Task Scheduling Strategies to Mitigate Hardware Variability in Embedded Shared Memory Clusters

Abbas Rahimi[‡], Daniele Cesarini[†], Andrea Marongiu[†§], Rajesh K. Gupta[‡], and Luca Benini[†§]

[‡]CSE, UC San Diego, La Jolla, CA 92093, USA
[†]DEI, University of Bologna, 40136 Bologna, Italy
[§]IIS, Swiss Federal Institute of Technology, 8092 Zurich, Switzerland
{abbas, gupta}@cs.ucsd.edu, daniele.cesarini@studio.unibo.it, {a.marongiu, luca.benini}@iis.ee.ethz.ch

Abstract—**Manufacturing and environmental variations cause timing errors that are typically avoided by conservative design guardbands or corrected by circuit level error detection and correction. These measures incur energy and performance penalties. This paper considers methods to reduce this cost by expanding the scope of variability mitigation through the software stack. In particular, we propose workload deployment methods that reduce the likelihood of timing errors in shared memory clusters of processor cores. This and other methods are incorporated in a runtime layer in the OpenMP framework that enables parsimonious countermeasures against timing errors induced by hardware variability. The runtime system "introspectively" monitors the costs of tasks execution on various cores and transparently associates *descriptive* metadata with the tasks. By utilizing the characterized metadata, we propose several policies that enhance the cluster choices for scheduling tasks to cores according to measured hardware variability and system workload. We devise efficient task scheduling strategies for simultaneous management of variability and workload by exploiting centralized and distributed approaches to workload distribution. Both schedulers surpass current state-of-the-art approaches; the distributed (or the centralized) achieves on average 30% (or 17%) energy, and 17% (4%) performance improvement.**

I. INTRODUCTION

Hardware variability arises from different physical sources and includes static and dynamic components. Static process variations manifest themselves in the fabricated chips as die-to-die (D2D) and within-die (WID) variations. D2D and WID variations induce performance and power mismatches between the cores. Dynamic components include the effect of environment in which a core is used. Examples of these dynamic variations include supply voltage ripples and temperature fluctuations. To mitigate the effect of these variations, IC designers commonly use conservative guardbands for the clock speed or the supply voltage, leading to a loss of operational efficiency [1].

The most common effect of the variations is on path delay that causes delay uncertainty. By reducing the guardband, this delay uncertainty manifests itself as an intermittent timing error[1] [2], [3]. Timing errors violate the setup or the hold time constraints of the sequential element connected at the end of a path that may be used by an instruction. Thus, from a software point of view, a timing error could cause erroneous instruction execution with wrong output value being generated or stored.

Circuit level error detection and correction (EDAC) techniques [2], [3], [4] combat the timing errors through various recovery mechanisms. These techniques share a common

[1] 'Error' here refers only to timing error, though there can be other types of errors as well, e.g., radiation-induced soft errors.

drawback: *increased cost of computation in the presence of hardware variability*. In other words, the time-variant errors impose different execution costs across the manufactured parts and overtime. Hence, a nominally homogeneous array of processing cores ultimately behaves as a heterogeneous system because each core may result in different execution time and energy. If these effects are ignored at the software level, parallel programs written for homogeneous multiprocessing fabrics experience performance and energy overheads. Recently researchers have focused on methods to reduce the cost of mitigating variability by exposing its effects to the software stack [5], [6], [7], [8], [9], [10], [11], [12]. Given that the parallel programming itself is a difficult task – especially in resource-constrained embedded systems – programmer attention to variation-induced effects (such as load-balancing due to heterogeneity) is impractical. Instead, we seek methods to systematically address variability effects through run-time via an easy-to-use and widely adopted programming model, where efficient variability-aware workload distribution policies can be implemented.

This paper makes three contributions to improve both *cost* and *scalability* of variability mitigation in a heterogeneous embedded system. **First**, we expose the cost of a task execution from hardware to runtime layer in a task-based programming model based on OpenMP v3.0 as the *de facto* standard for programming on shared memory systems. The runtime layer is enhanced with cost-effective countermeasures against hardware variability using two techniques: 1) *Concurrent* task characterization procedures that introspectively monitor execution of different tasks on every core, and upon the tasks completion associate *descriptive* metadata to capture the cost of task execution; 2) Enhanced scheduling methods that infer task-to-core assignments from the characterized metadata for improving energy and performance. **Second**, we explore several task scheduling strategies that use centralized and distributed dispatching methods to reduce the overall cost of execution. This is accomplished through efficient implementation of scalable tasking queues that enables simultaneous variability and workload managements. **Third**, we demonstrate the effectiveness of the proposed runtime layer, for each strategy, in a shared memory cluster with 16 cores in the presence of hardware variations. Our experimental results show that the proposed tasking strategies significantly reduce the cost of error correction compared to a baseline in which variability-agnostic schedulers [15] blindly rely on an EDAC technique [3] to combat the variations. On average, the

centralized approach saves 17% energy and executes 4% faster compared to the baseline. The distributed tasking achieves on average 30% energy saving and 33% better performance. Both scheduling strategies also surpass other state-of-the-art approaches [11].

In Section II, we survey prior work in this area. Cluster architectural details and tasking supports for variability-aware OpenMP execution are described in Section III and Section IV, respectively. Section V presents our centralized and distributed task scheduling strategies. In Section VI, we explain our methodology and present experimental results followed by conclusions in Section VII.

II. RELATED WORK

Circuit level EDAC techniques [2], [3], [4] continuously monitor path delay variations and raise a warning signal when a timing error occurs; the error is then corrected by a recovery mechanism. For instance, an Intel resilient scalar core [3] places circuit monitors [4] in the critical paths of the pipeline stages. Upon error detection, the core ensures error-free completion of the erroneous instruction using one of these two techniques [3]: 1) the erroneous instruction is replayed at half clock frequency; or 2) at the same clock frequency, the erroneous instruction is replayed multiple times in succession[2]. Although EDAC mechanisms guarantee functional correctness, they impose an inevitable cost for error recovery: $\sim 3 \times N$ cycles latency and energy penalties per error, where N is the number of pipeline stages [3]. The cost of EDAC is further exacerbated in a many-core chip where cores operate at near-threshold voltage [13] to save power. We reduce the cost of error recovery in EDAC by exposing the manifestation of variability to software for decisions over much larger time scales and scope of operations.

Software level approaches strive to expose the variations instead of solely relying upon the error recovery mechanisms that guarantee correct program execution eventually. A recent work makes the observation that some sequence of instructions can have a significant impact on the error rate, and therefore introduces improved code transformations [5]. At the next granularity of execution, various approaches focus on coarse-grained tasks [6], and threads [7]. However, these approaches suffer from some drawbacks: 1) Limited applicability since they are not tied to a standard parallel programming language, or consider only a specific class of programs amenable to approximate computation; 2) Lack of efficient support for scheduling and migration incurs high costs. For instance, a large penalty is imposed in [7] any time a migration is requested due to transfer of the entire of instruction and data memories in one tile to another. In the same vein, the proposed technique in [6] incurs cost of over thousand cycles for any migration, and is well-suited for coarse-grained isolated tasks that can be executed entirely with approximation.

Tasking execution model in OpenMP, enables asynchronous and dynamic creation of fine-grained units of work [14]. It naturally enables balanced and continued activity among many cores. A dynamic triple modular redundancy (TMR) technique for OpenMP tasking is presented in [8]. The programmer needs to manually annotate a reliable task through an extended

[2]Correct execution of the last replayed instruction is guaranteed by equalizing the number of replica instructions to the number of stages [3].
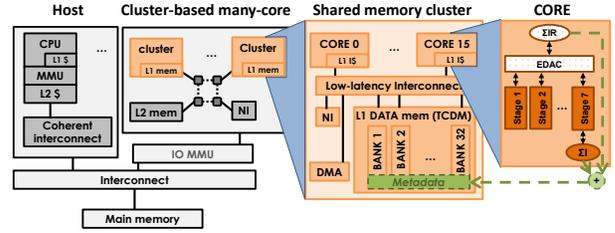

Fig. 1. Target platform high level view.

OpenMP task construct. Therefore, to assure fault tolerance, when a parent task creates a reliable child task into the runtime environment, it will dynamically replicate and submit three redundant children tasks, and finally a majority voting is applied. Similarly, Bolchini et al. [9] propose an application-level TMR scheme for P2012 [16], in which the cluster controller generates three replicas of the main thread. An OS support for TMR multithreading is also proposed to handle the errors during the execution of user-level applications [10]. These techniques impose a large penalty up to $1.8\times$ performance degradation [8].

OpenMP recovery-based approaches for tasking [11] and broader constructs [12] strive to reduce the recovery cost, incurred by EDAC techniques through better scheduling. In a variability-affected cluster of cores, equal workloads do not mean equal executions time (and hence energy). Thus, scheduling flexibility can be used to improve core utilization under variability.

III. VARIATION-TOLERANT SHARED MEMORY CLUSTER

Fig. 1 shows the block diagram of a modern heterogeneous embedded system. A powerful general-purpose processor (the *host*), featuring multi-level coherent-cache memory hierarchy and capable of running full-fledged operating systems, is coupled to a programmable many-core accelerator (PMCA) composed of tens of simple cores, where critical computation kernels of an application can be offloaded to improve overall performance/watt [16], [17], [18].

In this paper our focus in on the PMCA. More specifically, a many-core accelerator that leverages a multi-cluster design to overcome scalability limitations [16], [19], [20]. In a cluster, up to 16 32-bit in-order RISC processors feature private L1 instruction caches, and share a L1 tightly-coupled data memory (TCDM) – a scratchpad memory. The TCDM is configured as a shared multi-banked scratchpad memory that enables concurrent access to different memory locations. The cores in the cluster might display different error rates due to the process and environmental variations [7], [11], [6]. To handle these errors each core uses the EDAC mechanism with the multiple-issue instruction replay as the recovery technique [3]. This enables every core to independently recover from the timing errors at a penalty cost to performance and energy.

IV. VARIABILITY-AWARE OPENMP TASKING

OpenMP v3.0 uses a task-centric model of execution [14]. The new `task` directive is used to dynamically generate units of parallel work that can be executed by every thread in a parallel team. In response to this directive, a running thread prepares a task descriptor consisting of the code to be executed, plus a data environment inherited from the enclosing
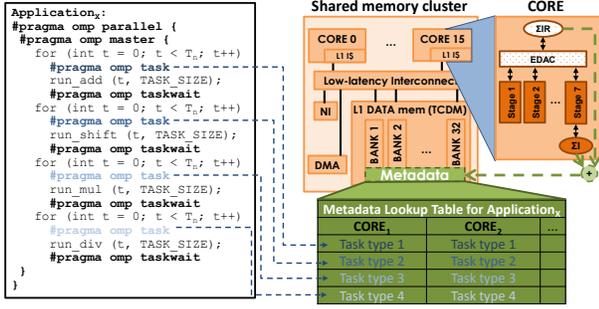
```
Application_x:
#pragma omp parallel {
  #pragma omp master {
    for (int t = 0; t < T_n; t++)
      #pragma omp task
      run_add (t, TASK_SIZE);
      #pragma omp taskwait
    for (int t = 0; t < T_n; t++)
      #pragma omp task
      run_shift (t, TASK_SIZE);
      #pragma omp taskwait
    for (int t = 0; t < T_n; t++)
      #pragma omp task
      run_mul (t, TASK_SIZE);
      #pragma omp taskwait
    for (int t = 0; t < T_n; t++)
      #pragma omp task
      run_div (t, TASK_SIZE);
      #pragma omp taskwait
  }
}
```

Fig. 2. Metadata for OpenMP synthetic workload with 4 task *types* (1200 dynamic tasks with T_n=300).

structured block. All the available (i.e., idle) threads in a parallel team can immediately start executing new tasks, as the specification deals with independent units of work that can be executed in any order within specific synchronization points[3]. OpenMP tasking is considered a convenient programming abstraction for embedded multi- and many-cores [14], [21], [22], [8]. Our OpenMP implementation is designed to operate on 'bare metal', as it is built directly on top of the hardware abstraction layer that provides the lowest-level software services. The OpenMP directives allow the programmer to statically identify several task *types* in a program. Every `task` directive syntactically delimits a unique workload (a task type). There are as many task types in a program as there are `#pragma omp task` directives in its code. For instance, code snippet in Fig. 2 has four task types.

A. Concurrent Task Characterization

To expose the effects of variability to the software, we need to characterize the cost of a task execution in terms of performance and energy. Task execution cost (TEC) is a function of task type and hardware variability of the underlying core. We define TEC as metadata to estimate this cost for the timing errors. TEC captures the variability-induced heterogeneity in the execution time of each task type (i) per each core (j); it is defined as follows:

$$TEC(task_i, core_j) = \#I(task_i, core_j) + \#RI(task_i, core_j) \quad (1)$$

where `#I` is the number of error-free executed instructions and `#RI` is the number of replayed instructions during execution of $task_j$ on $core_i$. These statistics are reported by the available counters. For instance, a resilient LEON-3 [3] supports a counter to account the replica instructions and once the counter passes a threshold, the core adaptively changes the clock frequency. Intuitively, if all the instructions run without any error, TEC is equal to `#I` as the total error-free dynamic instruction count. In case of errors, TEC also accounts for the additional replica instructions which incur higher performance and energy penalties. It then dynamically characterizes both vulnerability and execution time of the executed task[4].

Whenever a task is scheduled on a core for execution, the TEC characterization procedure is executed on the core to compute Equation 1. The characterization procedure reads

the required statistics from the counter, before and after task execution. Upon a task completion, the procedure associates the computed TEC metadata to the task type executed on the selected core. This results in a two-dimensional lookup table (LUT) across the task types and the number of cores. The LUT is physically distributed across the 32 banks of the TCDM, thus it can be quickly accessed from multiple threads at the same time. The right hand side of Fig. 2 illustrates the architectural components involved in TEC characterization. The host processor launches one application on the cores, and every application typically has a few tens of task types (K). Since TEC is a 32-bit word, the LUT has a very small footprint of $K \times 4 \times C$ Bytes per application, where C is the number of the cores in the cluster.

The online characterization procedure is distributed among all the cores in the cluster, thus enabling fully concurrent task-level monitoring and metadata characterization. Note that in principle it would be strictly necessary to characterize a couple $<task_{type}, core_{id}>$ only once. We rather keep the characterization active at every scheduling event and apply a history-based weighted average calculation between the new characterized TEC value and the previously TEC value stored in the LUT. This continued calibration better reflects the recent effects of the dynamic variations on the cores. Based on TEC metadata, a task scheduler can optimize the cluster efficiency by matching the variability-affected cores characteristics to tasks.

V. OPENMP TASK SCHEDULING STRATEGIES

The task scheduler in OpenMP is typically implemented using a centralized queue that collects the task descriptors [21], [22], [8], [11]. The central queue design reduces the overhead for task management which is an important design choice for the resource-constrained systems. This works well for homogeneous systems, but places inherent limitations on applying efficient scheduling policies in the presence of variability-induced heterogeneity across the computational resources. As an example, a notion of task-level vulnerability (TLV) for OpenMP tasking is defined in [11]. OpenMP runtime scheduler is modified to prevent task-to-core assignments for highly vulnerable tasks. However, the central queue imposes limitations on the task scheduling decisions due to a rigid task insertion interface. More specifically, when a `task` directive is encountered, the thread that creates the descriptor cannot utilize the characterized information since there is only a central queue in the system. This limits deriving an efficient scheduling decision. Only upon a task extraction from the queue, a core can utilize this information to decide whether to proceed to the execution of this task or leave it there for another suitable core. A de-centralized tasking queue eliminates this limitations and provides capabilities to enhance scheduling choices. In the following, we describe the design of OpenMP tasking framework based on a distributed queue system.

A. Distributed Task Queues

We propose distributed task queues consisting of private task queue for each core implemented as a standard linked list. Every thread can access a queue using two basic operations:

---

[3]The latest specifications v4.0 has introduced the notion of dependent tasks, our approach can be naturally extended to cover these as well.

[4]Note that the characterization is very accurate, as the OpenMP support for the targeted accelerator runs on top of bare metal (no OS) and only one application at a time is allowed. This removes any source of interference.

insert and extract, which are translated into lock-protected[5] operations on a queue descriptor. The queue descriptors are stored in TCDM for minimal access time. These descriptors are statically instantiated during the initialization of the runtime to avoid the timing overheads for dynamic memory management. Upon each insert operation, a counter for the number of tasks in the queue descriptor is atomically increased. The insertion of a task in a queue immediately wakes up the associated core for the task execution, while the cores with an empty queue are set to a low-power idle mode. The operating mode of every core (*executing*, or *sleeping*) is annotated to its queue descriptor. The thread that inserts the task in a remote queue is responsible for inspecting the operation mode and waking up the destination core to resume execution of the newly inserted task. In addition, the queue descriptor holds synchronization locks used for the `taskwait` directive.

Extracting a task from a queue updates the queue descriptor as well. To do so we use lock-protected operations, since we allow all the threads to extract work from any queue. This enables work stealing policies on top of a variability-aware scheduling. The tasks are extracted only from the head of the queue, while the task insertion can be done at both head or tail of the queue.

As a baseline, we implement a simple round-robin scheduler **(RRS)** [15] which is variation-agnostic. This policy aims at balancing the number of tasks assigned among all the cores, and introduces a minimal runtime overhead due to a lightweight implementation[6]. To account for tasks with different durations, we enhance the scheduling policies with a unified task stealing algorithm. The task stealing algorithm searches the remote queues in a round-robin fashion for finding tasks to steal. The task stealing occurs from the head of the remote queue, and then the task is inserted in the target queue. In the following, we describe two more advanced scheduling policies for task-to-core assignment using the distributed tasking queues.

B. Centralized Variability- and Load-aware Scheduler (CVLS)

The first policy is called centralized variability- and load-aware scheduler **(CVLS)**. This scheduler leverages the characterized TEC to allocate tasks to cores so as to minimize the number of instruction replays while maintaining the overall load balance across all the cores. The main goals of CVLS are: 1) matching assignment of the tasks to the variability-affected cores in a system where task failure has critical consequences; 2) avoiding unbalanced situations by extending its awareness of the load on each queue. CVLS may not assign a task to the best core, but to the second best core in such a way to balance the overall workload. Each $queue_j$ descriptor is enhanced with a load status register ($load_{Qj}$) that estimates the overall weight – in terms of dynamic instruction count – of all the tasks available in that queue. This is a better metric for workload-awareness than just the total task count, because different task types present in the queue may have various computational weight. For a given $task_i$, CVLS assigns a $core_j$ where `TEC`($task_i$, $core_j$)`+load`$_{Qj}$ is minimum across the cluster. CVLS is a centralized scheduler, executed only by one master thread.

[5] Based on test-and-set HW primitives available in the platform.
[6] RRS only increments a non-atomic private counter for each core.
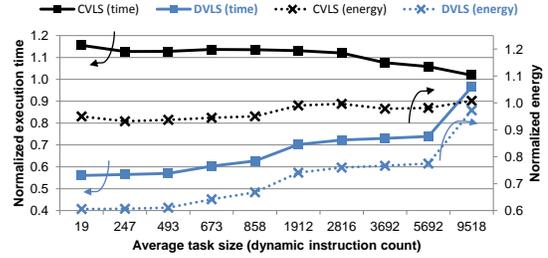

Fig. 3. Execution time and energy of CVLS and DVLS normalized to RRS for synthetic workload with a wide range of task sizes.

At system startup, when there is no information about TEC, CVLS operates in round-robin mode. Since OpenMP tasking assumes completely independent tasks, it is allowed to execute the tasks in any order. We use this property to insert the tasks for which TEC is not available yet, at the head of the queue. This task inserting at the head of the queue is useful to prioritize the execution of the non-characterized tasks; hence, we enable *out-of-order* task characterization. It gives higher priority to the non-characterized task types, therefore speeds up the characterization of TEC.

1) Synthetic Workload: To evaluate the effectiveness, we use a synthetic workload consisting of four consecutive loops, each iteration enclosed within a `task` directive as illustrated in Fig. 2. After each loop there is a `taskwait` synchronization point. Since there are four `task` directives, the compiler statically identifies four task types. Every loop generates a fixed number (T_n) of tasks. Each task is also composed of a loop with a parametrizable number (*TASK_SIZE*) of iterations that controls the size of task in terms of dynamic instruction count. Within this loop, each task type iteratively executes a specific arithmetic operation.

Fig. 3 shows the execution time and energy of CVLS normalized to RRS for the synthetic workload. The execution time and energy is obtained from a variability-affected cluster – more details in Section VI-B. The x-axis shows the average dynamic instruction count for all the executed tasks. As shown, CVLS has 6% lower energy consumption and 15% slower execution time compared to RRS when executing tiny tasks with an average dynamic instruction count of 19. This performance overhead is because of CVLS execution by the single master thread. Once the master thread encounters a new `task` directive, it consumes more cycles to compute CVLS and find a suitable core for the task assignment, compared to a simple counter increment in RRS. Since the size of the tasks is small, the cost of variation-tolerant scheduling is much higher than the benefit of executing the tasks in the favored cores where fewer errors will occur. CVLS approaches RRS as the size of the task increases. Finally, when executing large tasks, with an average ∼9,500 dynamic instruction count, the overhead of CVLS execution time diminishes to 1%.

C. Distributed Variability- and Load-aware Scheduler (DVLS)

As shown in Fig. 3, the centralized execution of the variability- and load-aware scheduler, or CVLS, by a master thread is a bottleneck especially so for the *tiny* tasks. The master thread slows down upon encountering a `task` directive to insert the task on the queue of an appropriate core. Moreover, the inserted task will be executed immediately by the
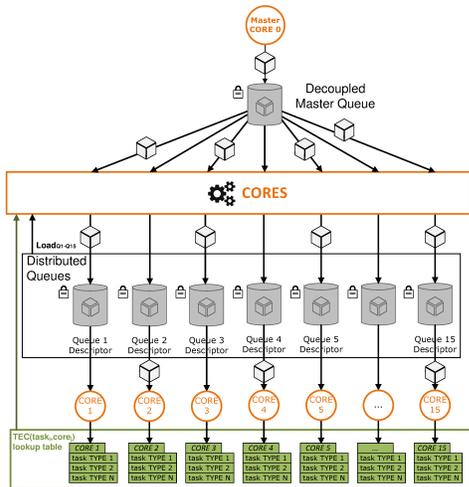
Fig. 4. Distributed Variability- and Load-aware Scheduler (DVLS).

TABLE I
ARCHITECTURAL PARAMETERS OF CLUSTER.

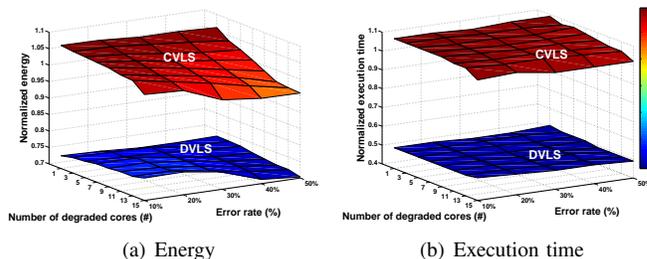| ARM v6 core	16	TCDM banks	16
I\$ size	16KB per core	TCDM latency	2 cycles
I\$ line	4 words	TCDM size	256KB
Latency hit	1 cycle	L3 latency	$\geqslant 60$ cycles
Latency miss	$\geqslant 59$ cycles	L3 size	256MB

(a) Energy

(b) Execution time

Fig. 5. Execution time and energy for CVLS and DVLS, normalized to RRS. The 16 cores cluster with different degradation scenarios: increasing number of degraded cores (1 to 15), and increasing error rate (10%–50%).

suitable core, while the master thread is still computing CVLS for the next task, therefore avoiding full parallel utilization of all the cores. To account for this performance effects, CVLS is enhanced with a distributed execution among all the cores which is called distributed variability- and load-aware scheduler (**DVLS**). As shown in Fig. 4, DVLS shares the computational load of CVLS from one master thread to multiple slave threads, therefore it can boost the performance. The master thread simply pushes the tasks in a *decoupled* master queue for the task deployment without performing any complex computation. Then each slave thread will pick up a task from the master queue and execute the scheduler code of CVLS independently to decide where the task should be assigned across the distributed queues. This "decoupling" between the master queue and the distributed queues is highly beneficial as the master thread proceed fast to push the tasks, while the rest of threads cooperatively will schedule the tasks among themselves, hence maintaining the full utilization. Once the master thread finishes pushing the tasks it itself can join to this cooperating team.

Fig. 3 also shows the normalized execution time and energy of DVLS compared to RSS. DVLS reaches to a simultaneous energy saving of 40% and 44% faster execution for the tiny tasks with the average instruction count of 19. DVLS maintains the gain across a wide range of the task sizes; even for the largest task size, it spends less time (4%) and energy (3%) than of RRS. This is accomplished through simultaneous variability management and load balancing. DVLS assigns proper task types to the degraded cores while maintaining the load balancing across all the distributed queues, thanks to utilization of TEC and the continuous queue monitoring. To quantify the overheads for TEC characterization and computation of DVLS, we further assess its effectiveness in a cluster without any variation, i.e., ideally a perfect hardware. Compared to RSS, DVLS incurs no performance penalty thanks to its distributed nature that decouples executions across all the cores. Without variability, DVLS also saves 6% energy for the large tasks.

VI. EXPERIMENTAL RESULTS

We demonstrate our approach on a SystemC-based virtual platform [23] modeling the tightly-coupled cluster described in Section III. Table I summarizes the main architectural parameters. To emulate the variations on the virtual platform, we have integrated variations models at the level of individual instructions using the characterization methodology presented in [24]. Integration of the instruction vulnerability models for every core enables online assessment of the presence or absence of the errant instructions at a given amount of variations. We re-characterized the variability models of an in-order LEON-3 [25], an open-source RISC core. First, we synthesized LEON-3 with the 45nm TSMC technology library. The frontend flow with normal V_{TH} cells has been performed using *Synopsys DesignCompiler*, while *Synopsys IC Compiler* has been used for the back-end where the core is optimized for performance. We use *Synopsys PrimeTime VX* to compute energy models and probability of timing errors for individual cores detailed in Section VI-B. Finally, we back annotate these models on the virtual platform [23] where the models are used in an instruction-by-instruction fashion. This enables simulation of erroneous instructions during the program execution based on the available amount of variations on individual cores.

Our OpenMP implementation is based on [21]. To evaluate the effectiveness of the proposed variation-tolerant policies, we conduct two sets of experiments. First, we use the synthetic workload to assess our scheduling policies on a generic cluster with a full range of degradations. Second, we use eight widely adopted computational kernels mainly from the image processing domain on a variability-affected cluster.

A. Generic Degraded Cluster

We consider a generic degraded cluster in which the number of degraded cores and the error rates are varied. In Fig. 5, the x-axis indicates an increasing number of the cores that experience a fixed error rate. The y-axis indicates the timing error rate[7] – i.e., the percentage of instructions that need to be replayed – for the degraded cores. We use the tiny tasks in the synthetic workload; the choice of the tiny tasks is

---

[7]The degraded cores with error rates of >50% can effectively guarantee the error-free execution of the erroneous instructions at half frequency [3] without any synchronization penalty across multiple clock domains.
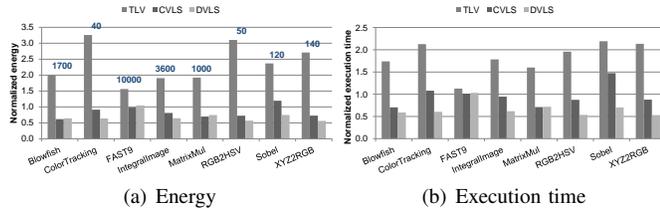
Fig. 6. CVLS, DVLS, and TLV normalized to RRS. Each benchmark has one task *type* and 900 dynamic tasks.

mainly because of this task granularity is representative of real embedded workloads where *light-weight* irregular parallelism is exploited [22], [21].

We compare the energy consumption (Fig. 5(a)) and the total execution time (Fig. 5(b)) for CVLS and DVLS policies with the fast RRS as the normalized baseline. Focusing on the energy, under all 8×5 degradation scenarios DVLS consumes on average 27% lower energy compared to RRS. This is because DVLS reduces significantly the number of replica instructions, across different degradation scenarios, that saves energy. DVLS also reaches an average 49% faster execution. However, in comparison with RRS, CVLS imposes performance cost of 2%–9% slower execution. This is a consequence of the slow centralized scheduler that lengthens the overall execution time of the tiny tasks. CVLS is able to save energy when the cluster faces nine or more 30%–50% degraded cores.

B. Variability-affected Cluster

For this experiment we consider a 16-core cluster that is affected by the process variation. To observe the effect of static process variation on the clock frequency of individual cores within the cluster, we analyze how critical paths of each core are affected due to WID and D2D variations, following a methodology in [11]. As a result, only ten cores out of 16 cores within the cluster can meet the design time target clock frequency, while the remaining six cores face different error rates based on the instruction types and the operating condition.

We compare our policies and the TLV technique [11] with the baseline RRS policy. Fig. 6(a) shows the energy consumption for different benchmarks. The average tasks size for each benchmark is illustrated on top of the bars in Fig. 6(a). DVLS has a distributed focus on allocating tasks among more reliable cores for a given workload type, as it achieves on average 30% (and up to 44%) better energy efficiency than RRS. CVLS also reaches to an average energy saving of 17% and up to 38% compared to RRS. The TLV policy does not consider the overall system workload that led to an imbalanced system which is on average 100% less energy efficient than RRS.

Fig. 6(b) shows the execution time for the benchmarks. DVLS achieves up to 47% (33% on average) faster execution than RRS, thanks to its distributed execution within the cluster that aims to maintain the load balancing. In the other hand, CVLS exhibits slower execution for benchmarks with tiny tasks (e.g., Sobel and ColorTracking) where the centralized dispatcher is a bottleneck and cannot utilize all the cores. Overall, CVLS reaches up to 29% (4% on average) faster execution. The TLV policy, with the limited task insertion interface due to the single queue design, displays on average 83% slower execution than RRS with the distributed queues.

VII. CONCLUSION

Manufacturing and environmental variability makes even homogenous processing fabrics behave as heterogeneous processing elements. The variability-induced timing errors have been addressed as circuit or microarchitectural issues in reliable processor designs with significant power and performance overheads due to limited scope of the timing recovery mechanisms. Instead, our approach consists of an enhanced OpenMP runtime system that uses tasking scheduling strategies to reduce the increased cost. The runtime layer enables introspective task monitoring and utilizes characterized metadata to infer a proper task-to-core assignment while reducing the likelihood of timing errors and overall system workload. We have explored centralized and de-centralized scheduling approaches on distributed tasking queues that highly surpass state-of-the-art approaches in terms of both energy and performance. Our proposed DVLS (CVLS) reduces energy on average 30% (17%) compared to RRS. On average, DVLS achieves 33% better performance than RRS, as opposed to the CVLS' 4%.

REFERENCES

[1] S. Ghosh and K. Roy, "Parameter variation tolerance and error resiliency: New design paradigm for the nanoscale era," *Proceedings of the IEEE*, vol. 98, pp. 1718–1751, Oct 2010.

[2] D. Ernst, et al., "Razor: a low-power pipeline based on circuit-level timing speculation," *IEEE MICRO*, pp. 7–18, Dec 2003.

[3] K. Bowman, et al., "A 45 nm resilient microprocessor core for dynamic variation tolerance," *IEEE JSSC*, vol. 46, pp. 194–208, Jan 2011.

[4] K. Bowman, et al., "Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance," *JSSC*, vol. 44, pp. 49–63, Jan 2009.

[5] G. Hoang, et al., "Exploring circuit timing-aware language and compilation," *ACM ASPLOS*, pp. 345–356, 2011.

[6] H. Cho, et al., "Ersa: Error resilient system architecture for probabilistic applications," *IEEE TCAD*, vol. 31, pp. 546–558, April 2012.

[7] S. Dighe, et al., "Within-die variation-aware dynamic-voltage-frequency-scaling with optimal core allocation and thread hopping for the 80-core teraflops processor," *IEEE JSSC*, vol. 46, pp. 184–193, Jan 2011.

[8] O. Tahan, et al., "Using dynamic task level redundancy for openmp fault tolerance," *Springer-Verlag ARCS*, pp. 25–36, 2012.

[9] C. Bolchini, et al., "An adaptive approach for online fault management in many-core architectures," *IEEE DATE*, pp. 1429–1432, March 2012.

[10] B. Dobel, et al., "Operating system support for redundant multithreading," *ACM EMSOFT*, pp. 83–92, 2012.

[11] A. Rahimi, et al., "Variation-tolerant openmp tasking on tightly-coupled processor clusters," *IEEE DATE*, pp. 541–546, March 2013.

[12] A. Rahimi, et al., "Improving resilience to timing errors by exposing variability effects to software in tightly-coupled processor clusters," *IEEE JETCAS*, vol. 4, pp. 216–229, June 2014.

[13] M. Kakoee, et al., "Variation-tolerant architecture for ultra low power shared-l1 processor clusters," *IEEE TCAS II*, vol. 59, pp. 927–931, Dec 2012.

[14] E. Ayguade, et al., "The design of openmp tasks," *IEEE TPDS*, vol. 20, pp. 404–418, March 2009.

[15] The GNU Project. GOMP - An OpenMP implementation for GCC [Online]. Available: http://gcc.gnu.org/projects/gomp

[16] D. Melpignano, et al., "Platform 2012, a many-core computing accelerator for embedded socs: Performance evaluation of visual analytics applications," *IEEE DAC*, pp. 1137–1142, June 2012.

[17] Texas Instruments Inc., "Multicore DSP+ARM KeyStone II System-on-Chip (SoC)," [Online]. Available: www.ti.com/lit/ds/symlink/66ak2h12.pdf

[18] Adapteva Inc., "Parallela Reference Manual," [Online]. Available: www.parallella.org/wp-content/uploads/2013/01/parallellan gen1n reference.pdf

[19] Kalray S.A., "MPPA MANYCORE," [Online]. Available: http://www.kalray.eu/IMG/pdf/FLYERn MPPAnMANYCORE-3.pdf

[20] Plurality Ltd., "Plurality HyperCore Architecture," White Paper.

[21] P. Burgio, et al., "Enabling fine-grained openmp tasking on tightly-coupled shared memory clusters," *IEEE DATE*, pp. 1504–1509, March 2013.

[22] S. Agathos, et al., "Deploying openmp on an embedded multicore accelerator," *IEEE SAMOS XIII*, pp. 180–187, July 2013.

[23] D. Bortolotti, et al., "Virtualsoc: A full-system simulation environment for massively parallel heterogeneous system-on-chip," *IEEE IPDPS*, pp. 2182–2187, 2013.

[24] A. Rahimi, et al., "Analysis of instruction-level vulnerability to dynamic voltage and temperature variations," *IEEE DATE*, pp. 1102–1105, March 2012.

[25] LEON-3 [Online]. Available: http://www.gaisler.com/cms