Variation-tolerant OpenMP Tasking on Tightly-coupled Processor Clusters

Abbas Rahimi[†], Andrea Marongiu[‡], Paolo Burgio[‡], Rajesh K. Gupta[†], Luca Benini[‡]
[†]Department of Computer Science and Engineering, UC San Diego, La Jolla, CA 92093, USA
[‡]Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna, 40136 Bologna, Italy
{abbas, gupta}@cs.ucsd.edu, {a.marongiu, paolo.burgio, luca.benini}@unibo.it

ABSTRACT

We present a variation-tolerant tasking technique for tightly-coupled shared memory processor clusters that relies upon modeling advance across the hardware/software interface. This is implemented as an extension to the OpenMP 3.0 tasking programming model. Using the notion of *Task-Level Vulnerability* (TLV) proposed here, we capture dynamic variations caused by circuit-level variability as a high-level software knowledge. This is accomplished through a variation-aware hardware/software codesign where: (i) Hardware features variability monitors in conjunction with online per-core characterization of TLV metadata; (ii) Software supports a *Task-level Errant Instruction Management* (TEIM) technique to utilize TLV metadata in the runtime OpenMP task scheduler. This method greatly reduces the number of recovery cycles compared to the baseline scheduler of OpenMP [22], consequently instruction per cycle (IPC) of a 16-core processor cluster is increased up to 1.51× (1.17× on average). We evaluate the effectiveness of our approach with various number of cores (4,8,12,16), and across a wide temperature range(ΔT=90°C).

1. INTRODUCTION

While shrinking CMOS minimum feature sizes and higher transistor density open the way to many-core processor chips [1], they also come with the side effects of increased variability [2]. The emerging billion-transistor dies with multiple parametric variabilities poses serious static and dynamic variability challenges [3]. Static process variations manifest themselves as die-to-die (D2D) and within-die (WID) variations. D2D variations affect all cores on a die equally, whereas WID variations induce different characteristics for each computing core (differ core-to-core characteristics). Other variations that impact cores are dynamic in nature and depend on the environment in which a core is used. Examples of these types of variations include dynamic voltage droop, and on-die hot spots. These factors are expected to be worse in future technologies [4], and are exacerbated in large-area many-core systems where advanced variability management is already employed [3]. Hence a significant timing error and performance loss takes place if variability is not properly addressed.

Resilient circuit techniques, including Error-Detection Sequential (EDS) [5], and Bubble Razor [6], focus on measures to combat variability at the circuit level. A common strategy is to detect timing errors, then use architectural instruction *replay*, time borrowing and/or tuning CMOS knobs to correct errors. Detection circuits typically utilize double sampling using shadow latches. In an ARM Cortex-M3 core if an instruction arrives late in the pipeline, Bubble Razor flags an error that causes a latch to skip its next transparent clock phase, giving it an additional cycle for correct instruction to arrive. Recent 45nm Intel resilient core [7] places EDS within the critical paths of the pipeline stages for detecting dynamic variations. Once a timing error is detected, the core prevents the errant instruction from corrupting the architectural state and initially flushes the pipeline to resolve any complex bypass register issues. To ensure error recovery, the core supports two separate techniques: (I) instruction *replay* at half frequency, and (II) multiple-issue, e.g. *N*, instruction *replay* at the same frequency; the first *N*-1 issues are replica instructions, while the *N*-th issue is a valid instruction. The former technique imposes 28 extra Recovery Cycles Per Error (RCPE), while the latter pays 21 RCPE as the cost of compensation with *N*=8 [7].

At higher levels, where instructions come into focus as the most fine-grained abstraction of the processor's functionality, several efforts have tried to characterize and use variability related information. Rahimi et al. introduce a notion of Instruction-Level Vulnerability (ILV) [8] to expose dynamic variations and its effects to the software stack. Another technique, operating at the level of sequence, is proposed by Gupta et al. [9] to determine sequences of instructions that have a significant impact on timing error rate. Therefore, code transformations have been introduced for improving their timing speculation. Raising further the level of abstraction, Rahimi et al. also define a notion of Procedure-Level Vulnerability (PLV) [10] for guiding a runtime system to mitigate dynamic voltage variations by hopping a procedure (subroutine) from one core to a favor core within a shared-L1 processor clusters. Dighe et al. propose a thread hopping scheme in conjunction with DVFS to mitigate the within-die variation across 80-core [11]. F. Paterna et al. [12] also propose a run time variability-aware workload distribution technique for enhancing predictability and energy efficiency of parallel multiprocessor arrays.

This paper makes three contributions. **First**, we propose a notion of *Task-Level Vulnerability* (TLV) as metadata to characterize dynamic variations in Section 3. In fact, TLV is a vertical abstraction: TLV reflects manifestation of circuit-level hardware variability in specific software context for parallel execution model. **Second**, we devise a variation-aware synergetic hardware/software approach. Our cluster is equipped with the circuit sensors for online measurement of variability and per-core introspective characterization of TLV metadata. Fast access to the TLV metadata for each *type* of task is guaranteed by carefully placing these key data structures in tightly-coupled shared-L1 memory. Section 4 covers these details. The OpenMP runtime scheduler utilizes TLV metadata to support a *Task-level Errant Instruction Management* (TEIM) technique for reducing the cost of recovery, described in Section 5. Both TLV metadata characterization and TEIM operate at the level of task, where tasks are abstractly expressed by the programmer through code annotations. **Third**, we demonstrate the effectiveness of our approach on a variability-affected tightly-coupled processor cluster with accurate ILV models in 45-nm TSMC technology. Our experimental results in Section 6 indicate that the IPC of the cluster is increased to up to 1.51× (on average 1.17×). The technique exhibits a consistent and robust behavior for a wide range of temperature fluctuation (ΔT=90°C) and when the number of cores in the cluster is varied to 4, 8, 12, 16.

2. RELATED WORK

Circuit-level techniques like EDS [5], and Bubble Razor [6] raise a warning signal when a timing error is detected in case of any

978-3-9815370-0-0/DATE13/©2013 EDAA

variations. Then, recovery mechanisms compensate the error while incurring extra RCPE cost. Their cost of recovery is shown to be high in face of frequent timing errors, especially so in aggressive voltage over-scaling and near-threshold computation [13]. Moreover, these per-core detection-correction mechanisms that seek to act for every instance of timing error may be inefficient in the system-level that feature a cluster of tightly-coupled processors. 'Higher level' techniques are needed not only to support individual per-core recovery mechanisms, but also to reduce the cost of RCPE across a cluster and thus improve overall IPC.

Some higher-level approaches have been proposed to mitigate timing errors on individual instructions [8] as well as sequence of instructions [9]. Such fine granularities are expensive to control due to the need of fast hardware support. Moving up to a coarser granularity, techniques have been proposed to address a dynamic series of instructions, in various levels: procedure [10], thread [11], task [14], and workload [12]. The main drawbacks of these techniques are the following. (I) [10][14] do not support online characterization. (II) [14] applies task mapping only during application initialization, and does not support any dynamic scheduling and management after mapping. [10] defines a generic notion of dynamic procedure hopping which does not tie to a standard parallel execution model, and thus imposes intrusive changes through runtime support. (III) [11][12] target coarse-grained many-core systems that incur a high penalty any time that a migration is required; e.g., [11] needs to transfer the entire contents of instruction and data memory in one tile to another over a packet-switched router. [12] also needs a dedicated host processor to dispatch workload and take decisions about the allocation.

Programmers need a simple way to identify independent units of work and not concern themselves with scheduling these work units. A tasking model, such as one in OpenMP 3.0 [15], makes it possible to do so by expressing irregular and unstructured parallelism in a simple way. A dynamic Triple Modular Redundancy (TMR) technique for OpenMP tasking is presented in [16]. Programmer needs to manually define a reliable task through extended OpenMP **`task`** construct with a *reliable* clause ("#pragma omp task reliable"). 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, [17] proposes a loosely coupled application-level TMR schema for P2012 [3], in which the cluster controller generates three replicas of the main thread. However, these technique target a generic fault model, and impose a large penalty due TMR, e.g., up to 1.8× slow down [16].

3. TASK-LEVEL VULNERABILITY (TLV) AND OPENMP TASKS

The OpenMP specification version 3.0 introduces a task-centric model of execution. The new **`task`** directive is used to dynamically generate units of parallel work that can be executed by every thread in a parallel team. When an executing thread encounters the **`task`** construct, it prepares a task descriptor consisting of the code to be executed, plus a data environment inherited from the enclosing structured block. The tasking programming model is considered as a convenient abstraction for application development in shared memory multi-cores [18]. Thus we integrate TLV metadata as an extension to the OpenMP tasks. A **`task`** directive outlines an execution unit which runs a sequence of instructions. The OpenMP directives allow the programmer to statically identify several task *types* in the program. Every **`task`** directive syntactically delimits a unique stream of instructions. While at runtime the same stream may be dynamically instantiated several times (e.g., a **`task`** directive nested within a loop), from the point of view of our characterization it uniquely identifies a single task *type*. As a direct consequence, there are as many *types* of tasks in a program as there are **`task`** directives in its code.

In a variability-affected core, ILV [8] is not uniform across the instruction set. In fact, ILV data partitions instructions into three classes: (i) logical/arithmetic class, (ii) memory class, and (iii) multiply/divide class. ILV indicates that the classes of instructions have different levels of vulnerability to variations depending on the way they exercise the non-uniform critical paths across the various pipeline stages. For instance, in an in-order RISC core the *execution* and *memory* stages are highly vulnerable to dynamic variations, and the memory class has a higher vulnerability in comparison to the logical/arithmetic class [8]. We note that complex high-performance cores such as IBM POWER6 also confirm that vulnerability is not uniform across the instructions set [19]. We extend the notion of ILV to a more coarse-grained task-level metric, TLV. The vulnerability of a task *type* varies based on the class of instructions that it will execute. TLV is also a per-core metric since the amount of variation affecting different classes of instructions changes from one core to another. Therefore, each dynamic task (dynamic instance of a task *type*), can potentially face a different density of the errant instructions imposed by both software context and hardware variations.

While the identification of task *types* can be done statically (i.e., at compile time), their characterization has to be done online due to two reasons. First, dynamic instances of the same task *type* may exercise the processor pipeline in a non-identical manner due to data-dependent control flow that executes different classes of instructions. Second, the characterization must reflect the variability-affected characteristic of every core (not known a priori) on every task *type*. Therefore, we define the notion of TLV as a metric to characterize vulnerability of each task *type* per each core, in the following:

$$\mathrm{TLV}_{(i,j)} = \frac{\Sigma\,\mathrm{EI}}{\mathrm{L}}\ ,\ \forall \mathrm{core}_i\ ,\ \forall \mathrm{task}_j \qquad (1)$$

where $\sum$EI is the number of *Errant Instructions* during execution of task_j on core_i that are reported by the circuit sensors and need to be *replayed* for correct execution; L is the total number of executed instructions. Intuitively, if all the instructions run without any timing error, TLV is *0*; on the other hand, TLV is *1* if every instruction causes at least one timing error. The lower TLV, the lower the number of errant instructions, the lower the cost of recovery, and thus the higher the IPC.

3.1 TLV Across Various Types of Tasks

We examine intra-corner TLV for a core_i that runs a synthetic benchmark consisting of six distinct types of tasks. Each task is composed of a loop with a parametrizable number of iterations (10 to 100). Within the loop, each task executes a different class of instructions illustrated in Figure 1. The core_i works in the typical operating condition, i.e., room temperature of 25°C and voltage supply of 1.1V. This operating corner is fixed, thus there will be no environmental variation during task execution. The TLV characterization for each task *type* is shown in Figure 1. As shown, TLV of each *type* of tasks is different even within the fixed operating condition in the core_i. For instance, TLV of task $type_1$ ($\mathrm{TLV}_{(i,1)}$) is 9× higher than the $\mathrm{TLV}_{(i,6)}$ indicating a considerable variation across the *type* of tasks. Furthermore, within same *type* of task, TLV can also be affected by the data-dependent control-flow that can cause execution of different classes of instructions. For example, task $type_1$ consists of mix classes of instruc-

tions that will be executed conditionally upon the current iteration of the loop. As a result, $TLV_{(i,1)}$ is changed from 0.033 to 0.041 when the number of iterations changes from 10 to 100. This TLV variation across task *types* indicates the need of online monitoring for every task *types*.

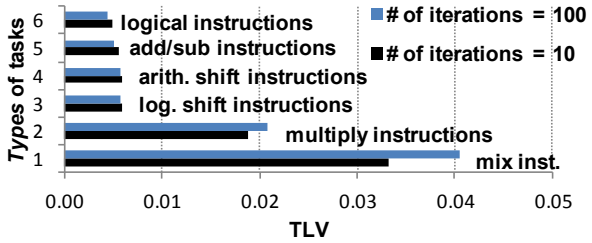

Figure 1. Intra-corner TLV for six distinct task *types*.

3.2 Inter-corner TLV to Dynamic Variations

We examine the TLV across different operating conditions. Specifically, we analyze the effects of a full range of dynamic variations, a temperature range of 20°C−140°C, and a voltage range of 0.88V−1.1V. As shown in Figure 2, the average TLV of the six types of tasks is an increasing function of temperature. With a fixed voltage of 1.1V, by increasing the temperature the delay of critical paths is increased, thus more instructions will face the timing error which causes TLV to increase up to 0.096 at 140°C. In contrast, decreasing the voltage from the nominal point of 1.1V increases TLV. In lower voltages, the delay of critical paths highly increases, thus imposing a high rate of the errant instructions. For example, a dynamic voltage variation of 0.2V (ΔV=1.1V-0.9V) causes a TLV of 0.507, which implies that more than half of the total executed instructions within tasks failed due to the timing errors of the voltage variation. Figure 1 illustrates TLV values across different *types* of task in the typical operating corner and Figure 2 highlights that TLV of tasks is further magnified across various corners of operating conditions, thus TLV should be characterized for every different operating condition.

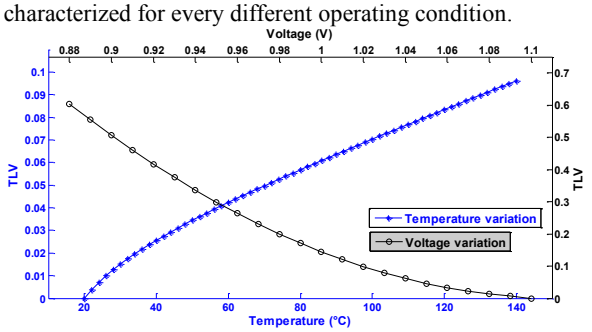

Figure 2. TLV to dynamic voltage and temperature variations.

4. VARIATION-TOLERANT CLUSTER

In this section, we describe the architectural details of the proposed variation-tolerant processing cluster shown in Figure 3. The cluster is inspired by tightly-coupled clusters in STMicroelectronics P2012 [3] as the essential component of a many-core fabric. In our implementation, each cluster consists of sixteen 32-bit in-order RISC cores, a L1 software-managed Tightly Coupled Data Memory (TCDM) and a low-latency logarithmic interconnection [20]. The TCDM is configured as a shared, multi-ported, multi-banked scratchpad memory that is directly connected to the logarithmic interconnection. The number of TCDM ports is equal to the number of banks to enable concurrent access to different memory locations. Note that one bank of the TCDM provides test-and-set read operations, which we use to implement basic synchronization primitives (e.g., locks). The logarithmic interconnection is composed of mesh-of-trees networks to support single cycle communication between processors and memories. When a read/write request is brought to the memory interface, the data is available on the negative edge of the same clock cycle, leading to two clock cycles latency for a conflict-free TCDM access.

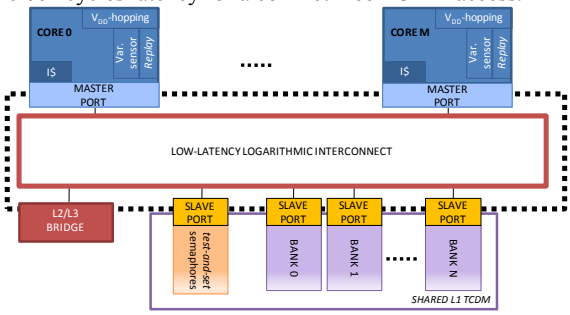

Figure 3. Variation-tolerant tightly-coupled processor cluster.

The cluster is equipped with two core-level resiliency techniques. First, each core relies on the circuit sensors to detect any timing error due to dynamic delay variation. To recover the errant instruction without changing the clock frequency, the core employs the multiple-issue instruction *replay* mechanism [7] in its recovery unit; seven replica instructions followed by a valid instruction, thus RCPE=21. Second, the cluster supports a per-core V_{DD}-hopping technique [21] for tuning the voltage of each core individually to compensate the impact of static process variation. The core-level V_{DD}-hopping is employed in a variability-affected tightly-coupled cluster [10]. The V_{DD}-hopping improves the clock speed of the slow cores, thus enabling all components of the variability-affected cluster to work at same frequency (with memories at a 180° phase shift). This technique avoids the inter-core synchronization that would significantly increase L1 TCDM latency.

To observe the effect of static process variation on the frequency of individual cores within the cluster, [10] analyzed how critical paths of each core are affected due to WID and D2D process parameters variation. The maximum frequency distribution of every core is shown in Figure 4 (left), in which each core's maximum frequency varies significantly due to the process variation. As a result, six cores (C_0, C_2, C_4, C_{10}, C_{13}, C_{14}) cannot meet the *design time* target frequency of 850 MHz. To compensate this core-to-core frequency variation, the V_{DD}-hopping technique measures the delay variation of each core and then applies the appropriate voltage accordingly (higher voltage for slow cores). The technique utilizes three discrete voltage modes (V_{DD}-high, V_{DD}-medium, V_{DD}-low), consequently, the cluster mitigates the core-to-core variations, and all cores can work with the *design time* target frequency. More details of V_{DD}-hopping and process variation analysis on the cluster is provided in [10].

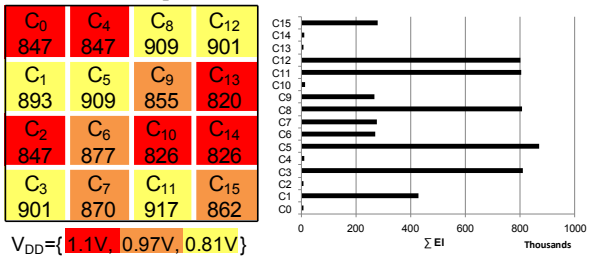

Figure 4. V_{DD}-Hopping in the variability-affected cluster (left); Number of errant instructions during synthetic bench (right).

In V_{DD}-hopping, cores in various voltage islands display different characteristics. Figure 4 (right) shows that the number of errant instructions significantly varies across cores cooperating together within a single cluster for executing available tasks. For instance,

C_0 faces 7.3K errant instructions, whereas C_1 has more than 428K errant instructions during the synthetic benchmark execution. As shown in Figure 2, a core with lower voltage has higher TLV (higher ∑EI), and will impose higher extra cycles to correct those errant instructions. Thus a task scheduler that is aware of the individual core characteristics and tasks is better able to match them to reduce the overall penalty for correcting the errant instructions.

4.1 Decentralized TLV Characterization

To reduce the cost of recovery, TLV metadata guides the runtime scheduler. Since TLV depends on the *type* of task, we consider individual TLV characterization for every task *type*. As we already explained, TLV metadata is defined for a given core because different cores can display different variability characteristics. Therefore, each core needs to be characterized during online execution of a task. This results in TLV as a two-dimensional lookup table across tasks and cores. This lookup table is physically distributed across all the 32 banks of TCDM, thus it can be written/read with a two-cycle latency in case of conflict-free communication. Since TLV metadata is 32-bit, and every application will have a bounded number of N supported task *types*[1], the cluster needs to allocate a maximum of $N \times 4 \times C$ Bytes for the lookup table, where C is the number of cores in the cluster.

```
void handle_tasks () {
   while (HAVE_TASKS) {  // Task scheduling loop
     task_desc_t *t = EXTRACT_TASK ();
     if (t) {
      float old_mdata =
        tlv_read_task_metadata (core_id);
      /* Reset counter for this core */
      tlv_reset_task_metadata (core_id);
      /* EXEC! */
      t->task_fn (t->task_data);
      /* We executed. Fetch TLV ...*/
      float mdata = tlv_read_task_metadata (core_id);
  // Update MDATA by averaging the new and old values
       tlv_table_write (t->task_type_id,
                        core_id,(mdata+old_mdata)/2);
     }   }   }
```

Figure 5. Pseudo-code to perform TLV characterization.

The online characterization mechanism is distributed among all the cores in the cluster, thus enables fully parallel task-level monitoring and characterization. The cluster employs the circuit sensors and the error recovery unit of every core to perform characterization. To quantify TLV, the core collects the statistics of ∑EI and L for Equation (1) through available counters. For instance, [7] includes a counter for the errant instructions (∑EI) to change the frequency when the number of errors is above a certain threshold. Two function calls for profiling TLV of current task are inserted in the runtime library, right before and after actual execution (see Figure 5), and then the lookup table is updated with the new value. The former (`tlv_reset_task_metadata`) restarts the counters, and the latter two (`tlv_read_task_metadata` and `tlv_table_write`) transfers the characterized TLV metadata at the end of task execution to the lookup table for future inspection. Thus, the cluster pays 15−20 cycles latency per each task characterization thanks to the shared TCDM and fast interconnection.

5. TLV-AWARE OPENMP TASKING

5.1 Centralized Variation-tolerant Scheduler

The lookup table for the characterized TLV metadata acts as a software-accessible monitor that provides information to the runtime systems to guide task scheduling. We propose a reactive variation-tolerant scheduler that we call *Task-level Errant Instruction Management* (TEIM). The OpenMP implementation that we consider [22] leverages a centralized task queue, where all the threads involved in parallel computation actively push and pop job descriptors. Typically, to avoid redundant computation, only a single thread from a parallel team executes the code within the **`task`** directive (pushing its task descriptor in the queue). The rest of the threads remain idle in wait for work to do. Whenever a thread is idle it tries to extract a task from the queue, thus tasks are scheduled to threads on a *first-come, first-served* basis.

Our TEIM technique enhances the above baseline scheduler with additional conditional checks. It utilizes TLV metadata to determine whether the querying thread is well suited to run the task on the head of the queue. The overall goal is a guided scheduling of tasks to cores, which reduces the number of errant instructions so that the *replay* logic is exercised less frequently. In other words, the scheduler tries to match the variability-affected characteristics of the cores with the level of vulnerability of tasks, thus reducing unnecessary recovery cycles. At each scheduling point, an idle $core_i$ runs the scheduler. Then, the scheduler checks two conditions to decide whether the core should execute a $task_j$ in the head of queue, or should skip it and lets other *favoured* cores execute it later. First, the scheduler reads the TLV metadata entry corresponding to the combination of $task_j$ and $core_i$. If $TLV_{(i,j)}$ is greater than a predefined target threshold (TLV_THR), there is no match between the characteristics of $core_i$ and $task_j$ (execution of $task_j$ on $core_i$ may cause at least TLV_THR×L errant instructions, see Equation (1)), so the scheduling attempt fails. $Task_j$ remains in the queue, ready to be reconsidered for scheduling at the next attempt (thus, the rest of cores can potentially execute it). Second, to avoid starvation, each core can skip tasks for a maximum number of ESCAPE_THR times. Beyond this threshold the core has to execute at least one task, independent of its TLV value. The TEIM algorithm is shown in the following:

```
TLV(i,j) = read_TLV_LUT (corei, taskj);
if (TLV(i,j)> TLV_THR && corei_escape_cnt <ESCAPE_THR){
      corei_escape_cnt ++;
      escape (taskj);}
else { assign_to_corei (taskj);
      corei_escape_cnt = 0;}
```

Figure 6. TEIM algorithm in the variation-tolerant scheduler.

Thus far, we assumed that TLV characterization information is available for the scheduler to take decisions. When the program starts there is no such information for any task *type*. If no information is available in the lookup table for mapping of a particular task *type* on a particular core, a TLV of 0 will be returned, so the scheduler simply assigns the task to the requesting core, and enables online characterization. Once a task *type* is characterized, this information could be used for all the successive instances of the same *type* and thus the online characterization could be stopped. However, we rather keep the characterization active at every scheduling event and average the new characterized TLV value with the already TLV metadata available in the lookup table. This results in a better characterization for tasks that exhibit data-dependent control flow. Moreover, it also incorporates recent effects of dynamic variations on cores, including temperature fluctuation. Therefore, the scheduler uses the latest metadata generated from monitoring recent changes in both hardware and software. For each task scheduling point, the scheduler overhead for such decision-making is highly amortized over task execution as discussed in the next section.

5.2 Effectiveness Analysis of TEIM

We analyze the effectiveness of TEIM technique with utilization of TLV metadata. The technique reduces the total *Recovery Cycles Per Cluster* (RCPC) for every $task_j$. The total RCPC is the summation of recovery cycles of all 16 cores in the cluster. Equation (2) defines the desired ΔRCPC saving that can be achieved

[1] The number of task *types* corresponds to the number of **`task`** directives in the code, thus it is likely to be limited to a few tens.

by using TEIM technique instead of the baseline scheduler (with the naive algorithm described in Section 5.1):

$$\Delta\text{RCPC}_j = \text{L}\times\text{RCPE}\sum_{i=0}^{15}\text{TLV}_{(i,j)}\times(\text{n}_{i-\text{base}}-\text{n}_{i-\text{TEIM}})>0\ ,\forall\text{task}_j \quad (2)$$

Where L is the average number of clock cycles to execute a variation-free instance of $task_j$; RCPE is the penalty for recovering an error (described in Section 4) and has a constant value of 21 cycles, $TLV_{(i,j)}$ is TLV of $task_j$ when executing on $core_i$; and n_{i-base} (n_{i-TEIM}) is the total number of instances of $task_j$ executed on $core_i$ using baseline (TEIM) scheduler. Among the available parameters to increase ΔRCPC, TEIM can only control n_{i-TEIM}, since L is determined by the task, RCPE is imposed by the hardware recovery unit, and TLV is a function of core and task characteristics. With a positive ΔRCPC, the cluster spends fewer recovery cycles during execution of $task_j$ (higher throughput) in comparison with the baseline scheduler. However, there is an extra cost for running TEIM algorithm. Equation (3) defines *TEIM Cycles Per Cluster* (TCPC), the total number of clock cycles that the cluster consumes for all executions of TEIM during $task_j$:

$$\text{TCPC}_j = F(\text{L}_\text{S},\ \text{m}_{(i,j)},\ \text{RCPE},\ \text{TEIM-V}_i) \quad (3)$$

Where L_S is the number of clock cycles for one variation-free execution of TEIM; $m_{(i,j)}$ is the total number of times that $core_i$ runs TEIM for $task_j$ (each core contributes for executing TEIM); TEIM-V_i is the TEIM Vulnerability to variations on $core_i$. Since execution of TEIM's instructions is not variation-immune, it may incur recovery cycles depending of both characteristics of TEIM's instructions and the core that will run it. To achieve throughput benefit for $task_j$, TEIM must guarantee that ΔRCPC is always greater than the total number of cycles that cluster spends for running TEIM scheduler (TCPC). In other words, the *Decreased Cycles Per Cluster* (DCPC) has to be positive:

$$\text{DCPC}_j = \Delta\text{RCPC}_j - \text{TCPC}_j > 0 \quad (4)$$

To satisfy this constraint, the threshold parameters of TEIM (see Figure 6) are tuned accordingly, based on the following parameters. (i) L_S=250, L>3000, since TEIM takes 250 cycles to execute in the worst-case, whereas the tasks in our benchmarks take above 3000 cycles. (ii) RCPE=21. (iii) TEIM-V_i < $TLV_{(i,j)}$, since TEIM is composed of few instructions, its vulnerability is lower than any task. Thus, to satisfy (4) for every $task_j$, we need to restrict $m_{(i,j)}$ that determines how many times a $core_i$ can run TEIM scheduler to amortize $TCPC_j$ over the benefit of a desired task scheduling. The desired task scheduling is achieved by setting TLV_THR to 0.2 which avoids the cores with voltages lower than 0.97V from executing a high vulnerable task (see Figure 2). Thus, tasks with a TLV range of [0.6,0.2] will be executed on the *favoured* cores (with voltage higher than 0.97V) where they will display a lower TLV range of [0,0.2). In average, by the desired scheduling, a task reduces its TLV from 0.4 to 0.1, which based on (2) will save 6.3L recovery cycles compared to the baseline. Substituting this value and (i)−(iii) in (4) restricts $m_{(i,j)}$ < 6. This means $core_i$ during execution of $task_j$ can skip up to 6 tasks (ESCAPE_THR=6), but to prevent starvation and achieve positive DCPC, it has to execute at least one task every 6 skipped tasks.

The synthetic benchmark with the six *types* of tasks was run on the variability-affected cluster described in Section 4. Figure 7 shows the average values of ΔRCPC, TCPC, and DCPC for each dynamic task. As shown, TEIM with the tuned thresholds achieves positive DCPC values for all *types* of tasks − satisfies (4). For instance, for each dynamic task of $type_1$: (i) TEIM highly reduces the recovery cycles per cluster compared to the baseline scheduler −ΔRCPC=7K; (ii) TEIM amortizes the cost of the variation-tolerant scheduling−TCPC=1.5K; (iii) TEIM finally decreases the total cycles per cluster−DCPC=5.5K. For all types of tasks, TEIM achieves an average ΔRCPC (DCPC) of 4.1K (2.8K) per-dynamic task, in which the cost of TEIM execution is deliberately amortized (TCPC has an average value of 1.3K per-dynamic task).

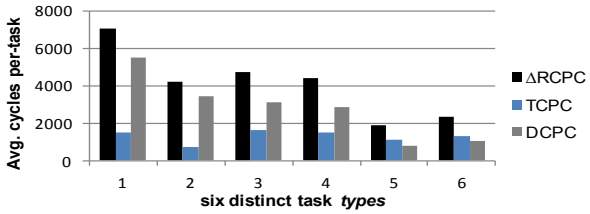

Figure 7. Average ΔRCPC, TCPC, and DCPC per-task.

6. EXPERIMENTAL RESULT

We demonstrate our approach on a SystemC-based virtual platform [24] modeling the tightly-coupled cluster described in Section 4. Table I summarizes its parameters.

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	256 KB
Latency hit	1 cycle	L3 latency	≥ 60 cycles
Latency miss	≥ 59 cycles	L3 size	256MB

To emulate variations on the virtual platform, we have integrated variations models at the level of individual instructions using the ILV characterization methodology presented in [8]. Integration of ILV models for every core enables online assessment of presence or absence of errant instructions at the certain amount of dynamic voltage and temperature variations. We re-characterized ILV models of an in-order RISC LEON-3 [23] core for 45-nm. This choice is because of availability of an advanced open-source RISC core that provides full back-end details for variation analysis. First, we synthesized the VHDL code of LEON-3 with the 45-nm TSMC technology library, general-purpose process. The front-end 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.

To observe the effects of a full range of dynamic voltage and temperature variations, we analyze the delay variability on the individual instructions, leveraging voltage-temperature scaling features of *Synopsys PrimeTime* for the composite current source approach of modeling cell behavior. Finally, delay variability is annotated to the gate-level simulations for creating ILV models. To utilize ILV models on the virtual platform, each core maps ARM v6 instructions to the corresponding ILV models in an instruction-by-instruction fashion during execution of tasks. Therefore, every core will face the errant instructions during tasks execution on the variability-affected cluster described in Section 4.

Our OpenMP implementation for the target cluster is based on [25]. To evaluate the effectiveness of the variation-tolerant technique, seven widely used computational kernel from the image processing domain are parallelized using OpenMP tasking. To quantify improvement of our technique, we have used normalized IPC of the cluster as a metric which divides the IPC of the cluster when using TEIM scheduler by the IPC of the cluster when using the baseline scheduler. First, we have quantified the overhead of TEIM technique on a variation-immune cluster (none of cores is affected by variations). Figure 8 shows the normalized IPC of the variation-immune cluster for the benchmarks. On average, the normalized IPC of the cluster (the effective instructions) is slightly decreased by 0.998×. This tiny overhead is imposed by reading the TLV lookup table, and checking the conditions mentioned in

Figure 6. During executions, the TLV lookup table only occupies 104−448 Bytes depending upon the number of task *types*. The number of dynamic tasks for each benchmark is illustrated on top of the bars in Figure 8.

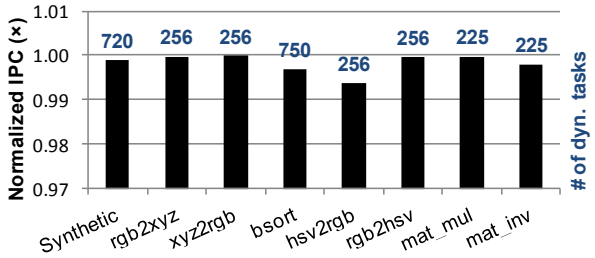

Figure 8. Overhead of the variation-tolerant scheduler.

The variation-tolerant scheduler imposes negligible IPC degradation in the variation-immune cluster, while it outperforms the baseline scheduler in the variability-affected clusters and effectively amortizes the cost of TCPC. Figure 9 shows the normalized IPC improvement of the variability-affected cluster (shown in Figure 4). As shown, the normalized IPC is increased for all benchmarks, e.g., at 10°C, IPC of *bsort* is increased by a factor of 1.51× (1.17× on average for all benchmarks). TEIM technique decreases the number of cycles per cluster for each type of tasks, because cores incur fewer errant instructions and spend lower cycles for recovery. Thus, the effective IPC is increased (compared to the baseline scheduler, the cluster spends fewer cycles for the same amount of work). Moreover, this saving is consistent across a wide range of temperature variations with a slight decrease due to the slower critical paths. At temperature of 100°C (ΔT=90°C), TEIM achieves 1.15× IPC improvement, on average, thanks to the online TLV metadata characterization which reflects the latest variations, thus enables the scheduler to react accordingly. Figure 9 also shows the average number of times that TEIM postponing the execution of the task in the head of queue (M). On average[2], each task is escaped 2.1 times because of no matching core. Overall, it shows that the tasks are postponed for a short latency in the queue, thus the performance penalty is avoided in the synchronization of tasks on a barrier.

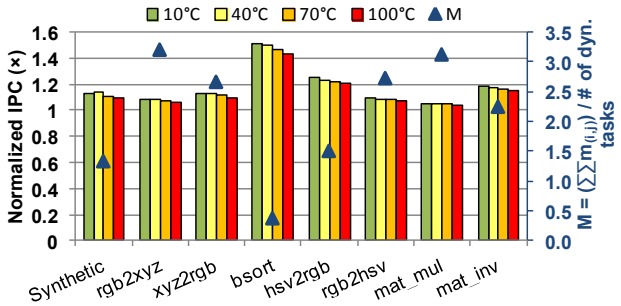

Figure 9. Normalized IPC improvement of the variability-affected cluster using TEIM across a wide temperature range.

Figure 10 shows the normalized IPC improvement of the cluster, when dedicating different number of cores for execution of tasks. On average, at 10°C, TEIM achieves 1.17×, 1.11×, 1.11×, and 1.07× IPC improvement when using only 16, 12, 8, and 4 cores, respectively. It shows effectiveness of TEIM in presence of various hardware resources, and variation scenario. TEIM achieves higher normalized IPC across higher number of cores (where there are higher variations and more voltage islands − see Figure 4). TEIM is also effective with a 4-core scenario (C_0−C_3) in which the available two voltage islands are proactively utilized.

[2] at maximum 3.2 times which is less than ESCAPE_THR

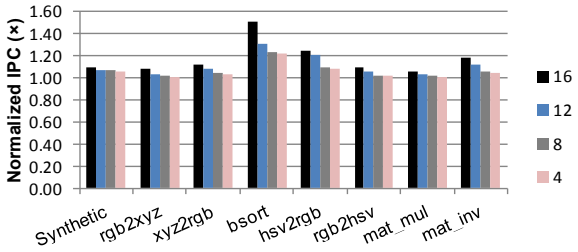

Figure 10. Cluster IPC improvement using TEIM.

7. CONCLUSION

We propose a method for vertical abstraction of circuit-level variations into a high-level parallel software execution (OpenMP 3.0 tasking). Our method characterizes and mitigates variations at the level of *tasks*, identified by the programmer through annotations. The vulnerability of tasks is characterized by TLV metadata during introspective execution on individual cores. A variation-tolerant runtime scheduler (TEIM) is proposed to utilize characterized TLV metadata. TEIM matches different characteristics of each variability-affected core to various levels of vulnerability of tasks. Therefore, it enhances normalized IPC (compared to the baseline scheduler [22]) of a 16-core variability-affected cluster up to 1.51×. On average, it achieves 1.15×−1.17× normalized IPC improvement for a wide range of temperature fluctuation.

8. ACKNOWLEDGMENTS

This work was supported by the NSF under award n. 1029783, ERC-AdG MultiTherman (291125), and FP7 Virtical (288574).

9. REFERENCES

[1] S. Borkar, "Thousand core chips−A technology perspective," Proc. *DAC*, 2007.
[2] S. Borkar, et al., "Parameter variations and impact on circuits and microarchitecture," Proc. *DAC*, pp. 338-342, 2003.
[3] L. Benini, et al., "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," Proc. *DATE*, pp. 983-987, 2012.
[4] ITRS [Online]. Available: http://public.itrs.net
[5] K.A. Bowman, et al., "Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance," IEEE Journal of Solid-State Circuits, 44(1): 49-63, 2009.
[6] M. Fojtik, et al., "Bubble Razor: An architecture independent approach to timing error detection and correction," Proc. *ISSCC*, pp. 488-490, 2012.
[7] K.A. Bowman, et al., "A 45 nm resilient microprocessor core for dynamic variation tolerance," IEEE Journal of Solid-State Circuits, 46(1): 194-208, Jan. 2011.
[8] A. Rahimi, L. Benini, R.K. Gupta, "Analysis of instruction-level vulnerability to dynamic voltage and temperature variations," Proc. *DATE*, pp. 1102-1105, 2012.
[9] M.S. Gupta, V.J. Reddi, G. Holloway, G. Wei, D.M. Brooks, "An event-guided approach to reducing voltage noise in processors," Proc. *DATE*, pp.160-165, 2009.
[10] A. Rahimi, L. Benini, R.K. Gupta, "Procedure hopping: A low overhead solution to mitigate variability in shared-L1 processor clusters," Proc. *ISLPED*, pp. 415-420, 2012.
[11] 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 J. of Solid-State Circuits, 46(1): 184-193, Jan. 2011.
[12] F. Paterna, et al., "Variability-aware task allocation for energy-efficient quality of service provisioning in embedded streaming multimedia applications," IEEE Transactions on Computers, 61(7): 939-953, 2011.
[13] R. Pawlowski, et al., "A 530mV 10-lane SIMD processor with variation resiliency in 45nm SOI," Proc. *ISSCC*, pp. 492-494, 2012.
[14] F. Chaix, G. Bizot, M. Nicolaidis, N.-E. Zergainoh, "Variability-aware task mapping strategies for many-cores processor chips," Proc. *IOLTS*, pp.55-60, 2011.
[15] OpenMP API v.3.1 [online]. Avail.: http://www.openmp.org/mp documents/OpenMP3.1.pdf
[16] O. Tahan, M. Shawky, "Using dynamic task level redundancy for OpenMP fault tolerance," Proc. *ARCS*, pp. 25-36, 2012.
[17] C. Bolchini, A. Miele, D. Sciuto, "An adaptive approach for online fault management in many-core architectures," Proc. *DATE*, pp.1429-1432, 2012.
[18] E. Ayguadè, et al., "The Design of OpenMP Tasks," IEEE Trans. Par. Distrib. Sys., 2009.
[19] P. N. Sanda, et al., "Soft-error resilience of the IBM POWER6 processor," IBM Journal of Research and Development , 52(3): 275-284, May 2008.
[20] A. Rahimi, I. Loi, M.R. Kakoee, L. Benini, "A fully-synthesizable single-cycle interconnection network for shared- ll processor clusters," Proc. *DATE*, pp.1-6, 2011.
[21] A. Rahimi, et al., "History-based dynamic voltage scaling with few number of voltage modes for GALS NoC," Proc. FutureTech, 2010.
[22] FSF - The GNU Project. GOMP - An OpenMP implementation for GCC [online]. Available: http://gcc.gnu.org/projects/gomp
[23] LEON3 [Online]. Available: http://www.gaisler.com/cms/
[24] D. Bortolotti et al., "Exploring instruction caching strategies for tightly-coupled shared-memory clusters," Proc. *Intern.Symposium on System on Chip* (*SoC*), pp.34-41, 2011.
[25] A. Marongiu et al., "Fast and lightweight support for nested parallelism on cluster-based embedded many-cores, " Proc. *DATE*, pp.105-110, 2012.