Approximate Computation With Outlier Detection in Topaz

Sara Achour
MIT CSAIL
sachour@csail.mit.edu

Martin C. Rinard
MIT CSAIL
rinard@csail.mit.edu

Abstract

We present Topaz, a new task-based language for computations that execute on approximate computing platforms that may occasionally produce arbitrarily inaccurate results. Topaz maps tasks onto the approximate hardware and integrates the generated results into the main computation. To prevent unacceptably inaccurate task results from corrupting the main computation, Topaz deploys a novel outlier detection mechanism that recognizes and precisely reexecutes outlier tasks. Outlier detection enables Topaz to work effectively with approximate hardware platforms that have complex fault characteristics, including platforms with bit pattern dependent faults (in which the presence of faults may depend on values stored in adjacent memory cells). Our experimental results show that, for our set of benchmark applications, outlier detection enables Topaz to deliver acceptably accurate results (less than 1% error) on our target approximate hardware platforms. Depending on the application and the hardware platform, the overall energy savings range from 5 to 13 percent. Without outlier detection, only one of the applications produces acceptably accurate results.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Processors – Optimization

General Terms Languages, Reliability, Performance

Keywords Approximate Computing, Control Systems, Dynamic Systems

1. Introduction

Errors and approximation are fundamental concerns in computing. Although many applications can easily tolerate reasonable errors and approximations, developers today design applications with the expectation that they will execute on precise, error-free computing platforms.

Motivated by the benefits available via relaxing stringent correctness and precision requirements, researchers have proposed a variety of approximate computing platforms [6, 9, 10, 19, 22, 26, 28, 30, 36, 45, 47]. Many of these platforms are populated with components that feature higher error rates and/or less accurate execution in return for reduced energy consumption and/or increased performance. Despite the significant amount of research in this area, no unified model has emerged that can provide predictable reliability or approximation guarantees. Moreover, many approximate components (such as approximate memories and caches) exhibit complex fault behavior and can easily deliver arbitrarily unreliable or inaccurate results. To execute successfully on such platforms, applications require techniques that can successfully mitigate large errors that would otherwise produce unacceptably inaccurate end-to-end results.

1.1 Topaz

We present Topaz, a new language for approximate computing. Unlike many previous languages and systems, Topaz is designed to deliver acceptably accurate end-to-end execution even on approximate computing platforms with complex fault characteristics and arbitrarily inaccurate results. Topaz structures the computation as (1) a main computation that executes precisely without error and (2) sets of small, self-contained, correctable approximate tasks whose results are integrated into the main computation. Topaz utilizes outlier detectors, which recognize and precisely reexecute tasks with unacceptably accurate results, to enable Topaz programs to execute successfully even on challenging platforms with large errors and inaccurate approximations.

We evaluate Topaz on a set of benchmark approximate applications running on two simulated approximate hardware platforms. The results indicate:

- **Critical Regions:** The main computations of all of the applications contain *critical regions* that must execute precisely without error (otherwise the application crashes).
- **Outlier Detection:** While the outlier detector rejects at most a few percent of the tasks, including the results from this small number of rejected tasks into the main computation produces unacceptably inaccurate end-to-end results.

- **Acceptable Accuracy:** Outlier detection plus reexecution produces end-to-end results that are within a fraction of a percent of the correct results produced by fully precise execution.
- **Energy Efficiency:** On our two simulated approximate hardware platforms, Topaz delivers energy savings of between 5 and 13 percent (out of a maximum achievable energy savings of 19.23 percent).

Together, these facts highlight how Topaz, by enabling approximate applications to execute successfully despite arbitrarily inaccurate results, effectively supports current and future approximate platforms and maximizes the design space available to developers of new approximate platforms.

1.2 Potential Benefits

In this paper we focus on energy efficiency. However, energy efficiency is only one of multiple potential benefits that approximate computing with Topaz can deliver:

- **Successful Execution on Current Hardware:** In practice, hardware faults in current hardware platforms are far more common than initially thought [21, 38, 43]. Topaz can enhance the ability of applications to execute successfully on current hardware platforms despite the presence of these errors.
- **Reduced Design and Test Effort:** Vendors currently expend significant design and test resources in an attempt to minimize defects in shipped components. By increasing error tolerance, Topaz can help vendors reduce the design and test effort required to obtain acceptable components. We note that it is possible to apply this approach to both hardware and software components.
- **Increased Yield:** Vendors currently discard a substantial percentage of hardware components because of manufacturing defects [24]. Topaz can help increase the effective yield by enabling vendors to salvage and redeploy defective components to execute approximate tasks.
- **New Hardware Substrates:** Revolutionary new hardware substrates (such as carbon nanotubes) feature far superior energy consumption profiles but less reliable fabrication characteristics than standard substrates [40]. By reducing the reliability requirements of components that execute approximate tasks, Topaz can help enable the deployment of components built from these new substrates.
- **New Software Marketing Models:** While this paper focuses on approximate hardware platforms, Topaz also enables new, more flexible pricing models for software components. Maximally accurate/reliable versions would command the highest prices, with less accurate/reliable versions available at less cost. To avoid the expense of multiple development efforts, vendors could use purposeful detuning to produce cheaper, less accurate/reliable versions of a single base software product.

All of these benefits address current issues in hardware and software design, development, and manufacture. And given technology trends that emphasize increased functionality and complexity, smaller feature sizes, and energy efficiency, these issues will only increase in importance in the future.

1.3 Approximate Checkers and AOVs

The Topaz outlier detectors use *approximate checkers* to deliver acceptably accurate computations on approximate computing platforms with complex approximation and error characteristics. Previous research focuses on exact checkers [3, 6, 20], which must efficiently determine whether a given result is correct or incorrect. Approximate checkers, in contrast, only aspire to check if the result is *accurate enough*. This additional flexibility makes it possible to develop checkers for a much broader range of computations.

It is possible to perform outlier detection directly on the raw task results. But to facilitate more efficient and effective outlier detection, Topaz first works with the task inputs and results to compute *abstract output vectors* (AOVs). Each AOV is a vector of numbers upon which the outlier detector operates. AOVs typically select relevant results and combine results (typically via a reduction such as summing the results) to reduce the dimensionality of the vector on which the outlier detector operates. Topaz also supports more specialized AOVs that exploit application semantics to more appropriately abstract the task results for outlier detection.

1.4 Reexecution

When an outlier detector rejects a task, Topaz reexecutes the task on the precise platform. It then integrates the correct result from this precise reexecution into the main computation. This mechanism provides several benefits:

- **Corrects Unacceptably Inaccurate Results:** When the approximate hardware produces an unacceptably inaccurate result, the Topaz outlier detector detects and replaces the incorrect result with the correct result.
- **Learns Online:** Topaz uses an online algorithm that utilizes the reexecuted tasks to train the outlier detector. There is no need for offline training.
- **Adapts to Change:** The outlier detector contains a control system that allows the outlier detector to adapt to the characteristics of the tasks as they evolve over time.
- **Refreshes Corrupt Data:** Topaz refreshes any stable data on the approximate machine when an error is detected, eliminating any accrued data corruptions (hardware platforms with approximate memories are particularly vulnerable to errors that accrue over time).
- **Accepts Reasonable Results:** The Topaz outlier detectors work with correct tasks to maintain ranges of data within which they expect acceptable tasks to fall. Any false negative tasks (incorrect tasks that the outlier detector accepts) therefore produce results that are close to the results that correct tasks produce. This strategy inter-

acts well with the characteristics of our set of target approximate computations, which easily tolerate small deviations from correct task results.

The drawback of reexecution, of course, is the energy required to reexecute the task. Topaz must therefore reexecute few enough tasks to deliver significant energy savings.

1.5 Topaz Task Design Rationale

Topaz tasks enable Topaz to work with a wide range of approximate computing platforms, including approximate computing platforms that can occasionally produce wildly inaccurate results. Long-lived computations tend to interact poorly with such platforms — the chance of encountering a fatal error or wildly inaccurate intermediate result that unacceptably corrupts the final result can become unacceptably high. Topaz enables developers to identify small idempotent computations whose results can be checked for acceptability and, if necessary, reexecuted. Topaz therefore promotes the effective decomposition of the computation into small correctable pieces that can be combined to obtain an acceptably accurate final result [42].

Topaz tasks also enable Topaz to work effectively on hardware platforms in which errors accumulate over time (platforms with approximate caches and memories, for example, often have this error pattern). Each Topaz task identifies *stable* inputs (which have the same value for multiple tasks) and *transient* inputs (which have different values for different tasks). The Topaz implementation transmits the stable data only once for multiple tasks. Because Topaz refreshes the stable data when it detects three consecutive incorrect tasks, it can detect and correct any stable data corruption errors soon after they begin to accumulate.

Topaz tasks also enable Topaz computations to interoperate well with approximate components that may sporadically fail or require periodic microreboots. Because Topaz tasks implement small standalone pieces of computation that can typically complete before a failure or microreboot interrupts their execution, they can effectively exploit unreliable computing platforms with these kinds of components.

1.6 Contributions

This paper makes the following contributions:

- **Topaz:** It presents Topaz, a task-based language for approximate computation. Topaz supports a reliable main computation and approximate tasks that can execute on approximate computing platforms.
  The Topaz language design exposes a key computational pattern (small, self-contained, correctable tasks) that enables the successful utilization of approximate computing platforms that may produce arbitrarily inaccurate results.
- **Adaptive Outlier Detection:** Approximate computations must deliver acceptably accurate results. The Topaz implementation uses outlier detection to ensure that unacceptably inaccurate task results are not integrated into the main computation.
- **Abstract Output Vectors (AOVs):** It presents AOVs, which produce abstract versions of task results appropriate for efficient and effective outlier detection.
- **Precise Outlier Task Reexecution:** Topaz reliably reexecutes outlier tasks and integrates the resulting correct results into the computation. This strategy delivers a relatively low reexecution rate while detecting and correcting results that might otherwise unacceptably affect the final result.
- **Experimental Evaluation:** We evaluate Topaz on a set of benchmark Topaz programs executing on two approximate computing platforms. The results show that outlier detection and reexecution enables Topaz to deliver acceptably accurate end-to-end results (less than 1% error) and energy savings of between 5 and 13 percent depending on the benchmark and the hardware platform.

Topaz provides a new, task-based model of approximate computation that can successfully exploit approximate computational platforms that may deliver arbitrarily inaccurate results. Our experimental evaluation provides facts that justify the Topaz language design and characterizes the ability of Topaz to deliver acceptably accurate, energy-efficient approximate computations.

2. Example

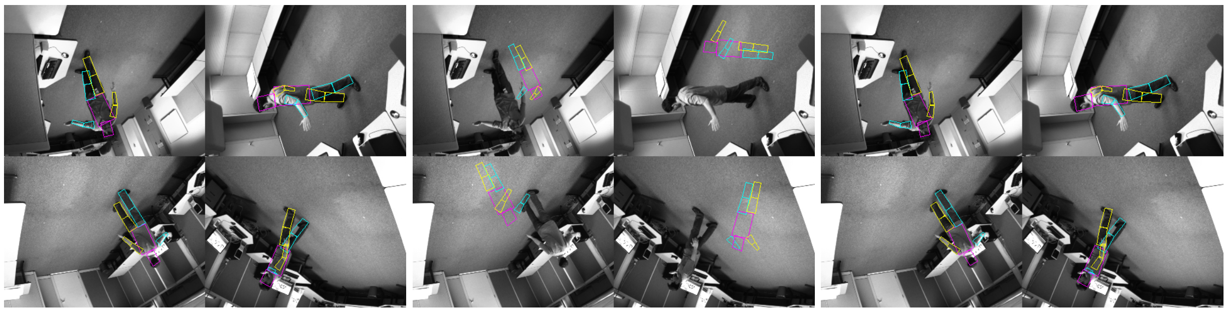
We next present an example that illustrates (1) how to express a computation in Topaz and (2) the resulting Topaz execution on an approximate computing platform.

```
// computes the weights for each valid pose.
taskset calcweights(i=0; i < particles.size(); i+=1){
  compute in (
   float tpartP_SIZE =  (float*) particlesi,
   const float tmodelM_SIZE = (float*) mdl_prim,
   const char timgI_SIZE = (char *) img_prim,
   const int nCams = mModel->NCameras(),
   const int nBits = mModel->getBytesPerPixel(),
   const int width = mModel->getWidth(),
   const int height =mModel->getHeight()
  ) out (float tweight) {
      tweight = CalcWeight(tpart,
          tmodel, timg, nCams, width, height, nBits);
  } transform out (bweight, bp1, bp2, bp3) {
      bweight =  tweight;
      bp1=tpart3; bp2=tpart4; bp3=tpart5;
  } combine {
      mWeightsi = tweight;
  }
}
```

Figure 1: Example Topaz Program

2.1 Example Topaz Program

Bodytrack is a machine vision application from the Parsec benchmark suite [2]. In each step, bodytrack computes the estimated position of a human in a frame [2]. Figure 1 presents the bodytrack `calcweights` computation. Given a tracking model, this computation scores the fit of each pose



(a) Precise Execution (b) No Outlier Detection (c) Outlier Detection

Figure 2: Output Quality for Bodytrack

in a set of poses against an image. The computation is implemented as a set of Topaz tasks defined by the Topaz taskset construct. When it executes, the taskset generates a set of tasks, each of which executes three main components: the task `compute` block (which defines the approximate computation of the task), `transform` block (which defines an output abstraction), and `combine` block (which integrates the computed result into the main computation).

`compute` block: The compute block defines the inputs (with section `in`), outputs (with section `out`) and computation (following code block) associated with an approximate task. In our example, the `compute` block computes the weight for a particular pose by invoking the `CalcWeight` function, which is a standard C function. The `taskset` executes a compute task for each pose. In total, it executes `particles.size()` tasks.

Each task takes as input two kinds of data: *stable* data, which has the same value for all tasks in the taskset (stable data is identified with the `const` keyword) and *transient* data (which may have different values for each task in the taskset). In our example each task has one transient input, `tpart` (which identifies the pose to score) and several stable inputs (that remain unchanged throughout the lifetime of the taskset): (1) `tmodel`, the model information (2) `timg`, the frame to process, (3) `nCams`, the number of cameras, and (4) `nBits`, the bit depth of the image. The task produces one result: `tweight`,

`transform` block: The `transform` construct defines the AOV, which captures the key features of the task result. Topaz performs outlier detection on this AOV when the task finishes execution. In this example we use the pose position from the task input (`bp1,bp2,bp3`) and pose weight from the task result (`bweight`) as the AOV.

`combine` block: When the task completes, its `combine` block executes to incorporate the results from the `out` parameters into the main Topaz computation (provided the outlier detector accepts the task). In our example, the `combine` block updates the `mWeights` global program variable.

2.2 Approximate Execution

The current Topaz implementation runs on an approximate computing platform with a precise processor and an approximate processor. The precise processor executes all non-taskset code and the `transform` and `combine` blocks of executed tasks. The approximate processor executes the `compute` bodies of the executed tasks. Since the `taskset` construct is executed approximately, the computed results may be inaccurate. In our example the task result, `tweight` (the computed pose weight), may be inaccurate. That is, the computed pose may be assigned an inappropriately high or low weight for the provided frame.

If integrated into the main computation, significantly inaccurate tasks can unacceptably corrupt the end-to-end result that the program produces. In our example, the `combine` blocks write the computed weights into a persistent weight array `mWeights`. Bodytrack uses this weight array to select the highest weighted pose for each frame. If the computation incorporates an incorrect weight, it may select an incorrect pose as the best pose for the frame.

2.3 Outlier Detection

Topaz therefore deploys an *outlier detector* that is designed to detect tasks that produce unacceptably inaccurate results. The outlier detector uses the user defined `transform` block to convert the task inputs and results into a numerical vector, which we call the *abstract output vector* (AOV). This vector captures the key features of the task result. The `transform` block in our example defines a four dimensional AOV `<bweight,bp1,bp2,bp3>` comprised of the weight and pose position. We use this abstraction because we expect poses in the same location to be scored similarly.

The outlier detector uses this numerical vector to perform outlier detection online (see Section 5.4 for more details). The outlier detector *accepts* correct AOVs and *rejects* AOVs that are outliers. In our example, the outlier detector rejects tasks that produce high weights for bad poses – poses that are positioned in parts of the frame without a person. Similarly, the outlier detector rejects tasks that produce low weights for good poses – poses near the person.

2.4 Precise Reexecution

When the outlier detector rejects a task, Topaz precisely reexecutes the task on the precise processor to obtain the correct task result, which it then integrates into the main computation. The numerical vector attained by transforming the task result is used to further update the outlier detector. To minimize any error-related skew, Topaz uses only correct results attained via reexecution to train the outlier detector. In our example, the tasks that assign abnormally high weights to bad poses or low weights to good poses are rejected and reexecuted. After all task results have been integrated and the taskset has completed execution, the highest weight in the persistent weight array belongs to a pose close to the correct one. This technique maximizes the accuracy of the overall computation.

Figure 2a presents the correct pose for a particular video frame when the computation has no error. Figure 2b presents the selected pose for the same frame run on approximate hardware with no outlier detection. Note that the selected pose is unacceptably inaccurate and does not even loosely cover the figure. Figure 2c presents the selected pose for the same frame on the same approximate hardware with outlier detection. Observe that the chosen pose is visually indistinguishable from the correct pose (Figure 2a). The numerical error for the computation with outlier detection is 0.161% (compared to 73.63% error for the computation with no outlier detection). The outlier detector reexecutes 4.72% of the tasks, causing a 4.10% energy degradation when compared to the savings without detection (12.70% vs. 8.60%). With the Topaz overhead included, the reported energy savings are 7.69% (0.91% loss from Topaz overhead).

3. The Topaz Language

We implement Topaz as an extension to C. Topaz adds a single statement, the *taskset* construct. When a taskset construct executes, it creates a set of approximate tasks that execute to produce results that are eventually combined back into the main computation. Figure 3 presents the general form of this construct.

```
taskset name(int i = l; i < u; i++) {
  compute in  (d1 x1 = e1, ..., dn xn = en)
         out (o1 y1, ..., oj yj) {
    <task body>
  }
  transform out(v1, ..., vk) {
    <output abstraction>
  }
  combine { <combine body> }
}
```

Figure 3: Topaz Taskset Construct

Each taskset construct creates an indexed set of tasks. Referring to Figure 3, the tasks are indexed by `i`, which ranges from `l` to `u-1`. Each task uses an `in` clause to specify a set of in parameters `x1` through `xn`, each declared as `d1` through `dn` and initialized to expressions `e1` through `en`. Topaz supports scalar declarations of the form `double x`, `float x`, and `int x`, as well as array declarations of the form `double x[N]`, `float x[N]`, and `int x[N]`, where `N` is a compile-time constant. To specify that input parameters are stable and do not change throughout the lifetime of a `taskset`, we prepend the `const` identifier to the input type.

Each task also uses an `out` clause to specify a set of out parameters `y1` through `ym`, each declared as `o1` through `om`. The task writes the results that it computes into these out parameters. As for `in` parameters, `out` parameters can be scalar or array variables.

Topaz imposes the requirement that the `in` and `out` parameters are disjoint and distinct from the variables in the scope surrounding the `taskset` construct. Task bodies write all of their results into the `out` parameters and have no other externally visible side effects.

The `transform` body computes over the in parameters `x1, ..., xn` and out parameters `o1, ..., oj` to produce the numerical AOV `v1, ..., vk`. The outlier detector operates on the AOV.

The `combine` body integrates the task results into the main body of the computation. All of the task bodies for a given taskset are independent and can execute in parallel — only the combine bodies may have dependences.

4. Abstract Output Vectors

A well-designed AOV is an efficient abstraction that exposes sufficient result quality information to the Topaz outlier detector. To work well, AOVs should be efficiently computable and produce a small set of numbers that make unacceptably inaccurate results readily apparent. Each AOV produces a floating point vector computed in the `transform` block. In general, larger dimension vectors and more complex transform blocks incur greater overhead.

4.1 Constructing an AOV

We consider two techniques for designing effective AOVs:

- **Dimensionality Reduction:** The transform block performs a reduction on the task results to reduce the dimensionality of the AOV.
- **Domain-Specific Normalization**: The programmer utilizes domain knowledge to construct an AOV that removes the input correlation from the task results without explicitly including the inputs in the AOV.

Dimensionality reduction is an easy-to-apply generalized method that may be applied to any task. Normalization is typically closely tied to the application semantics. We use the blackscholes benchmark (see Section 6) to illustrate these two techniques. Blackscholes predicts the future price of a set of options, given the current prices and a set of option parameters (`strike`,`volatility`,`time`,`type`,`rate`). Each Topaz task computes prices for 64 options.

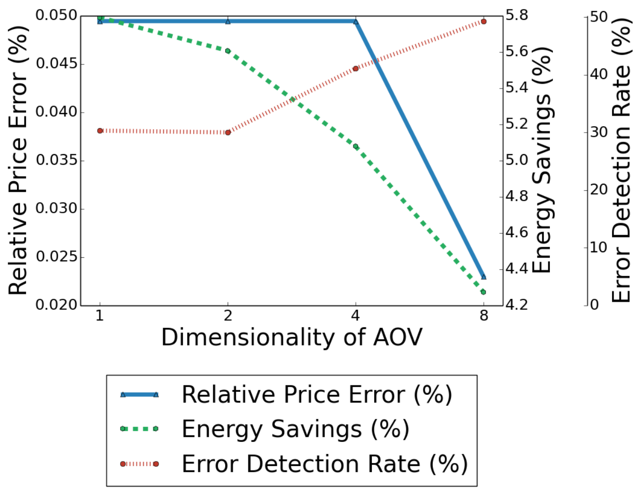


Figure 4: Effect of Performing Output Reduction

Dimensionality Reduction: We consider four blackscholes AOVs: (1) a one-dimensional AOV which sums all 64 predicted prices to obtain a single element for outlier detection, (2) a two-dimensional AOV in which each element is the sum of 32 prices, (3) a four-dimensional AOV in which each element is the sum of 16 prices, and (4) an eight-dimensional AOV in which each element is the sum of eight prices.

Figure 4 presents the effect of dimensionality reduction on the error detection rate, option price error, and energy savings. As the dimensionality of the AOV increases, the error detection rate increases. In response, the computation becomes more accurate and the relative option price error decreases, but from an already very small base error (specifically, from 0.05% to 0.02%). Also in response, the energy savings drops. There are two contributing factors. First, increasing the dimensionality of the result increases the precision of the outlier detector, which detects more errors and reexecutes more tasks as the dimensionality increases.[1] Second, the increase in the dimensionality of the AOV increases the outlier detector overhead.

Contraindications for Dimensionality Reduction: Dimensionality reduction has almost no impact on the overall accuracy of blackscholes (in part because the correct option prices all have similar magnitudes). Combining results from radically different distributions may significantly degrade Topaz's ability to perform error detection and noticeably impact result quality. For example, if one of the results tends to be large and the other result is small, the larger result may mask errors in the smaller result. In the barnes benchmark (see Section 6), the magnitude ranges of the acceleration and velocity task results are drastically different (-5 to 5 vs -150 to 150). Topaz detects 52.50% of the errors when the AOV combines like results and only 30.61% of the errors (20% less) when the AOV combines unlike results. The corresponding energy savings are 6.40% (when combining like results) and 7.91% (when combining unlike results).

Dimensionality Reduction and Batching: The natural task granularity of some Topaz applications may be too small to profitably amortize the Topaz task management overhead. In such situations *batching* — combining multiple natural tasks into a single larger Topaz task — is one way to drive the Topaz task management overhead down to acceptable levels. Batched tasks typically produce multiple sets of results (one for each natural task in the batch). Batching can therefore often be productively combined with dimensionality reduction for effective outlier detection, with like results from natural tasks combined into a single element for outlier detection. Blackscholes, for example, batches 64 natural tasks into a single Topaz task for a batching factor of 64. For this application (but not all Topaz applications, see Section 6), batching is essential for successfully amortizing the Topaz task management overhead.

Domain-Specific Normalization: We derive a domain-specific normalization for blackscholes using the no-arbitrage bound for European put-call options [11]. The no-arbitrage bound is a sane maximum and minimum option price, given the option parameters and price. The benchmark utilizes European put-call options. The original inequality is as follows:

`if put:` $s \cdot e^{-r \cdot t} - p_0 \leq p \leq p_0$
`if call:` $p_0 - s \cdot e^{-r \cdot t} \leq p \leq s \cdot e^{-r \cdot t}$

Where s is the strike price, r is the rate, t is the time, p_0 is the input price and p is the result price. First, we normalize the lower bound to attain an expression for p.

`if put:` $0 \leq p - s \cdot e^{-r \cdot t} + p_0 \leq 2 \cdot p_0 - s \cdot e^{-r \cdot t}$
`if call:` $0 \leq p - p_0 + s \cdot e^{-r \cdot t} \leq 2 \cdot s \cdot e^{-r \cdot t} - p_0$

We then normalize the expression against the upper bound by dividing out the upper bound.

`if put:` $0 \leq (p - s \cdot e^{-r \cdot t} + p_0) \cdot (2 \cdot p_0 - s \cdot e^{-r \cdot t})^{-1} \leq 1$
`if call:` $0 \leq (p - p_0 + s \cdot e^{-r \cdot t}) \cdot (2 \cdot s \cdot e^{-r \cdot t} - p_0)^{-1} \leq 1$

Note that the `type` variable determines whether the operation is a `put` or `call` operation. The above expression provides a normalized representation of p, where it is unlikely that p falls outside of the range $[0, 1]$.

With the no-arbitrage normalization applied, the Topaz outlier detector catches 8% more errors (reexecuting 0.02% more tasks) and loses 7% energy savings in comparison with an AOV that simply uses the computed option price. We attribute the lost energy savings to the more complex AOV computation as reflected in the formulas above. This example illustrates how an effective AOV must balance a number of factors including reexecution rate, accuracy, and AOV compute time.

[1] The target reexecution rate for these executions is zero. The outlier detector therefore does not contract its bounds in an attempt to obtain a higher reexecution rate.

4.2 AOVs and Quality Metrics

Note that although we use result quality metrics to evaluate the quality of the AOV, we decouple the AOV definition from the quality metric of the application when constructing the AOV. We intentionally decouple the two phenomena because quality metrics introduce the following complexities:

- **Expensive to Compute:** Some quality metrics can be more expensive to specify and compute than AOVs. For example, a quality metric may involve a comparison with a reference result which can be expensive to compute.
- **Indirect Measure of Detector Performance:** Quality metrics often have an indirect relationship to the efficacy of the outlier detector since the computed taskset result flows through the rest of the program, which may further amplify or dampen any taskset inaccuracy.

In contrast, AOVs are typically easier to specify and compute — they are specified at the task level, work only with task inputs and results, and are designed only to enable the detection of unacceptably inaccurate results, rather than provide a full quantitative measure of quality. They therefore typically involve less specification overhead than a full-blown quality metric and are more closely tied to the performance of the outlier detector.

On Synthesizing AOVs: It is possible to automatically synthesize an AOV given an application, representative input(s), and target energy savings and result quality goals. The synthesis procedure would empirically explore the space of reductions applied to the task results to find an AOV that satisfied the target energy savings and result quality goals for the application running on the representative input(s).

4.3 Application Classes and AOVs

There are some classes of applications for which designing an AOV may be challenging or difficult. For these applications, the programmer may have to choose between carefully designing a domain-specific AOV or using an AOV which is a suboptimal proxy for the quality of the task results. We next discuss two potentially problematic classes of applications.

Applications with Irreducible, Complex Task Outputs: Applications which contain large or complex irreducible task results may complicate the design of efficient AOVs. For such applications AOVs that work with all or most of the result components may be unacceptably inefficient. Computations with tasks that produce large segments of unstructured data (such as image data produced by graphics applications) may exhibit these characteristics.

Uncheckable Applications: In some applications an appropriate result distribution may be difficult to learn or infeasible to compute without imposing unacceptable overhead. Tasks with complex result constraints or, conversely, completely unpredictable results, may exhibit these characteristics.

5. Topaz Implementation

The Topaz implementation is designed to run on an approximate computing platform with one precise processor and one approximate processor. The precise processor executes the main Topaz computation, which maps the Topaz tasks onto the approximate processors for execution. The Topaz implementation contains four components: (1) a front end that translates the taskset construct into C-level Topaz API calls, (2) a message-passing layer for dispatching tasks and collecting results, (3) an outlier detector that prevents unacceptably inaccurate task results from corrupting the final result, and (4) a fault handler for detecting failed tasks.

5.1 Topaz Compiler

As is standard in implementations of task-based languages that are designed to run on distributed computing platforms [34], the Topaz implementation contains a compiler that translates the Topaz `taskset` construct into API calls to the Topaz runtime. The Topaz front-end is written in OCaml. The front-end generates task dispatch calls that pass in the input parameters defined by the `in` construct, the task index, and the identifier of the task to execute. The front end generates the code required to send the task results (the task's `out` parameters) to the precise processor running the main computation. The front end also extracts the different pieces of the taskset (the task body, `combine` block, and `transform` block) into invocable functions.

The Topaz compiler generates code for the Topaz data marshalling API, which coordinates the movement of tasks and results between the precise and approximate processors.

5.2 Topaz Runtime

The Topaz implementation works with a distributed model of computation with separate processes running on the precise and approximate processors. Topaz includes a standard task management component that coordinates the transfer of tasks and data between the precise and approximate processors. The current Topaz implementation uses MPI [12] as the communication substrate that enables these separate processes to interact. It is also obviously possible to implement Topaz on shared memory machines using standard techniques [34].

To amortize the communication overhead, Topaz divides task data into two types: *stable* data and *transient* data. Stable data does not vary over the lifetime of the taskset. It is therefore sent over with the first task, reused in place for subsequent tasks, and refreshed (sent over again) if the Topaz implementation detects three consecutive incorrect tasks. Transient data may vary across tasks and is sent over with every task.

To execute a taskset, Topaz first sends the stable data to the approximate processor. It then executes the tasks in the taskset as follows:

- **Task Creation:** The precise processor sends the task, including the transient data, to the approximate processor.
- **Task Execution:** The approximate processor receives the task and executes the task's `compute` block to obtain the task results.
- **Task Response:** The approximate processor sends a message containing the task results back to the precise processor running the Topaz main computation.
- **Outlier Detection:** The precise processor executes the task's `transform` block to compute the abstract output vector (AOV) for the task. It then runs the outlier detector on the AOV. If the detector accepts the result, Topaz runs the task's `combine` body to integrate the result into the main computation. If the detector rejects the result:
  - **Task Reexecution:** It reexecutes the task on the precise processor to obtain the correct result. It then runs the `combine` block to integrate this correct result into the Topaz main computation.
  - **Outlier Detector Update:** It uses the correct result to update the outlier detector (see Section 5.4).
  - **Stable Data Resend:** If the task is the third consecutive incorrect task, Topaz resends the stable data to the approximate processor. This feature was implemented to counteract the degradation of stable data in approximate memories (the values of stable data may degrade/change over time as the memory cells lose capacitance). Because the degradation of stable memory is only one of several sources of error, and to avoid unnecessary communication overhead, the Topaz implementation resends the stable data only after it observes three consecutive incorrect tasks.

5.3 Failed Task Detection

Topaz also handles cases where the task fails to complete execution on the approximate processor. In the event a task fails, the approximate processor notifies the main processor. Topaz reexecutes the task on the main processor and refreshes the stable data on the approximate machine. It is straightforward to extend Topaz to handle infinite loops by modelling the task runtimes and reexecuting tasks that take unacceptably long. If the precise reexecution also contains an infinite loop, it is possible to apply the same model and discard the task [5, 17, 32].

5.4 Outlier Detection

For each taskset, Topaz builds an outlier detector designed to separate incorrect AOVs from correct AOVs. Topaz performs outlier detection when a task is finished but prior to result integration. It first runs the `transform` block to obtain the AOV for the task. The outlier detector then processes the AOV to either accept or reject the task. If it rejects the AOV, Topaz reexecutes the task on the precise processor to obtain correct results and re-sends the stable task data. The AOV of the correct result is used to train the outlier detector.
We use the following notation:

- $\boldsymbol{t}$: We use t to denote a task vector, which is comprised of task data and the task results. We denote the task vector from the precise machine as t_p.
- **aov**$(\boldsymbol{t}), ||$**aov**$(\boldsymbol{t})|| = \boldsymbol{d}$: The function notation for applying the transform block, which transforms the task vector into an AOV. The resulting AOV has dimensionality d.
- **reexec**$(\boldsymbol{t})$: Reexecute the task using inputs defined by the task vector t on the reliable machine, to obtain a new (correct) task vector.
- $\boldsymbol{w}, \boldsymbol{w}_\perp, \boldsymbol{w}_\top, \boldsymbol{w}_\mu, \boldsymbol{w}_n$: We denote outlier detector regions with w. Regions are d dimensional hypercubes, where we define the bottom corner, top corner, center of mass and number of test points in a region w using $w_\perp, w_\top, w_\mu, w_n$ respectively.
- $\boldsymbol{W} = \{\boldsymbol{w}\}$: We define W as the set of regions used by the outlier detector. We use $||W||$ to denote the number of regions the detector is using.
- $\boldsymbol{l}$: The user-specified maximum number of regions the outlier detector may use. This bound limits the complexity of the outlier detector.

Testing: The outlier detector maintains a set of regions, where each region models a hypercube that encapsulates a part of the distribution of the correct AOVs. If the AOV that is being tested falls into any of these regions, the task result is considered a valid result and accepted; otherwise, the task result is rejected and the task is reexecuted. Consider t, a task vector produced by an approximate worker machine. Figure 5 presents the algorithm for determining whether to accept or reject the task:

```
bool is_acceptable($t$):
  $v$ = aov($t$)
  for $w \in W$:
    bool enc = true
    for $0 \leq i < d$:
      enc = enc && ($w_\perp(i) \leq v(i) \leq w_\top(i)$)
    if enc:
              $w_n$ += 1
       return true;
  end
  return false
```

Figure 5: Task Outlier Detection Algorithm

Training: Figure 8 presents the algorithm for training the outlier detector. The outlier detector runs this algorithm only after it rejects a task, then reruns the task on the precise processor to obtain the correct task vector t_p. If the correct AOV is already included in one of the regions of the outlier detector, the outlier detector updates the region's center of mass. Otherwise, it creates a new region containing just the

```
double score($w$,$y$):
   overlap = 1
   for $0 \leq i < d$:
      if $w_\perp(i)$ > $y_\perp(i)$:
        ov = $w_\top(i) - y_{\perp(i)}$
      else:
        ov = $y_\top(i) - w_{\perp(i)}$

      if ov $<$ 0:
        ov = 0
      overlap = overlap*ov

   if overlap $>$ 0:
      return overlap

   center_distance = 0
   for $0 \leq i < d$:
      center_distance += $(w_\mu(i) - y_\mu(i))^2$

   return -center_distance
```

Figure 6: Region Proximity Scoring Algorithm

```
region merge($w$,$y$):
   z = new region()
   $z_n = w_n + y_n$
   for $0 \leq i < d$:
      $z_\top(i)$ = max($w_\top(i)$,$y_\top(i)$)
      $z_\perp(i)$ = min($w_\perp(i)$,$y_\perp(i)$)
      $z_\mu(i) = \frac{1}{z_n} (w_\mu(i) \cdot w_n + y_\mu(i) \cdot y_n)$
   return z
```

Figure 7: Region Merge Algorithm

```
bool in_region($v$,$w$):
   enc = true
   for $0 \leq i < d$:
      enc = enc && $(w_\perp(i) \leq v(i) \leq w_\top(i))$
   return enc

tuple update_detector($t$):
  $t_p$ = reexec($t$)
  $v$ = aov($t_p$)

  for $w$ in $W$:
     if in_region($v$,$w$):
       $w_\mu$ = update_mean($w_\mu$, $v$)
       return $t_p$

  $z$ = new region()
  $z_\top = z_\perp = z_\mu = v$
  $W = W \cup v$

  if $||W|| > l$:
     best_score = 0
     best_pair = (nil,nil)
     for $(w,y) \in W \times W$:
        curr_score = score($w$,$y$)

        if best_score $<$ curr_score:
           best_score = curr_score
           best_pair = ($w$,$y$)

     merge(best_pair)
  return $t_p$
```

Figure 8: Outlier Detector Training Algorithm

AOV and adds it to the set of regions. If the outlier detector contains more than the maximum number of regions, it uses the scoring algorithm to find the two most similar regions and merges them (using the merge algorithm).

- **Scoring**: If the regions overlap, the score is the product of the magnitude of the overlapping region along each axis. The regions must overlap across all axes in order for them to be overlapping. If the regions are non-overlapping, the score is the negative distance between centers (to penalize distant, non-overlapping regions). The higher the score, the more similar the regions.
- **Merging**: The merged region is an n-dimensional hypercube enclosing both regions with a center of mass that is the weighted average of the two regions' centers of mass.

5.5 Outlier Detector with Contraction

The outlier detector has an extension which contracts the region's bounds when the reexecution rate falls below the user-defined target reexecution rate. Contracting the region's bounds increases the reexecution rate because the outlier detector rejects tasks whose AOVs would have previously fallen on the fringe of the hypercube defined by the region. This mechanism allows the outlier detector to disclude fringe values from the region and adapt to changing AOV distributions.

We augment the outlier detector so that each region also contains a Proportional-Integral-Derivative (PID) control system, which controls the region contraction. We define the notation for the control system as follows:

- $\gamma = \boldsymbol{\gamma_r}, \boldsymbol{\gamma_c}, \boldsymbol{\gamma_\int}, \boldsymbol{\gamma_\partial}$: The Proportional-Integral-Derivative (PID) control system γ comprised of the running reexecution rate ($\boldsymbol{\gamma_r}$) and the current ($\boldsymbol{\gamma_c}$), integral ($\boldsymbol{\gamma_\int}$), and discrete derivative ($\boldsymbol{\gamma_\partial}$) of the difference between the target and running reexecution rate.
- $\boldsymbol{K_c}, \boldsymbol{K_\int}, \boldsymbol{K_\partial}$: Weights associated with the PID control system.
- **ctrl**$(\boldsymbol{w})$: The control system associated with region w.
- $\boldsymbol{g}$: The user defined target reexecution rate. This bound limits the reexecution overhead and controls the contraction of the regions.

- **update_control(γ,u)**: given u, where u is 0 if the AOV was accepted or 1 if the AOV was rejected, update the control system γ.
- **get_contract**(γ): get the region contraction factor for the given control system γ.
- **update_running_rate**(γ_r, u): Given the running reexecution rate γ_r and u (as defined above), update and return the running reexecution rate.
- **update_running_integral**($\gamma_\int, \gamma_c$): Given the current integral of the difference between the target and running reexecution rate, update and return the integral.

We next discuss two algorithms: (1) updating the control system and the (2) contracting the region bounds.

Control System Get: Figure 9 presents the algorithm for getting the contraction rate. To compute the contraction rate, we use the PID control formula, a standard technique in control systems. This formula uses the integral, derivative and current reexecution rate to compute the contraction rate.

```
double get_contract($\gamma$):
    $F$ = $K_c \cdot \gamma_c + K_\int \cdot \gamma_\int + K_\partial \cdot \gamma_\partial$
    return $F$
```

Figure 9: Contraction Rate Computation

Control System Update: Figure 10 presents the algorithm for updating the control system. It first updates the running reexecution rate to include the outcome of the testing the last task. It then updates the derivative, running integral, and present difference between the running reexecution rate and target rate and returns the new control system.

```
object update_control($g$,$\gamma$,$u$):
    $\gamma_r$ = update_running_rate($\gamma_r$,$u$)
    $i$ = $\gamma_r$- $g$
    $\gamma_\partial$ = $i - \gamma_c$
    $\gamma_c$ = $i$
    $\gamma_\int$ = update_running_integral($\gamma_\int$,$\gamma_c$)
    return $\gamma$
```

Figure 10: Control System Update

Region Contract: Given the target rate, the approximate AOV, whether the AOV was accepted, and the region the AOV belongs to, the algorithm in Figure 11 adjusts the region. The algorithm operates as follows.

It first updates the region's rejection rate based on whether the last point was accepted or rejected. It then queries the control system to obtain F, the factor by which to contract the region. It clamps the contracting factor to $(0.5, 1)$ so that region doesn't shrink by more than half its size. It finally scales the region relative to its center of mass.

```
function contract_region($g$,$w$,$u$)
  $\gamma$ = ctrl($w$)
  $\gamma$ = update_control($g$, $\gamma$, $u$)

  contraction_factor = get_contract($\gamma$)
  F = min(max(contraction_factor,1),0.5)

  for $0 \leq i < d$
     $w_\top(i)$ = $w_\mu(i) + F \cdot (w_\top(i) - w_\mu(i))$
     $w_\perp(i)$ = $w_\mu(i) - F \cdot (w_\mu(i) - w_\perp(i))$
```

Figure 11: Region Contract

5.6 Mathematical Model

We next present a simple mathematical model that may be used to predict energy savings and outlier detector efficacy across the space of reexecution rates, taskset sizes, and abstraction configurations. To simplify the presentation, we assume the application executes one taskset and that all tasks in that taskset execute the same number of instructions. It is straightforward to generalize the model to include multiple tasksets and tasks with varying numbers of instructions:

- $\boldsymbol{n}$: The number of tasks in the taskset.
- $\boldsymbol{r}$: The number of reexecuted tasks. This may be approximated from a reexecution rate v using: $r = n \cdot v$
- $\boldsymbol{c}$: The number of stable data transfers.
- $\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{o}$: The size of transient data (p), stable data (q), and task results (o), in bytes.
- $\boldsymbol{d}$: The dimensionality of the AOV emitted by the transform block.
- $\boldsymbol{l}$: The maximum number of regions used by the outlier detector.
- $\boldsymbol{T}, \boldsymbol{V}, \boldsymbol{M}$: The number of instructions in a task (T), AOV transformation (V), and the total number of non-taskset instructions (M).

We also consider the following Topaz parameters:

- $\boldsymbol{\imath_p}, \boldsymbol{\imath_a}$: Topaz initialization overhead for the precise ($\imath_p$) and approximate ($\imath_a$) processors.
- $\boldsymbol{\sigma}$,$\boldsymbol{\rho}$: Per-task Topaz instruction overhead for communicating data, per byte, for sending (σ) and receiving (ρ) data.
- $\boldsymbol{\tau_m}$, $\boldsymbol{\tau_w}$: Per-task Topaz instruction overhead not associated with data communication, precise (τ_p) and approximate (τ_a) processors.
- $\boldsymbol{\alpha}$: Per-task AOV computation overhead.
- $\boldsymbol{\delta_t}$, $\boldsymbol{\delta_r}$: Per-task, per detector region, and per AOV dimension detector testing (δ_t) and training (δ_r) overhead.

We also consider Δ, the energy consumption of the approximate processor relative to the precise processor. We approximate the Topaz overhead on the precise (O_p) and approximate (O_a) processors as follows:

$$O_p = \imath_p + \sigma \cdot c \cdot q + \sigma \cdot n \cdot p + \rho \cdot n \cdot o + \tau_p \cdot n$$

$$O_a = \imath_a + \rho \cdot c \cdot q + \gamma \cdot n \cdot p + \sigma \cdot n \cdot o + \tau_a \cdot n$$

Essentially, we sum up the initialization overhead, communication overheads and per-task runtime overheads to determine the Topaz overhead.
Next we compute the detection overhead:

$$D = \delta_r \cdot r \cdot d \cdot l + \delta_t \cdot n \cdot d \cdot l + V \cdot (n + r)$$

Finally, we model the energy savings as follows:

$$S = 1 - \frac{T \cdot r + M + D + O_p + \Delta \cdot [T \cdot n + O_a]}{M + T \cdot n}$$

We use this model to approximate the energy consumption, given a particular set of parameters.

Software and Hardware Interdependences: The above model does not attempt to derive task error rates as a function of hardware error rates because the hardware error characteristics and software characteristics are often interdependent:

- **Stateful Errors**: Some classes of errors, such as memory errors, are stateful and may persist across tasks.
- **Hardware-Dependent Errors:** Some classes of errors, such as cache and memory errors, depend on which data is loaded into cache and memory (as well as the values of that data), which depends on the behavior of the application.
- **Time-Dependent Errors**: Some classes of errors, such as memory decay errors, have error rates that are functions of time (time since last refresh). Since data is refreshed when it is written, these errors can also depend on the behavior of the application.

6. Experimental Results

We present experimental results for Topaz implementations of our set of five benchmark computations:

- **blackscholes:** A financial analysis application from the Parsec benchmark suite that solves a partial differential equation to compute the price of a portfolio of European options [2].
- **water:** A computation from the SPLASH2 benchmark suite that simulates liquid water [33, 48].
- **barnes:** A computation from the SPLASH2 benchmark suite that simulates a system of N interacting bodies (such as molecules, stars, or galaxies) [48]. At each step of the simulation, the computation determines the forces acting on each body, then uses these forces to update the positions, velocities, and accelerations of the bodies [1].
- **bodytrack:** A computation from the Parsec benchmark suite that tracks the pose of the subject, given a series of frames [2].
- **streamcluster:** A computation from the Parsec benchmark suite that performs hierarchical k-means clustering [2].

We provide an artifact containing detailed instructions for installing Topaz and replicating the experiments, as well as a Virtualbox virtual machine image and EC2 AMI image containing the Topaz environment [37].

6.1 Experimental Setup

We perform our experiments on a simulated computational platform with one precise processor and one approximate processor. There is one process per processor; the processes communicate using the MPI message passing interface [12].
Caches and Memories: The computational platform contains a memory hierarchy with a mix of reliable and unreliable components.
Caches: The precise processor has a precise L1 instruction and data cache, while the approximate processor has a precise L1 instruction cache and mixed precise-approximate L1 data cache. Each processor has a larger, mixed precise-approximate L2 cache with the same cache line size as the L1 cache. The mixed precise-approximate caches are dual voltage caches where cache lines may be flagged to use reduced voltage [9]. In our evaluation all critical data is stored in high voltage cache lines, while task data is stored in low voltage cache lines. The fault characteristics for these approximate caches are modelled as probabilistic write corruptions and read corruptions [35]. Table 1 presents the fault and energy characteristics for approximate caches. In our cache hierarchy, we model 16K,4 Way,16 byte/line L1i and L1d caches and a 64K,8 Way,16 byte/line L2 cache.

Main Memory: Main memory is divided into precise, high refresh DIMMs and no refresh DIMMs. Prior work models DRAM faults as random corruptions that occur in non-refreshed cells over time. For our evaluation, we construct the per-millisecond bit flip probability model by modelling the regression of Flikker's DRAM error rates and use the energy consumption associated with DRAM refresh to compute energy savings [22]. We also consider a more sophisticated fault model where the per-millisecond error rate per row fluctuates based on DIMM activity [21]. Table 1 presents the fault and energy characteristics for approximate DRAMs. We store all stable and transient data associated with task execution in the no-refresh DRAMs. We assume 40% percent of the memory banks are approximate.
Bit Pattern Dependence: Recent research has found a significant fraction of DRAM errors are dependent on the values of the data stored in DRAM [21]. In a DIMM, bitline-bitline and bitline-wordline coupling effects impact the cell leakage. That is, neighboring cells may influence the value read

Property	Error Probability
Sram read upset probability†	$10^{-7.4}$ / read
Sram write upset probability†	$10^{-4.94}$ / write
Supply Power Saved†	80%
Fraction energy consumption†	0.2996%
L1 Cache Size	16K
L1 Cache Associativity	4 ways
L1 Cache Line Size	16 bytes
L2 Cache Size	64K
L2 Cache Associativity	4 ways
L2 Cache Line Size	16 bytes
Dram per-word error probability over time (cell/ms)††	$3 \cdot 10^{-7} \cdot t^{2.6908}$
Dram Energy Saved††	33%
Fraction energy consumption†	0.7009
Fraction DRAM no-refresh	0.40

Table 1: Hardware Model Parameters. †: from Enerj Paper ([35]). ††: from Flikkr Paper [22].

from the target cell[14]. The effect can be described as follows: if the target cell contains a logic 1 and the adjacent cell also contains a logic 1, the target cell is considered stressed and may be read as a 0. If the target cell contains a logic 0 and the adjacent cell contains a logic 0, the target cell is considered stressed and may be read as a 1.

Hardware Models: In the `basic` hardware model, we use the fixed DRAM error rate seen in previous work [35]. In our `ddep` hardware model, we model the dependence between the retention time of data in a row and the data stored in the DIMM as described above.

Hardware Emulator: To simulate the approximate hardware we implemented an approximate hardware emulator using the Pin [23] binary instrumentor. We based this emulator on on iAct, Intel's approximate hardware tool [27]. Our tool instruments memory and stack reads to access simulated versions of the caches and main memory described above, using the fault parameters from Table 1. We have included a specialized API for setting the hardware model and enabling/disabling fault injection. We track various statistics, such as the cache hit and miss rates and the average fraction of unreliable lines in cache, to populate the energy model.

We simulate the time-dependence of DRAM error probability by increasing the probability of a DRAM error in accordance with the model every 10^4 instructions, or 0.01 milliseconds (for a 2GHZ machine). For the `ddep` model, we simulate the data-dependence of the DRAM error probability by modelling a simplified version of bitline coupling. We attach a multiplier (up to 2x) to the nominal error rate in Table 1 $w_i \in \{1.0, 2.0\}$ depending on the parity of the neighbors compared to the bit being read. This model assumes contiguous regions of memory are spatially co-located in the DIMMs.

Hardware Energy Model: We next present our hardware energy savings model. We break up energy consumption into the SRAM energy consumption and DRAM energy consumption.

Cache Energy Savings: For each cache C, we consider the following parameters: the average fraction of approximate lines in the cache β_c, the size of the cache S_c, the relative energy consumption (1-savings) for low power cache lines ψ_{low}, and the energy consumption for high power cache lines ψ_{high} (1) (see Table 1). We consider the following caches: $C = \{L1I, L1D, L2\}$. The energy consumption of the approximate cache hierarchy relative to the precise cache hierarchy is as follows:

$$\Delta_C = \frac{\sum_{c \in C} \beta_c \cdot S_c \cdot \psi_{low} + (1 - \alpha_c) \cdot S_c \cdot \psi_{high}}{\sum_{c \in C} S_c \cdot \psi_{high}}$$

Main Memory Energy Consumption: We assume a constant fraction of memory, β_m, is no-refresh. We assume all banks are in use. Given the energy consumption for no refresh memories ψ_{noref} and refresh memories ψ_{ref} (see Table 1), we compute the approximate DRAM consumption relative to the fully precise energy consumption as follows:

$$\Delta_M = \frac{\beta_m \cdot \psi_{noref} + (1 - \beta_m) \cdot \psi_{ref}}{\psi_{ref}}$$

Cache + Memory Energy Consumption: We parameterize the formula with f_d, the fraction of system energy consumption from memories and f_c, the fraction of system energy consumption for caches. See Table 1 for parameters. We compute the energy consumption of the approximate cache+memory system relative to the precise system as follows:

$$\Delta = \Delta_M f_d + \Delta_C f_c$$

6.2 Benchmark Executions

We present experimental results that characterize the accuracy and energy consumption of Topaz benchmarks under a variety of conditions:

- **Precise:** We execute the entire computation, Topaz tasks included, on the precise processor. This execution provides the precise, fully accurate results that we use to evaluate the accuracy of the other executions.
- **No Topaz:** We attempt to execute the full computation, main Topaz computation included, on the approximate processor. For all the benchmarks, this computation terminates with a segmentation violation.
- **No Outlier Detection:** We execute the Topaz main computation on the precise processor and the Topaz tasks on the approximate processor with no outlier detection. If a task crashes and does not return a result, the Topaz implementation reexecutes the task on the precise processor. Stable data is only resent when three consecutive tasks crash. We integrate all of the results from approximate tasks that do not crash into the main computation.
- **Outlier Detection:** We execute the Topaz main computation on the precise processor and the Topaz tasks on the approximate processor with outlier detection and reexecution as described in Section 5.

Benchmark	Model	No Outlier Detector	Outlier Detector
barnes	`basic`	inf	0.158229%
blackscholes	`basic`	inf	0.135584%
bodytrack	`basic`	73.6327%	0.161024%
streamcluster	`basic`	0.6219	0.6344
water	`basic`	nan	0.000469%
barnes	`ddep`	inf	0.075927%
blackscholes	`ddep`	inf	0.025791%
bodytrack	`ddep`	73.6327%	0.317984%
streamcluster	`ddep`	0.6321	0.6344
water	`ddep`	nan	0.000383%

Table 2: End-to-End Output Quality

6.3 Benchmark AOVs

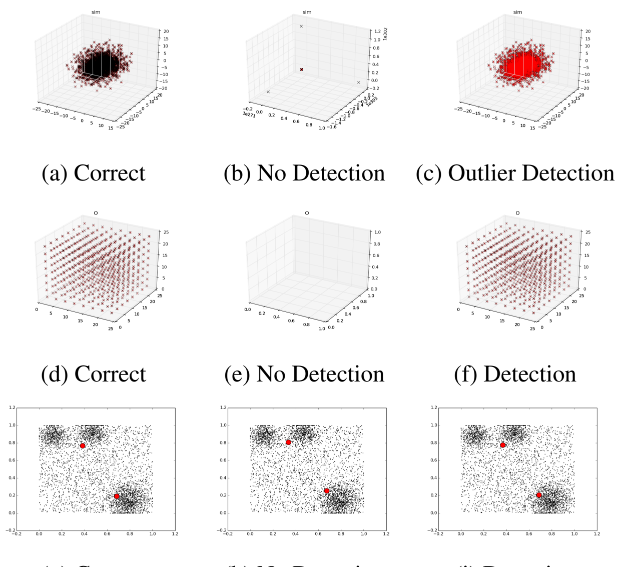
We use the following AOVs:

- **barnes**: Each task is the force calculation computation for a particular body. The tasks are batched such that each task computes the velocity and acceleration of two bodies. The AOV is the amplitude of the velocity and acceleration vectors and the scalar result phi.
- **bodytrack**: Each task is the weight calculation of a particular pose in a given frame on the approximate machine. The AOV is the position vector and weight.
- **streamcluster**: Each task is a subclustering operation in the bi-level clustering scheme. The AOV is the gain of opening/closing the chosen centers and the sum of weights assigned to the points in the subclustering operation.
- **water:** Two computations execute on the approximate machine: (1) the intermolecular force calculation, which determines the motion of the water molecules (interf) (2) the potential energy estimation, which determines the potential energy of each molecule (poteng). The tasks are batched such that each task computes sixty four forces/energies. The AOVs are as follows:
  - **interf**: The AOV is the cumulative force exerted on the H,O,H atoms in the x,y and z directions and the scalar result incr.
  - **poteng**: The AOV is the sum of the magnitude of potential energy in the x,y and z directions.
- **blackscholes**: Each task is a price prediction on the approximate machine. These tasks are batched such that each task computes 64 prices. The single dimensional AOV is the sum of the 64 computed prices.

6.4 End-to-End Output Quality

Table 2 presents the end-to-end output quality metrics for the (1) No Outlier Detection and (2) Outlier Detection cases. With one exception (streamcluster), executing the benchmarks with no outlier detection produces unacceptably inaccurate results. With outlier detection, the output quality is always acceptable and typically very small.

barnes: The output quality metric is the percent positional error (PPE) of each body. With outlier detection, the PPE between the precise and approximate executions is a fraction of a percent and visually indistinguishable in the output. See Figures 12a and 12c, which plot the positions of the bodies at the end of the simulation. Figure 12c overlays the positions of the bodies from the precise execution (in red) and the approximate execution with outlier detection (in black). Figure 12b plots the output from the approximate computation without outlier detection (almost all of the bodies lie outside the plotted range).


(a) Correct (b) No Detection (c) Outlier Detection

(d) Correct (e) No Detection (f) Detection

(g) Correct (h) No Detection (i) Detection

Figure 12: Visual Representation of Output Quality. (a-c) Barnes. (d-f) Water. (g-i) Streamcluster.

blackscholes: The quality metric is the cumulative error of the predicted stock prices with respect to the total portfolio value. Without outlier detection, the percent portfolio error is many times more than the value of the portfolio. With outlier detection, the portfolio error is a fraction of a percent of the value of the portfolio.

bodytrack: The quality metric is the weighted percent error in the selected pose for each frame. In both the `basic` and `ddep` hardware models, we observed wildly incorrect pose choices. With outlier detection, the selected poses are visually good matches for the body. Figures 2c and 2a present a visual comparison.

streamcluster: The quality metric is the weighted silhouette score, which is a measure of cluster quality - the closer to one, the better the clusters. The silhouette score with fully precise execution is 0.557. Figures 12g-12i) plot the input data (black dots) and the cluster centers (red dots). The precise (Figure 12g) cluster centers are visually similar to the cluster centers produced by the approximate machine with no outlier detection (Figure 12h) and with outlier detection (Figure 12i). We attribute this result the robustness of the computation, which is inherently resilient to outliers. More-

Benchmark	Hardware Model	Correct Accepted (%)	Correct Rejected (%)	Error Accepted (%)	Error Rejected (%)	Rejection Accuracy (%)	Errors Detected (%)
barnes	`basic`	94.48%	0.19%	2.94%	2.38%	92.58%	44.74%
bodytrack	`basic`	87.58%	0.16%	7.67%	4.58%	96.62%	37.39%
water-interf	`basic`	95.30%	0.32%	1.71%	2.67%	89.37%	60.96%
water-poteng	`basic`	99.51%	0.26%	0.02%	0.20%	43.59%	89.47%
blackscholes	`basic`	98.57%	0.04%	1.06%	0.33%	90.00%	24.06%
streamcluster	`basic`	98.34%	0.14%	0.37%	1.15%	89.15%	75.66%
barnes	`ddep`	94.22%	0.20%	3.11%	2.47%	92.59%	44.26%
bodytrack	`ddep`	77.34%	0.15%	16.04%	6.46%	97.67%	28.71%
water-interf	`ddep`	95.44%	0.33%	1.62%	2.61%	88.81%	61.71%
water-poteng	`ddep`	99.49%	0.26%	0.04%	0.21%	44.54%	85.48%
blackscholes	`ddep`	98.70%	0.04%	0.94%	0.33%	89.80%	25.88%
streamcluster	`ddep`	62.24%	0.11%	36.68%	0.98%	89.90%	2.59%

Table 3: Overall Outlier Detector Effectiveness

over, the precise machine performs the top-level clustering operation, since errors in the top level clustering operation would have a disproportionate effect on the output. These two properties make introducing non-existent clusters and eliminating well-represented clusters unlikely — for this to occur, the approximate hardware would have to systematically produce errors that result in selecting (or excluding) particular subsets of points.

water: The output quality metric is the percent positional error (PPE) of each molecule. With outlier detection, the PPE between the precise and approximate executions is a fraction of a percent. The approximate molecule positions with outlier detection (Figure 12f) are visually indistinguishable from the precise molecule positions (Figure 12d). Figures 12d-12f plot the positions from precise execution (in red), and the approximate execution (in black). Figure 12e plots the output from the approximate execution without outlier detection (almost all of the molecules lie outside the plotted range).

6.5 Outlier Detector Effectiveness

Table 3 presents data that characterizes the overall effectiveness of the outlier detector. The third through sixth columns present a breakdown of all of the tasks into correct tasks that were accepted (third column) or rejected (fourth column) by the outlier detector and incorrect tasks that were accepted (fifth column) or rejected (sixth column). The seventh column (Rejection Accuracy) presents the percentage of rejected tasks that contain errors. The eighth column (Errors Detected) presents the percentage of tasks with errors that were rejected.

These numbers indicate that, in general, (1) the majority of the tasks are correct and accepted by the outlier detector (column three, Correct Accepted) and (2) a large percentage of the rejected tasks contain errors (with the exception of water-poteng) (column seven, Rejection Accuracy). Even though the outlier detector rejects at most a few percent of the tasks, the data in Table 2 show that if the outputs from these few percent of the tasks are incorporated into the main computation, they produce unacceptably inaccurate results.

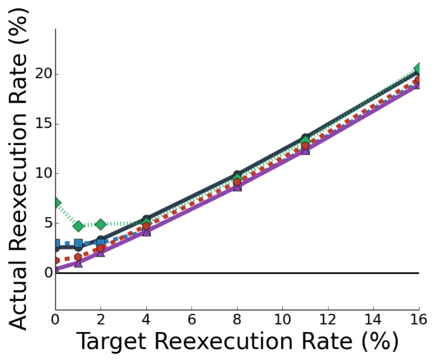
Benchmarks	Model	Baseline	Detect & Reexecute	Full Topaz
barnes	`basic`	17.47%	14.77%	13.02%
blackscholes	`basic`	16.20%	14.62%	9.94%
bodytrack	`basic`	12.70%	8.60%	7.69%
streamcluster	`basic`	16.87%	15.62%	11.03%
water	`basic`	18.41%	15.12%	12.43%
barnes	`ddep`	17.47%	14.76%	13.02%
blackscholes	`ddep`	16.02%	14.41%	9.70%
bodytrack	`ddep`	12.88%	6.51%	5.02%
streamcluster	`ddep`	16.89%	15.58%	11.03%
water	`ddep`	18.41%	15.37%	12.82%

Table 4: Energy Savings, `basic` and `ddep` Hardware Models

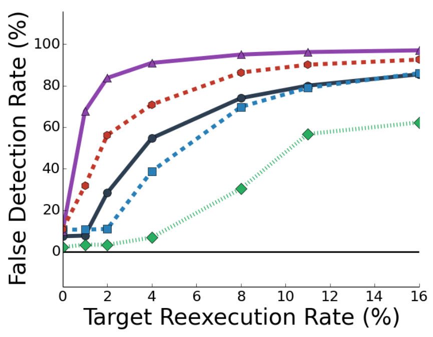
Across applications, many of the incorrect tasks produce results that are embedded within the range of correct outputs. The outlier detector therefore accepts these tasks (compare column five, Error Accepted, and column six, Error Rejected). Since the errors for these tasks are small, they are acceptably inaccurate and have (very) acceptable impact on the overall end-to-end output quality. The outlier detector is effective at detecting the (in most applications relatively few) unacceptably inaccurate tasks that would cause the application to produce unacceptable end-to-end quality.

6.6 Energy Savings

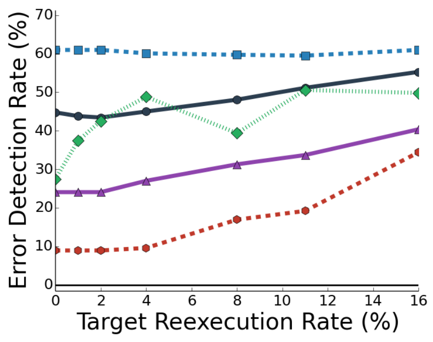
Table 4 presents the energy savings for our benchmark applications. We break the computation into the following components: instruction count of the main computation on the precise processor M, instruction count of tasks executing on the approximate processor A, instruction count of outlier detection routine and task reexecution on the precise processor R, and the overhead from the Topaz implementation and marshalling/unmarshalling task data L. For the simple model presented in Section 5.6 (a Topaz program with a single taskset with n tasks, each of which executes T instructions, r reexecuted tasks, D detector overhead, and O_p and O_a Topaz overhead on the precise and approximate processors, respectively), $A = T \cdot n$, $R = T \cdot r + D$, and $L = O_p + \Delta \cdot O_a$. For each benchmark, Table 4 presents the following three energy savings metrics:



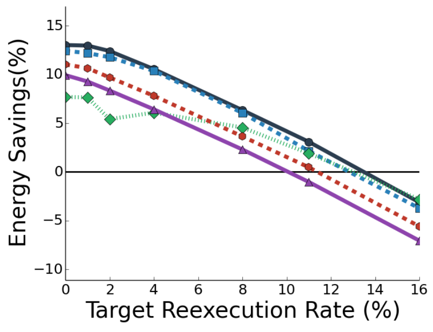
(a) Actual Reexecution Rate



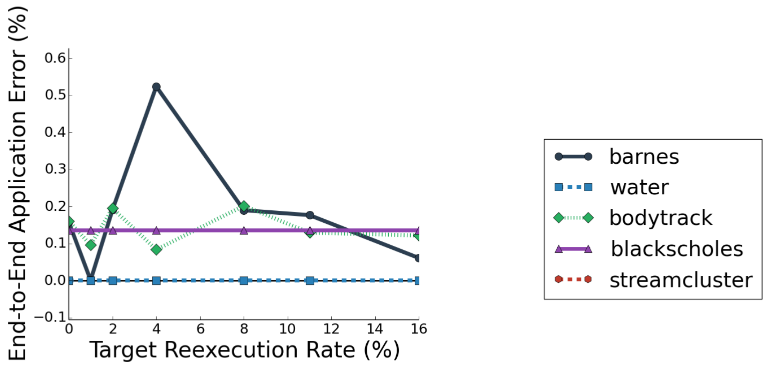
(b) False Detection Date



(c) Error Detection Rate



(d) Energy Savings



(e) End-to-End Application Error

Figure 13: Tradeoff Analysis Graphs

Target Reexecution Rate	0.00%	1.00%	2.00%	4.00%	8.00%	11.00%	16.00%
Silhouette Score	0.634	0.635	0.629	0.635	0.634	-1.000	0.635

Table 5: Output Quality vs Reexecution Rate for Streamcluster

Baseline: The baseline energy savings available from executing tasks on the approximate processor:

$$1 - \frac{M + \Delta \cdot A}{M + A}$$

Detect & Reexecute: The energy savings including outlier detection and task reexecution on the precise processor:

$$1 - \frac{M + \Delta \cdot A + R}{M + A}$$

Full Topaz: The energy savings, with outlier detection and task reexecution, accounting for the Topaz overhead, including the overhead devoted to coordinating the distribution of the computation across the precise and approximate processors:

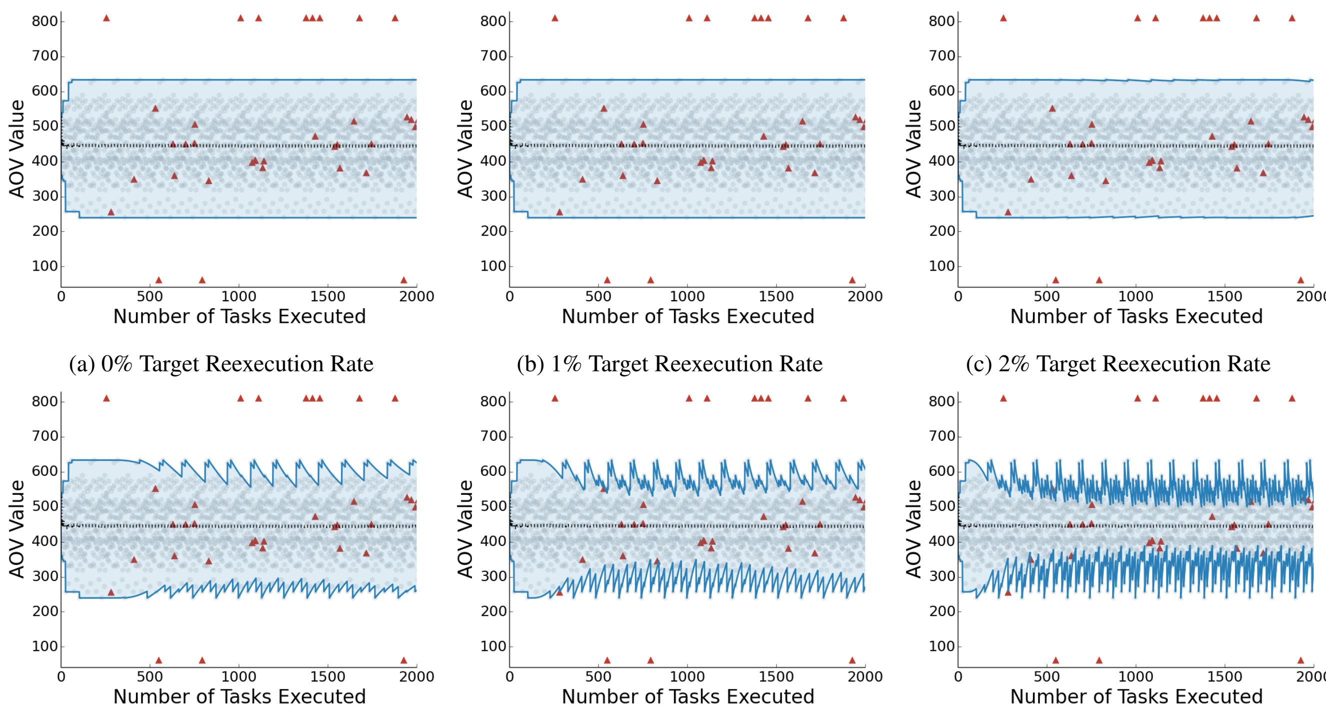
$$1 - \frac{M + \Delta \cdot A + R + L}{M + A}$$

Each row presents the Baseline, Detect & Reexecute, and Full Topaz savings metrics for the application specified in the first column (Benchmark). The maximum attainable savings is 19.185%, which is attained when all of the lines in the approximate caches are approximate lines, there is no overhead, and all of the computation executes on the approximate processor. The numbers in the table are for a target reexecution rate of zero — the outlier detector bounds grow to include the correct task results and never contract. At higher target reexecution rates, the outlier detector control algorithm contracts the bounds, detects more incorrect tasks, and the energy consumption and number of reexecuted tasks grows (see Figures 13d and 13c).

6.7 Crossover Analysis

We evaluate the effect of different target reexecution rates on percent errors detected and output quality. Figures 13a-13e present the results for each benchmark (we report results for the `basic` hardware model).

For all applications, the Topaz control system effectively matches the actual task reexecution rate with the target reexecution rate (Figure 13a). For some benchmarks, the control system fails to meet the target rate for low reexecution rates



(a) 0% Target Reexecution Rate

(b) 1% Target Reexecution Rate

(c) 2% Target Reexecution Rate

(d) 4% Target Reexecution Rate

(e) 8% Target Reexecution Rate

(f) 16% Target Reexecution Rate

Figure 14: Time Series Visual Representation of Outlier Detector Behavior. Gray circles indicate correct tasks, red triangles indicate tasks with errors, the blue lines indicate the outlier detector bounds, and the black dashed line is the center of mass of the tasks that the outlier detector accepts. The shaded region is the region the outlier detector accepts.

because too many AOVs fall outside the envelope defined by previously seen AOVs to attain the reexecution rate.

As the target reexecution rate increases, the percentage of detected errors (and reexecuted tasks) increases (Figure 13c). Increasing the target reexecution rate also drastically increases the percent of reexecuted tasks that have no errors (Figure 13b) — increasing the reexecution rate causes the detector to contract its regions, cutting into the periphery of the AOV distribution. Although there are some errors at the periphery, most of the AOVs are correct, which is why the false detection rate increases more quickly than the error detection rate.

The task reexecutions decrease the energy savings (Figure 13d), with the crossover point from energy savings to increased energy consumption occurring between a target reexecution rate of 10% (blackscholes) to 14% (barnes,water). The crossover point occurs when the application curve intersects the 0% energy savings axis (black horizontal line, Figure 13d).

Figure 13e presents the end-to-end application error for all applications except streamcluster (all of these application error metrics are computed relative to the output from the precise execution). Table 5 separately presents the Silhouette end-to-end quality metric for streamcluster. The end-to-end output quality (Figure 13e and Table 5) is largely unaffected by the target reexecution rate. We attribute this phenomenon to (1) the outlier detector's ability to effectively detect and reexecute unacceptably inaccurate tasks at all target reexecution rates and (2) the tolerance of our set of approximate applications to the remaining acceptably inaccurate tasks. Note that streamcluster has an anomalous score (-1) at 11% reexecution rate. We attribute this anomaly to the algorithm dropping a cluster (two to one cluster) — where the silhouette score of one cluster is always -1.

Figures 14a-14f graphically illustrate the operation of the outlier detector for blackscholes running with different target reexecution rates. Each gray circle indicates the AOV value for a correctly executed task. Each red triangle indicates the AOV value for a task with an error. The x axis plots time (measured in the number of Topaz tasks executed) while the y axis plots AOV value. So a task that executes at time x with AOV value y will generate a red triangle or gray circle at the point x,y in the figure.

The blue lines indicate the upper and lower bounds of the outlier detector as a function of time. The black dashed line is the center of mass of the region. For small target reexecution rates, the bounds contract slowly about the center of mass (or not at all for a zero target reexecution rate). For larger target reexecution rates, the bounds contract more aggressively. When a reexecuted task produces an AOV value outside of the outlier detector bounds, the outlier detector expands one of the bounds to include the produced AOV value.

The range of the y axis includes all correct AOV values. Incorrect AOV values that lie outside the range of the y axis are pinned at either the minimum or maximum visible y values. The graphs clearly show the bimodal distribution of AOV values. Some of the incorrect values are embedded within the gray region of correct AOV values, while others (the lines of red triangles along the top and bottom of each graph) are widely removed from the gray region of correct AOV values. These graphs highlight how the outlier detector (1) accepts incorrect tasks whose results are within the region of observed correct results and (2) detects and reexecutes tasks with clearly unacceptably inaccurate results.

These graphs also indicate that most of the incorrect tasks produce results that are within the gray region of correct AOV values (also see Table 3). The ability of blackscholes to incorporate the values produced by these accepted incorrect tasks into its computation while still producing acceptably accurate end-to-end results (Figure 13e) highlights its robustness to small inaccuracies in the results that the tasks produce.

7. Discussion

We next discuss appropriate and inappropriate application classes for Topaz, the guarantees Topaz provides and the interplay between the Topaz language design, hardware and implementation. We first note that any application in which exact results are both feasible to obtain and either required or highly desirable is not appropriate for Topaz. Examples of such applications include compilers, traditional relational databases, theorem provers, and much operating system code. For Topaz to be at all applicable, the application must have some flexibility to generate approximate results. A secondary requirement is that the majority of the computation must consist of independent tasks that fit the basic Topaz taskset construct. Examples of applications that may not fit naturally into this pattern include applications with long sequential dependence chains such as text formatters and standard compression/decompression computations.

7.1 Appropriate Application Classes

We next outline several application characteristics that make applications a good match for Topaz:

- **Effective Sanity Checks:** For the Topaz outlier detector to function effectively, it must be able to efficiently detect and discard unacceptably inaccurate tasks — in other words, there must be an efficient and acceptably accurate sanity check for acceptably accurate tasks. And it must be possible to express this sanity check within the Topaz AOV framework. Examples of computations with such sanity checks include checkable computations (whose correctness can be checked exactly with an efficient checker) [26] and computations whose unacceptably inaccurate tasks fall within a range that is disjoint from any acceptably accurate (including correct) tasks. Examples include scientific computations, machine learning computations, big data computations, and financial applications [26]. For all of these applications acceptably accurate results are typically constrained by the underlying domain-specific phenomena to fall within a relatively narrow range suitable for outlier detection.
- **Amortizable Tasks**: The use of Topaz entails task management overheads, including the AOV and outlier detector overheads. Topaz therefore works best with tasks that can profitably amortize the Topaz overhead. In our experience, the natural task granularity of many applications is large enough for this purpose. And even when the natural granularity is too small, we have found that it is often possible to batch the application's natural tasks together into Topaz tasks that are more than large enough to amortize the Topaz overhead. Once again, scientific, machine learning, and financial applications typically perform more than enough independent computations to enable the construction of Topaz tasks that are large enough to profitably amortize the Topaz overhead.
- **Acceptable Task Error Rates:** Topaz detects unacceptably inaccurate computations at the granularity of tasks. The larger the task, the higher the probability that the task will encounter an error that introduces an unacceptable inaccuracy. The tasks must therefore be small enough so that an acceptably small number of tasks encounters such errors. Acceptable Topaz tasks sizes are therefore bounded below by the need to amortize Topaz task management overhead and bounded above by the need to avoid large numbers of unacceptably inaccurate tasks.

We next discuss two concerns that can make applications a poor fit for the basic Topaz approach:

- **Externally Visible Effects:** Tasks with externally visible effects (such as interacting with the network or manipulating sensors, actuators, or system configurations) are typically a poor match for Topaz since they are (in general) not idempotent and cannot be successfully reexecuted if discarded by the Topaz outlier detector.
- **Outlier Detector Accuracy:** The Topaz outlier detector relies on relatively uniform task input/output behavior to operate successfully. If this behavior changes too quickly or unpredictably, the outlier detector may reexecute too many tasks (so that any gains are overcome by repeated task executions). Rapidly shifting input/output distributions that cannot be accurately captured with an appropriate AOV or monotonically increasing outputs are two examples of task input/output behavior that can cause counterproductive outlier detector behavior. A related concern is short, nonrepeating tasksets. In this case the initial learning phase of the Topaz outlier detector, which exhibits a high reexecution rate as the outlier detector converges to its steady state, may comprise a large percent-

age of the total taskset execution with the initial reexecution overhead not successfully amortized by the steady state taskset execution.

Examples of applications with rapidly changing characteristics that may interact poorly with outlier detection include simulations of turbulent fluid flow and explosive combustion simulations.

These concerns highlight how Topaz is most appropriate for compute-intensive applications with many appropriately sized tasks.

7.2 Static Accuracy Guarantees and Topaz

There are two kinds of static accuracy guarantees that would be appropriate for Topaz programs: taskset guarantees, which would characterize the accuracy of the task and taskset results, and end-to-end guarantees, which would characterize the overall accuracy of Topaz computations. Topaz currently provides neither kind of guarantee. We next outline the challenges associated with obtaining each kind of guarantee:

- **Taskset Guarantees:** Topaz is designed to work with hardware platforms that may generate arbitrarily inaccurate results, with the Topaz outlier detector detecting and reexecuting unacceptably inaccurate results. The key to obtaining taskset guarantees is bounding the inaccuracy resulting from tasks that the outlier detector accepts. With the current Topaz model of computation, there is no bound on this inaccuracy — an approximate task may produce an incorrect result that is very close to previously observed results (so the outlier detector accepts the incorrect result) while the correct result is arbitrarily far away. Additional guarantees about the operation of the underlying hardware platform or the values of task results would be required to obtain any Topaz taskset accuracy guarantees. For example, it should be possible to obtain Topaz taskset accuracy guarantees for Topaz programs running on approximate hardware that provides bounds on the inaccuracy of the results that it may compute.
- **End-to-End Guarantees:** To obtain an end-to-end accuracy guarantee, one must first obtain a Topaz taskset guarantee, then reason about how the bounded taskset inaccuracy propagates through the Topaz application to influence the outputs that the application produces. Potential options for such guarantees include worst-case and probabilistic guarantees. While techniques exist for providing such guarantees, they have yet to be shown to scale to large applications.

7.3 Interplay Between Language Design, Hardware, and Runtime System

The Topaz language design decisions enable Topaz to work with approximate computing platforms that may produce arbitrarily inaccurate results:

- **Distributed Memory Model**: Topaz employs a distributed memory model for program data. This mechanism confines any data corruption within the worker process executing the approximate tasks. Topaz therefore supports platforms with hardware errors that may trigger cascading software faults or data corruption.
- **Self-Contained, Stateless Tasks**: Tasks are stateless and self contained. In the event the Topaz worker process (or the approximate processor) crashes, the approximate processor can begin receiving tasks after reboot without any checkpointing.
- **Abstract Output Vectors**: AOVs provide a powerful mechanism for obtaining the data on which the outlier detector operates. AOVs also support efforts to automate the selection of an appropriate abstraction for outlier detection.
- **Outlier Detector**: The outlier detector enables Topaz to exclude unacceptable results and adapt to the error characteristics of the system and the application (via the control system). This technique is vital to operating on systems with a difficult to formalize and volatile error characteristics.

8. Related Work

We discuss related work in software and hardware approximate computing.

Software Approximate Computing. Task skipping [32] and loop perforation [41] both apply approximate computing at the software level with no hardware support. The results indicate that this approach can deliver significant performance improvements and energy savings.

Approximate Hardware Platforms. Researchers have previously proposed multiple hardware architecture designs that improve the performance or energy consumption of processors by reducing precision or increasing the incidence of error [8, 9, 15, 18, 19, 28, 30, 35, 44, 46, 47]. Researchers have also proposed a variety of approximate DRAM and SRAM designs. To save DRAM energy, researchers have proposed downgrading the refresh rate [9, 22] or using a different underlying substrate [36], For SRAM, researchers have proposed decreasing operating voltage [9, 39]. Researchers have also designed hardware platforms with a single precise core and multiple approximate cores [19, 29]. Topaz targets all of these approximate hardware models and has been tested on a subset of these models.

Not all approximate hardware is easy to model. Researchers have done experiments on systems with hard to quantify fault characteristics, such as undervolted and overclocked hardware, and found them hard to model or quantify [16, 19, 25]. In industry, hardware component manufacturers sacrifice some performance, die space and energy into error correction hardware to reduce the effects of manufacturing defects [13]. These difficult to model pieces of hardware are good targets for Topaz.

Programming Models for Approximate Hardware. Researchers have previously investigated using task checkers for approximate hardware. Relax is a task-based programming language and ISA extension that supports manually developed checker computations [7]. If a checker computation fails, Relax allows a developer to specify a custom recovery mechanism. Although Topaz is also task-based, it operates entirely in software, and does not require ISA extensions. In contrast to Relax, Topaz uses online outlier detection as an automatic checking mechanism and automatically re-executes tasks as a default recovery mechanism.

Jade is a task-based language for parallel computing [34]. It is possible to execute Jade tasks on unreliable hardware or software platforms, with the implementation discarding failed tasks to produce acceptably accurate approximate results [32]. The Topaz implementation, in contrast, reexecutes failed tasks (as well as tasks that the outlier detector rejects). It is obviously possible to modify the Topaz implementation to discard (instead of reexecuting) failed and/or outlier tasks. The Jade results suggest that this alternate implementation strategy would also deliver acceptably accurate results.

Researchers have also investigated language features that allow developers to annotate approximate data and operations. Flikker provides a C API, which developers can use to specify data to store in unreliable memory [22]. Enerj provides a type system to specify data that may be placed in unreliable memories or computed using approximate computations [35]. Rely provides a specification language that developers can use to specify quantitative reliability requirements and an analysis that verifies that Rely programs satisfy these requirements when run on approximate hardware [6]. Unlike these systems, Topaz does not require the developer to provide such fine grain annotations.

Recently, researchers have sought to lessen programmer burden by automatically determining the placement of approximate structures and annotations. Researchers have demonstrated that it is possible to automatically find critical input regions and code [4, 32]. Chisel allows a developer to express accuracy and reliability constraints and automates the placement of unreliable operations for a Rely program by framing the provided energy model and program as an integer programming problem [26]. Like Chisel, ExpAX searches for approximations that minimize energy usage subject to accuracy and reliability constraints using genetic programming [31]. Both Chisel and ExpAX remove the need for the programmer to specify approximate operations. But, unlike Topaz, they still require the programmer provide software and hardware specifications.

9. Conclusion

Topaz enables developers to cleanly express the approximate tasks present in their approximate computations. The Topaz implementation then maps the tasks appropriately onto the underlying approximate computing platform and manages the resulting distributed approximate execution. The Topaz execution model gives approximate hardware and software designers the freedom and flexibility they need to produce maximally efficient approximate hardware and software — the Topaz outlier detection and reexecution algorithms enable Topaz computations to work with approximate computing platforms even if the platform may occasionally produce arbitrarily inaccurate results. Topaz therefore supports emerging and future approximate computing platforms that promise to provide an effective, energy-efficient computing substrate for existing and future approximate computations.

Acknowledgments

This research was supported by DARPA (Grant FA8650-11-C-7192). We would like to thank the anonymous paper and artifact reviewers for their useful feedback. The second author would like to thank Subhasish Mitra for interesting discussions on approximate computing.

References

[1] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(4):446–449, 1986.

[2] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[3] M. Blum and S. Kanna. Designing programs that check their work. STOC, 1989.

[4] M. Carbin and M. Rinard. Automatically identifying critical input regions and code in applications. ISSTA, 2010.

[5] M. Carbin, S. Misailovic, M. Kling, and M. Rinard. Detecting and escaping infinite loops with Jolt. ECOOP, 2011.

[6] M. Carbin, S. Misailovic, and M. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. OOPSLA, 2013.

[7] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. ISCA '10.

[8] P. Düben, J. Joven, A. Lingamneni, H. McNamara, G. De Micheli, K. Palem, and T. Palmer. On the use of inexact, pruned hardware in atmospheric modelling. *Philosophical Transactions of the Royal Society*, 372, 2014.

[9] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, pages 301–312, 2012.

[10] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. MICRO, 2012.

[11] S. Fedotov. Lecture 7, introduction to financial mathematics. www.maths.manchester.ac.uk/ sf/20912lecture7.pdf.

[12] W. Gropp and E. Lusk. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994.

[13] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *DAC*, 2013.

[14] S. Irobi, Z. Al-Ars, and S. Hamdioui. Detecting memory faults in the presence of bit line coupling in sram devices. In *ITC*, 2010.

[15] H. Kaul, M. Anders, S. Mathew, S. Hsu, A. Agarwal, F. Sheikh, R. Krishnamurthy, and S. Borkar. A 1.45ghz 52-to-162gflops/w variable-precision floating-point fused multiply-add unit with certainty tracking in 32nm cmos. In *ISSCC*, pages 182–184, 2012.

[16] V. Kiriansky and S. Amarasinghe. Reliable computation on unreliable hardware: Can we have our digital cake and eat it?

[17] M. Kling, S. Misailovic, M. Carbin, and M. Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. OOPSLA, 2012.

[18] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Mitigating soft error failures for multimedia applications by selective data protection. CASES, 2006.

[19] L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra. Ersa: error resilient system architecture for probabilistic applications. DATE'10.

[20] N. Leveson, S. Cha, J. C. Knight, and T. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE TSE*, 1990.

[21] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu. An experimental study of data retention behavior in modern dram devices: Implications for retention time profiling mechanisms. In *ISCA*, 2013.

[22] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: saving dram refresh-power through critical data partitioning. In *ASPLOS*, pages 213–224, 2011.

[23] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[24] G. S. May and C. J. Spanos. *Fundamentals of semiconductor manufacturing and process control*. John Wiley & Sons, 2006.

[25] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta. Evaluating the effectiveness of model-based power characterization. In *USENIX Annual Technical Conf*, 2011.

[26] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: reliability-and accuracy-aware optimization of approximate computational kernels. In *OOPSLA*, 2014.

[27] A. K. Mishra, R. Barik, and S. Paul. iact: A software-hardware framework for understanding the scope of approximate computing. 2014.

[28] S. Narayanan, J. Sartori, R. Kumar, and D. Jones. Scalable stochastic processors. DATE, 2010.

[29] D. S. Nikolopoulos, H. Vandierendonck, N. Bellas, C. D. Antonopoulos, S. Lalis, G. Karakonstantis, A. Burg, and U. Naumann. Energy efficiency through significance-based computing. *Computer*, 47(7):82–85, 2014.

[30] K. Palem. Energy aware computing through probabilistic switching: A study of limits. *TC*, 2005.

[31] J. Park, X. Zhang, K. Ni, H. Esmaeilzadeh, and M. Naik. Expectation-oriented framework for automating approximate programming. Technical Report GT-CS-14-05, Georgia Institute of Technology, 2014.

[32] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. ICS, 2006.

[33] M. C. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS*, pages 324–334, 2006.

[34] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20 (3):483–545, 1998.

[35] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[36] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[37] A. Sara and M. C. Rinard. Topaz artifact. rhino.csail.mit.edu/sachour/topaz.oopsla15.artifact, 2015.

[38] B. Schroeder, E. Pinheiro, and W.-D. Weber. Dram errors in the wild: a large-scale field study. In *PER*, 2009.

[39] M. Shoushtari, A. Banaiyan, and N. Dutt. Relaxing manufacturing guard-bands in memories for energy savings. Technical Report CECS TR 10-04, UCI, 2014.

[40] M. M. Shulaker, G. Hills, N. Patil, H. Wei, H.-Y. Chen, H. S. P. Wong, and S. Mitra. Carbon nanotube computer. *Nature*, 501 (7468):526–530, 09 2013.

[41] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. FSE, 2011.

[42] H. A. Simon. *The architecture of complexity*. Springer, 1991.

[43] V. Sridharan and D. Liberty. A study of dram failures in the field. In *SC*, 2012.

[44] R. St Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger. General-purpose code acceleration with limited-precision analog computation. ISCA, 2014.

[45] P. Stanley-Marbell and M. Rinard. Lax: Driver interfaces for approximate sensor device access. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, 2015.

[46] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Trans. VLSI Syst.*, 8(3):273–286, 2000.

[47] S. Venkataramani, V. Chippa, S. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. MICRO, 2013.

[48] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 24–36, 1995.