discussion and debate in an important and relatively unexplored research direction.

4 Green Evaluation

We performed two types of experiments. First, we show that Green can produce significant improvements in performance and reduction in energy consumption with little QoS degradation. Next, we show that the QoS models Green constructs are robust and in conjunction with runtime recalibration provide strong QoS guarantees. For evaluation, we use four applications including `Live Search`, a back-end implementation of a commercial web-search engine, and three desktop applications such as `Eon` from SPEC CPU2000 [18], Cluster GA (`CGA`) [11], and Discrete Fourier Transformation (`DFT`) [6]. Detailed application descriptions are provided later in this section.

4.1 Environment

We use two different machines for our experiments. A desktop machine is used for experiments with `Eon`, `CGA`, and `DFT`. The desktop machine runs an Intel Core 2 Duo (3 GHz) processor with 4 GB (dual channel DDR2 667 MHz) main memory. A server-class machine is used for experiments with `Live Search`. The server machine has two Intel 64-bit Xeon Quad Core (2.33 GHz) with 8 GB main memory.

For each application, we compare the approximated versions generated by the Green compiler implemented using the Phoenix compiler framework [15] against their corresponding precise (base) versions. For evaluation, we measure three key parameters for each version: performance, energy consumption, and QoS loss. For the desktop applications, the wall-clock time between start and end of each run is used for performance evaluation. For `Live Search`, we first run a set of warmup queries and use the measured throughput (i.e., queries per second (QPS)) while serving the test queries as the performance metric. To measure the energy consumption, we use an instrumentation device that measures the entire system energy consumption by periodically sampling the current and voltage values from the main power cable. The sampling period of the device is 1 second. Since the execution time of the applications we study are significantly longer, this sampling period is acceptable. Finally, we compute the QoS loss of each approximate version by comparing against results from the base versions. The QoS metric used for each application will be discussed later. We also attempted to measure the overhead of Green by having each call to QoS_Approx() eventually return false and found the performance to be indistinguishable from the base versions of the applications without the Green code.

4.2 Applications

In this section, we provide a high-level description and discuss Green approximation opportunities for each application. In addition, we discuss the evaluation metrics and input data-sets used.

4.2.1 Live Search

Description: `Live Search` is a back-end implementation of a commercial web-search engine that accepts a stream of user queries, searches its index for all documents that match the query, and ranks these documents before returning the top N documents that match the query in rank order. Web crawling and index updates are disabled. There are a number of places in this and subsequent sections where additional information about the Live Search application may have been appropriate but where protecting Lives business interests require us to reduce some level of detail. For this reason, performance metrics are normalized to the base version and the absolute number of documents processed are not disclosed.

Opportunities for Approximation: The base

version of `Live Search` processes all the matching candidate documents. Instead, we can limit the maximum number of documents (M) that each query must process to improve performance and reduce energy consumption while still attempting to provide a high QoS.
Evaluation Metrics: We use QPS as the performance metric since throughput is key for server applications. We use *Joules per Query* as the energy consumption metric. Finally, for our QoS loss metric, we use the percentage of queries that either return a different set of top N documents or return the same set of top N documents but in a different rank order, as compared to the base version.
Input data-sets: The `Live Search` experiments are performed with a production index file and production query logs obtained from our data center. Each performance run uses two sets of queries: (1) warm-up queries: 200K queries to warm up the system and (2) test queries: 550K queries to measure the performance of the system.

4.2.2 Eon

Description: `Eon` is a probabilistic ray tracer that sends N^2 rays to rasterize a 3D polygonal model [18]. Among the three implemented algorithms in `Eon`, we only used the Kajiya algorithm [9].
Opportunities for Approximation: The main loop in `Eon` iterates N^2 iterations and sends a ray at each iteration to refine the rasterization. As the loop iteration count goes higher, QoS improvement per iteration can become more marginal. In this case, the main loop can be early terminated while still attempting to meet QoS requirements.
Evaluation Metrics: We measure the execution time and energy consumption to rasterize an input 3D model. To quantify the QoS loss of approximate versions, we compute the average normalized difference of pixel values between the precise and approximate versions.
Input data-sets: We generated 100 input data-sets by randomly changing the camera view using a reference input 3D model of `Eon`.

4.2.3 Cluster GA

Description: `Cluster GA (CGA)` solves the problem of scheduling a parallel program using a genetic algorithm [11]. `CGA` takes a task graph as an input where the execution time of each task, dependencies among tasks, and communication costs between processors are encoded using node weights, directed edges, and edge weights, respectively. `CGA` refines the QoS until it reaches the maximum generation (G). The output of `CGA` is the execution time of a parallel program scheduled by `CGA`.
Opportunities for Approximation: Depending on the size and characteristics of a problem, `CGA` can converge to a near-optimal solution even before reaching G. In addition, similar to `Eon`, QoS improvement per iteration can become more marginal at higher iteration counts (i.e., generation). By terminating the main loop earlier, we can achieve significant improvement in performance and reduction in energy consumption with little QoS degradation.
Evaluation Metrics: We use the same metrics for performance and energy consumption as for `Eon`. For a QoS metric, we compute the normalized difference in the execution time of a parallel program scheduled by the base and approximate versions.
Input data-sets: We use 30 randomly generated task graphs described in [12]. To ensure various characteristics in the constructed task graphs, the number of nodes varies from 50 to 500 and communication to computation ratio (CCR) varies from 0.1 to 10 in randomly generating task graphs.

4.2.4 Discrete Fourier Transform

Description: `Discrete Fourier Transform (DFT)` is one of the most widely used signal processing applications [6] that transforms signals in time domain to signals in frequency domain.
Opportunities for Approximation: In the core of `DFT`, `sin` and `cos` functions are heavily used. Since the precise version implemented in standard libraries can be expensive especially when the underlying architecture does not support complex FP operations, the approximated version of `sin` and `cos` functions can be effectively used if it provides sufficient QoS. We implement several approximated versions of `sin` and `cos` functions [8] and apply them to our `DFT` application.
Evaluation Metrics: We use the same metrics for performance and energy consumption as `Eon`. As a QoS metric, we compute the normalized difference in each output sample of `DFT` between the precise and approximated versions.
Input data-sets: We randomly generate 100 different input data-sets. Each input sample has a random real value from 0 to 1.

4.3 Experimental Results with `Live Search`

Figure 8 demonstrates the tradeoff between the QoS loss and the improvement in performance and

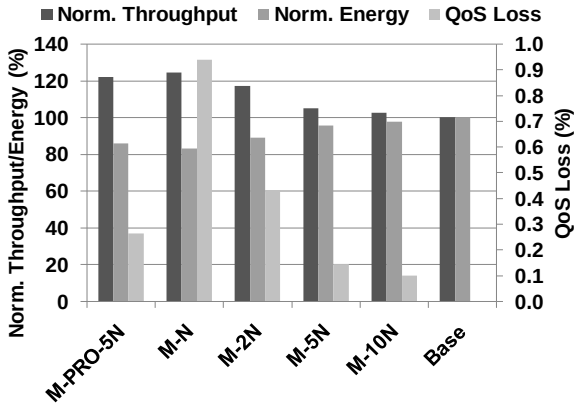

Figure 8. The tradeoff between QoS loss and the improvement in performance and energy consumption of `Live Search`.

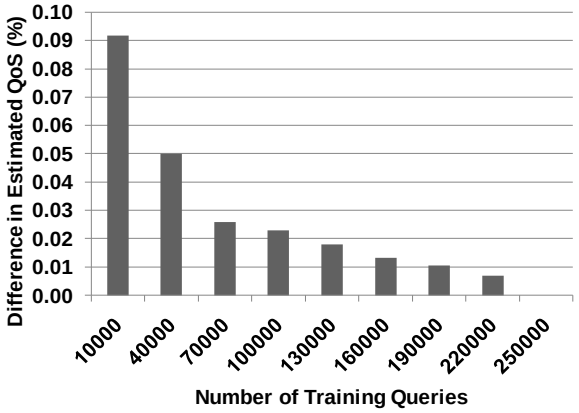

Figure 9. Sensitivity of Green's QoS model of `Live Search` to the size of training data-sets.

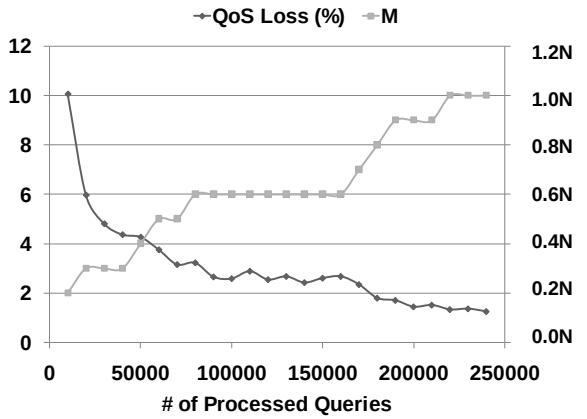

Figure 10. The effectiveness of Green's re-calibration mechanism for `Live Search`.

reduction in energy consumption. Base version is the current implementation of `Live Search`, while M-* versions are approximated. Specifically, M-*N statically terminates the main loop after processing *N matching documents for each query. M-PRO-0.5N samples QoS improvement after processing every 0.5N documents and adaptively terminates the main loop when there is no QoS improvement in the current period. As can be seen, some approximated versions significantly improve the performance and reduce the energy consumption (i.e., Joules per query) with very little QoS loss. For example, M-N improves throughput by 24.3% and reduces energy consumption by 17% with 0.94% of QoS loss. Another interesting point is that M-PRO-0.5N that uses the adaptive approximation leads to slightly better performance and less energy consumption while providing better QoS compared to M-2N version that uses the static approximation. This showcases the potential of adaptive techniques to optimize `Live Search`.

To study the sensitivity of Green's QoS model to the training data-set size, we randomly permuted the warm-up and test queries and injected different number of queries ranging from 10K to 250K queries to build the QoS model. Figure 9 demonstrates the difference in estimated QoS loss (when M=N) with the varying size of training data-sets. QoS models generated with much fewer number of queries are very close to the one generated with 250K. For example, the QoS model generated using only 10K queries differs by only 0.1% compared to the one generated using 250K inputs. This provides empirical evidence that Green can construct a robust QoS model for `Live Search` without requiring huge training data-sets.

To evaluate the effectiveness of Green's re-calibration mechanism, we performed an experiment simulating an imperfect QoS model. Say a user indicates his/her desired QoS target as 2%, but the constructed QoS model incorrectly supplies M = 0.1N (which typically results in a 10% QoS loss). Thus, without re-calibration `Live Search` will perform poorly and not meet the target QoS. Figure 10 demonstrates how can Green provide robust QoS guarantees even when supplied with an inaccurate QoS model. After processing every 10K queries, Green monitors the next 100 consecutive queries (i.e., Sample_QoS=1%) by running the precise version while also computing the QoS loss, if it had

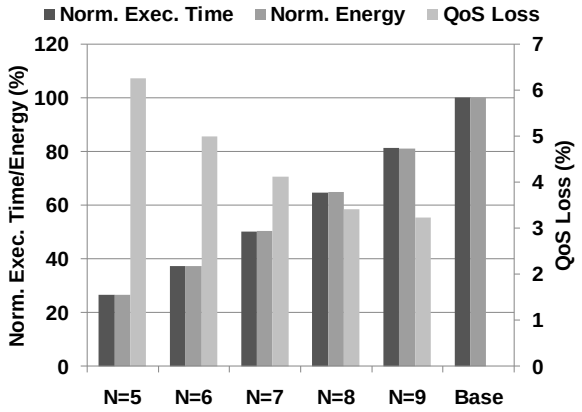

Figure 11. The tradeoff between QoS loss and the improvement in performance and energy consumption of `Eon`.

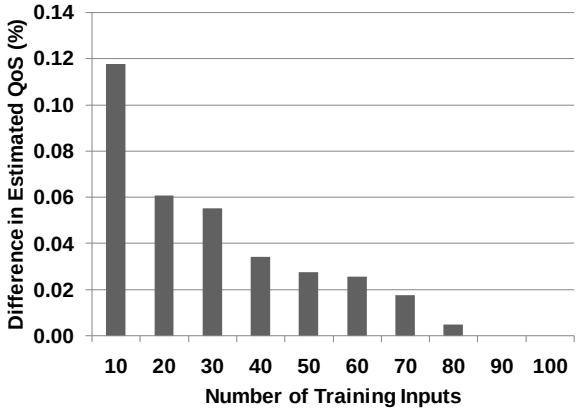

Figure 12. Sensitivity of Green's QoS model of `Eon` to the size of training data-sets.

used the approximated version for those 100 queries. Since the current QoS model is not accurate enough, the monitored results will keep reporting low QoS. Then, Green's re-calibration mechanism keeps increasing the accuracy level (i.e., by increasing the M value by 0.1N) until it satisfies the user-defined QoS target. In Figure 10, Green meets the QoS target after processing 180K queries. The user could use a period smaller than 10K to make Green adapt faster but there is a tradeoff between quick adaptation and degradation in performance and energy consumption caused by more frequent monitoring.

4.4 Experimental Results with Desktop Applications

Figure 11 shows the results for `Eon` using 100 randomly generated input data-sets. Similar to `Live Search`, approximated versions of `Eon` significantly improve the performance and energy consumption with relatively low QoS loss. Figure 12 demonstrates the sensitivity of Green's QoS model for `Eon` to the size of training data-sets. We varied the training data size from 10 to 100, and compared the estimated QoS loss difference (when N=9) of the generated QoS model to the one generated using 100 inputs. Figure 12 provides empirical evidence that Green's QoS model for `Eon` can be constructed robustly with relatively small training data-sets. For example, the QoS model generated using 10 inputs differs by only 0.12% compared to the one generated using 100 inputs.

Figure 13 demonstrates the Green model of `CGA` using 30 randomly generated input data-sets. Up to

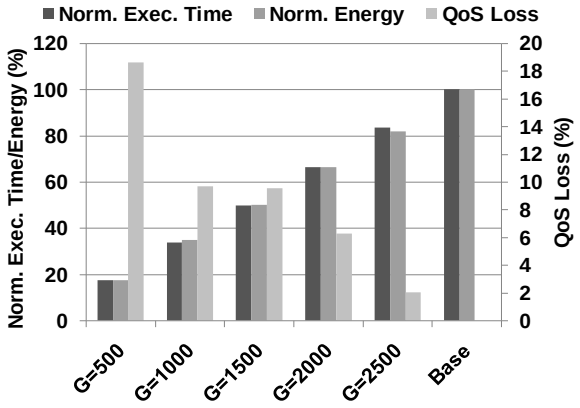

Figure 13. The tradeoff between QoS loss and the improvement in performance and energy consumption of `CGA`.

G=1000, QoS loss is reasonable (<10%), while significantly improving performance and energy consumption by 66.2% and 65.0%, respectively. In Figure 14, We also present the sensitivity of Green's QoS model of `CGA` to the training data-set size. We varied the number of training inputs from 5 to 30 and compared the estimated QoS loss (G=2500) from the generated QoS model to the one generated using 30 inputs. While the difference in the estimated QoS loss is higher than other applications due to the discrete nature of the outcome of a parallel task scheduling problem, the difference is still low (<1.5% even when 5 inputs are used).

Figure 15 shows the tradeoff between QoS loss and improvement in performance and energy con-

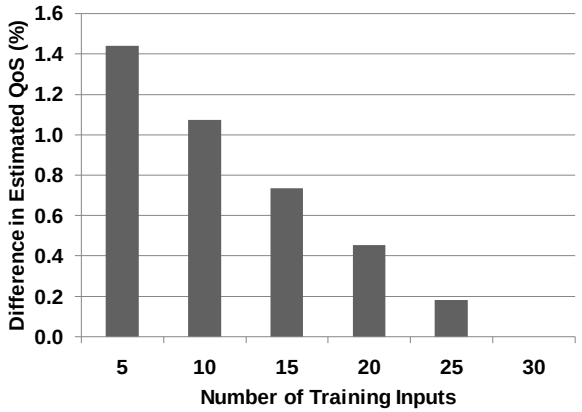

Figure 14. Sensitivity of Green's QoS model of `CGA` to the size of training datasets.

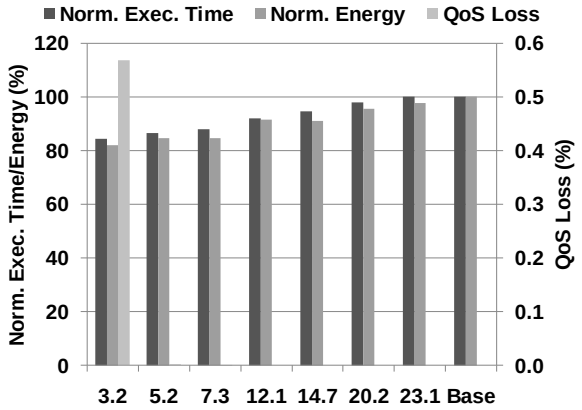

Figure 15. The tradeoff between QoS loss and the improvement in performance and energy consumption of `DFT`.

sumption using various approximated versions of `DFT` generated by Green. More specifically, each `DFT` version uses `sin` and `cos` functions that provide different accuracy ranging from 3.2 digits to 23.1 digits [8]. Up to the accuracy of 12.1 digits, no accuracy loss is observed while improving performance and energy consumption of `DFT` by 8.1% and 8.4%, respectively. Even using the accuracy of 3.2 digits, the observed QoS loss is very low (0.57%), while improving performance and energy consumption of `DFT` by 15.8% and 18.1%, respectively. This clearly indicates the potential of function-level approximation using Green. While not shown, the QoS model constructed for `DFT` is also similarly robust and can be accurately generated with very few inputs.

5 Related Work

There are several application domains such as machine learning and multimedia data processing where applications exhibit *soft computing* properties [4]. The common soft computing properties are *user-defined, relaxed correctness*, *redundancy in computation*, and *adaptivity to errors* [2, 13]. Researchers have studied improving the performance, energy consumption, and fault tolerance of applications and systems by exploiting these soft computing properties. However, to the best of our knowledge, we believe Green is the first system that provides a simple, yet flexible framework and programming support for principled approximations that attempts to meet specified QoS requirements.

Green is most similar to Rinard's previous work [16, 17] in the sense of proposing a probabilistic QoS model (i.e., *distortion model* in [16]) and exploiting the tradeoff between the performance improvement and QoS loss. However, our proposal significantly differs in three aspects. First, Green provides language constructs at the function- or loop-level which can often allow more modular, fine-grained, and efficient optimizations for C/C++ programs whereas [16, 17] considered tasks as an optimization granularity. Second, Green introduces the re-calibration mechanism that can effectively provide strong QoS guarantees even in the presence of some degree of inaccuracy in the constructed QoS model. Finally, we experimentally demonstrate that the principled approximation can be effective for a wider range of application domains (i.e., not only scientific applications) including a production-quality, real-world web-search engine.

Several researches focused on floating-point approximation techniques [1, 19]. Alvarez et al. proposed *fuzzy memoization* for floating-point (FP) multimedia applications [1]. Unlike the classical instruction memoization, fuzzy memoization associates similar inputs to the same output. Tong et al. proposed a *bitwidth reduction technique* that learns the fewest required bits in the FP representation for a set of signal processing applications to reduce the power dissipation in FP units without sacrificing any QoS [19]. While effective in improving performance and energy consumption of FP applications, applications with infrequent use of FP operations will not benefit from these schemes significantly. In addition, they do not provide any runtime re-calibration support for statistical QoS guarantees. In contrast, Green is more general, targets a wider range of applications (i.e., not only FP applications) and attempts to meet specified QoS requirements.

Several researches studied the impact of the soft computing properties on the error tolerance of systems in the presence of defects or faults in chips [5, 14]. Breuer et al. demonstrated that many VLSI implementations of multimedia-related algorithms are error-tolerant due to the relaxed correctness. Based on this, they proposed design techniques to implement more error-resilient multimedia chips [5]. Li and Yeung investigated the fault tolerance of soft computations by performing fault-injection experiments [14]. They demonstrated that soft computations are much more resilient to faults than conventional workloads due to the relaxed program correctness. Our work differs as it focuses on performance and energy optimizations that meet specified QoS requirements instead of fault tolerance.

6 Conclusions

This paper describes the Green system that supports energy-conscious programming using principled approximation for expensive loops and functions. Green generates a calibration version of the program that it executes to construct a QoS model that quantifies the impact of the approximation. Next, it uses this QoS model to synthesize an approximate version of the program that attempts to meet a user-specified QoS target. Green also provides a runtime re-calibration mechanism to adjust the approximation decision logic to meet the QoS target.

To evaluate the effectiveness of Green, we built a prototype implementation using the Phoenix compiler framework and applied it to four programs including a real-world search application. The experimental results demonstrate that the Green version of these applications perform significantly better and consume less energy with only a small loss in QoS. In particular, we improved the performance and energy consumption of `Live Search` by 22.0% and 14.0% respectively with 0.27% of QoS degradation. We also showed that the QoS models constructed for these applications are robust. In conjunction with Green's runtime re-calibration mechanism, this enables approximated applications to meet user-specified QoS targets.

7 Acknowledgements

We would like to thank Preet Bawa, William Casperson, Engin Ipek, Utkarsh Jain, Benjamin Lee, Onur Mutlu, Xuehai Qian, Gaurav Sareen, Neil Sharman, and Kushagra Vaid, who made contributions to this paper in the form of productive discussions and help with the evaluation infrastructure.

References

[1] C. Alvarez and J. Corbal. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.*, 54(7):922–927, 2005.

[2] W. Baek, J. Chung, C. Cao Minh, C. Kozyrakis, and K. Olukotun. Towards soft optimization techniques for parallel cognitive applications. In *19th ACM Symposium on Parallelism in Algorithms and Architectures*. June 2007.

[3] L. A. Barroso. Warehouse-scale computers. In *USENIX Annual Technical Conference*, 2007.

[4] P. P. Bonissone. Soft computing: the convergence of emerging reasoning technologies. *Soft Computing—A Fusion of Foundations, Methodologies and Applications*, 1(1):6–18, 1997.

[5] M. A. Breuer, S. K. Gupta, and T. Mak. Defect and error tolerance in the presence of massive numbers of defects. *IEEE Design and Test of Computers*, 21(3):216–227, 2004.

[6] E. O. Brigham. *The fast Fourier transform and its applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[8] J. G. Ganssle. A Guide to Approximation. `http://www.ganssle.com/approx/approx.pdf`.

[9] J. T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, 1986.

[10] R. Katz. Research directions in internet-scale computing. In *3rd International Week on Management of Networks and Services*, 2007.

[11] V. Kianzad and S. S. Bhattacharyya. Multiprocessor clustering for embedded systems. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 697–701, London, UK, 2001. Springer-Verlag.

[12] Y.-K. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the Symposium on Paral-*

lel and Distributed Processing 1998, pages 531–537, Mar-3 Apr 1998.

[13] X. Li and D. Yeung. Exploiting soft computing for increased fault tolerance. In *In Proceedings of Workshop on Architectural Support for Gigascale Integration*, 2006.

[14] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 181–192, Washington, DC, USA, 2007. IEEE Computer Society.

[15] Phoenix Academic Program. `http://research.microsoft.com/Phoenix/`.

[16] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 324–334, New York, NY, USA, 2006. ACM.

[17] M. C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 369–386, New York, NY, USA, 2007. ACM.

[18] Standard Performance Evaluation Corporation, *SPEC CPU Benchmarks*. `http://www.specbench.org/`, 1995–2000.

[19] J. Tong, D. Nagle, and R. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8(3):273–286, Jun 2000.