Significance-Aware Program Execution on Unreliable Hardware

KONSTANTINOS PARASYRIS, VASSILIS VASSILIADIS,
CHRISTOS D. ANTONOPOULOS, SPYROS LALIS, and NIKOLAOS BELLAS,
Centre for Research and Technology, Hellas & University of Thessaly

This article introduces a significance-centric programming model and runtime support that sets the supply voltage in a multicore CPU to sub-nominal values to reduce the energy footprint and provide mechanisms to control output quality. The developers specify the significance of application tasks respecting their contribution to the output quality and provide check and repair functions for handling faults. On a multicore system, we evaluate five benchmarks using an energy model that quantifies the energy reduction. When executing the least-significant tasks unreliably, our approach leads to 20% CPU energy reduction with respect to a reliable execution and has minimal quality degradation.

CCS Concepts: • **Computer systems organization** → **Reliability**; • **Theory of computation** → **Program semantics;**

Additional Key Words and Phrases: Significance aware computing, unreliable hardware, energy efficiency, quality of output

ACM Reference Format:
Konstantinos Parasyris, Vassilis Vassiliadis, Christos D. Antonopoulos, Spyros Lalis, and Nikolaos Bellas. 2017. Significance-aware program execution on unreliable hardware. ACM Trans. Archit. Code Optim. 14, 2, Article 12 (April 2017), 25 pages.
DOI: http://dx.doi.org/10.1145/3058980

1. INTRODUCTION

The scalability of semiconductor manufacturing process, as predicted by Moore's law, has been the driving force of the increase in the capabilities of computer systems. However, scaling transistors to lower geometries tends to amplify the effects of manufacturing variability, resulting in less deterministic electrical—and thus performance and power—transistor characteristics. The stochasticity of transistor characteristics leads to reduced chip yields and increased voltage and frequency guard bands. These guard bands are pessimistic, as they have to compensate for the worst-case scenarios and combinations of non-determinism, switching patterns, temperature and aging effects. According to Das et al. [2006] the average power cost of guard bands is roughly 35%. However, most of the time, these guard bands represent mere overhead, as worst-case scenarios and combinations will appear very seldom during application execution.

The main reason for having these pessimistic guard bands and energy inefficiency is that modern computing systems execute programs under strict correctness requirements. But in several application domains, it is not the precise result that matters to

This work is supported by the FP7/FET Open research programme of the European Commision, under grant agreement FP7- 323872, project SCoRPiO.
Authors' addresses: K. Parasyris; email: koparasy@inf.uth.gr; V. Vassiliadis; email: vasiliad@inf.uth.gr; C. D. Antonopoulos; email: cda@inf.uth.gr; S. Lalis; email: lalis@inf.uth.gr; N. Bellas; email: nbellas@inf.uth.gr.

DOI: http://dx.doi.org/10.1145/3058980

the user but rather an approximation of the output [Doucet et al. 2000; Goiri et al. 2015]. For example, for big data analytics performed in large-scale clusters, one may be interested in rough data classification rather than the exact value of the end result. More generally, within each program, all computations are treated as equally important for the quality of the end result, although in most cases this assumption is not true.

Approximate computing has recently attracted the interest of the research community as a means to improve the energy efficiency of such computations [Sampson et al. 2011; Baek and Chilimbi 2010]. The basic idea is to introduce simpler and less accurate versions of the less-significant parts of the application to trade off output quality with faster and thus more energy-efficient execution. The notion of significance is semantically defined in Vassiliadis et al. [2016]. In this work, less-significant parts of the application are executed on lower-power yet unreliable hardware. The crucial difference to approximate computing is that such unreliable execution can lead to arbitrary errors, which are not easily controllable and may cause large disruptions to the output or even crash the application program. This necessitates mechanisms to isolate/protect unreliable computations from reliable ones, in conjunction with methods for detecting and correcting severe and silent errors.

Based on these observations, we introduce a framework for expressing and exploiting application-level knowledge about the significance of different parts of a program. Developers use a task-based programming model to declare the significance of computations, depending on how strongly they contribute to the quality of the result. A suitable runtime system executes less-significant tasks on unreliable but lower-power cores. To contain errors and silent data corruptions that may occur due to the unreliable execution of non-significant tasks, the developer can provide result-check functions that are called by the runtime when task execution completes (or fails) to inspect the task status and produced output and, if needed, take corrective action.

There has already been extensive work on error-tolerant and approximate computing. Our work proposes a new framework that exploits algorithmic significance in conjunction with suitable system software mechanisms to reduce the energy footprint of applications by purposefully driving hardware operation beyond the pessimistic guard bands and dealing with potential errors in a controlled way.

The main contributions of this article are the following: (i) We introduce a significance-centric programming model and runtime system, which allows computations to be executed on potentially unreliable but low-power hardware in a controlled way to trade off output quality for greater energy efficiency. (ii) We introduce a single knob, called *ratio*, to trade off quality for energy gains in a flexible way. The *ratio* can be set at execution time, thereby allowing the user or a higher-level framework to control the energy/quality tradeoff to meet dynamically varying application requirements. (iii) We evaluate our framework using five benchmarks with different computational characteristics and significance patterns. (iv) We study the effectiveness of the ratio knob and the different error detection and protection/repair mechanisms provided by our framework and the respective energy gains that can be achieved as a function of output quality degradation.

The rest of the article is structured as follows. Section 2 introduces the programming model, and Section 3 discusses the supporting runtime system. Section 4 presents the application benchmarks used to evaluate our framework, while Section 5 provides an overview of the evaluation approach. Section 6 describes and validates the performance and energy models. Section 7 describes the fault model and fault injection approach used in the evaluation. Section 8 presents the evaluation experiments and discusses the results. Section 9 provides an overview of related work. Finally, Section 10 concludes our article and presents directions for future work.

2. SIGNIFICANCE-CENTRIC, FAULT-TOLERANT, TASK-BASED PROGRAMMING MODEL

Our programming model can develop applications that target unreliable hardware, without uncontrolled degradation of the quality. The features are the following:

Significance characterization: The developer can specify the significance of the computations based on how strongly they contribute to the quality or correctness of the result. Significant computations need to be executed correctly on reliable hardware, while non-significant ones may be executed on potentially unreliable hardware.

Error isolation and control: During unreliable execution, errors that manifest on non-significant tasks should not propagate to the rest of the computation in an uncontrolled way. Moreover, the developer can provide application-specific code for checking and repairing the output of the parts that are executed unreliably.

Exploitation of unreliable cores: It is possible to exploit cores configured to operate beyond their nominal guard bands to improve the energy efficiency of the non-significant parts of the computation at the cost of potential unreliability.

We adopt a task-based paradigm in the spirit of the latest version of the OpenMP standard, where task boundaries and task synchronization are expressed using using *#pragma* compiler directives. We extend these directives so the programmer can specify task significance, task result-checking, and relaxed task synchronization. Apart from expressing parallelism and data dependencies, tasks are a natural unit for structuring code for containing, checking and repairing errors explicitly. The *#pragma* directives also facilitate non-invasive and progressive code transformations, without requiring a complete code rewrite, which is important when targeting existing code.

2.1. Task Definition and Significance Characterization

Tasks are defined using the *task* directive (Listing 1). The *significance*() clause specifies the significance of the task and takes values in the range $[0.0, 1.0]$, indicating the importance of the task with respect to the output quality/correctness of the result. Depending on their significance, tasks may be executed on top of reliable or unreliable hardware. The significance expression is evaluated at execution time, thus allowing the programmer to parameterize task significance with user input.

The *taskcheck*() clause specifies a result-check function, which is invoked only if the task is executed unreliably. The result-check function is always executed reliably and can be used by the developer to (i) inspect the task status to see if it completed its execution normally or has crashed, (ii) assess whether the task output is wrong, (iii) assign meaningful default values to the task output, and (iv) request a re-execution of the task. The result-check function has implicitly access to all arguments of the corresponding task and may return either *TRC_SUCCESS* or *TRC_REDO* to the runtime. In the latter case, the task is re-executed reliably.

Finally, the programmer can define the inputs and outputs of the task via the *in*() and *out*() clauses, respectively. This information can be used by the runtime to infer task dependencies and schedule tasks accordingly.

```
1 #pragma omp task  [significance(...)] [taskcheck(resultcheck())] [in(...)] [out(...)]
```

Listing 1. #pragma omp task.

2.2. Synchronization

Explicit barrierlike synchronization is achieved via the *taskwait* directive (Listing 2). This is used to wait until all tasks that have been created so far are run to completion.

The *taskwait* can be used to control the degree of reliable (and unreliable) task execution. Specifically, via the *ratio*() clause, the developer specifies the minimum

```
#pragma omp taskwait [on(...)] [ratio(...), time(...)] [groupcheck(resultcheck())]
```

Listing 2. #pragma omp taskwait.

percentage of tasks that should be executed in a reliable way, while *respecting* task significance—that is, a less significant task will not be executed reliably at the expense of a more significant task being executed unreliably. The task ratio takes values in the range [0.0, 1.0] and serves as a knob to control the quality of the result. Small ratios give the runtime energy reduction opportunities but with a loss in the quality.

Given that some of the non-significant tasks may be executed unreliably, *taskwait* also allows for a more *relaxed* synchronization. Namely, the programmer can use the *time*() clause to define a timeout after which execution will continue, provided that the most significant tasks (as per the *ratio*() setting) have completed. If some non-significant tasks have not completed their execution yet, then they are stopped, and the respective result-check functions are invoked (requests to re-execute a task are ignored). Note that such timeouts are task dependent as is the case in most soft real-time applications. The programmer may introduce a result-check function for all tasks that have been created so far via the *groupcheck*() clause. This function is called when the conditions of *taskwait* are fulfilled to perform checks and repairs on the aggregate output produced by the tasks.

2.3. Example

Listing 3 presents a task-based implementation of Discrete Cosine Transform (DCT) using our programming model. Line 15 defines a task to compute the frequency coefficients of a specific 2×4 sub-block. All tasks created in this loop have varying significance depending on their position in the 8×8 block: Upper left sub-blocks have higher significance than lower right, as encoded in the *sgnf* array. In line 15, *dct_taskrescheck*() is specified as the result-check function. This function checks whether the task crashed (Line 2) or whether its output is wrong (Line 4). In both cases, a the corrections sets the respective coefficients to 0. Since this correction does not require task re-execution the function returns *TRC_SUCCESS* (Line 6).

```
1  int dct_taskrescheck(...) {    /* DCT task result-check function.       */
2    if ( task_crashed() )          /* Takes the same arguments as the task, */
3      coeff = 0;                   /*  returns int.                         */
4    else if ( abnormal(coeff) )
5      coeff = 0;
6    return TRC_SUCCESS;
7  }
8  void dct_task(...) {     /* Calculate the coefficients for a specific 2x4 block */
9    ...                    /* over a number of different 8x8 blocks.              */
10 }
11 void DCT(...,double taskratio){/*DCT calculation. The taskratio is an extra parameter.*/
12   float sgnf[] = {1.0,  0.9,  0.7,  0.3,    /* Significance look up table    */
13                   0.8,  0.4,  0.3,  0.1};   /*for each of the 2x4 sub-blocks */
14   for each 2x4 sub-block K {/* Iterate over the blocks of the DCT coefficient space. */
15     #pragma omp task  significance(sgnf[K]) taskcheck(dct_taskrescheck())
16         dct_task(...);/* Task to calculate the Kth 2x4 sub-block, over all 8x8 blocks */
17   }
18   #pragma taskwait ratio(taskratio) time(16)
19 }
```

Listing 3. Programming model use case: DCT pseudo-code.

In Line 18 of Listing 3, the barrier for all *dct* tasks is specified with a timeout of 16 ms; this corresponds to a target frame rate of 30fps, assuming DCT corresponds to almost 50% of the computation time for each frame. Note that the *taskratio* is an open parameter that is supplied when the program is invoked. In effect, it serves as

a knob to set the "borderline" between the most-significant sub-blocks that have to be computed reliably and the less-significant sub-blocks that may be computed unreliably. No group-level result-check function is used in the example, because task-level result checks and repairs are sufficient.

2.4. Programmer Insight

The programming model assumes that the developer is sufficiently familiar with the application to take good decisions as to how to structure the computation in tasks, which tasks to characterize as more significant, and which result-check functions to provide. Similarly to parallelism, significance is a key algorithmic aspect that requires the programmer's full attention, but, unlike parallelism, task significance is orthogonal to the underlying platform architecture.

A formal definition of significance can be provided as follows. Assuming that a task implements the function $y = f(\mathbf{x})$, where $\mathbf{x}$ is the vector of task inputs, the significance of $\mathbf{x}$ to the output y can be defined using interval arithmetic [Moore et al. 2009] and first-order adjoint analysis. The range of possible input values is the input interval vector $[\mathbf{x}] = [\underline{\mathbf{x}}, \overline{\mathbf{x}}] = \{\mathbf{x} \in I\!R^n | \underline{\mathbf{x}} \leq \mathbf{x} \leq \overline{\mathbf{x}}\}$, and an evaluation of f in interval arithmetic is obtained by replacing all variables and intermediate elementary functions ϕ with their interval version. The significance of an input element $x_i \in \mathbf{x}$ to the final result y is equal to

$$S_y(x_i) = w([x_i] \cdot \nabla_{[x_i]}[y]),$$

where $w(\cdot)$ is the width of the interval. The first-order derivative $\nabla_{[x_i]}[y] = \frac{\partial f[\mathbf{x}_i]}{\partial [x_i]}$ is the derivative of the function result $[y]$ with respect to the input variable $[x_i]$. In other words, the bounds of interval derivative $\nabla_{[x_i]}[y]$ are the steepest downward and upward slopes, respectively, of $y = f(\mathbf{x})$ in the interval $[x_i]$, which quantify the impact of all possible values from $[x_i]$ on the final result y. If the range (width) of S_y is large, then x_i strongly affects the value of y. As such, the code that produces the value of x_i is highly significant for the accuracy of the final output y. More information on the algorithmic property of significance and a methodology for determining the significance of computations automatically can be found in Vassiliadis et al. [2016].

Choosing result-check functions is also important. If the result-check function is too complex, then it is practically useless, as the same result could be achieved simply by declaring the task as significant and executing it reliably in the first place. If too simple, then the result-check function may erroneously mis-characterize and destroy good task output, possibly deteriorating the end result of the computation.

Finally, task granularity is an important parameter that should be considered when using this programming model. Fine-grained tasks may allow for a richer (more diverse) significance characterization, which in turn can be exploited to achieve a smoother degradation of output quality at increased energy gains. The downside is that having many small tasks will also increase the task management overhead of the runtime system in terms of both time and energy consumption.

2.5. Application Characteristics

Several application domains offer the opportunity to trade off quality of output for significant improvements in energy consumption. **Visualization applications** are amenable to approximations, because their output is typically consumed by humans. The correction part of the result-check function can exploit the perceptual limitations of the human eye to approximate computations without inflicting noticeable quality degradation to their output. In our evaluation, we use two benchmarks from this category: *DCT* and *Sobel*. **Streaming** applications are inherently amenable to approximations, since they do not maintain a large state. They consume input data, perform

computations, and produce output data. If an error occurs during the computation of a specific output data batch, then the next batch will not be severely affected. In that sense, streaming applications inherently exhibit computational isolation. *Blackscholes*, one of the benchmarks used in the experimental evaluation, falls into this category. Some **iterative methods** tend to be self healing. For example, in the presence of errors, Monte Carlo simulations or iterative numerical methods still tend to converge to a correct solution but will typically require more iterations. Such applications in our evaluation are *Jacobi* and *k-means*. We wish to clarify that the proposed significance-based computing model does not fit all applications. For instance, task significance may be highly input dependent, hard to specify at design time, and difficult or costly to extract even at runtime. Also, some programs may require all tasks to be executed without any inaccuracy or any chance of data corruption.

3. SIGNIFICANCE-AWARE RUNTIME SYSTEM

The runtime system is designed for a multicore shared memory platform, in which cores can be set to operate in various voltage-frequency configurations (V, f), even in ones below nominal values. Unsafe settings only apply to the cores of the Central Processing Unit (CPU), including the integer and Floating Point Unit (FPU) pipeline logic as well as the L1 and L2 caches. Modules critical to the correct operation of all cores, such as buses, memory controllers, and cache coherence mechanisms, are set to a safe setting and thus always operate reliably. Our power model takes this into account, and all reported energy gains are gained from undervolting the core part. A user-level library implements the runtime system and runs on top of the Linux operating system. A source-to-source compiler, which we developed based on Zakkak et al. [2012], lowers programs that use the primitives of our programming model to code with calls to the runtime system API. Finally, the produced source code is compiled into machine code using the standard *gcc* tool chain.

3.1. Runtime Execution Management

We consider three different configurations, *FastRel*, *SlowRel*, and *FastUnRel*. The *FastRel* configuration is a high-performance nominal point of operation, with high voltage/frequency (V_h, f_h), where a core executes code fast, whereas *SlowRel* is a slower nominal operation point, with lower voltage/frequency (V_l, f_l). Furthermore, cores can be set in the non-nominal and unsafe *FastUnRel* configuration (V_l, f_h), with the same (low) voltage as *SlowRel* and the same (high) frequency as *FastRel*. Code execution in *FastUnRel* is equally fast as in *FastRel* yet more energy efficient. At the same time, execution is potentially unreliable due to timing faults, since *FastUnRel* is outside the nominal range of operation. We assume that the runtime system can switch the operation of cores dynamically. Due to the difference in their voltage, the transition between *FastRel* and *SlowRel* requires a *voltage and frequency scaling* step, which introduces significant delay. In contrast, given that *SlowRel* and *FastUnRel* have the same voltage, the transition between them can be done quickly via clock stretching [Constantin et al. 2015]. Figure 1 illustrates the principle of operation.

The main application thread and the master runtime thread are executed reliably in the *FastRel* configuration. The tasks of the application can be executed reliably in the *FastRel* configuration, or unreliably in the *FastUnRel* configuration, depending on their relative significance and the user-supplied task ratio (see Section 2). Task execution is done using separate worker threads, with each worker being placed in a different core. To reduce the number of voltage transitions, task scheduling is done in two alternating phases. In the first phase, workers are configured to operate in *FastRel*, and the master thread schedules all the tasks in the ready list that have been flagged for reliable execution. Before the second phase starts, all workers soft-checkpoint crucial context

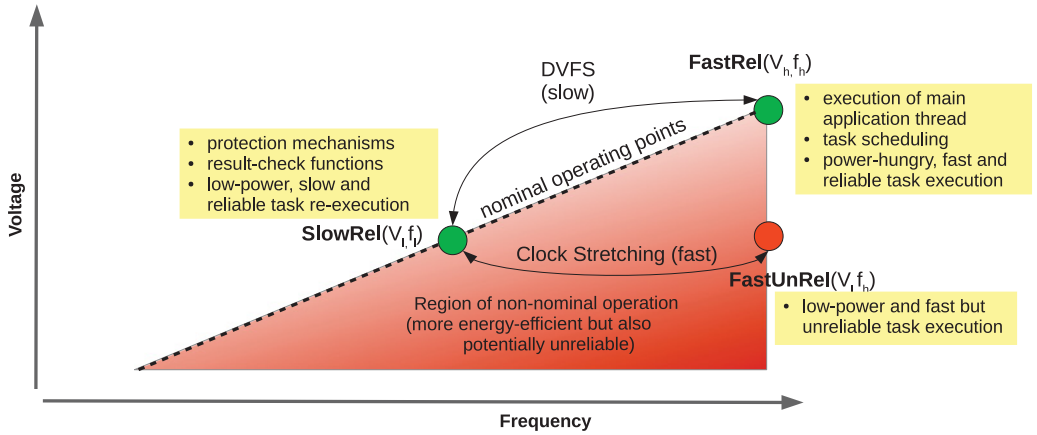

Fig. 1. The configurations *FastRel*, *SlowRel*, and *FastUnRel* used by the runtime system to reduce the energy footprint by exploiting the significance of computations. Our approach exploits non-nominal configurations, that are energy efficient but unreliable.

information to use it to recover in case of corruption from faults.[1] Afterwards, the main thread requests the memory allocator (discussed in Section 3.2) to protect all the memory pages as well as the stack of the main application thread. This actually forces all data, including non-significant output data, to a read-only state. Reliable task input/output data can be mixed with unreliable input/output data in the same memory page. However, the Operating System (OS) assigns privileges at the granularity of a page; therefore, when locking a page to a read-only state, even unreliable tasks cannot write to their output data locations. To overcome this, each worker allocates extra memory in which the non-significant tasks will store their results. These memory locations have read-write permissions.

At this point the second phase starts. Workers switch from *FastRel* to *SlowRel*, and the master thread proceeds with the scheduling of all the tasks that have been flagged for unreliable execution. When a worker is assigned with a task, it switches to the *FastUnRel* configuration and executes the task. If during task execution an event causes the OS to take over (e.g., an I/O event), then the worker switches to *SlowRel* prior to executing the kernel code and switches back to *FastUnRel* mode when it resumes the execution of the application task. When the task completes or crashes, the core is switched back to *SlowRel*, the previously saved state is restored, and the result-check function of the task is invoked.

If the result-check function requests task re-execution, then the worker repeats the execution but maintains the core in the reliable *SlowRel* configuration. When all tasks have finished their execution or the synchronization timing constraint is reached, the main thread requests from the allocator to revert the protected memory privileges to their previous state. Afterwards, the main thread copies the computed output data from unreliable tasks back to their original memory locations. In case the group result-check function requests re-execution of the task group, the master thread configures all workers to operate in *FastRel*. Then all tasks in the group that have been flagged for unreliable execution are re-scheduled from scratch and are executed reliably. The overhead of switching to a different voltage level is amortized by the execution of a large number of tasks.

[1]We use the Linux *getcontext()* function. The state is copied to a read-only memory page to prevent it from being written accidentally.

The runtime supports the following levels of protection:

No Protection (NP): The runtime system does not employ any error detection/correction mechanism or programmer-supplied significance information. All tasks of the application are executed unreliably (*FastUnRel* configuration) and are susceptible to faults. A task crash leads to the abrupt termination of the entire application.

Basic Protection (BP): All applications tasks are executed unreliably as in *NP*, but the runtime system identifies and handles errors using the standard processor/OS protection mechanisms, including the internal soft-checkpointing of critical state and the memory protection mechanism. As a result, task crashes are properly caught. However, the programmer-supplied result-check functions are ignored.

Basic & Result Checking (B-RC): In addition to *BP*, when an application task completes its execution normally or crashes, the runtime system invokes the programmer-supplied result-check function to detect and correct possible errors.

Basic & Significance (B-SF): On top of *BP*, the runtime system takes into account the programmer-supplied significance of tasks and ratio and schedules them for execution accordingly. As a consequence, the most significant tasks are executed reliably (in the *FastRel* configuration), while the less-significant tasks are executed unreliably (in the *FastUnRel* configuration). Task crashes are caught and handled as in *BP*, and the programmer-supplied result-check functions are ignored.

Full System (FS): The entire protection arsenal is employed, including basic runtime system protection, task scheduling based on the programmer-supplied significance information, and invocation of the result-check functions for unreliable tasks.

Full System & Re-Execution (FS-RE): Like *FS*, but if the task result-check functions detect a task crash or invalid output, then they request a full task re-execution rather than try to repair the task output.

3.2. Memory Management

Our framework has been developed on a shared memory system, which is the worst-case scenario in terms of reliability. Erroneous stores by code that executes on an unreliable core may affect global data structures or the memory of significant computations.

The runtime system utilizes a custom dynamic memory manager that requests memory slabs from the OS at the granularity of pages and serves dynamic allocations from either the application or the runtime system. When switching to an unreliable execution phase, the memory manager assigns read-only privileges to all used heap pages to protect them from rogue stores from tasks executed on non-reliable cores. Should such a store be attempted, it leads to an exception that is handled accordingly by the runtime system. Besides the heap, faults can also corrupt the stack. The runtime system allocates its internal data structures dynamically, and thus there is no information within the stack or the global data space to be protected from application faults. All memory pages used as stack by the main application thread are also set to be read-only prior to executing unreliable code. The current framework does not implement any global data protection as this requires compiler support.

4. BENCHMARKS

We calibrated, validated, and evaluated our models as well as the significance centric framework on top of a Intel Quad Core i7 IvyBridge CPU platform. We use five benchmarks listed in Table I.

We apply three different methodologies to perform significance characterization on these benchmarks. In *DCT*, we use **domain expertise** to identify the significance of different parts of the computation. The tasks that compute low-frequency coefficients are close to the upper left corner of each 8×8 frequency block and are more significant than the ones computing coefficients towards the lower-right corner of the block. In

Table I. Lines of Code (LOC) for the Tasks and Corresponding Result-Check and Correction Functions for Each Benchmark. The Result-Check Functions Are Implemented Based on the Original Task Code, Which Was Modified to Reduce Its Computational Complexity

			Lines of Code	
Benchmark	Domain	Sgnf. Characterization	Task	TRC function
DCT	Multimedia	Domain expertise	39	34
Sobel	Image Filter	Randomly	54	42
Blackscholes	Finance	Profile-driven	117	105
K-means	Data mining	Profile-driven	141	57
Jacobi	Numerical Solver	Profile-driven	62	39

Blackscholes and the iterative benchmarks *k-means, Jacobi* we employed a **profile-driven** approach. More specifically, in *Blackscholes* we injected bitflips in the input data and observed the output quality. All parts of the code appear to be equally significant, since faults had similar manifestations regardless or task computations. Therefore, all tasks are assigned equal values of significance, since all stock options are considered equally important. In *Jacobi* and *k-means*, we injected bit-flips in the input data of a randomly chosen iteration and compared the relative error of the faulty execution with an error-free one. In both *Jacobi and k-means*, we observe that errors in the last few iterations tend to severely reduce the output quality and thus infer that these are the most significant ones. Finally, in *Sobel*, we exploit the perceptual properties of the human eye and **randomly** distribute the significance among tasks. This way errors are spread across the entire output image and the loss of quality is not clustered in a specific area of the image.

In all benchmarks, we used a very simple result-check function. The result-check function of *DCT* detects errors in the task output via a heuristic out-of-bounds check; coefficients that do not respect the bounds are set to zero. In *Sobel*, the task result-check function corrects only tasks that crashed during their execution by running an approximate version of the Sobel filter, using a lightweight stencil with just 2/3 of the filter taps. *Blackscholes* is a benchmark of the Parsec suite [Bienia et al. 2008]. Results are checked with the *isfinite()* macro. This is a *glibc* floating point classification macro; it returns a non-zero value if the value under inspection is not *NaN* or *infinite*. If the check fails, then the function uses a faster implementation of the *Blackscholes* formula by substituting costly mathematical operations (such as $expr(), sqrt(), log()$) with approximate versions. In *k-means*, the result-check function of non-significant tasks is minimalistic, exploiting the error-tolerant nature of this iterative application: If a point attempts to subscribe itself to cluster but miscalculates the cluster's id, then it reverts to its previous cluster. Also, if the runtime system reports an error, then all points computed by the task are subscribed back to their previous clusters. In *Jacobi*, it is hard to create an error detection mechanism, since assessment of the quality of results is associated with the application in which the solver is used. We implement a simple result-check function that uses the *glibc isfinite()* macro to detect obvious errors to the output of tasks. In the event of detecting such an error, the current solution estimate is replaced with that of the previous iteration.

In our benchmarks, the result-check part was simple, mostly based on range checks. For the correction part, we reused the original task code and modified it to perform the computation approximately. Table I shows that result-check functions are almost as big as the tasks themselves. Nevertheless, since we heavily reused the existing task code, the actual effort to implement the result-check function was minimal.

5. EVALUATION METHODOLOGY

Commercially available platforms do not allow individual cores to be operated below nominal settings and hence cannot be used to support the runtime model that was

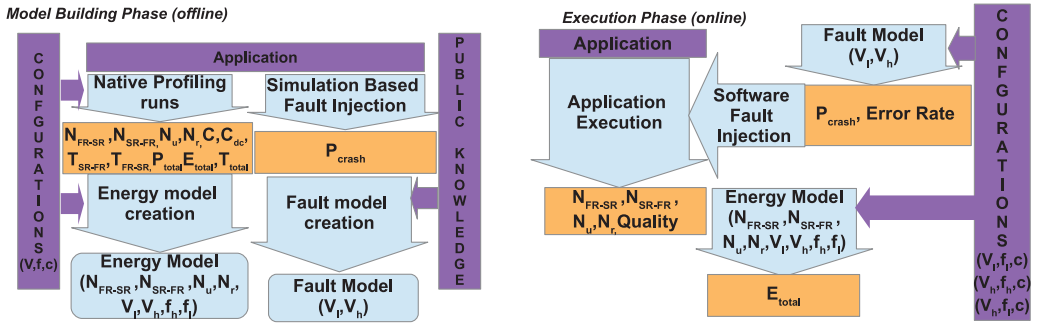

Fig. 2. Evaluation approach: we build the performance, energy and fault models (left), and use these models to drive experiments and estimate energy consumption (right).

described in Section 3. We note that operation in sub-nominal (V, f) values is possible in x86 CPUs using BIOS settings but only for the entire CPU. This is not useful for the purposes of our work, because at least one core has to always work reliably to boot the machine and safely run the OS and the runtime system. As a consequence, we cannot take real measurements on the performance, energy consumption, and fault rate/behavior of the system.

To evaluate our framework, we use a suitable model for estimating the execution time and energy consumption of a computation as a function of the voltage-frequency settings of the *FastRel*, *SlowRel*, and *FastUnRel* configurations and the tasks that are executed in these configurations. The model also takes into account the task management overheads of the runtime system, as well as the cost for performing the voltage and frequency scaling steps needed for a switch between *FastRel* and *SlowRel*/*FastUnRel*. This allows us to run computations on a real platform, trace its execution, profile the performance and power parameters used as input to the model for the *FastRel* and *SlowRel* configurations, and extrapolate estimates for the *FastUnRel* configuration. The performance model is described in detail in Section 6.

However, this performance modeling is insufficient. When executing in the *FastUnRel* configuration, the hardware may experience timing errors, which in turn trigger the respective runtime protection mechanisms. A major challenge is to associate the operation in the *FastUnRel* configuration with the probability of hardware faults due to timing violations. Another issue is how to assess the impact of such faults to the actual execution and outputs of a given task, which is entirely application specific. We use a model that estimates the fault rate as a function of the voltage-frequency setting of *FastUnRel*. We use a combination of simulation-based and software-based fault injection to observe the impact of faults on the benchmarks.

Figure 2 illustrates the workflow of our evaluation approach, which is split into two phases. During the model building (offline) phase, we run the benchmarks in the GemFI simulator (see Section 7.2) to obtain the probability of crash (P_{crash}). We also perform native executions and measure the number of reliable and unreliable tasks (N_r, N_u), the number of transitions from *FastRel* to *SlowRel* and vice versa (N_{FR-SR}, N_{SR-FR}), the average time required to perform a voltage and frequency transition from *FastRel* to *SlowRel* and vice versa (T_{FR-SR}, T_{SR-FR}), the length (in cycles) of a task (C), the length of its result-check function (C_{dc}), the total energy and power consumption (E_{total}, P_{total}) of the program, as well as its total execution time (T_{total}). We create the energy model using the above information, as well as the operating frequency (f), the supply voltage (V), and the number of cores (c). The energy model input parameters are the number

of reliable/unreliable tasks (N_r, N_u), the number of active cores (c), the number of voltage and frequency transitions (N_{FR-SR}, N_{SR-FR}), and the supply voltage and the operating frequency of all configurations. Its goal is to estimate the energy cost of an unreliable execution. We model application error resiliency via simulation-based fault injection combined with observations in the literature. The fault model input arguments are the supply voltage of the *FastRel* and the *FastUnRel* configuration. The energy and fault models are discussed in more detail in the following sections. During the execution phase, the models are used to inject errors and to estimate the energy consumption of the execution. After each execution, the system reports the values for the number of tasks that were executed reliably (N_r) and unreliably (N_u) and the number of transitions between the *FastRel* and *SlowRel/FastUnRel* (N_{FR-SR} and N_{SR-FR}) configurations. Using these data, we estimate the energy footprint of the computation.

6. EXECUTION TIME AND ENERGY CONSUMPTION MODEL

We introduce an analytical model for the performance and energy consumption of a program as a function of the core frequency, the voltage, the number of tasks that are executed reliably and unreliably, and the number of voltage and frequency transitions. Our model is agnostic to the CPU structure and captures the execution phases of an application. Therefore, it accounts for both the *core* and *uncore* components of the CPU. The model is validated for our CPU platform, where it predicts the actual energy consumption of our benchmark applications with high accuracy over a wide range of different (nominal and thus reliable) core configurations.

6.1. Execution Time Modeling

As discussed, the runtime uses three different voltage/frequency configurations, $FastRel = (V_h, f_h)$, $SlowRel = (V_l, f_l)$, and $FastUnRel = (V_l, f_h)$. Equation (1) expresses the time for executing a given piece of code N times, where C denotes the number of cycles spent for code execution and f is the frequency of the core depending on its configuration setting (f_h for *FastRel/FastUnRel* and f_l for *SlowRel*),

$$T(N, f, C) = \frac{C}{f} \times N. \tag{1}$$

Tasks can be executed in parallel by the workers of the runtime system on different cores. In addition to task execution itself, the system software spends additional time to schedule tasks and to manage unreliable task execution. Equations (2) give the total execution time of an application,

$$\begin{aligned} &T_{FastRel} = T(N_r, C, f_h), T_{SlowRel} = T(N_u, C_{dc}, f_l), T_{FastUnRel} = T(N_u, C, f_h) \\ &T_{vfs} = N_{FR\to SR} \times T_{FR\to SR} + N_{SR\to FR} \times T_{SR\to FR} \\ &T_{Total} = \max_{i=1}^{Workers} \left(T_{FastRel_i}\right) + \max_{i=1}^{Workers} \left(T_{SlowRel_i} + T_{FastUnRel_i}\right) + T_{vfs}. \end{aligned} \tag{2}$$

The execution time for each worker in each configuration is expressed by $T_{FastRel}$, $T_{SlowRel}$, and $T_{FastUnRel}$. Variable C is the average number of cycles required to execute a task in *FastRel/FastUnRel*, while C_{dc} is the average number of cycles required by the runtime system to prepare for an unreliable task execution and to execute the result-check/repair function in the *SlowRel* configuration. Variables N_r and N_u express the number of reliable and unreliable tasks executed by the worker, respectively. T_{vfs} captures the time required to switch between the *FastRel* and *SlowRel* configurations. Variable $N_{FR\to SR}$ denotes the number of times the runtime system switches from the *FastRel* to *SlowRel*, and $T_{FR\to SR}$ is the average time required to perform this transition.

Similar parameters apply for the reverse direction. Finally, the total execution time of the application is the maximum execution time among all workers for the first task scheduling phase (in the *FastRel* configuration), plus the maximum execution time among all workers for the second task scheduling phase (in the *SlowRel/FastUnRel* configurations), plus the time spent on the respective voltage and frequency transitions.

6.2. Power and Energy Modeling

The total power dissipation of a CMOS circuit is given by Equations (3). P_{dyn} is the dynamic power dissipation, P_{leak} is the power dissipation due to transistor leakage current, and P_{shortC} is the power dissipation due to the short circuit formed when both the PMOS and NMOS transistor tree momentarily conduct current during CMOS switching. Since modern fabrication technologies that use high-k dielectric materials can control leakage current, it is the P_{dyn} component that dominates power dissipation. Therefore, our model considers the idle power consumption of a processor as a constant P_{idle} and equal the sum of P_{leak} and P_{shortC}. The uncore power consumption of the CPU is included in P_{idle}. Since the P_{idle} is a constant, all the energy gains are a result of the undervolted core part of the *CPU*. P_{dyn} is the product of the supplied voltage squared (V^2), the frequency (f), and the activity factor $A(\vec{c})$. We have observed that the activation of a new core in our multicore platform results to power steps. The number of cores used by the application are captured via vector $\vec{c}$, where $\vec{c}[n]$ is 1 if n cores are active , else 0. $A(\vec{c})$ is the dot product of $\vec{c}$ and a vector $\vec{w}$ containing per-core switching capacitance values that are obtained via regression.

$$\begin{aligned} &P_{Total} = P_{idle} + P_{dyn},\ P_{idle} = P_{leak} + P_{shortC} \\ &P_{dyn}(\vec{c}, V, f) = A(\vec{c}) \times V^2 \times f,\ A(\vec{c}) = \vec{c} \cdot \vec{w}. \end{aligned} \tag{3}$$

The total energy dissipation E_{Total} is given by Equations (4). In general, this depends on the hardware/core configuration and the time spent to execute the runtime management functions, the application tasks, and their result-check/repair functions, as discussed above,

$$\begin{aligned} &E_{FastRel} = P(\vec{c}, V_h, f_h) \times \max_{i=1}^{Workers} \left(T_{FastRel}\right), E_{FastUnRel} = P(\vec{c}, V_l, f_h) \times \max_{i=1}^{Workers} \left(T_{FastUnRel}\right) \\ &E_{SlowRel} = P(\vec{c}, V_l, f_l) \times \max_{i=1}^{Workers} \left(T_{SlowRel}\right) \\ &E_{vfs} = N_{FR\rightarrow SR} \times T_{FR\rightarrow SR} \times P(V_h, f_h) + N_{SR\rightarrow FR} \times T_{SR\rightarrow FR} \times P(V_l, f_l) \\ &E_{Total} = E_{FastRel} + E_{SlowRel} + E_{FastUnRel} + E_{vfs}. \end{aligned} \tag{4}$$

6.3. Calibration and Validation

We calibrate and validate the timing and energy models based on measurements taken on our platform for the benchmarks presented in Section 5. The parameters f_h and f_l are known while N_r, N_u, $N_{FR\rightarrow SR}$, $N_{SR\rightarrow FR}$ can be measured. C and C_{dc} are profiled using *likwid* [Treibig et al. 2010] by accessing the x86 performance counters. Similarly, $T_{FR\rightarrow SR}$ and $T_{SR\rightarrow FR}$ are profiled using the *FTaLaT* tool [Mazouz et al. 2014]. Finally, the transition overhead between the *SlowRel* and *FastUnRel* configurations is negligible, since clock adjustment is very fast.

As a first step, we execute all tasks of each application reliably under different configurations $V, f, \vec{c}$, and measure the power consumption. We then perform linear regression using least-squares to derive the parameters $\vec{w}$ and P_{idle} of the power model. Finally, we validate the accuracy of our model by forcing the runtime system to execute tasks in different (V, f) configurations. To this end, we execute half of the tasks of

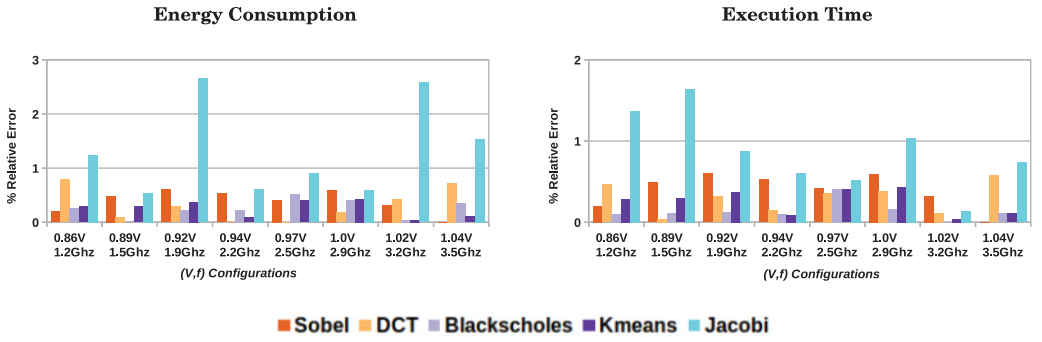

Fig. 3. Relative error for the execution time and energy as predicted by our model vs. a real execution for our application benchmarks when half of the tasks execute in the $FastRel = (3.7(\text{GHz}), 1.06\text{V})$ configuration and the other half in a lower-power *SlowRel* configuration. All *SlowRel* configurations are shown in the x-axis.

each application in the $FastRel = (1.06V, 3.7GHz)$ configuration and the other half in various lower power but still reliable configurations. The latter are different candidates for *SlowRel*. Cores enter these configurations, which correspond to different P-states,[2] by applying a software-driven voltage and frequency transition.

Figure 3 depicts the relative error of model-based estimates vs. the execution time and energy that was measured using *likwid*. Our model closely matches the real numbers for various *SlowRel* configurations, with the relative error ranging from 0.004% to 2.7%. In *Jacobi*, the increased error is due to load-balancing issues. Different executions of the benchmark result in different tasks to worker assignment. This impacts the execution time of the benchmark, and hence there is an increase in the relative error.

Note that our x86 platform does not allow placing individual cores in a non-nominal configuration, where actual timing violations and faults might occur. Thus it is impossible to validate the execution time and energy consumption estimates of the model for non-nominal *FastUnRel* configurations. Still, the accuracy of the model for this wide range of real operating points gives us sufficient confidence to use the model to extrapolate for non-nominal *FastUnRel* configurations as well.

7. FAULT MODEL AND FAULT INJECTION METHODOLOGY

This section introduces the fault model we use for different unreliable execution. We discuss how we combine simulation-based and software-based fault injection to map the fault rates derived from the model into actual errors at the application level.

Note that it was impossible to conduct our full evaluation using a purely simulated execution platform. Given the vast number of fault injection experiments required to acquire statistically significant results, we would have to limit executions to non-realistic input sizes, despite using a large compute farm for the simulations. Therefore, we adopt a hybrid approach. Initially, we use detailed simulations for injecting faults at the architectural level of an x86 CPU model and observe the impact they have on each of our benchmarks. Afterwards, we use these observations to drive fault-injection via software when running the benchmarks and our runtime system natively on our platform. The latter setup, in conjunction with the performance model discussed in Section 6, makes it possible to evaluate the fault-tolerance mechanisms that are provided by our framework for different *SlowRel*/*FastUnRel* configurations.

[2]P-states are voltage-frequency pairs that specify the performance and power consumption of a processor.

7.1. Fault Modeling

A key challenge is to associate the operation of a core in the unreliable *FastUnRel* configuration with the probability of hardware faults due to timing violations. Besides undervolting (or overclocking), the number and distribution of faults in a CPU also depends on the type of instructions executed: Instructions that activate long paths, which are close to the critical path, tend to fail more frequently [Rahimi et al. 2012]. The failure probability of each instruction is also closely related to the micro-architectural design of the CPU; optimizations used by synthesis, placement, and routing CAD tools; the manufacturing process and process variability; ambient temperature; IR drops; aging; and so on. Even identical chips with the same micro-architecture, using the same technology libraries, and running identical code, can have highly different behaviors [Das et al. 2006]. Moreover, whether a fault manifests as an error not only depends on the paths that are activated during the current cycles but also on the paths that were activated in previous cycles [Tziantzioulis et al. 2015].

It is almost impossible to model such complex phenomena in a practical way, as the conclusions are specific to the particular system used to make the observations and create the model and cannot be generalized to other systems. To the best of our knowledge there is no modern-CPU fault model that combines all the observations in a unified and applicable method. For these reasons, we abstract out the instruction mix of applications by taking into account only the effects of voltage scaling.

The Point of First Failure (PoFF) is used to indicate the point at which circuits start to exhibit massive errors (one error every ∼10 million cycles). Prior to this point, errors still occur, however, at rates that are orders of magnitude lower [Das et al. 2006]. If one goes beyond the PoFF, then the fault rate increases exponentially by one order of magnitude for every 10mV drop of the supply voltage [Blaauw et al. 2008; Das et al. 2006].

To guarantee functional correctness, designers typically account for parameter variations by imposing conservative margins that guard against worst-case scenarios. The extent of the voltage margins required for fault-free operation for all operating conditions of the chip is on average around 15% [Gupta et al. 2009; Reddi et al. 2010; James et al. 2007]. We determine the *PoFF* based on Equation (5), where ρ is the percentage of the extra voltage margin to guarantee fault-free operation and V_n is the nominal supply voltage. A CPU part with tight margins has a low ρ and, therefore, low energy benefits when using our approach. We select the average case, $\rho = 15\%$, which is consistent with several observations in the literature [Gupta et al. 2009; Blaauw et al. 2008; Das et al. 2006]. Based on the same reports, we model the fault rate as shown in Equation (6). The parameters are the voltage V_{PoFF}, which can be obtained using Equation (5) using as an input argument the nominal voltage V_n and the voltage of the requested (unreliable) operating point V_u. In our case, $V_n = V_h$, the voltage setting of *FastRel*, and $V_u = V_l$ is the voltage setting of the *FastUnRel* configuration. Our model obtains the constants β, γ via regression on the data provided in Blaauw et al. [2008] and Das et al. [2006]. Note that Equation (6) is CPU agnostic,

$$V_{PoFF} = \frac{(100 - \rho)}{100} \times V_n \tag{5}$$

$$Err(V_{PoFF}, V_u) = \beta \times 10^{\gamma * (V_u - V_{PoFF})} \tag{6}$$

$$V_n(f) = \delta \times f + \eta. \tag{7}$$

Finally, the nominal supply voltage V_n is linearly dependent on the operating frequency, as modeled by Equation (7). Parameters δ and, η depend on the system configuration. We deduce their values by monitoring the supply voltage of the CPU of our x86 platform, while commanding changes of the operating frequency.

7.2. Simulation-Based Fault Injection

We use the GemFI framework [Parasyris et al. 2014] to execute our benchmarks on a simulated out-of-order CPU supporting the x86 Instruction Set Architecture (ISA). GemFI injects faults at different CPU pipeline stages. In the fetch stage, a fault corrupts a single bit of the instruction. In the decoding stage, the selection of registers is corrupted so the instruction in question reads from, or writes to, a different register. In the execution stage, faults corrupt a single bit of the computed result. Finally, faults in the memory stage corrupt a single bit of the value being transferred from/to memory. Even though we only inject faults to a subset of the CPU modules, these faults can be propagated to the majority of the CPU modules. For example, when a fault is injected during the execution stage of an integer instruction, the fault corrupts the result of the operation. If the result is stored in a register, then the fault propagates and corrupts the register file. Also, when injecting a fault to a memory write, the fault can corrupt the data cache hierarchy and even propagate to the main memory. Note that we model transient faults, that is, the injection of the fault only lasts for one clock cycle.

Simulated fault injection captures the "default" impact of faults on an application executed on top of unreliable hardware without employing any of the protection mechanisms provided by our framework. Consequently, all application tasks are susceptible to faults, and the result-check functions are ignored. The number of experiments for each application and pipeline stage (see above) is determined based on the methodology described in Leveugle et al. [2009] for a 99% confidence level and 1% error margin. For the purpose of our evaluation, we categorize the outcome of program execution in three bins: (i) crash if the program did not terminate normally, (ii) inexact if the result is not bitwise identical to that of a reliable execution, and (iii) exact if the result is bitwise identical to that of a reliable execution. The output of this phase is the probability for a single fault to result in a crash (P_{crash}) for each benchmark. This probability is used by the software fault injection mechanisms during native execution.

7.3. Software-Based Fault Injection during Native Execution

For the native (fast) executions of the benchmarks on our platform, we use software-based fault injection. This is designed to have two possible effects: (i) It forces a crash, and (ii) it corrupts a randomly chosen register of the CPU. The former is done with the probability P_{crash} computed in the GemFI simulation, and the latter with probability $1 - P_{crash}$. As in the simulation experiments, all protection mechanisms are disabled, and faults are injected in all application tasks. To validate that software-based fault injection yields realistic results, we compare the outcome of the native executions with the respective outcomes of simulated executions on GemFI. Figure 4, which summarizes the results for all benchmarks, shows that the software-based fault injection has practically equivalent effects to simulation-based fault injections using GemFI.

Finally, we support native execution scenarios with multiple fault injection. The runtime selectively executes application tasks in the reliable or unreliable mode and where the different protection mechanisms of our framework come into play. The voltage and frequency settings for the *FastRel* and the *FastUnRel* configurations are decided as follows. We pick f_h to maximize performance and derive respective nominal voltage V_h from Equation (7). We then set $V_l = \epsilon \times V_h | \epsilon < 1.0$. Frequency f_l is derived from Equation (7), and the fault rate of the *FastUnRel* configuration is derived from Equation (6) using V_l and V_h as parameters. The rate increases for smaller values of ϵ. Given a target fault rate, we randomly generate a set of fault injection intervals, expressed as number of cycles between faults, using a uniform distribution with a mean value equal to the target fault rate. We then use the performance counter infrastructure of x86 CPUs to interrupt application execution at those intervals and invoke the software-based

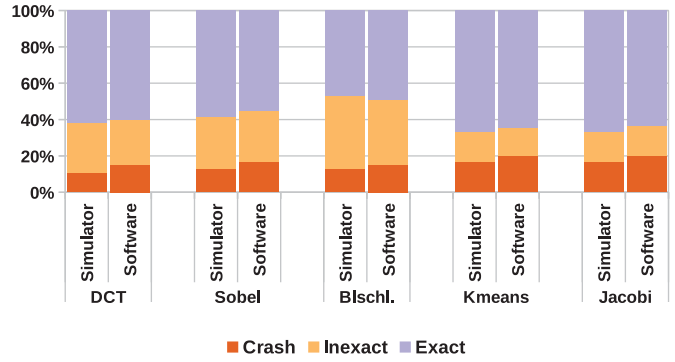

Fig. 4. Effects of single fault injection, using the GemFI simulator at the architectural CPU level, and the software-based approach during native execution.

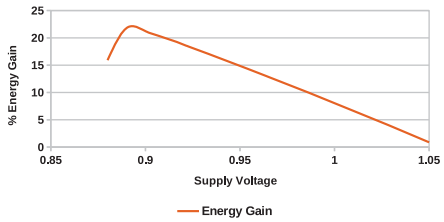

Fig. 5. Energy gains of a single task for *Sobel* executed at voltages $V_l < V_h$ for constant frequency $f_h = 3.7$GHz.

Table II. *SlowRel* and *FastUnRel* Configuration Settings Used in Our Evaluation, and Average Fault Rates of the *FastUnRel* Configurations

SlowRel		FastUnRel		
Freq.	**Voltage**	**Freq.**	**Voltage**	**Fault Rate**
1.67 GHz	0.90V	3.7 GHz	0.90V	10^{-7}
1.54 GHz	0.89V		0.89V	10^{-6}
1.41 GHz	0.88V		0.88V	10^{-5}

fault-injection logic. For each application, combination of protection mechanisms, and voltage level (fault rate), we perform 10,000 multiple fault injection experiments for a confidence interval of 95% and an error margin of 2.5%.

8. EXPERIMENTAL RESULTS

We study the behavior of benchmarks for the different protection levels supported by our runtime system (Section 3) using the methodology discussed in Section 5 on our CPU clocked up to 3.70GHz. We fix the *FastRel* configuration to the highest performance configuration, with $V_h = 1.06$V, $f_h = 3.7$GHz. To determine proper *FastUnRel* configurations, we run our benchmarks for different values for V_l while keeping frequency to f_h, observe their behavior, and compute the corresponding energy gains.

Figure 5 demonstrates the energy gains for a single task of *Sobel* when executed at different *FastUnRel* configurations in comparison with an execution in the *FastRel* configuration. The "sweet spot" is around 0.88V. If we further undervolt, inducing faults at higher rates, then tasks are practically certain to crash the CPU. This increases the overhead due to the activation of protection and task correction mechanisms in the *SlowRel* configuration and cancels any energy gains. In contrast, when a core operates in voltage regions higher than the PoFF, the failure rate is very small, and the functionality of our framework is rarely activated. Since these effects are observed in all the application benchmarks in our evaluation, we focus on the "promising" voltage range from 0.88V to 0.90V. In our evaluation, we set $FastRel = (1.06\text{V}, 3.7\text{GHz})$. Table II summarizes the configurations used in our experiments.

Figure 6 breaks down the execution time of a task for each benchmark using a fixed frequency of 3.7GHz. The time spent by the runtime system to create and schedule

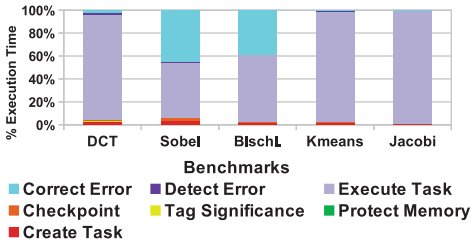

Fig. 6. Breakdown of task execution time for each benchmark.

Table III. Average Task Execution Time in Cycles (Thousands), Number of Tasks Executed Reliably/Unreliably, and Number of Voltage and Frequency Transitions for Each Benchmark

Bench.	C	N_r	N_u	$N_{FR \to SR}$	$N_{SR \to FR}$
DCT	133K	4096	28672	1	1
Sobel	50K	410	3684	1	1
Blscls	197K	90	10	1	1
Kmeans	283K	1500	13500	83	83
Jacobi	594K	830	7470	83	83

tasks, to protect the memory, and to checkpoint the state of each task before execution is less than 5% of the total task execution time. Task creation and scheduling overhead is practically constant, at about 5,000 cycles. The same applies to checkpointing, which costs approximately 2,000 cycles. Noticeably, in *Sobel* and *Blackscholes*, the overhead of correction is comparable with the task execution time. These two benchmarks execute an approximate version of the computation as a correction heuristic, whereas the rest simply discard the computed erroneous solutions, which incurs almost zero overhead.

Figure 7 summarizes our experimental results for a range of voltage settings and protection mechanisms. For each benchmark, we present two diagrams. The left one depicts the *cumulative distribution function (CDF)* of the percentage of experiments (y-axis) achieving a specific quality threshold (x-axis) under different protection mechanisms (different lines). For the media benchmarks (DCT, Sobel) the quality metric is Peak Signal To Noise Ratio (PSNR) (higher value is better). For the remaining benchmarks, quality is quantified by the relative error w.r.t the fully reliable execution (lower value is better). The two extreme bins of the x-axis correspond, on the one side, to experiments that resulted in bitwise exact results and, on the other side, to experiments producing very bad output quality. The percentage of crashed experiments can be deduced by subtracting the percentage of worst-quality experiments from 100%. The percentage of experiments that resulted in bitwise exact results are equal to the percentage of experiments that provide the best quality in the *CDF*. For a specific quality target, the height of each CDF line at the specific quality corresponds to the percentage of experiments that achieve the specific quality of results. Intuitively, the sooner (to the left) and the higher the lines raise, the better the respective protection configurations.

The diagrams to the right depict the average energy gains against a fully reliable execution (*FastRel* state) using our runtime in the *NP* configuration. The number of voltage and frequency transitions, the average execution time of a task in cycles, as well as the number of reliable and non-reliable tasks are given in Table III. In all scenarios where task significance information is taken into account, the task ratio is fixed to 10%, except *DCT*, in which the requested ratio is 13%. In *DCT*, all tasks that compute the upper-left coefficient corner need to be executed reliably. These tasks correspond to the 13% of the total number of tasks. In scenarios that do not exploit significance information, all tasks are executed unreliably.

When no protection mechanism is active, all experiments result in crashes. Basic protection eliminates crashes and can even lead to satisfactory behavior as long as the fault rate remains moderate. As expected, error resilience increases as more protection mechanisms are employed. As an exception, result-check functions (*B-RC*) may produce worse results compared with *BP* by discarding partially good results produced by tasks before they crash. On the other hand, energy gains are typically reduced as the amount of protection increases. Therefore, we select naive result check (*RC*) functions, which do not spend a lot of time to detect and correct an error. This increases the energy gains;

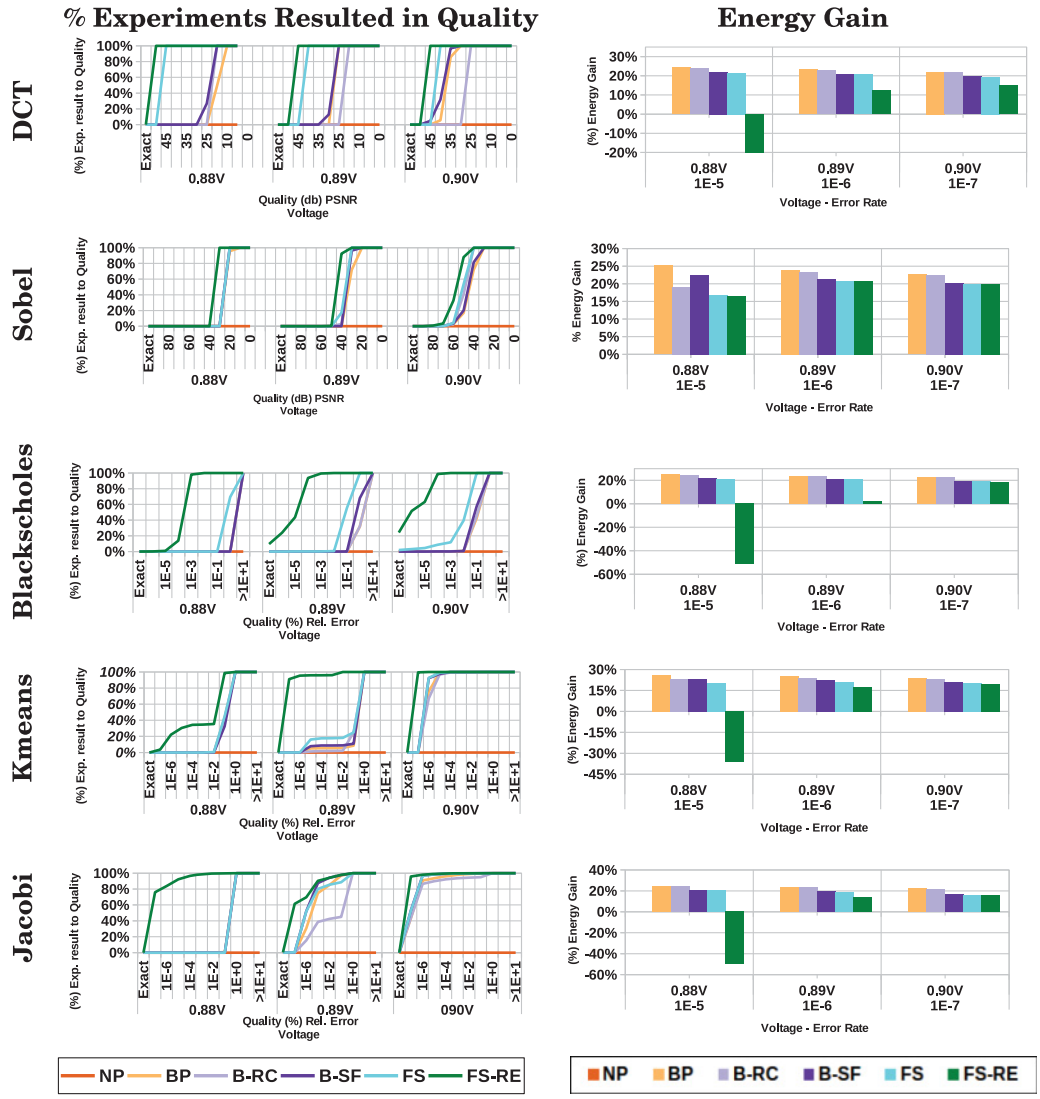

Fig. 7. Experimental results for different V_l values for the *SlowRel* and *FastUnRel* configurations. Percentage of experiments that achieved a certain quality (left), and energy gains with each protection scenario (right).

however, it decreases the quality of the end result. Another interesting observation is that task re-execution (*FS-RE*) does not guarantee perfect results, as is clearly visible from the CDFs in Figure 7. A task is re-executed reliably only if it crashes or the result-check function requests a re-execution. Since the result-check functions are simple, they miss silent data corruptions, which in turn lead to imperfect results. Finally, when combining all protection mechanisms, the application error resiliency is pushed to significantly higher fault rates. In the following paragraphs, we discuss the behavior of each application in more detail.

The two image processing benchmarks demonstrate a similar behavior. The transition from *NP* to *BP* completely eliminates any program crashes. However, there is

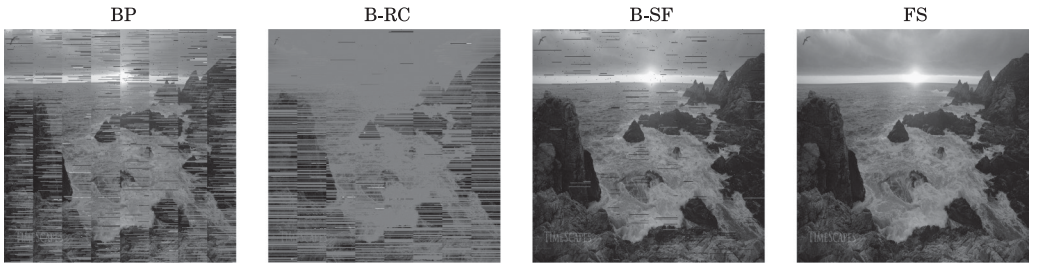

Fig. 8. *DCT* output at 0.89V, with one fault injected every 100,000 cycles. The images correspond (from left to right) to the *BP*, *B-RC*, *B-SF*, and *FS* protection configurations, resulting in PSNRs of 12-, 13-, 15-, and 37dB, respectively. A fault free execution leads results in a PSNR of 43dB. *NP* deterministically leads to crashes.

no guarantee for the quality of the output. The produced outputs are of unacceptable quality when executed in all *FastUnRel* configurations. Even the addition of a *result-check function (B-RC)* does not increase the quality; the same is observed for the *B-SF* scenario. In DCT *B-RC*, the detection part of the result-check function is able to detect many errors; however, when errors corrupt tasks that should had been significant, there is no efficient way to correct them. This motivates the usage of the significance information by our runtime system. On the other hand, in *Sobel*, the detection part is incapable of detecting many errors. In the *B-SF* scenario, significant tasks are protected by the software stack; however, there is no increase in the quality of the output. In the case of *DCT*, the absence of a result-check function allows faults that manifest on non-significant tasks to negatively impact the end quality. Figure 8 illustrates the output of four protection configurations (excluding *NP* and *FS-RE*) for the *DCT* benchmark. The corrupted images show the effect of faults when protection is not adequate, while the rightmost image shows that even in a highly faulty environment, our approach almost eliminates visible artifacts. In *Sobel*, the significance characterization of tasks simply spreads unreliability uniformly within the output; however, *PSNR* does not capture such effects. It is interesting to note that for *Sobel* at 0.88V, the *B-RC* leads to smaller energy gains than *B-SF*. Under such high error rates, tasks tend to crash frequently, which is detectable by the runtime system and therefore the correction part is invoked. However, in *Sobel*, the correction part of the result-check function is almost as costly as the task itself (Table III), so correcting a large number of tasks incurs excessive overhead. The combination of the result-check function with the significance values (*FS* scenario) results in increased quality. Even in the highest fault rates, the *FS* scenario delivers quality higher than *35*dB for *DCT* and *30*dB for *Sobel*, respectively, for all experiments. Similar behavior is observed for the *FS-RE* scenario. In the case of *Sobel*, the detection part of the result-check function is unable to detect most faults except the ones that lead to task crashes. Therefore the correction part (re-execution in FS-RE) is rarely executed. Consequently, the negative (energywise) impact of the re-execution is not captured in this benchmark.

In the *FS* configuration when the voltage is decreased from 0.90V to 0.88, the energy gains of *DCT* slightly increase from 18% to 21%, whereas in *Sobel* the energy gain is reduced from 20.0% to 16.0% The result-check function of *DCT* sets a default value (0) to the faulty output. For *Sobel*, an approximate version of the task is executed. Therefore, the energy gains due to undervolting are eliminated by executing the result-check function more frequently due to the higher fault rate. A similar trend is observed for *DCT* in the *FS-RE* configuration. Re-executing the entire task every time its output is detected as erroneous outweighs all energy savings and results in energy losses.

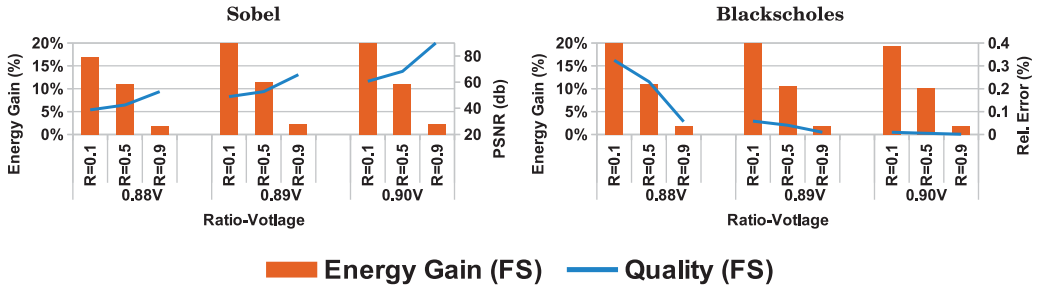

Fig. 9. Quality vs. energy tradeoffs using the *ratio* parameter in the FS configuration.

The computationally intensive *Blackscholes* uses mathematical functions, such as logarithms, square roots, and so on, that return Not A Number (NaN) or *inf* when arguments are outside their definition range. Detecting such errors is easy. Since many faults can be detected, *FS-RE* computes exact outputs 24% of the time when operating at 0.90V. The resulting output quality is exceptional with a relative error less than 0.03% across all experiments. However, at high fault frequencies, the application results in energy losses, since a large number of tasks need to be re-executed in the *SlowRel* domain.

k-means and *Jacobi* demonstrate similar characteristics. At 0.90V, in all protection scenarios, both applications result in a relative error less than 10^{-6}%. In *k-means*, the quality decreases rapidly for higher fault rates. Neither the result-check function nor the significance values increase output quality. The result check function has no efficient way to correct errors, and the small subset of the last significant iterations is unable to assign the points to the correct centers. For *Jacobi*, at 0.89V *BP* has better quality than *B-RC*. In *B-RC*, when an error is detected, the current solution estimate is replaced with that of the previous iteration. At high error rates, faults are frequently detected, and therefore the respective iterations are discarded. In *Jacobi*, it is better to rely on the self-healing attributes than correct the result.

Figure 9 presents experiments in which we vary the *ratio* parameter and record the energy savings and output quality for different values for the *Sobel* and *Blackscholes* benchmarks under the FS configuration. The *ratio* knob allows the user to select the percentage of reliably executed tasks and can effectively control the tradeoff between energy savings and quality loss. A similar behavior is observed in all benchmarks.

Note that energy gains are best obtained when shaving voltage guard bands to the point of first failure. More aggressive voltage scaling diminishes improvements in energy efficiency due to the overhead of error detection and correction.

9. RELATED WORK

In this section, we review the research effort on software and hardware techniques to detect platform errors and alleviate their effects on functionality and performance.

Topaz [Achour and Rinard 2015] is a task-based framework that executes computations unreliably. An online outlier detection mechanism detects and then re-computes unacceptable task results reliably. Chisel [Misailovic et al. 2014] selects approximate kernel operations to minimize an application's energy consumption while satisfying its accuracy specification. Our framework can dynamically operate at different energy gain/output qualities configurations using the *ratio* knob to gracefully trade off output quality with energy reduction.

Rinard et al., in one of the chronologically earlier efforts on task-based error-tolerant computing, propose a software mechanism that allows the programmer to identify

task blocks and then creates a profile-driven probabilistic fault model for each task [Rinard 2006]. This is accomplished by injecting faults at task execution and observing the resulting output distortion and output failure rates. Task Level Vulnerability (TLV) [Rahimi et al. 2013] captures dynamic circuit-level variability for each OpenMP task running in a specific processing core TLV meta-data are gathered during execution by circuit sensors and error detection units to provide characterization at the context of an OpenMP task. Based on TLV metadata, the OpenMP runtime apportions tasks to cores aiming at minimizing the number of instructions that incur errors. TLV does not consider error recovery and user-specified approximate execution paths.

Rehman et al. present a framework for reliable code generation and execution using reliability driven compilation [Rehman et al. 2016]. A compiler generates multiple, functionally equivalent, versions of a given function that differ in terms of vulnerability and execution time. On profiling the versions, the runtime system selects one that both increases the reliability of the system and meets the application's real-time constraints. Their work enforces functional correctness but does not exploit the algorithmic characteristic of significance. Salehi et al. [2015] introduces a system that selects a reliability robustness mechanism (Triple or Double Modular Redundancy, Dual Modular Redundancy (DMR)/Triple Modular Redundancy (TMR)) as well as the CPU operating voltage and frequency. Its goal is to minimize power consumption while achieving the reliability and timing requirements of the system. In our work, we do not seek functional correctness, but we offer a mechanism to exploit the algorithmic significance to allow errors to manifest only on non-significant computations. [Rehman et al. 2011] introduces the instruction vulnerability index (IVI) for software reliability estimation. Vulnerability indexes at the granularity of the function and the application is computed based on IVI. Given a user-specified tolerable performance overhead constraint, they perform compiler transformation to enhance code reliability. In our work, we do not take into account the instruction vulnerability. We consider the algorithmic property of significance to steer application execution on reliable and unreliable cores. Relax is an architectural framework that lets programmers turn off the recovery mechanism as well as annotate regions of code for which hardware errors can occur [de Kruijf et al. 2010]. The hardware supports error detection and a C/C++ language-level recovery mechanism provides error recovery from hardware faults at different levels of code granularity.

An example of a domain-specific approach to improve fault resiliency is described by Schmoll et al. [2013]. Algorithmic and static analysis is performed to detect which variables must be computed reliably and which variables can be computed approximately in an H.264 video decoder. Correction techniques are applied at execution time only to errors that have been predetermined to have the largest impact to the output result. We follow a domain-agnostic approach in our article, yet we provide sufficient abstraction for implementing domain-specific approximation methods.

Hardware support for error-tolerant and approximate computing spans designs to novel architectures. Razor is a processor design that is based on dynamic detection and correction of timing failures of the critical paths due to below-nominal supply voltage [Ernst et al. 2003]. The key idea is to tune supply voltage by monitoring the error rate during operation using shadow latches controlled by delayed clocks.

The observation that the sequence of instructions in an application binary can have a significant impact on timing error rate is studied in Hoang et al. [2011]. A number of simple, yet effective, code transformations that reduce error rate are introduced.

In Iyer et al. [2005], a hardware module monitors the processor pipeline and checks for possible control flow violations (infinite loops). This module is used by the OS/compiler/application to detect errors and take corrective action. ERSA is a multicore architecture where cores are either fully reliable or have relaxed reliability [Leem et al.

2010]. ERSA uses an explicit and application-specific mapping of code to cores with different levels of reliability. Our work follows a different approach, the programmer uses significance to indicate code with relaxed correctness semantics, and the framework implements error detection and recovery, potentially approximating the task output.

EnerJ proposes a programming model that explicitly declares data structures that may be subject to unreliable computation in return for increased performance or fault tolerance [Sampson et al. 2011]. EnerJ allows operations to be computed in aggressively voltage-scaled processors and data structures to be stored in DRAM with low refresh rate and SRAM with low supply voltage. Exposing such computing to the programmer requires expanding the processor ISA with instructions that offer only an expectation, rather than a guarantee, that a certain operation will be performed correctly [Esmaeilzadeh et al. 2012]. Contrary to our framework, EnerJ specifies significance in the granularity of data, whereas we use as a vehicle the granularity of a task.

Approximate computing refers to the deliberate and controllable injection of imprecise computations aiming at energy and performance improvements.

Vassiliadis et al. [2015] exploit the significance of task-based computations by executing approximate alternatives of the least-significant ones. Their work focuses on approximate computing in which the task execution semantics are fully controllable: The runtime system can execute a task, or a approximate version of the task, or even drop the task altogether. In this article, we study how task significance can be exploited to run tasks unreliably and present a different programming model and runtime system, designed to operate on top of unreliable hardware and gracefully tolerate faults that may mainifest due to unreliable task execution. Green allows the programmer to write several versions of a single function with varying precision [Baek and Chilimbi 2010]. A runtime system then monitors the application's quality online to select the degree of approximation that meets a target Quality Of Service (QoS) level. Sloan et al. provides manual code transformation guidelines for approximate computing and fault recovery [Sloan et al. 2012]. These frameworks rely on application-specific invariants. In our case, the programmer uses a higher level of abstraction for fault detection and recovery, namely computational significance and result-check functions, while the runtime system supports error recovery mechanisms.

ApproxIt is a framework for approximate computation of iterative-based methods, based on a lightweight quality control mechanism [Zhang et al. 2014]. The error-resilient and error-sensitive parts of each application are identified during an offline phase. Then, at the online stage, reconfiguration of approximation modes are performed at certain iterations considering the time-varying resiliency requirements of each application and respecting the user-specified output quality. Unlike our work, ApproxIt uses approximation at the level of one loop iteration and not at the task level.

Laurenzano et al. [2016] describe input responsive approximation. A canary input (a much smaller representation of the full input) is used to dynamically predict the accuracy and speedup characteristics of the full input for a number of approximation options. Using the canary input, they can easily explore all approximate options and choose the fastest option that achieves the desired level of accuracy.

There has been a considerable amount of work on compilers to trade off accuracy for other benefits such as reduced energy consumption. Loop perforation is a compiler technique that drops non-critical loop iterations. It considers critical those that have to be always executed for the application to produce an acceptable quality [Sidiroglou-Douskos et al. 2011]. Our approach corrects an error using a variety of user-provided methods that differs from just skipping erroneous or insignificant loop iterations.

Ringenburg et al. [2015] introduces offline debugging and online monitoring mechanisms for approximate programs. The offline mechanisms detect correlations between Quality Of Result (QoR) and approximate operations to evaluate the degree to which

approximate operations affect approximate variables. The online mechanisms consist of verification functions that detect and compensate QoR loss while maintaining energy gains. Khudia et al. [2015] present an output-quality monitoring and management technique that meets a given output quality threshold based on the observation that simple prediction approaches can predict approximation errors. The authors use an error detection module that tracks predicted errors to find the elements that need correction. The recovery module, which runs in parallel to the detection module, re-executes the iterations that lead to high errors. Lightweight checks are available for a set of benchmarks [Kadric et al. 2014]. When the checks detect a fault, the results are re-computed. Our work is complementary to theirs; the proposed result-check functions can be realized through the use of the aforementioned frameworks.

10. CONCLUSIONS

In this article, we introduce and evaluate a framework that enables execution on platforms operating unreliably, outside their normal voltage/frequency envelope, to aggressively reduce energy consumption. We present a programming model for the development of error-resilient programs in a disciplined manner. Our model exploits programmer wisdom to characterize task significance to provide checks and repair mechanisms to the outputs of tasks that are executed unreliably. We evaluate the effectiveness of different protection mechanisms. We show that traditional system software protection mechanisms are not adequate; however, their combination with programmer wisdom provides effective protection against crashing and silent data corruptions, while enabling considerable energy gains. Interestingly, modern processors with the assistance of our framework can produce acceptable results until they reach the Point of First Failure. Below that point, either additional energy gains are too low or massive failure rates defeat any software-based realistic protection mechanism.

A key direction for future work is to investigate significance in conjunction with parallelism. The approach supports parallel execution. Sometimes, significance is expected to be orthogonal with parallelism; however, equally often significance and parallelism may be conflicting concerns (in terms of task creation). Optimizing programs to simultaneously address both concerns is an open problem. Introducing parallelism also opens research directions in the areas of runtime policies for resource management and system configuration provisioning, depending on the dynamic requirements of applications (number and requirements of significant and non-significant tasks).

REFERENCES

Sara Achour and Martin C. Rinard. 2015. Approximate computation with outlier detection in topaz. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Woongki Baek and Trishul M. Chilimbi. 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

David Blaauw, Sudherssen Kalaiselvan, Kevin Lai, Wei-Hsiang Ma, Sanjay Pant, Carlos Tokunaga, Shidhartha Das, and David M. Bull. 2008. Razor II: In situ error detection and correction for PVT and SER tolerance. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC), Digest of Technical Papers*.

Jeremy Constantin, Lai Wang, Georgios Karakonstantis, Anupam Chattopadhyay, and Andreas Burg. 2015. Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*.

Shidhartha Das, David Roberts, Seokwoo Lee, Sanjay Pant, David Blaauw, Todd Austin, Krisztián Flautner, and Trevor Mudge. 2006. A self-tuning DVS processor using delay-error detection and correction. *IEEE J. Solid-State Circ.* 41, 4 (2006).

Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. 2010. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th International Symposium on Computer Architecture*.

Arnaud Doucet, Simon Godsill, and Christophe Andrieu. 2000. On sequential monte carlo sampling methods for bayesian filtering. *Stat. Comput.* 10, 3 (2000), 197–208.

Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. 2003. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*.

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Architecture support for disciplined approximate programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*.

Íñigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. 2015. ApproxHadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the 22th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Meeta S. Gupta, Jude A. Rivers, Pradip Bose, Gu-Yeon Wei, and David Brooks. 2009. Tribeca: Design for PVT variations with local recovery and fine-grained adaptation. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*.

Giang Hoang, Robby Bruce Findler, and Russ Joseph. 2011. Exploring circuit timing-aware language and compilation. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Ravishankar K. Iyer, Nithin M. Nakka, Zbigniew T. Kalbarczyk, and Subhasish Mitra. 2005. Recent advances and new avenues in hardware-level reliability support. *IEEE Micro* 25, 6 (2005).

Norman James, Phillip Restle, Joshua Friedrich, Bill Huott, and Bradley McCredie. 2007. Comparison of split-versus connected-core supplies in the POWER6 microprocessor. In *Proceedings of the 2007 IEEE International. Solid-State Circuits Conference. Digest of Technical Papers*.

Edin Kadric, Kunal Mahajan, and André DeHon. 2014. Energy reduction through differential reliability and lightweight checking. In *Proceedings of the 22nd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*.

Daya S. Khudia, Babak Zamirai, Mehrzad Samadi, and Scott Mahlke. 2015. Rumba: An online quality management system for approximate computing. *SIGARCH Comput. Archit. News* 43, 3 (2015).

Michael A. Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. 2016. Input responsiveness: Using canary inputs to dynamically steer approximation. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Larkhoon Leem, Hyungmin Cho, Jason Bau, Quinn A. Jacobson, and Subhasish Mitra. 2010. ERSA: Error resilient system architecture for probabilistic applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*.

Régis Leveugle, A. Calvez, Paolo Maistri, and Pierre Vanhauwaert. 2009. Statistical fault injection: Quantified error and confidence. In *Proceedings of the 2009 Design, Automation & Test in Europe Conference & Exhibition*.

Abdelhafid Mazouz, Alexandre Laurent, Benoît Pradelle, and William Jalby. 2014. Evaluation of CPU frequency transition latency. *Comput. Sci.* 29, 3–4 (2014), 187–195.

Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. *SIGPLAN Not.* 49, 10 (2014).

Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. 2009. *Introduction to Interval Analysis* (1st ed.). Society for Industrial and Applied Mathematics.

K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas. 2014. GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates. In *Proceedings of the 2014 44th Annual IEEE/IFIP Int. Conference on Dependable Systems and Networks (DSN)*.

Abbas Rahimi, Luca Benini, and Rajesh K. Gupta. 2012. Analysis of instruction-level vulnerability to dynamic voltage and temperature variations. In *Proceedings of the 2012Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

Abbas Rahimi, Andrea Marongiu, Paolo Burgio, Rajesh K. Gupta, and Luca Benini. 2013. Variation-tolerant OpenMP tasking on tightly-coupled processor clusters. In *Proceedings of the Conference on Design, Automation and Test in Europe*.

Vijay Janapa Reddi, Svilen Kanev, Wonyoung Kim, Simone Campanoni, Michael D. Smith, Gu-Yeon Wei, and David Brooks. 2010. Voltage smoothing: Characterizing and mitigating voltage noise in production processors via software-guided thread scheduling. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*.

Semeen Rehman, Kuan-Hsun Chen, Florian Kriebel, Anas Toma, Muhammad Shafique, Jian-Jia Chen, and Jörg Henkel. 2016. Cross-layer software dependability on unreliable hardware. *IEEE Trans. Comput.* 65, 1 (2016), 80–94.

Semeen Rehman, Muhammad Shafique, Florian Kriebel, and Jörg Henkel. 2011. Reliable software for unreliable hardware: Embedded code generation aiming at reliability. In *Proceedings of the 7th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*.

Martin Rinard. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS'06*. 324–334.

Michael Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. 2015. Monitoring and debugging the quality of results in approximate programs. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Mohammad Salehi, Mohammad Khavari Tavana, Semeen Rehman, Florian Kriebel, Muhammad Shafique, Alireza Ejlali, and Jörg Henkel. 2015. DRVS: Power-efficient reliability management through dynamic redundancy and voltage scaling under variations. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design*.

Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Florian Schmoll, Andreas Heinig, Peter Marwedel, and Michael Engel. 2013. Improving the fault resilience of an H.264 decoder using static analysis methods. *ACM Trans. Embedded Comput. Syst.* 13, 1s (2013).

Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*.

Joseph Sloan, John Sartori, and Rakesh Kumar. 2012. On software design for stochastic processors. In *Proceedings of the 49th Annual Design Automation Conference*.

Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*.

G. Tziantzioulis, A. M. Gok, S. M. Faisal, N. Hardavellas, S. Ogrenci-Memik, and S. Parthasarathy. 2015. b-HiVE: A bit-level history-based error model with value correlation for voltage-scaled integer and floating point units. In *Proceedings of the 52nd Annual Design Automation Conference*.

Vassilis Vassiliadis, Charalampos Chalios, Konstantinos Parasyris, Christos D. Antonopoulos, Spyros Lalis, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2015. A significance-driven programming framework for energy-constrained approximate computing. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*.

Vassilis Vassiliadis, Jan Riehme, Jens Deussen, Konstantinos Parasyris, Christos D. Antonopoulos, Nikolaos Bellas, Spyros Lalis, and Uwe Naumann. 2016. Towards automatic significance analysis for approximate computing. In *Proceedings of the Internationla Symposium on Code Generation and Optimization*.

Foivos S. Zakkak, Dimitrios Chasapis, Polyvios Pratikakis, Angelos Bilas, and Dimitrios S. Nikolopoulos. 2012. Inference and declaration of independence: Impact on deterministic task parallelism. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*.

Qian Zhang, Feng Yuan, Rong Ye, and Qiang Xu. 2014. ApproxIt: An approximate computing framework for iterative methods. In *Proceedings of the 51st Annual Design Automation Conference 2014 (DAC'14)*.

Received August 2016; revised February 2017; accepted February 2017