A Significance-Aware Software Stack for Computing on Unreliable Hardware

Konstantinos Parasyris[1,2], Vassilis Vassiliadis[1,2],
Christos D. Antonopoulos[1,2], Spyros Lalis[1,2] and Nikolaos Bellas[1,2]

[1]Dept. of Electrical and Computer Engineering, University of Thessaly, Volos, Greece
[2]I.RE.TE.TH., Centre for Research and Technology, Hellas, Volos, Greece
koparasy, vasiliad, cda, lalis, nbellas@inf.uth.gr

Abstract

Next-generation many-core architectures may feature cores that operate at very low voltage levels but occasionally produce faults. In this paper, we present a task-based programming model and runtime that allow selected parts of a computation to execute on such unreliable cores. Using our programming model, the developer can specify the tasks that do not contribute significantly to the quality of the final result as candidates for unreliable execution, while providing suitable result-check/repair functions. In addition, to limit the propagation of faults to the rest of the computation, the runtime provides additional protection via read-only memory regions, and support for task re-execution. We present an evaluation of our approach on top of both a simulated fault injection environment, and a native system using software fault injection. We quantify the contribution as well as the overhead of each application- and runtime-level protection mechanism. We also evaluate the effect of traditional and emerging fault detection mechanisms offered by the hardware and OS layers.

1. Introduction

Task-based programming modelshave been successfully used to enable the parallel execution of a wide range of applications on top of multi- and many-core platforms. These programming models assume that the underlying hardware is fully reliable. Recent technology trends suggest that this may no longer be the case in the near future. The significant energy cost of guardbands on the operating frequency or voltage of circuits in order to guarantee error-free operation even when subjected to worst-case combination of process, voltage and temperature (PVT) non-idealities, as well as the continued efforts towards even denser structures, have pushed researchers towards relaxing strict enforcement of precise hardware functionality.

At the same time, as shown by previous work on approximate computing, in several classes of applications, not all parts or execution phases of a program affect the quality of its output in an equal way. In fact, the output may remain virtually unaffected even if some parts of the computation produce incorrect results or fail completely. Multimedia and visualization kernels as well as iterative and stochastic algorithms can all tolerate imprecision in some phases of their computation and up to a certain extent.

Based on this observation, we present a task-based model for capturing application-level knowledge about the significance of different computations, so that non-significant code can be executed on potentially unreliable cores, while handling the errors that may occur in a controlled way. Using our programming model, the developer explicitly declares the significance of tasks, depending on how strongly they contributes to the quality of the final output. For non-significant tasks (or entire task groups), which may execute on unreliable cores, the developer also provides a result-check function that is called by the runtime when task execution completes in order to inspect the task status and produced output, and, if needed, take corrective action by repeating the task on a reliable core or simply by returning an approximate/default output value.

The main contributions of this paper are the following: (i) We propose a programming model for task-based execution on top of unreliable cores which allows the developer to identify significant tasks, and make application-level provisions for error recovery while controlling the degradation of program output; (ii) We discuss a proof-of-concept runtime system that provides support for this programming model; (iii) We evaluate our approach via extensive simulations and native execution experiments, using three benchmarks from different application domains; (iv) We study the effect of different levels of error detection support from the underlying system layers (OS and hardware).

The experimental evaluation indicates that our mechanisms significantly improve the error resilience of applications in terms of both reduced execution failures and controlled degradation in the quality of results, in the presence of single faults. We show that our protection mechanisms may reduce the failure rate of an application from 35% to as low as 0.43% for a single fault injection. We find that although emerging hardware support for error monitoring and reporting is helpful, traditional OS and hardware mechanisms such as traps are highly effective when combined with programmer wisdom.

The rest of the paper is structured as follows. Section 2 briefly describes the hardware platform and error model assumed in our work. Section 3 introduces the programming model, using DCT as a running example. Section 4 discusses the runtime system that serves as the back-end of the compiler which implements the programming model. Section 5 presents the experimental evaluation on top of an extended version of the Gem5 simulator [2] that provides support for fault injection, showing the advantages and quantifying the overhead of our approach. Section 6 provides an overview of related work. Finally, Section 7 concludes the paper and identifies directions for future work.

2. Hardware Platform Assumptions

As a basis for our approach, we consider a general-purpose multi-core hardware architecture, that can be afflicted with transient faults. The system consists of several interconnected cores, which can work in a fully reliable or potentially unreliable mode. We focus on transient errors that are injected in the core for a single cycle.We assume that the mode of operation of individual cores can be set by the system software, at runtime, using techniques such as voltage scaling, and instruction replay [3].

The system software itself always runs in reliable mode, but there is room for flexibility when it comes to the execution of non-critical parts of the application code. Of course, in order to take meaningful task placement decisions, the system software must be provided some information regarding the significance of different parts of the application code with respect to execution robustness and output quality.

Without loss of generality, the effects of unreliable execution on the application program can be categorized as follows: **Program failure** is caused by errors that lead to a crash of the application. **Program corruption** occurs when the application terminates normally, but the output quality is severely affected by the error; this is often referred to as a Silent Data Corruption (SDC). **Correct program** execution occurs when the application terminates normally, and the output quality is qualitatively correct within some user-supplied or domain-specific tolerance; this can also be considered as a special case of SDC. **Bitwise exact** execution occurs when the application terminates normally, and the output is bit-exact compared to an error-free execution. We use this classification in Section 5, where we evaluate our programming model.

3. Significance-Centric Programming Model with Fault Tolerance Support

The proposed programming model offers programmers the expressiveness and mechanisms to enable execution on non-reliable hardware, without catastrophic results or uncontrolled degradation of the quality of the end-result. The main design objectives are the following:

Significance Characterization: The programming model should allow the developer to characterize computations according to their degree of significance in an easy and intuitive way. Significant computations need to be executed correctly, thus will always be scheduled on reliable cores, while non-significant computations may be executed on non-reliable cores and thus may produce incorrect results.

Safety - Isolation: In order to limit fault propagation throughout the application, it should be possible for the developer to define early error detection and possibly correction methods on the outputs of non-significant computations that can be executed on unreliable cores.

Synchronization: Due to the uncertain behavior of computations that execute on unreliable hardware, traditional synchronization mechanisms are overly stringent. More elastic synchronization support is needed, based on timeouts and/or requiring only a subset of the computations to terminate.

```
1  /* DCT Task Result Check function (TRC) */
2  /* Returns int, takes the same arguments as the task */
3  int dct_trc(...) {
4    if ( task_status_get() ) {
5      /* At least one fault was detected by HW/OS */
6      /* Set output to 0 for all sub-blocks that may have been
         affected */
7      ...
8      coeff = 0;
9    }
10   else {
11     /* Check for extreme coefficient values and in such cases set
          coefficients to 0 */
12     ...
13     if ( !isnormal(coeff) || (fabs(coeff)>MAX_DCT_COEFF)
14       coeff = 0;
15   }
16
17   /* The TRC function took care of faults, do not re-execute the
        task */
18   return TRC_SUCCESS;
19 }
20
21
22 /* Calculate the coefficients for a specific 2x4 block of a number
       of different 8x8 blocks */
23 void dct_task(...) {
24   ...
25 }
26
27
28 /*
29    Significance ...  higher is more significant
30    100  90  70  30
31    80   40  30  10
32
33    Each 8x8 block is further split into eight 2x4 blocks. */
34
35 /* DCT calculation. Beyond the arguments necessary for the
       computation, we provide the ratio of tasks that will be
       characterized as significant */
36 void DCT(..., double sgnfRatio)
37 {
38   /* Significance look up table for each 2x4 sub-block */
39   int sgnf_lut[] = {100,  90,  70,  30,
40                     80,  40,  30,  10};
41
42   /* Each task will compute a specific 2x4 sub-block of the DCT
        coefficient frequency space for a number of 8x8 blocks. */
43   for each 2x4 sub-block K {
44     /* Create a task to calculate that sub-block for a number of 8
          x8 blocks */
45     #pragma omp task label(dct) \
46       significance(expr(sgnf_lut[K]>=sgnfRatio*100))\
47       tasktolerance(dct_trc())
48         dct_task(...);
49   }
50   /* Wait for all tasks unless the wait times-out after 16 msec */
51   #pragma taskwait all label(dct) time(16)
52 }
```

Listing 1: Programming model use case: DCT pseudo-code

We adopt a task-based paradigm, similarly to OMPss [5]. Listing 1 illustrates the use of the main mechanisms of our programming model, using Discrete Cosine Transform (DCT) as a running example.

3.1. Task Definition and Significance Characterization

Tasks are created using the *#pragma omp task* compiler directive (Listing 2). Tasks can be named/grouped via *label(...)*

```
#pragma omp task  [significance(expr(...)] [label(...)]
  [tasktolerance([taskcheck()], [redo(...)])]
  [in(...)] [out(...)]
```

Listing 2: #pragma omp task

using a common identifier, which can then be used as a reference in order to set parameters of group behavior, or when placing a synchronization barrier (as will be discussed in the sequel). The *significance(expr(...)* clause is used to specify the significance of the task, via the boolean expression *expr*. When a task is about to be spawned and the respective *expr()* evaluates to true, the task will be characterized as significant, otherwise it will be tagged as non-significant.

In Listing 1, lines 45-46 a new task is created to compute the frequency coefficients of a specific 2x4 sub-block of a number of 8x8 pixel blocks. Upper left sub-blocks are more significant than lower right, as illustrated by the *sgnf_lut* array.

The clause *tasktolerance([taskcheck()], [redo(...)])* allows the programmer to specify a function (*taskcheck*) which will be executed reliably after each non-significant task completes or crashes. The main jobs of a result-check function are: (i) to perform a rough estimation of the correctness of task output, in order to detect and isolate errors early; (ii) to optionally assign default values or approximate the output of the task, should task execution have failed; (iii) to optionally change the task characterization and/or system configuration across task re-executions, in case the original task is to be re-executed. The result-check function has implicitly access to all arguments of the corresponding task. Should a task result-check function return TRC_REDO, the task is re-executed by the runtime system until it reaches the permitted maximum number of re-executions specified by the *redo(number)* option.

In line 47 of Listing 1 *dct_trc()* is specified as the result check function for the spawned tasks. The task result check function tries to detect errors, either using hardware and OS support (Line 4), or by checking the produced output (Line 13). In case of errors a simple, yet effective for the specific application, correction strategy is applied: the respective coefficients are set to 0. Correction in DCT is not based on re-executions, therefore the task result check function always returns *TRC_SUCCESS* (Line 18).

3.2. Synchronization

We extend the *taskwait* pragma of OMPss with extra clauses to allow for elastic synchronization for a group of tasks in the presence of failures, as well as for result-checking at the task group level (Listing 3). The *label()* clause identifies the respective group of tasks.

```
#pragma omp taskwait [on(...)] [label(...)]
  [time(...), ratio(...), all]
  [grouptolerance([groupcheck()]), [redo(...)]]
```

Listing 3: #pragma omp taskwait

Relaxed synchronization can be expressed in three different ways: *all*, waits for all tasks to finish. This is equivalent to the traditional, strict barrier synchronization semantics, and is the default behavior of our model. *ratio()*, allows the user to specify a percentage of non-significant tasks that must finish before resuming execution after the barrier (however still all significant tasks will have to finish). Finally, *time()*, allows the user to define a time watchdog. More than one options can be specified simultaneously. In such a case when any of the conditions is satisfied and all the significant tasks have finished their execution, execution resumes after the barrier and the runtime forcefully terminates the remaining non-significant tasks. Terminated tasks execute the user-defined task result-check function, however they can not be re-executed.

In Line 51 of Listing 1 a taskwait for tasks of the *dct* task group is specified, with a timeout of 16 msec.[1]

Similarly to task-level result-checking, we allow the programmer to specify a group-level result-checking function, which is evaluated at the respective synchronization point (*omp taskwait* with the task group identifier), after the execution of the corresponding task group.

4. Runtime System

The main purpose of the runtime system is to serve as a compiler back-end. At initialization the runtime spawns as many worker threads as the number of cores in the system, binds them one-to-one on the cores and designates them as reliable / non-reliable, according to the mode of operation of the core.

Whenever a new task group is created the runtime creates a set of work queues associated with the task group: (i) A queue of non-ready to execute tasks, which have unresolved data dependencies or pending data transfers. (ii) A queue of ready to execute tasks, polled by idling worker threads; to give priority to significant tasks, these are added to the head of the queue, whereas non-significant tasks are added to the tail. (iii) A queue of finished tasks, used only in case of group re-execution. A similar set of global queues is available for tasks that are not members of any task group.

Upon termination of a non-significant task, the runtime system invokes the task check function, if the programmer has defined one. This function is always executed in reliable mode, by using one of the mechanisms briefly described in Section 2. Similarly, task group result check functions are executed after the taskwait for a task group is fulfilled.

Another responsibility of the runtime system is to exploit the error detection mechanisms offered by the underlying system layers (operating system and hardware). On initialization it defines handlers for all catchable traps. Should a trap occur on a task, the runtime upcalls the task result check function. The fact that the task finished abnormally or experienced a soft error (detected through mechanisms such as razor flip-flops) can be inferred via primitives of the runtime API.

[1] It corresponds to a target frame rate of 30fps, assuming DCT corresponds to almost 50% of the computation time for each frame.

Our current implementation is designed for a shared memory system. Note that this is the worst-case scenario in terms of fault tolerance and reliability: erroneous stores by computations that execute on unreliable cores may affect the memory environment of computations that execute on reliable cores.

5. Evaluation

5.1. Simulation Approach

We use the Gem5 simulator [2], augmented with fault injection capability to simulate an out of order x86 Instruction Set. A fault is introduced in the form of a bit flip during a pipeline stage (Fetch, Decode, Execute or Memory). We support – and in a subset of the experiments we exploit – razor-like fault detection capability for timing faults.

We study three different scenarios in terms of the fault detection and correction (FDC) that takes place during task execution. In the first scenario, called *No-FDC*, all application tasks may be executed unreliably and are susceptible to faults. Tasks do not have any result check function, so faults have a direct impact on application behavior.

In the second scenario, called *T-FDC*, only non-significant tasks are executed unreliably. Also, the runtime in cooperation with the OS and hardware provides "traditional" error detection support, for traps/exceptions and timeout errors. The result check function of non-significant tasks is invoked when the respective execution completes. Note that undetected errors may still occur.

The third scenario, called *M-FDC*, represents a "modern" system equipped with razor-like technology, where the hardware can detect timing errors. This case is similar to *T-FDC*, but in addition timing error induced fault injections are reported to the runtime, which in turn passes this information to the application. As in *T-FDC*, the runtime invokes the result check function of non-significant tasks, making it possible for the application to check for and handle not only traps/exceptions and timeout errors, but also SDCs.

We conduct fault injection campaigns for each different scenario and benchmark, injecting a single fault in each experiment. The campaigns are generated based on the methodology described in [8], using 99% as a target confidence level and 1% as the error margin. In each experiment a bit flip is injected using a uniform distribution among all executed instructions and pipeline stages (Fetch, Decode, Execute and Mem).

5.2. Benchmarks

We use a set of three benchmarks *DCT, K-means* and *Jacobi*. *DCT* is a module of video compression kernels, which transforms a block of image pixels to a block of frequency coefficients. Only the tasks that compute low frequency coefficients, in the upper left corner of each 8x8 frequency block, are significant. In the *T-FDC* scenario, where SDCs are not detected by the hardware, if the runtime does not report an error then the result-check function attempts to detect SDCs in the task output via a simple out-of-bounds check; coefficients that do not respect the bounds are set to zero. Infinite loops are handled using the timeout-based option for task synchronization.

Benchmark	"Correct" (Acceptable Quality) Criterion
DCT	The PSNR of the final image is no more than 10% less than that achieved by the *golden* execution.
Jacobi	Relative error of the solution differs less than 10% from the defined tolerance, or is occasionally lower than that of the golden execution.
K-means	If the experiment output differs by less than 10% from the *golden* output then it is characterized as Correct.

Table 1: Benchmark-specific quality criteria for "correct" (acceptable quality) characterization of experiments

K-means is an iterative algorithm for grouping data points from a multi-dimensional space into k clusters. In the first phase of each iteration, the data points are assigned to different tasks, which independently determine the nearest cluster for each data point.In the second phase, another task group is used to update the cluster centers by taking into account the points that have moved. It is possible to tolerate errors in the movement of individual points in the first phase. In contrast, the second phase is significant as it is much harder to tolerate a wrong estimate of a cluster center. The result-check function of non-significant tasks is minimalistic, exploiting the fault-tolerant nature of this iterative application: if the runtime reports an error, the output of the current iteration for the points controlled by the task is ignored.

Jacobi is an iterative solver of diagonally dominant linear equation systems. In this case, an iteration is considered, in its entirety, either as significant or as non-significant, depending of the relative error between the user-specified tolerance and the estimated error of the current solution. The rationale is that one can afford errors during the initial iterations, but cannot do so in the last steps where the algorithm approaches the final solution. Following, as long as the current error is at least *10x* bigger than the tolerance given by the user, all tasks of the next iteration are flagged as non-significant, otherwise they are treated as significant. Relying on the well-known "self healing" nature of Jacobi, we again use a simple result-check function. In this case, if the runtime system reports an error, task output is replaced with the output of the previous iteration.

5.3. Fault Campaigns

Figure 1 depicts the behavior of benchmarks for the three different protection scenarios, whenever a single fault injection occurred in four different modules of the hardware pipeline. The criteria for classifying a result as correct for each application are summarized in Table 1.

In unprotected experiments (*No-FDC*) faults impacting the selection of registers during the decoding stage result to severe errors. Most instructions in the x86 ISA operate on a number of operands, involving at least one error sensitive address calculation. Moreover a single instruction is divided to multiple micro-ops in the micro architecture, hence a single error in the decoding stage may corrupt multiple micro-ops. The fetch stage is also highly susceptible to errors: many experiments failed due to execution of an illegal instruction. The failure

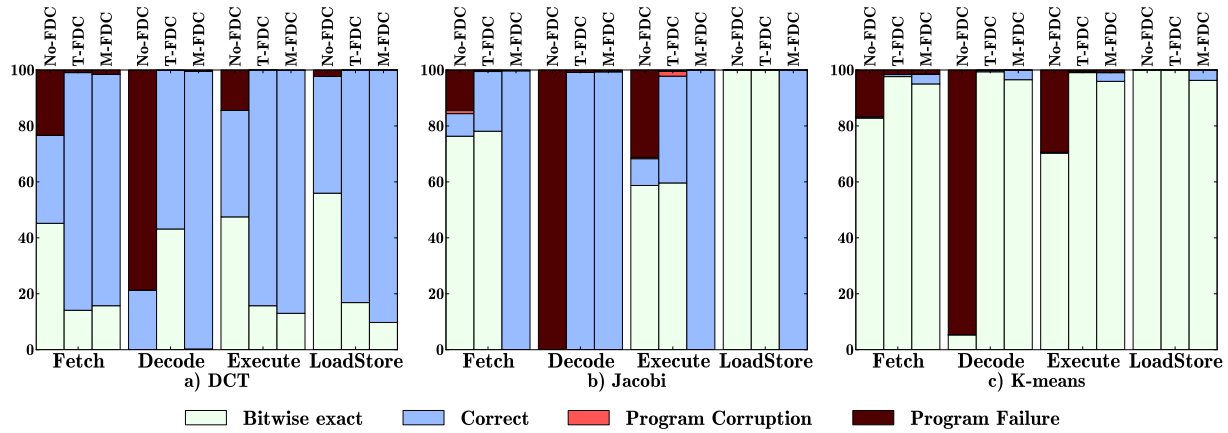

Figure 1: Error tolerance of applications in single fault injections, under different protection scenarios.

rate of experiments is even higher when faults are injected during the execution stage most noticeably due to a segment violation. Finally faults corrupting the values of loaded or stored arguments of an instruction usually do not result to failures but rather to SDCs. We observe failures only when the fault affects a value used in memory address calculations.

DCT results indicate that the transition from no protection to HW/SW protection (either *T-FDC* or *M-FDC*) is adequate to nearly eliminate application failures. 35% of the experiments executed under the *No-FDC* scenario failed. This number dramatically decreased to 0.43% and 0.58% for the *T-FDC* and *M-FDC* protection scenarios respectively. In the rare cases our framework did not manage to recover from the injected error, it was because the error propagated either to the runtime system or corrupted memory locations assigned to significant tasks. Error propagation to significant tasks would be naturally limited in a distributed address space setup.

When comparing the result quality of *T-FDC* and *M-FDC* in DCT, a counter-intuitive trend comes up. Even though *M-FDC* can detect errors more accurately than *T-FDC*, evidently it produces less bit-exact results in comparison with the later. This is due to the way the task result check (TRC) function is implemented. Whenever the runtime reports a detected error, we opt to drop all output generated by the task that suffered the fault. This decision stems from the fact that TRC functions are executed only for non-significant tasks, therefore the results produced by the task are not indispensable. Still, uncontrolled, extreme changes to the output of a task, even a non-significant one, will not necessarily have negligible effect to the overall output of the algorithm. Therefore, we choose to be aggressively pessimistic and drop all possibly affected output in the event of a confirmed error, even if that error did not actually manifest in the architectural state of the application. We further pursue this approach by checking in software for out-of-range output values within a TRC, if advanced hardware error detection is not available. On average DCT, which suffers the highest protection overhead of all three benchmarks, exhibits a 2.23% overhead with the *T-FDC* setup and 0.46% overhead with the *M-FDC* system configuration.

Jacobi is inherently fault tolerant and this confirmed by the fault injection campaigns. The *No-FDC* campaign indicates than when the application does not fail due to a critical error it will always produce acceptable results, potentially at the expense of additional iterations, thus at the expense of performance. The experiments indicate that *No-FDC* Jacobi did not converge close to the solution in only 0.8% of the experiments that completed, however the execution failed in 48% of the experiments. *T-FDC* and *M-FDC* further improve this number, by only experiencing *Program Failures* in 0.6% and 0.3% of the experiments respectively. Figure 1b also shows that the overall output quality is exceptional with 51% and 99.7% of the faults leading to correct results, whereas 48% and 0% lead to bitwise exact solutions for *T-FDC* and *M-FDC* respectively. When both bitwise exact and correct results are considered, the protected versions of Jacobi deliver an acceptable solution in more than 99% of the experiments.

K-Means proves to be the most fault tolerant benchmark when its execution is not abruptly terminated by a catastrophic fault. It terminates in 39% of the unprotected experiments and produces bitwise exact output in 99.6% of these cases. Only 0.4% of non failed executions result to corrupted results. Its error tolerance is further improved when the proposed protection mechanisms are used. Crashes are almost eliminated, with their probability dropping down to 0.6% in both cases. Once again, *T-FDC* appears to produce better results compared with *M-FDC* but this is only due to the pessimistic way we deal with detected faults. We perform error recovery when the runtime system reports that an error has been detected by the hardware and OS. Consequently, *T-FDC* manages to produce bitwise exact results in 98.87% of the experiments, correct results with a probability of 0.45% and only performs poorly in the form of corrupted results in 0.7% of the experiments. The respective numbers for *M-FDC* are 97.9%, 3% and 0.08%.

6. Related Work

Rinard, in one of the chronologically earlier efforts on task-based error-tolerant computing, proposes a software mecha-

nism that allows the programmer to specify tasks of interest and then creates a profile-driven probabilistic fault model for each task [10] by injecting faults at task execution and observing the resulting output distortion and output failure rates.

EnerJ proposes approximate programming by explicitly tagging data that may be subject to approximate computation in return for increased performance or fault tolerance [11].

Relax is an architectural framework that lets programmers annotate regions of code for which hardware errors can occur and, at the same time, recovery mechanisms can be turned off [4]. Green is a framework that allows the programmer to write several versions of a single function: a precise one and several of varying levels of precision [1]. A runtime system then monitors application QoS online and dynamically adapts to provide a target QoS value.

The concept of Task Level Vulnerability (TLV) captures dynamic circuit-level variability for each OpenMP task running on a specific processing core [9]. TLV metadata 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.

Finally, hardware support for error-tolerant and approximate computing spans designs with small hardware overhead to fundamentally novel architectures. Razor is a processor design which is based on dynamic detection and correction of timing failures of the critical paths due to below-nominal supply voltage [6]. Some additional ideas on hardware support for approximate computation are quality programmable vector processors using ISA extensions [12] and ISA extensions with approximate semantics in general purpose CPUs [7].

7. Conclusions

In this paper we introduced a task-based programming model that allows development of error resilient programs in a disciplined manner. It is based on exploiting programmer wisdom on the significance of tasks for the quality of the end result. Moreover, it allows the programmer to specify result check functions at the granularity of both tasks and groups of tasks. These check functions can also potentially correct (by approximation) the results produced by erroneous task executions. The programming model and the runtime system serving as its back-end also exploit traditional error detection mechanisms offered by hardware and the operating system (such as traps), as well as emerging, low-cost and efficient error detection and error rate monitoring mechanisms, such as razor flip-flops.

We evaluate our approach using three applications: DCT, K-means clustering and a Jacobi iterative solver. The experimental evaluation indicates that our mechanisms significantly improve the error resilience of applications, reducing crashes from 35% to 0.43% and increasing the percentage of executions that resulted to acceptable results from 56% to 91% on average. Despite the fact that the exploitation of additional error detection and monitoring support at the hardware level seems to improve the resilience of applications, we find that even the traditional mechanisms offered by hardware and the OS can yield remarkable improvements in error tolerance, when combined with programmer wisdom.

An interesting way to exploit error resilience offered by system- and application-software would be towards achieving minimal energy computation, on systems working outside their normal voltage / frequency envelope in order to aggressively reduce power consumption. We intend to evaluate the interesting trade-offs that arise under this setting. We also plan to evaluate the effectiveness and cost, in terms of power and performance, of selective protection mechanisms at the hardware level, such as ISA extensions offering reliable instructions on unreliable cores, and selective clock skewing and instructions replay to avoid timing errors.

Acknowledgments

This work has been supported by the "Aristeia II" action (Project "Centaurus") of the operational program Education and Lifelong Learning and is co-funded by the European Social Fund and Greek national resources.

This work has been supported by the EC within the 7th Framework Program under the FET-Open grant agreement SCoRPiO, grant number 323872.

References

[1] Woongki Baek and Trishul M. Chilimbi. Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[2] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[3] Keith A. Bowman, James W. Tschanz, Shih-Lien Lu, Paolo A. Aseron, Muhammad M. Khellah, Arijit Raychowdhury, Bibiche M. Geuskens, Carlos Tokunaga, Chris Wilkerson, Tanay Karnik, and Vivek K. De. A 45 nm resilient microprocessor core for dynamic variation tolerance. *J. Solid-State Circuits*, 46(1):194–208, 2011.

[4] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 497–508, New York, NY, USA, 2010. ACM.

[5] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: A Proposal for Programming Heterogeneous Multi-core Architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[6] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 7–, Washington, DC, USA, 2003. IEEE Computer Society.

[7] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture Support for Disciplined Approximate Programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 301–312, New York, NY, USA, 2012. ACM.

[8] Régis Leveugle, A Calvez, Paolo Maistri, and Pierre Vanhauwaert. Statistical fault injection: quantified error and confidence. pages 502–506. IEEE, 2009.

[9] Abbas Rahimi, Andrea Marongiu, Paolo Burgio, Rajesh K. Gupta, and Luca Benini. Variation-tolerant OpenMP Tasking on Tightly-coupled Processor Clusters. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 541–546, San Jose, CA, USA, 2013. EDA Consortium.

[10] Martin Rinard. Probabilistic Accuracy Bounds for Fault-tolerant Computations That Discard Tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 324–334, New York, NY, USA, 2006. ACM.

[11] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 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*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.

[12] Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Qouality Programmable Vector Processors for Approximate Computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 1–12, New York, NY, USA, 2013. ACM.