Improving Resilience to Timing Errors by Exposing Variability Effects to Software in Tightly-Coupled Processor Clusters

Abbas Rahimi, *Student Member, IEEE*, Daniele Cesarini, Andrea Marongiu, *Member, IEEE*, Rajesh K. Gupta, *Fellow, IEEE*, and Luca Benini, *Fellow, IEEE*

***Abstract*—Manufacturing and environmental variations cause timing errors in microelectronic processors that are typically avoided by ultra-conservative multi-corner design margins or corrected by error detection and recovery mechanisms at the circuit-level. In contrast, we present here runtime software support for cost-effective countermeasures against hardware timing failures during system operation. We propose a variability-aware OpenMP (VOMP) programming environment, suitable for tightly-coupled shared memory processor clusters, that relies upon modeling across the hardware/software interface. VOMP is implemented as an extension to the OpenMP v3.0 programming model that covers various parallel constructs, including `task`, `sections`, and `for`. Using the notion of work-unit vulnerability (WUV) proposed here, we capture timing errors caused by circuit-level variability as high-level software knowledge. WUV consists of descriptive *metadata* to characterize the impact of variability on different work-unit types running on various cores. As such, WUV provides a useful abstraction of hardware variability to efficiently allocate a given work-unit to a suitable core for execution. VOMP enables hardware/software collaboration with online variability monitors in hardware and runtime scheduling in software. The hardware provides online per-core characterization of WUV metadata. This metadata is made available by carefully placing key data structures in a shared L1 memory and is used by VOMP schedulerss. Our results show that VOMP greatly reduces the cost of timing error recovery compared to the baseline schedulers of OpenMP, yielding speedup of 3%–36% for tasks, and 26%–49% for sections. Further, VOMP reaches energy saving of 2%–46% and 15%–50% for tasks, and sections, respectively.**

***Index Terms*—Cross-layer variability management, OpenMP, processor clusters, recovery, robust system design, scheduling, timing errors, variations.**

Manuscript received February 01, 2014; revised February 21, 2014; accepted March 26, 2014. Date of publication April 18, 2014; date of current version June 09, 2014. This work was supported in part by NSF Variability Expeditions (1029783), in part by ERC-AdG MultiTherman (291125), and in part by FP7 Virtical (288574). This paper was recommended by Guest Editor A. Vega.

A. Rahimi and R. K. Gupta are with the Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093 USA (e-mail: abbas @cs.ucsd.edu; gupta@cs.ucsd.edu).

D. Cesarini is with the Department of Electrical, Electronic and Information Engineering, University of Bologna, 40136 Bologna, Italy (e-mail: daniele.cesarini@studio.unibo.it).

A. Marongiu and L. Benini are with the Department of Information Technology and Electrical Engineering, Swiss Federal Institute of Technology Zurich, 8092 Zurich, Switzerland, and also with the Department of Electrical, Electronic and Information Engineering, University of Bologna, 40136 Bologna, Italy (e-mail: a.marongiu@iis.ee.ethz.ch; lbenini@iis.ee.ethz.ch).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JETCAS.2014.2315883

I. Introduction

WHILE shrinking CMOS minimum feature sizes and higher transistor density open the way to many-core processor chips [1], these also come with the side effects of increased variability in device geometries and undesirable fluctuations in operating condition [2], [3]. Variations arise from different physical sources, and have static and dynamic components that are expected to be worse in future technologies [4]. The most immediate manifestation of variability is in path delay variations. Path delay variations cause violation of timing specification resulting in circuit-level timing errors. Timing errors can result in an *errant* instruction leading to a malfunction within the computing core. Hence, robust system design needs to ensure that systems perform correctly despite increasing timing failures caused by variability in many-core processor chips [5].

An important aspect of such robust system design is the ability to *detect* variations and adaptively *compensate* for their effects during system operation [6]. A wide range of dynamic variations includes fast changing voltage droops and comparatively slow changing temperature fluctuations that could occur locally as well as globally across the die area. Nowadays, multi-core processors include hardware support for dynamic thermal management—based on monitors and sensors—that enables online measurement and feedback control policies [7]. The use of *in situ* [8], [9] or replica [10], [11] circuit sensors has been investigated to detect the timing errors due to static process, and dynamic voltage and temperature variations. Razor [8] or error-detection sequential (EDS) [9] have been used to achieve robustness. A common strategy in these circuit/microarchitectural approaches to robustness is to detect incorrect circuit state values caused by timing errors.

To ensure correct functionality in the presence of timing error, these approaches rely upon error recovery mechanisms that guarantee correct program execution eventually. The timing failures are typically corrected by either adaptive tuning of CMOS control knobs to provide better-than-worst case guardband for error-free instruction execution [12], or by *replaying* the errant instruction [13]. For instance, a 45-nm Intel resilient core [13] places EDS at the endpoints of the critical paths of the pipeline stages. Once a timing error is detected during instruction execution, the core prevents the errant instruction from corrupting the architectural state and

an error control unit (ECU) triggers proper actions to ensure error recovery. The ECU first flushes the pipeline to resolve any complex bypass register issues, and then triggers one of the two recovery mechanisms: 1) instruction replay at half clock frequency; 2) multiple-issue instruction replay at the same clock frequency. These mechanisms impose energy overhead and latency penalty of up to 28 extra recovery cycles per error [13] which can adversely affect both performance and energy [14].

To achieve the required robustness while reducing these overheads, the variability-induced timing errors can be addressed through a combined hardware-software approach [15]–[17] that allows to evaluate the impact of timing errors on the overall system. A holistic cross-layer variability management can abstract the circuit-level timing error information into the vulnerability of individual (or streams of) instructions when executed on a particular core [18], [19]. For multi-core processors, this knowledge can be used by the runtime system to implement variability-tolerant parallel workload deployment for reducing the cost of timing error failure correction [20], [21]. We have earlier defined a set of hierarchically organized vulnerability measures—from instruction set architecture to a parallel programming model—to expose variations and their effects to the software stack. These measures include instruction-level vulnerability (ILV) [18], sequence-level vulnerability (SLV) [19], procedure-level vulnerability (PLV) [20], and finally task-level vulnerability (TLV) [21]. ILV characterizes individual instructions as the most fine-grained abstraction of the processor's functionality, while SLV determines streams of instructions that have a significant impact on the timing error rate. Raising further the level of abstraction, PLV exposes the effect of dynamic voltage variations for use in software preventive actions. Within a shared-memory multi-core computing cluster, PLV enables a runtime procedure hopping technique to mitigate the effect of variations by means of low-cost subroutine (procedure) migration to a less vulnerable core [20]. TLV is an extension to the OpenMP v3.0 tasking programming model to dynamically characterize the vulnerability of tasks. Here, the runtime system reduces the cost of error recovery by matching the characteristics of different variability-affected cores to the vulnerability of individual parallel tasks.

In this paper, we extend the definition of TLV to that of work-unit vulnerability (WUV), where the notion of a parallel work-unit (WU) is specialized into any of three OpenMP constructs to specify work-sharing among parallel threads: `task`, `sections` and `for`. Our goal is to provide runtime software support to increase cost-effective countermeasures against timing errors in hardware. We pursue this goal by exposing variability and its effect to the OpenMP programming model, thus enabling holistic variability management. Accordingly, we make three contributions.

1) We devise a variation-aware synergistic hardware/software approach. It enhances robustness of cluster-based processors through cost-effective software countermeasures against timing failures in hardware during system operation. On the hardware side, our multi-core cluster is equipped with circuit sensors for online measurement of variability and per-core introspective *metadata* characterization for a given workload. Fast access to metadata for each type of OpenMP work-sharing construct is guaranteed by carefully placing the key data structures on fast shared-L1 memory.
2) On the software side, we propose a fully variation-aware OpenMP (VOMP) environment, which supports `task`, `sections`, and `for`. VOMP provides online characterization of descriptive metadata for these constructs. Characterized WUV, or work-unit vulnerability, abstracts hardware variability that reflects the manifestation of circuit-level timing errors during the execution of an instance of a specific OpenMP construct. We also propose a set of scheduling algorithms, that implement software-only countermeasure schemes, one for each work-sharing construct. Hence, the OpenMP runtime scheduler utilizes WUV metadata during scheduling to efficiently mitigate the variability-induced timing errors at the level `task`, and `sections`. This leads to a holistic runtime management system that strives to reduce the cost of error recovery caused by execution of various work-sharing constructs.
3) 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 indicate the following. a) The entire cost of online software characterization and countermeasures is paid off for a variability-affected fabric. b) The proposed VOMP environment is able to save both energy and total execution time for a wide range of parallelized applications. VOMP reduces the execution time by 3%–36% and energy by 2%–46% for applications parallelized with `task` directives. VOMP also reaches to energy saving of 15%–50% and faster execution of 26%–49% for applications using `sections` directives. Further, we evaluate the robustness of our approach across $80\,^{\circ}\mathrm{C}$ temperature variations.

The rest of the paper is organized as follows. Section II surveys prior work in this specific topic area. Section III covers the architectural details to support VOMP. Section IV describes characterization of WUV metadata for every type of work-unit under a full range of dynamic voltage ($\Delta V = 0.22$ V) and temperature ($\Delta T = 140\,^{\circ}\mathrm{C}$) variations. The proposed runtime scheduling algorithms for each work-sharing construct are presented in Section V. In Section VI, we explain our methodology to capture variations, framework setup, and present experimental results followed by conclusions in Section VII.

II. Related Work

Various solutions have been proposed to mitigate the hardware timing failures, including adaptive management of guardbanding through "circuit failure prediction" mechanisms, "concurrent error detection and correction" techniques, and "cross-layer resiliency" approaches.

"Circuit failure prediction" mechanisms provide an early indication of the occurrence of a circuit failure and then adaptively avoid timing errors while reducing the conservative guardband [6], [23]. A model-based rule technique is proposed in [24] that enables focused adaptive guardbanding for various functional units at a given amount of variations. IBM POWER7 integrates adaptive power management techniques to proactively exploit

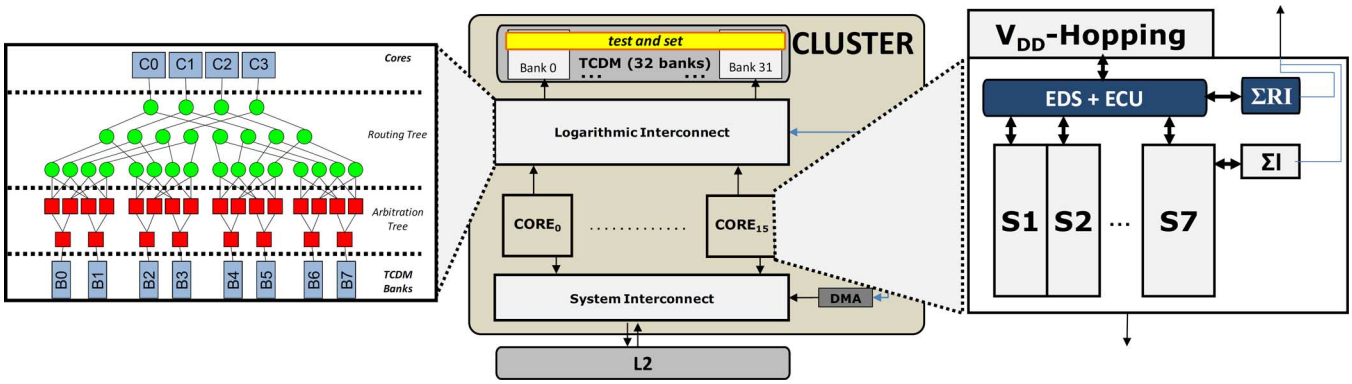

Fig. 1. Variation-tolerant tightly-coupled processor cluster for VOMP. The left part shows a 4×8 logarithmic interconnection [22]. The right part shows a resilient core that relies on EDS [9] and ECU [13] to correct timing errors by the replica instructions; ΣI is the number of error-free executed instructions, and ΣRI is the number of replayed instructions.

variations in workload as well as operating environment [25], [26]. The 8-core POWER7 prevents the timing errors by integrating five critical path monitors [11] per each core to capture PVT variations. The cores also employ two cooperating feedback controllers [27]: the first one is a clock frequency controller that reacts quickly to voltage droops by coupling between the monitors output and a phase-locked loop; the second controller dynamically adjusts the processor voltage to achieve a desired performance level on a longer time scale. These predictive techniques are well-suited for loosely-coupled processors, whereas our focus is on processor clusters. With frequent timing errors in aggressive voltage overscaling [28] and near-threshold computation [29], we opportunistically reduce guardbands by exploiting the opportunity given by *tightly-coupled* architecture to dynamically adjust workloads across the cores with minimal overhead.

"Concurrent error detection" techniques like Razor and EDS raise a warning signal to indicate timing error caused by variations [8]–[11]. Then, a recovery mechanism compensates the timing error while incurring extra recovery cycles [13]. This cost of recovery is shown to be high in face of frequent timing errors. Moreover, the circuit-level detection-correction mechanism that seeks to act for every instance of timing errors may be inefficient in large-area many-core systems that feature a cluster of tightly-coupled processors [5]. Higher level techniques are needed not only to support independent per-core recovery mechanisms, but also to reduce the cost of recovery across all processors within a cluster.

"Cross-layer resiliency" approaches have been proposed to mitigate timing errors on individual instructions [18] and sequences of instructions [19], [30]. Such fine granularly is expensive to control and requires fast hardware reactions. Moving up to a coarser granularity, techniques have been proposed to address variability at various levels, including procedure [20], thread [31], task [32], [33], and workload [34]. The main drawbacks of these techniques are the following. 1) Lack of online adaptation: for instance, [20], [33] do not support online characterization; similarly implemented policy in [32] applies static task mapping only during application initialization without providing any feedback for dynamic policy and management features. 2) Lack of standard execution environment: proposed techniques in [20], [33] define a generic notion of procedure or task execution which does not tie to a standard parallel execution model, and therefore requires intrusive changes through program source code. 3) Lack of architectural efficiency: target fabrics in [31], [33], [34] are based on coarse-grained many-core systems that impose high penalties any time that a migration is required. For instance, [31] needs to transfer the entire contents of instruction and data memories in one tile to another over a packet-switched router. This migration cost of over thousand cycles is simply too high to be useful.

OpenMP is an industry-standard combination of compiler directives and library routines, for shared-memory computers, that allows programmers to specify parallelism in their code without excessive details of parallel programming. Recently, there have been various extensions to OpenMP, as the de facto standard for shared memory multi-cores systems, to support resiliency [21], [35]–[37]. For instance, a set of extended custom directives allows a programmer to specify parts of a program that can be executed approximately or accurately [37]. Extended OpenMP task construct can also define a reliable task through a reliable clause (`#pragma omp task reliable`) [35]. Then, a dynamic triple modular redundancy (TMR) technique can be used for reliable OpenMP tasking execution. 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, [36] proposes a loosely coupled application-level TMR schema for P2012 [5], 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 to TMR, for example up to $1.8\times$ slower execution [35].

III. Architectural Support for VOMP

We now describe the architectural details of the variation-tolerant processing cluster, shown in Fig. 1. The architecture is inspired by STMicroelectronics Platform 2012 (P2012) [5], [38] as a programmable many-core accelerator for next-generation data-intensive embedded applications. The P2012 computing fabric is modular and scalable, since it is based on

multiple processor clusters such as those found in GP-GPUs [39] and clustered accelerators like HyperCore architecture line processors from Plurality [40], and Kalray multi-purpose processor array [41]. Every cluster has independent power and clock domain, therefore enabling fine-grained power and variability management [5]. The clusters are connected via a fully-asynchronous network-on-chip that enables them to work with different clock frequencies decided by a cluster controller for the power/variability management [5]. In our implementation, we focus on a single cluster consisting of sixteen tightly-coupled 32-bit in-order RISC cores, a level-one (L1) tightly coupled data memory (TCDM) and a low-latency 16×32 logarithmic interconnection [22]. The TCDM is a software-managed scratchpad memory, configured as a shared, multi-ported, multi-banked L1 memory that is directly connected to the logarithmic interconnection for fast accesses. The number of TCDM ports is equal to the number of banks (32) to enable concurrent access to different memory locations. Note that a range of addresses mapped on the TCDM space 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 the cores and TCDM banks (see the left part of Fig. 1). 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. The cores have direct access into the off-cluster L2 memory, also mapped in the global address space. Transactions to the L2 are routed to a logarithmic *peripheral interconnect* through a de-multiplexer stage. From there, they are conveyed to the L2 via the system interconnection which is based on the AHB bus. Since the TCDM has a small size (256 KB) the software must explicitly orchestrate continuous data transfers from L2 to L1, to ensure locality of computation. To allow for performance- and energy- efficient transfers, the cluster has a DMA engine. This can be controlled via memory-mapped registers, accessible through the peripheral interconnect.

In the embedded tightly-couple processor cluster, it is essential that all the cores within a cluster work with the same clock frequency to avoid the latency of the synchronization [5]. Synchronization across multiple frequencies increases the latency of the interconnection, and has a performance penalty as high as a L1 cache miss[1][22]. Therefore, the cores within the cluster are equipped with two circuit-level resiliency techniques. First, each core relies on the EDS [9] 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 [13] in its error recovery unit (ECU). It issues seven replica instructions (equal to the number of pipeline stages) followed by a valid instruction. Second, the cluster supports a V_{DD}-hopping technique [42] that discretely tunes the voltage of slow cores- the cores that are affected by static process variation. The V_{DD}-hopping improves the clock speed of the slow cores, thus enables all the 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. The core-level V_{DD}-hopping has been already employed in a variability-tolerant tightly-coupled cluster [20]. However, a core with higher vulnerability will impose extra cycles to correct the errant instructions.

[1]Eight cycles are required for synchronization between multiple clock domains for a read/write operation, while performance of the architecture relies on the fact that we have two cycles access to L1 memory.

IV. Work-Unit Vulnerability and VOMP Work-Sharing

OpenMP [43] consists of a set of compiler directives and library routines to specify parallel execution within a sequential code. Enclosing a code block within a `#pragma omp parallel` directive has the effect of launching multiple instances of that code over the available processors. Differentiating the actual work done by different processors in OpenMP is achieved by means of work-sharing constructs: `#pragma omp for`, `#pragma omp sections` and `#pragma omp task`. The for directive can only be associated to a loop nest, and distributes loop iterations over available processors. Within a `sections` directive multiple section blocks can be specified, each containing a different parallel work-unit. Sections have limited expressiveness for describing task parallelism. For this reason, the latest OpenMP specifications have included the new `task` directive, which supports sophisticated forms of task parallelism. However, `task` implies significant overheads, which makes `sections` more convenient to outline few coarse grained tasks in a program. In addition, it is easy to describe software pipeline parallelism with `sections`, by just adding point-to-point synchronization to enforce dependencies within parallel tasks. The latter is the main use we make of `sections` in this paper.

As discussed earlier in the introduction, to enable software-driven policies for variability-tolerant parallel workload scheduling we need to characterize parallel work-units, WU, in terms of vulnerability to timing errors[2]. Each OpenMP work-sharing construct outlines an execution unit which runs a sequence of instructions. Enclosing portions of code within any of these constructs allows the programmer to statically identify several WU *types* in the program, as every directive syntactically delimits a unique stream of instructions. While at runtime the same stream may be dynamically instantiated several times (e.g., a work-sharing directive nested within a loop), from the point of view of our characterization it uniquely identifies a single WU type. As a direct consequence, there are as many types of WUs in a program as there are work-sharing directives in its code, as shown in Fig. 2.

Intuitively, the closer we can associate information on variability-induced timing errors (*metadata*) to software abstractions of a parallel WU, the better we can schedule WUs to cores in a variation-tolerant manner. From this perspective, task-level vulnerability, or TLV, is an important metadata to address variability-tolerance within standard parallel programming models. The main limitation of TLV as described in [21] is that its implementation is specific to the `task` OpenMP construct. While

[2]Our platform does not have control over the errors happening while executing library code. The functionality is preserved as each core is equipped with the replay mechanism.

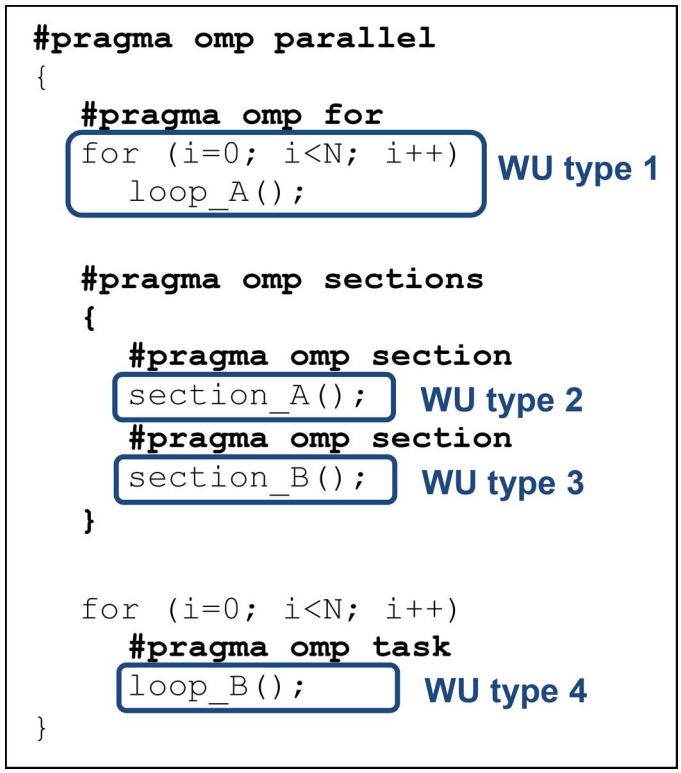

Fig. 2. Outlined WU types in a OpenMP program: `task`, `sections`, `for`.

this construct allows to express very flexible and sophisticated forms of dynamic parallelism, it is also true that several embedded workloads focus on more regular forms of parallelism, at the loop- or procedure-level [44]. Until the specification v2.5 OpenMP used to be focused on exactly those types of parallelism, through the `for` and `sections` constructs.

In our previous work [18] we have introduced ILV or instruction-level vulnerability as a metric to expose to the software stack the effect of variations on the performance of a processing core, at the level of individual instructions. In a variability-affected core ILV is not uniform across the instruction set. In fact, ILV partitions instructions into three classes: 1) logical/arithmetic, 2) memory, 3) hardware multiply/divide. Instructions belonging to different classes have different vulnerability to variations depending on the way they exercise the nonuniform 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. We note that complex out-of-order core such as IBM POWER6 also confirms that vulnerability is not uniform across the instructions set [45].

Here we extend the notion of ILV to a more coarse-grained (in terms of software execution units) metric: parallel *work-unit vulnerability* (WUV). WUV is a metric to estimate execution time of each WU type per each core, under variability. This metric is quite useful for the purpose of simultaneous vulnerability measurement and load balancing. The vulnerability of a WU type varies based on the class of instructions that it executes. WUV is clearly a per-core metric since the amount of variation affecting different classes of instructions changes from one core to another. Therefore, different dynamic instances of the same WU type can face different degrees of variability-induced timing errors.

While the identification of WU types can be done statically (i.e., at compile time), WUV characterization has to be done online due to two main reasons. First, dynamic instances of the same WU type may exercise the processor pipeline in a non-identical manner due to data-dependent control flow that results in the execution of different (classes of) instructions. Second, the characterization must reflect the variability-affected characteristic of every core (not known *a priori*) on every WU type. WUV is defined as follows:

$$\mathrm{WUV}_{(i,j)} = \sum I + \sum RI \mid \forall \mathrm{core}_i, \forall \mathrm{WUtype}_j \quad (1)$$

where ΣI is the number of error-free executed instructions; ΣRI is the number of replayed instructions[3] during execution of WU type j on core i, as reported by the ECU. Intuitively, for a given WU type if all the instructions run without any timing error, the corresponding WUV is equal to ΣI as the total error-free dynamic instruction count. In the event of timing errors, WUV also accounts for the additional replica instructions. The lower the WUV, the lower number of recovery cycles, the lower the dynamic instruction count, and thus the higher throughput and energy efficiency. WUV dynamically characterizes both vulnerability and execution time of WU types. Hence, based on WUV values, VOMP runtime schedulers can optimize the system performance or energy efficiency by matching variability-affected core characteristics to WU types.

A. Intra- and Inter-Corner WUV

For (1) WUV is the dynamic instruction count, including the replica instructions, for a given WU type. Similar to ILV, WUV is also not uniform across different variability-affected cores, which may exhibit different vulnerability to specific instruction classes. To demonstrate how this effect is propagated to the programming model level, we measure WUV across different WU types. More specifically, we use OpenMP constructs to outline software execution units, or WUs, which iterate several times over an identical instruction. We build four WU types each stressing a different instruction, as shown in Fig. 3.

In the following, we repeat the same experiment with different OpenMP work-sharing constructs. This synthetic experiment allows to stress a use case where we can estimate the variations in WUV among the software execution units. Fig. 4 illustrates the synthetic benchmark parallelized with the `#pragma omp task` construct, while the synthetic benchmark in Fig. 5 uses the `#pragma omp sections` construct. For the sake of clarity we organize the presentation of this experiment in following three consecutive subsections, one per each OpenMP construct. Section VI-A provides details of our simulation setup.

1) `task`*-Level WUV:* Fig. 4 shows the synthetic benchmark parallelized using the `#pragma omp task` construct. We measure WUV for different WU (here, `task`) types when executing on fixed and variable operating corners (current voltage and

[3]Proportional to the number of errant instructions

```
#define OP_MUL   1
#define OP_ADD   2
#define OP_DIV   3
#define OP_SHIFT 4
int A[][][], B[][][], C[][][];
void WU_run (int z, int OP)
{
  for (int y = 0; y < N; y++)
    for (int x = 0; x < N; x++)
    {
      switch(OP)
      {
        case OP_MUL:   C[x][y][z] =
            A[x][y][z] * B[x][y][z];
            break;

        case OP_ADD:   C[x][y][z] =
            A[x][y][z] + B[x][y][z];
            break;

        case OP_DIV:   C[x][y][z] =
            A[x][y][z] / B[x][y][z];
            break;

        case OP_SHIFT: C[x][y][z] =
            A[x][y][z] ≫ B[x][y][z];
            break;
      }
    }
}
```

Fig. 3. WU types each stressing a different class of instructions.

```
#pragma omp parallel
{
  #pragma omp master
  {
    for (int z = 0; z < N; z++)
      #pragma omp task
        WU_run (z, OP_MUL);

    for (int z = 0; z < N; z++)
      #pragma omp task
        WU_run (z, OP_ADD);

    for (int z = 0; z < N; z++)
      #pragma omp task
        WU_run (z, OP_DIV);

    for (int z = 0; z < N; z++)
      #pragma omp task
        WU_run (z, OP_SHIFT);
  }
}
```

Fig. 4. Synthetic benchmark using OpenMP task.

temperature). Specifically, we analyze the effects of a full range of operating corners, a temperature range of 0 °C–140 °C, and a voltage range of 0.88–1.1 V. For sake of simplicity, in this section we illustrate a *normalized* WUV (thereafter called NWUV) as a metric which divides WUV value to its ΣI, therefore this normalized metric will have a range of values greater than or equal to 1. For instance, if NWUV displays a value of 1, it indicates that there is no replica instructions $(\Sigma RI = 0)$.

Fig. 6 shows the task-level WUV for a core that works at fixed voltage supply of 1.1 V, while the environmental temperature is varied. As shown, the `task`-level vulnerability is an increasing function of temperature; for instance, the execution of task type one (task_1) at a temperature of 0 °C results in an NWUV value of 1.0017, while executing the same task at 140 °C causes an NWUV of 1.09 that increases the vulnerability of task_1 by 9%. This inter-corner WUV variation is the direct manifestation of dynamic temperature fluctuation. At supply voltage of 1.1 V, higher temperature leads to a higher timing error rate that increases the number of errant instructions, as mirrored by the WUV values.

Apart from the inter-corner WUV variation, for a given (fixed) temperature point there is an intra-corner WUV variation among the four types of WUs (tasks). As shown in Fig. 6, at the fixed temperature of 0°C, the WUV value of task_3 is 6% higher than the WUV of task_2, indicating a considerable variation across task types. WUV of each task type is different, even within the fixed operating conditions and in the absence of environmental variations, since each task type executes distinct classes of instructions experiencing different rates of the errant instructions.

Fig. 7 shows the `task`-level WUV for the core operating at a fixed temperature of 10 °C, while voltage is dynamically varied. As shown by the plot, NWUV is a decreasing function of voltage. Higher voltages result in shorter critical path delay, thus lower error rate and finally lower NWUV values. Similar to Fig. 6, intra-corner WUV variation can also be observed: WUV for different task types at the same operating corner is not equal because their instructions do not uniformly exercise the various critical paths of the pipeline. We have already seen that the vulnerability of instructions is not uniform [18] resulting in different levels of vulnerability for task types.

2) `sections`*-Level WUV:* Fig. 5 shows the code for the synthetic software pipeline implemented using parallel `sections` . Each WU type (here indicated as $\mathrm{section}_1$, $\mathrm{section}_2$, $\mathrm{section}_3$, and $\mathrm{section}_4$) is mapped on a different core. Synchronization between the pipeline stages is accomplished via simple point-to-point synchronization primitives that we implement on top of test-and-set semaphores. This guarantees that once computation of one pipeline stage is finished we can start the following stages. The `sections` construct is nested within a loop, which models the repetitions of the pipeline. It outlines four WUs, each dependent from the previous one. Note however that there is no dependence between the last stage of one iteration and the first stage of the next iteration.

In this parallel pattern, representative of image processing kernels where a set of filters is applied in sequence to independent image blocks (e.g., JPEG macro-blocks), there are N_{sec} stages, such that $N_{\mathrm{sec}} < N_{\mathrm{core}}$, where N_{core} is the number of

```
#pragma omp parallel
{
  for (int z = 0; z < N; z++)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
      {
        WU_run(z, OP_MUL);
        synch();
      }
      #pragma omp section {
        synch();
        WU_run(z, OP_ADD);
        synch();
      }
      #pragma omp section {
        synch();
        WU_run(z, OP_DIV);
        synch();
      }
      #pragma omp section {
        synch();
        WU_run(z, OP_SHIFT);
      }
    }
  }
}
```

Fig. 5. Software pipelined synthetic benchmark using OpenMP `sections`.

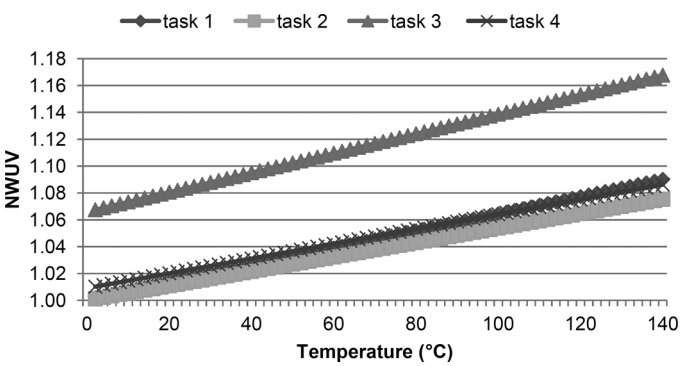

Fig. 6. Normalized WUV (NWUV) to temperature variations for `task` types.

available cores (16 cores in our platform). Normally, at the end of any work-sharing construct it is implied a barrier synchronization operation among all processors. However, we specify the `nowait` clause to skip this and allow the idle cores to start execution of the next pipeline iteration.

We now examine the `sections`-level WUV for different section types when executing on fixed and variable operating corners.Fig. 8 shows NWUV values for a core operating at fixed supply voltage of 1.1 V with a variable temperature range of 0 °C–140 °C, while Fig. 9 shows NWUV values for a fixed temperature of 10 °C with a supply voltage variation range

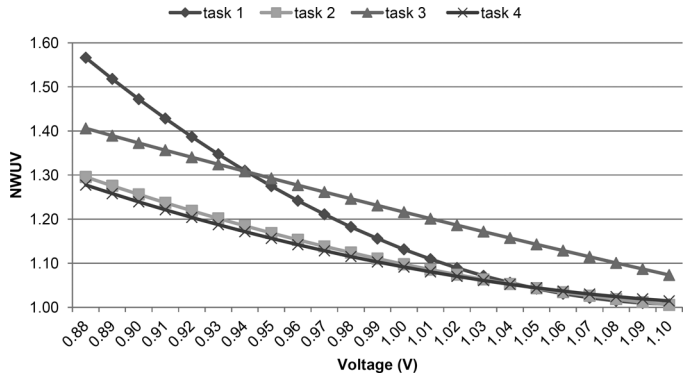

Fig. 7. Normalized WUV to voltage variations for `task` types.

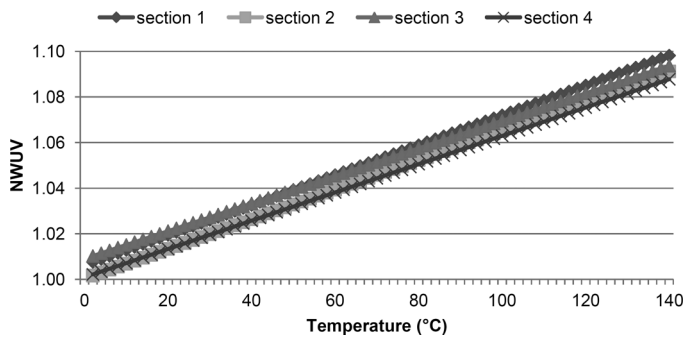

Fig. 8. Normalized WUV to temperature variations for `sections` types.

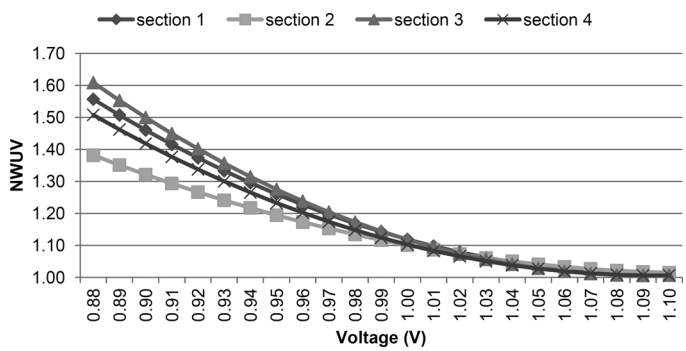

Fig. 9. Normalized WUV to voltage variations for `sections` types.

of 0.22 V. Akin to the `task`-level WUV, the `sections`-level WUV is an increasing function of temperature and a decreasing function of voltage. A temperature fluctuation of 140 °C increases the `sections`-level WUV by an average of 9%, and the voltage variation of 0.22 V increases the `sections`-level WUV by an average of 50%. Among the different section types, a maximum of 16% intra-corner WUV variation is observed at (10 °C, 1.09 V).

3) `for`*-Level WUV:* Applications running on multi-core systems often focus on a very common data parallel scenario where each core works on a portion of a data structure (e.g., array or matrix) and must synchronize with the others on a barrier. Similar parallelization schemes are typically focused on parallel loops, whose iterations are spread among several concurrent threads. Data-level parallelism, for instance parallel loops, can be exploited to distribute workload within a cluster. OpenMP v3.0 provides dynamic loop scheduling as another

work-sharing construct based on the notion of a work queue to parallelize loops locally inside a cluster. A `parallel for` directive describes a loop as a set of identical work-units; therefore the `parallel for` directive statically identifies one type of work-unit in the program. For every loop iteration, the work-unit is dynamically instantiated but it uniquely identifies a single type from our characterization point of view. In other words, the work-units generated from the `parallel for` directive are equivalent hence forming a *homogeneous* workload across all cores. This limits the capability of VOMP to schedule a single type work-unit to an appropriate core given that maintaining all cores busy.

4) Conclusion for WUV: The main conclusion that we can draw from the experiments presented in the aforementioned subsections is that WUV varies significantly: 1) among WU types; and 2) among the operating conditions. On one hand this is due to how different instruction streams exercise the variability-affected critical paths in the processor pipelines, which is the typical case for programs parallelized with `sections` or `task` directives, that outline several parallel tasks (i.e., WU types). This confirms the previous observation that executing different streams of instructions may result in various error rates [30]. For example, for any given operating condition the WUV of simple arithmetic operations (e.g., addition/shift) is lower than or equal to the WUV of complex arithmetic operations (e.g., MUL/DIV). Details sensitivity analysis of a sequence of instructions to changes in voltage and temperature are provided in [19]. On the other hand, even identical instruction streams behave differently on different cores in presence of dynamic temperature and voltage variations. This is particularly evident for the `#pragma omp for` construct, which always distributes among processors an identical work-unit type (i.e., the same instruction stream). Yet, WUV across cores varies significantly, because of the different vulnerability to specific instruction classes and to operating conditions.

This motivates the need to specialize WUV for different WU types and for online characterization. In the following section, we describe how we augment the VOPM runtime support for each of the work-sharing constructs to support online WUV characterization.

B. Online WUV Characterization

In the proposed VOMP, each core performs online characterization while executing a given WU type. To quantify WUV, the core collects ΣI and ΣRI statistics for (1) through a set of available counters in the ECU. The online characterization mechanism is distributed among all the cores in the cluster, thus enables full parallel WU execution monitoring and characterization. WUV is represented as a two-dimensional lookup table (LUT) for different WU types and cores. This lookup table is physically distributed across all the banks of the L1 TCDM for fast parallel read/write operations. Since each entry of the LUT consists of 32-bit integer data, and since each application includes a bounded[4] number (N_{WU}) of work-sharing directives, the LUT has a footprint of $N_{WU} \times 4 \times N_{\text{core}}$ Bytes, N_{core} being

[4]Up to a few tens, for large programs

```
When task_j is scheduled on core_i:
begin
  EXTRACT_TASK (task_j)
  WUV_old = LUT_rd (task_j, core_i)
  reset_WUV (core_i)
  EXECUTE_TASK (task_j)
  WUV_new = read_WUV (core_i)
  WUV_write = (WUV_new-(WUV_new≫3))+(WUV_old≫3)
  LUT_wr (task_j, core_i, WUV_write)
end
```

Fig. 10. Pseudo-code for task-level WUV characterization.

the number of cores in the cluster. We provide two simple functions for reading and writing the LUT, namely

```
int LUT_rd (int WUtype, int coreID);
void LUT_wr (int WUtype, int coreID,
    int WUV);.
```

In addition, we implement two functions for retrieving the calculated WUV of a task running on a core

```
int read_WUV (int coreID);
void reset_WUV (int coreID);.
```

The former function (`read_WUV`) reads the WUV value from per-core hardware counters, identified via the `coreID` parameter. These counters implement (1), accumulating instruction count and replica instruction count for the target core since the last reset. The second function (`reset_WUV`) resets the counter for the target core (`coreID`).

Based on these low-level APIs, we modify the OpenMP runtime schedulers to enable online WUV characterization as illustrated in Fig. 10 (our additions in **bold font**). While this pseudo-code explicitly refers to the `task` scheduler, we modify in an equivalent manner also the scheduler for `sections`. For what concerns loops the implementation is slightly more complicated. OpenMP allows to couple the `schedule(static|dynamic)` clause to the `#pragma omp for` directive. Choosing dynamic scheduling, chunks of iterations of user-defined size are scheduled to parallel cores in a first-come, first-served manner. This allows for better load balancing at runtime, but is implemented through calls to a runtime scheduler and implies higher over-head. For those cases where loop iterations contain identical amount of works it is often better to use static scheduling, which is implemented by statically inlining the code that precomputes the assigned iterations to any cores. Thus, for dynamic scheduling we instrument the runtime scheduler similar to Fig. 10. For static scheduling we modify the OpenMP compiler to inline the additional WUV characterization code during the loop expansion pass.

Note that in principle it would be strictly necessary to characterize a couple $<$ WUtype, coreID $>$ only once. Once a WU type is characterized for a given core the online characterization could be stopped. However, we rather keep the characterization active at every scheduling event and apply a his-

tory-based weighted average calculation between the new characterized WUV value and the previously WUV value stored in the LUT. This has been used to estimate power and time for a given interval [46]; and also better captures recent effects of dynamic variations on the cores, conditional code within WUs, and future workload. At each scheduling point, the encountering core incurs only a fixed negligible overhead for WU characterization. This is achieved by distributing the LUT in the multi-banked TCDM that enables not only *predictable* accesses, as opposed to cache-based hierarchical memories, but also fast parallel read/write operations among the cores.

From the observation point of view, our online characterization can reflect any changes in dynamic behavior of a core and the environment in which the core is used. More specifically, in our cluster each core can be powered at a different voltage (that could lead to different temperature points due to self-heating), but all the 16 cores have to work with a fixed clock frequency. Figs. 6–9 show the sensitivity of WUV to changes in the operating voltage and temperature. These figures illustrate that a wide range of dynamic variations can be reflected by WUV metric. From the controllability point of view, the cluster as an accelerator operate under the control of a main *host* processor, capable of running full-fledged operating systems (OS). The cluster itself, on the other hand, typically does not have all the necessary support to run unmodified OS. Resource management is demanded to custom lightweight middleware. In this respect, the OpenMP implementation that we leverage in this work [47] as a baseline to demonstrate our techniques is designed to operate on *bare metal*, as it is built directly on top of the hardware abstraction layer (HAL). The HAL provides the lowest-level software services for processor (thread) and memory management, as well as the power control APIs.

V. VOMP Schedulers

A. Variation-Aware Task Scheduling (VATS)

In this subsection we first explain our OpenMP tasking implementation followed by our specific variation-aware scheduling policy. OpenMP tasking has already been considered as a convenient programming abstraction for embedded multi- and many-cores [21], [35], [48], [49]. Typically in these approaches the task scheduler is implemented using a centralized queue which collects the task descriptors. The central FIFO design reduces the overhead for task management, which is usually a relevant design choice for energy- and resource-constrained systems. This design choice works well for homogeneous systems, but places limitations on applying efficient scheduling policies in presence of variability-induced heterogeneity across computational resources.

Our OpenMP implementation leverages distributed task queues (private queue per each core), where all the threads[5] involved in parallel computation can actively push and pop job descriptors. Fig. 11 shows the design of our OpenMP tasking framework based on a distributed queue system. Every thread can access a queue using two basic operations: *insert* and *extract*, which are translated into lock-protected operations

[5]There is a 1:1 correspondence between threads and cores, thus we will use the two terms interchangeably.

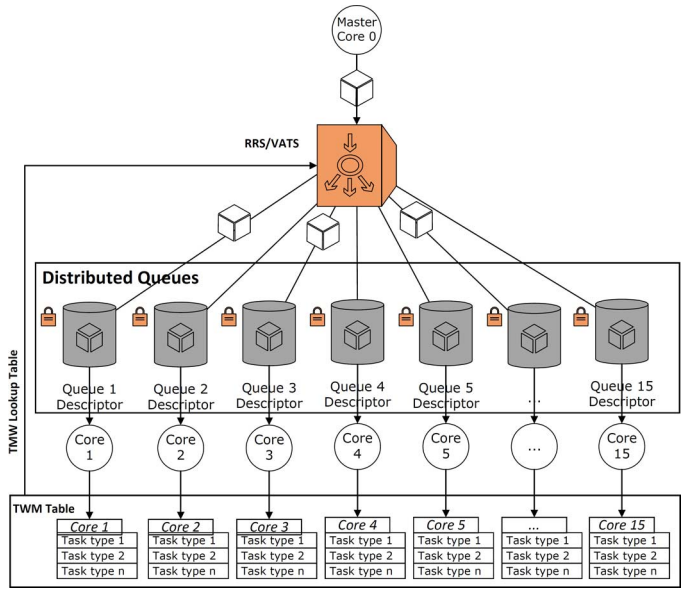

Fig. 11. Distributed queues for OpenMP tasking.

on a queue descriptor (stored in TCDM for minimal access time). Queue descriptors are statically instantiated during the initialization of the run-time to avoid the time overheads for dynamic memory management. Since threads with an empty queue are set to a low-power IDLE mode, the insertion of a task in a queue wakes up the associated core. This is achieved by inspecting an additional flag of the queue descriptor, where the destination core operating mode is annotated (*executing*, *sleeping*). The core that inserts the task in a remote queue is responsible for checking the flag and waking up the destination core to resume execution of the newly inserted task. In addition, the queue descriptor holds synchronization flags used for the `taskwait` directive. Extracting a task from a queue updates the queue descriptor in the dual manner. Note that also in this case we use lock-protected operations, since we allow all threads to extract work from any queue. Extracting tasks always occurs from the head of the queue, while insertion can be done at the head and tail. Insert operations at the head are useful to prioritize the execution of non-characterized tasks (in terms of vulnerability to errors). Stealing tasks occurs from the head of the queue.

As a baseline policy we implement a simple round-robin scheduler (RRS) [43]. This policy aims at balancing the number of tasks assigned among all cores, and introduces minimal run-time overhead due to a very lightweight implementation. To account for tasks of different durations, RRS is enhanced with a task stealing algorithm, which searches remote queues in a round-robin fashion for work to steal.

We propose a reactive policy for variability-aware task scheduling (VATS) shown in Algorithm V-1. This scheduler leverages the characterized WUV metadata to allocate tasks to cores so as to minimize both overall number of instruction replays and unbalanced loads. The main goal of this scheduler is to prevent allocation of tasks to unreliable cores, which is representative of a policy adopted in a system where task failure has critical consequences. At system startup, when there is no WUV available,

the scheduler operates in round-robin mode. Since the OpenMP tasking model assumes completely independent tasks, it is allowed to execute them in any order. We leverage this property to insert tasks for which WUV is not available yet at the head of the queue (*out-of-order* task characterization). This will give higher priority to non-characterized task types, thus speeding up the "system warm-up".

Algorithm V.1: VATS($task_j$)

for $i \leftarrow 1$ **to** N_{core}

$$\textbf{do} \begin{cases} load_i \leftarrow loadQueue_i + WUV(core_i, task_j) \\ \min \leftarrow findMinimum(load_i) \end{cases}$$

$Queue_{\min} \leftarrow insret(task_j)$

return $(\min)$

VATS scheduling policy strives to minimize the number of replayed instructions utilizing characterized WUV metadata. VATS also extends its awareness of the load on each queue, thus avoids heavily unbalanced situations that could increase the total execution time. Each queue descriptor is enhanced with a status register that estimates the overall load (*loadQueue*), in terms of dynamic instructions count, of all tasks present into 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.

To account for imbalance effects due to nonhomogeneous task durations and other system-level issues, VATS is further enhanced with a *most loaded queue-first* stealing algorithm. An additional array structure is used to keep the sorted workload over the various queues. This array is then traversed to steal work from the most loaded queues first. Note that after the execution of a stolen task we always check if in the meantime some tasks have been inserted in the local queue. It this case, we switch to the execution of the tasks with better WUV values, otherwise we continue executing the stealing algorithm until there is no task left in the system.

B. Variation-Aware Section Scheduling (VASS)

The default OpenMP section scheduling policy is to allocate a section to an available thread in a first-come, first-served (FCFS) fashion. When sections are used in a traditional manner to outline parallel tasks with no dependencies among each other Algorithm V.1 cab be applied. However, when sections are used to model software pipeline parallelism we have an additional constraint: avoiding the variability-induced errors (hence their instruction replays) that lengthen in an uncontrolled manner one or more sections. This effect dominates the overall pipeline duration. Since in a variability-affected computing cluster, there might be a set of cores that display poor performance—depending upon their software and hardware context—causing bottlenecks in the entire pipeline execution.

For these cases, we propose a variation-aware section scheduling (VASS) policy shown in Algorithm V.2. VASS has a *warm-up* phase which assigns execution of different section types to all cores for a constant[6] number of iterations. After execution of each section, the characterization process updates the corresponding WUV matadata in LUT using the mechanisms described in Section IV-B. When the warm-up phase is completed, the WUV metadata in the LUT are ready and can be inspected by the runtime environment to take decisions on workload distribution. Accordingly, VASS assigns the execution of each section to a set of suitable cores.

In this way, VASS strives to maintain all cores in the*executing* operating mode, while reducing the instruction replays and the overall pipeline duration. VASS sorts each section types based on their average WUV decreasingly. The first section type in the sorted list has either high instruction count (ΣI) or high replica instruction count (ΣRI). Therefore it should be executed on a set of suitable cores that display fewer error rate during its execution. Basically, every core has a private tag vector that lists the types of *permissible* sections for executing on this particular core. This constraint limits the participation of worse cores for executing long or high vulnerable types of sections. The worse cores instead may execute shorter sections or sections with lower vulnerability; therefore avoiding the latency penalty for the synchronization between the unbalanced stages and effectively utilizing all the resources in the variability-affected cluster.

As shown in Algorithm V.2, VASS assigns the execution of the longest section type to the best set of cores (those that display lower WUV values), then the execution of the second longest section type to the next best set of cores, and so on. In other words, VASS performs a one-to-many dynamic pipeline mapping between the section types (i.e., the stages) and the cores such that the overall execution time is reduced. After the section-to-core assignment, once a core_i encounters a section_j, VASS checks the condition to decide whether section_j is assigned for the execution on top of core_i. If section_j is assigned for core_i, it means that there is a match between the characteristics of core_i and section_j, therefore the execution will be performed. Otherwise VASS does not allocate the section_j to the core_i. Thanks to the `nowait` statement, for a `parallel sections` consists of N_{sec} sections, VASS replicates the entire `parallel sections` for $R = N_{\text{core}}/N_{\text{sec}}$ times to maintain all N_{core} cores active while reducing overall pipeline duration.

Algorithm V.2: VASS (sec_0:$sec_{N_{\text{sec}}}$)

$sortedSecList \leftarrow SortSectionsWUV(sec_0 : sec_{N_{\text{sec}}})$

while $sortedSecList \neq EMPTY$

$$\textbf{do} \begin{cases} secID \leftarrow extractTopList(sortedSecList) \\ \{coreIDs\} \leftarrow findBestSetCores(secID) \\ tag[\{coreIDs\}] \leftarrow tag[\{coreIDs\}] \bigcup secID \end{cases}$$

return $(tag[core_0 : core_{N_{\text{core}}}])$

[6] In our applications, it is selected as 2 iterations.

TABLE I
ARCHITECTURAL PARAMETERS OF THE 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	L2 latency	$\geqslant$ 60 cycles
Latency miss	$\geqslant$ 59 cycles	L2 size	256MB

VI. EXPERIMENTAL RESULTS

A. Framework Setup

We demonstrate our approach on an OpenMP-enabled SystemC-based virtual platform [50] modeling the tightly-coupled cluster described in Section III. The virtual platform supports tasking on top of a runtime [47] optimized for the target platform. Table I summarizes the main architectural parameters, a typical setup for the considered platform template (see [5]). 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 [18]. 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 [51] core for 45-nm, for which an advanced open-source RISC core with back-end details for variation analysis is available. 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. Therefore, every core will face the errant instructions during work-units execution based on the available amount of variations on the variability-affected cluster. From the same flow we also extract energy models for our cluster architecture.

For the following experiments we consider the cluster with 16 cores. 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 die-to-die and within-die process parameters variation, following the methodology presented in [20]. Each core maximum frequency varies significantly due to the process variation. As a result, six cores for 16-core cluster cannot meet the design time target clock frequency. To compensate this core-to-core frequency variation, the V_{DD}-hopping technique [42] uses the measured delay variation of each core and then selects one of available three discrete voltage modes: $\mathrm{V_{DD}}$-high, $\mathrm{V_{DD}}$-medium, $\mathrm{V_{DD}}$-low. This technique mitigates the core-to-core frequency variations within the variability-affected cluster: six cores are powered up with $\mathrm{V_{DD}}$-high, four cores with $\mathrm{V_{DD}}$-medium, and six cores with $\mathrm{V_{DD}}$-low. This ensures all cores work with the design time target frequency, but they face different error rate based on the instruction type and the operating condition.

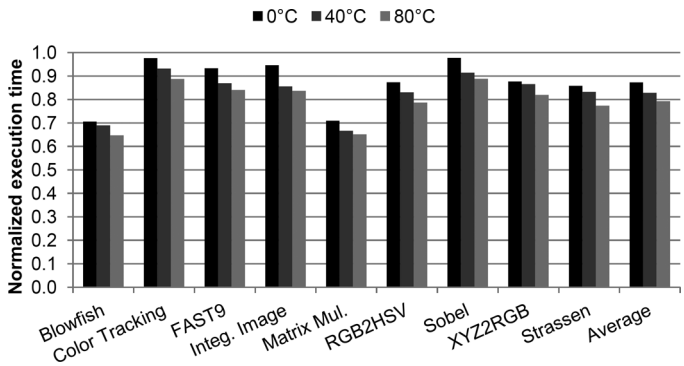

Fig. 12. Execution time for VATS normalized to RRS under temperature variation.

B. VOMP Results for Tasking

We use nine widely adopted computational kernels mainly from the image processing domain, that we parallelize using `task` directives. These kernels include *RGB-to-HSV* and *XYZ-to-RGB* for colormap conversions, *Integral image* and *Sobel* for filter operations, *FAST* for corner detection, *Color Tracking* , *Strassen* matrix multiplication, and *Blowfish* for encryption/decryption. Each kernel has one task type, therefore there is no task dependency during execution. We compare the total execution time and energy consumption of VATS, our variability-aware task scheduler, to the baseline RRS policy. Fig. 12 shows the execution time for all the kernels for three operating corners with temperature of $0\,^{\circ}\mathrm{C}$, $40\,^{\circ}\mathrm{C}$, and $80\,^{\circ}\mathrm{C}$. VATS aims at reducing the instruction replays by allocating tasks on reliable cores while taking into account the load of every queue. As a result, at an operating temperature of $0\,^{\circ}\mathrm{C}$, VATS achieves up to 30% better performance than RRS, and 13% on average. This clearly indicates that the entire overhead of the variation-tolerant technique is paid off, including the online task characterization, reading and updating WUV metadata, and cost of execution of Algorithm V.1. As shown, VATS displays a robust behavior across a wide range of temperature variations thanks to the reflection by the *always-on* characterizations. At higher temperature, VATS achieves better average performance gain of 17% (at $40\,^{\circ}\mathrm{C}$) and 21% ($80\,^{\circ}\mathrm{C}$), since WUV is increased at higher temperature.

Fig. 13 shows the energy consumption of the kernels for VATS normalized to RRS. VATS achieves on average 21% and up to 38% better energy efficiency than RRS at the temperature of $0\,^{\circ}\mathrm{C}$. VATS further reaches to an average energy saving of 31% at the operating temperature of $80\,^{\circ}\mathrm{C}$.

We also compare the TLV technique with the centralized queue proposed in [21]. TLV, which has variation-agnostic task insertion operations displays on average 75% slower execution than RRS. TLV is on average 100% less energy efficient than

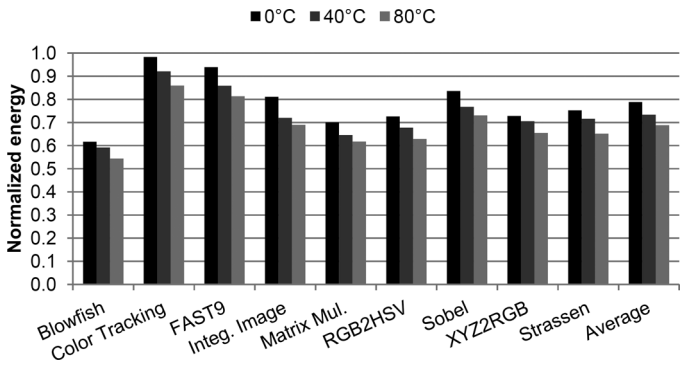

Fig. 13. Energy consumption for VATS normalized to RRS under temperature variation.

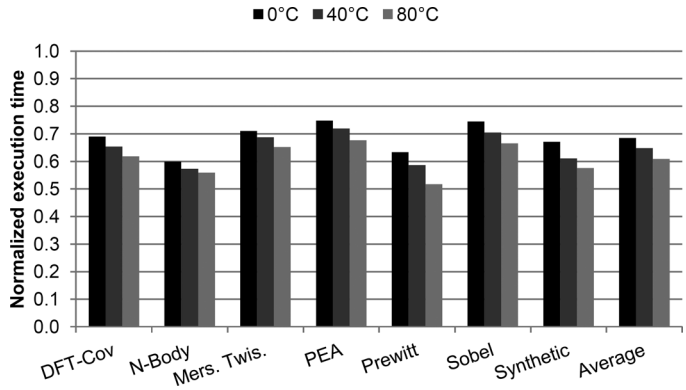

Fig. 14. Execution time for VASS normalized to FCFS under temperature variation.

RRS. This lack of efficient utilization of resources under variability is mainly because of TLV characterization that does not consider the overall system workload. Its single tasking queue also limits the potentials of task scheduling policies: a core can utilize TLV to only decide whether to proceed to the execution of a task or leave it in the single queue for other cores that leads to an imbalanced system.

C. VOMP Results for Sections

For evaluating VOMP in the `parallel sections`, we used seven computational intensive kernels amenable to software pipelining. Pitch extractor algorithm (*PEA*), and FFT with covariance matrix factorization (*DFT-COV*) are embedded signal processing kernels extracted from [52], [53].*Sobel* and *Prewitt* are filter operations useful in the edge detection algorithms. *N-body* is a simulation of a large number of particles under the influence of physical forces. *Mersenne twister* is a pseudorandom number generator. Synthetic is a microkernel implementing a four-stage parallel pipeline (see Fig. 5), representative of streaming applications [54]. We evaluate the effectiveness and robustness of our approach across a wide temperature range of 80 °C.

Fig. 14 shows the normalized performance (execution time) of VASS to FCFS for three operating corners with temperature of 0 °C, 40 °C, and 80 °C. At an operating temperature of 0 °C, the total execution time is reduced on average by 31% (and up to 40%) thanks to proper assignment of sections to those cores that avoid unbalanced pipelines. This is accomplished by preventing the worst cores from executing a section type that leads to the highest WUV. At the temperature of 80 °C, VASS reaches on average 39% performance improvement, thanks to the online WUV metadata characterization which reflects the latest temperature variations, thus enabling the scheduler to react accordingly.

Moreover, as shown in Fig. 15, VASS simultaneously reduces the total dynamic instruction count that yields an average of 28% (up to 35%) reduction in energy consumption at an operating temperature of 0 °C. A similar pattern for energy saving is observed under temperature fluctuations, confirming the robustness of our approach. VASS reduces energy consumption on average by 37% for high operating temperatures of 80 °C.

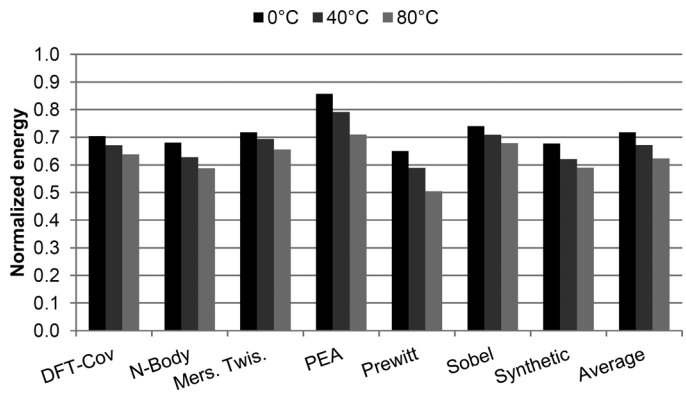

Fig. 15. Energy consumption for VASS normalized to FCFS under temperature variation.

VII. Conclusion

Circuit failures due to timing errors are considered an important concern in the design of reliable circuits. In this paper, we show that processing cores can be made robust against an important class of such errors, caused by manufacturing and environmental variabilities, by raising the visibility of such failures across the hardware/software boundary. This is achieved by attaching *metadata* that captures work-unit vulnerability (WUV) from hardware sensing circuits to the runtime system via the software stack. We specifically address its implementation in a parallel execution environment that associates WUV metadata to OpenMP parallel constructs: `task`, `sections`, and `for`. WUV metadata is characterized during work-unit execution on individual cores, and is used to efficiently schedule new instances of the same work-unit type. We have implemented our approach in VOMP, a variability-aware OpenMP execution environment. With VOMP, we propose scheduling algorithms for `tasks` and `sections` that use WUV metadata for countermeasures against variability-induced timing errors. This matches the characteristics of different variability-affected cores to the error-vulnerability of different work-unit types in the program, minimizing the need for timing error recovery and the associated costs. Across a wide operating temperature of 80 °C, VOMP effectively eliminates the timing error recovery in the 16-core cluster resulting in average 17% and 36% faster execution for

`task` and `sections`, respectively. VOMP achieves an average energy saving of 27% for `task` and 33% for `sections`.

REFERENCES

[1] S. Borkar, "Thousand core chips: A technology perspective," in *Proc. 44th Annu. Design Automat. Conf.*, New York, 2007, pp. 746–749.

[2] S. Borkar *et al.*, "Parameter variations and impact on circuits and microarchitecture," in *Proc. Design Automat. Conf.*., Jun. 2003, pp. 338–342.

[3] S. Ghosh and K. Roy, "Parameter variation tolerance and error resiliency: New design paradigm for the nanoscale era," *Proc. IEEE*, vol. 98, no. 10, pp. 1718–1751, Oct. 2010.

[4] The ITRS website [Online]. Available: http://public.itrs.net

[5] D. Melpignano *et al.*, "Platform 2012, a many-core computing accelerator for embedded SOCs: Performance evaluation of visual analytics applications," in *Proc. 49th ACM/EDAC/IEEE Design Automat. Conf.*, Jun. 2012, pp. 1137–1142.

[6] S. Mitra *et al.*, "Robust system design to overcome CMOS reliability challenges," *IEEE J. Emerg. Select. Topics Circuits Syst.*, vol. 1, no. 1, pp. 30–41, Mar. 2011.

[7] A. Bartolini, M. Cacciari, A. Tilli, and L. Benini, "Thermal and energy management of high-performance multicores: Distributed and self-calibrating model-predictive controller," *IEEE Trans. Parallel Distrib. Syst.,*, vol. 24, no. 1, pp. 170–183, Jan. 2013.

[8] S. Das *et al.*, "A self-tuning DVS processor using delay-error detection and correction," *IEEE J. Solid-State Circuits*, vol. 41, no. 4, pp. 792–804, Apr. 2006.

[9] K. Bowman *et al.*, "Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance," *IEEE J. Solid-State Circuits*, vol. 44, no. 1, pp. 49–63, Jan. 2009.

[10] J. Tschanz *et al.*, "Tunable replica circuits and adaptive voltage-frequency techniques for dynamic voltage, temperature, and aging variation tolerance," in *VLSI Circuits Symp.*, Jun. 2009, pp. 112–113.

[11] A. Drake *et al.*, "A distributed critical-path timing monitor for a 65 nm high-performance microprocessor," in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, Feb. 2007, pp. 398–399.

[12] L. d. L. Silva *et al.*, "Power efficient variability compensation through clustered tunable power-gating," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 1, no. 3, pp. 242–253, Sep. 2011.

[13] K. Bowman *et al.*, "A 45 nm resilient microprocessor core for dynamic variation tolerance," *IEEE J. Solid-State Circuits*, vol. 46, no. 1, pp. 194–208, Jan. 2011.

[14] H. Zakaria and L. Fesquet, "Designing a process variability robust energy-efficient control for complex SOCs," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 1, no. 2, pp. 160–172, Jun. 2011.

[15] P. Gupta *et al.*, "Underdesigned and opportunistic computing in presence of hardware variability," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 32, no. 1, pp. 8–23, Jan. 2013.

[16] G. Karakonstantis, A. Chatterjee, and K. Roy, "Containing the nanometer pandora-box: Cross-layer design techniques for variation aware low power systems," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 1, no. 1, pp. 19–29, Mar. 2011.

[17] L. Leem *et al.*, "Cross-layer error resilience for robust systems," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, Nov. 2010, pp. 177–180.

[18] A. Rahimi, L. Benini, and R. Gupta, "Analysis of instruction-level vulnerability to dynamic voltage and temperature variations," in *Design, Automat. Test Eur. Conf. Exhibit.*, Mar. 2012, pp. 1102–1105.

[19] A. Rahimi, L. Benini, and R. Gupta, "Application-adaptive guardbanding to mitigate static and dynamic variability," *IEEE Trans. Comput.*, 2013.

[20] A. Rahimi, L. Benini, and R. Gupta, "Procedure hopping: A low overhead solution to mitigate variability in shared-l1 processor clusters," in *Proc. 2012 ACM/IEEE Int. Symp. Low Power Electron Design*, New York, 2012, pp. 415–420.

[21] A. Rahimi *et al.*, "Variation-tolerant openmp tasking on tightly-coupled processor clusters," in *Design, Automat. Test Eur. Conf. Exhibit.*, Mar. 2013, pp. 541–546.

[22] A. Rahimi, I. Loi, M. Kakoee, and L. Benini, "A fully-synthesizable single-cycle interconnection network for shared-l1 processor clusters," in *Design, Automat. Test Eur. Conf. Exhibit.*, Mar. 2011, pp. 1–6.

[23] Y. Li *et al.*, "Overcoming early-life failure and aging for robust systems," *IEEE Design Test Comput.*, vol. 26, no. 6, pp. 28–39, Nov. 2009.

[24] A. Rahimi, L. Benini, and R. K. Gupta, "Hierarchically focused guardbanding: An adaptive approach to mitigate PVT variations and aging," in *Design, Automat. Test Eur. Conf. Exhibit.*, Mar. 2013, pp. 1695–1700.

[25] M. Floyd *et al.*, "Adaptive energy-management features of the IBM Power7 chip," *IBM J. Res. Develop.*, vol. 55, no. 3, pp. 8:1–8:18, May 2011.

[26] M. Floyd *et al.*, "Introducing the adaptive energy management features of the power7 chip," *IEEE Micro*, vol. 31, no. 2, pp. 60–75, Mar. 2011.

[27] C. Lefurgy *et al.*, "Active guardband management in power7+ to save energy and maintain reliability," *IEEE Micro*, vol. 33, no. 4, pp. 35–45, Jul. 2013.

[28] D. Jeon *et al.*, "Design methodology for voltage-overscaled ultralow-power systems," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 59, no. 12, pp. 952–956, Dec. 2012.

[29] M. Kakoee, I. Loi, and L. Benini, "Variation-tolerant architecture for ultra low power shared-11 processor clusters," *IEEE Trans. Circuits Syst. II, Exp., Briefs*, vol. 59, no. 12, pp. 927–931, Dec. 2012.

[30] G. Hoang, R. B. Findler, and R. Joseph, "Exploring circuit timingaware language and compilation," in *Proc. 16th Int. Conf. Archit. Support Program. Lang. Operat. Syst.*, NY, 2011, pp. 345–356.

[31] 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. Solid-State Circuits*, vol. 46, no. 1, pp. 184–193, Jan. 2011.

[32] F. Chaix, G. Bizot, M. Nicolaidis, and N.-E. Zergainoh, "Variabilityaware task mapping strategies for many-cores processor chips," in *Proc. 2011 IEEE 17th Int. On- Line Testing Symp.*, Jul. 2011, pp. 55–60.

[33] H. Cho, L. Leem, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," *IEEE Trans. Computer-Aided Design Integrat. Circuits Syst.*, vol. 31, no. 4, pp. 546–558, Apr. 2012.

[34] F. Paterna *et al.*, "Variability-aware task allocation for energy-efficient quality of service provisioning in embedded streaming multimedia applications," *IEEE Trans. Comput.*, vol. 61, no. 7, pp. 939–953, Jul. 2012.

[35] O. Tahan and M. Shawky, "Using dynamic task level redundancy for openmp fault tolerance," in *Proc. 25th Int. Conf. Archit. Comput. Syst.*, 2012, pp. 25–36.

[36] C. Bolchini, A. Miele, and D. Sciuto, "An adaptive approach for online fault management in many-core architectures," in *Design, Automat. Test Eur. Conf. Exhibit.*, Mar. 2012, pp. 1429–1432.

[37] A. Rahimi, A. Marongiu, R. Gupta, and L. Benini, "A variability-aware openMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor clusters," in *Proc. Int. Conf. Hardware/Software Codesign Syst. Synthesis*, Sep. 2013, pp. 1–10.

[38] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Design, Automat. Test Eur. Conf. Exhibit.*, Mar. 2012, pp. 983–987.

[39] NVIDIA's next generation CUDA compute architecture Fermi whitepaper.

[40] Plurality, the HyperCore processor [Online]. Available: http://www.plurality.com/hypercore.html

[41] KALRAY, MPPA [Online]. Available: http://www.kalray.eu/products/mppa-manycore-a-multicore-processors-family-13/

[42] S. Miermont, P. Vivet, and M. Renaudin, "A power supply selector for energy- and area-efficient local dynamic voltage scaling," in *Proc. 17th Int. Workshop Integrated Circuit Syst. Design. Power Timing Model., Optimizat. Simulat.*, , 2007, pp. 556–565 [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74442-9-54

[43] The GNU project, GOMP—An openMP implementation for GCC [Online]. Available: http://gcc.gnu.org/projects/gomp

[44] E. Ayguade *et al.*, "The design of openMP tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 3, pp. 404–418, Mar. 2009.

[45] P. Sanda *et al.*, "Soft-error resilience of the IBM Power6 processor," *IBM J. Res. Develop.*, vol. 52, no. 3, pp. 275–284, May 2008.

[46] E. K. Ardestani, E. Ebrahimi, G. Southern, and J. Renau, "Thermal-aware sampling in architectural simulation," in *Proc. 2012 ACM/IEEE Int. Symp. Low Power Electron. Design*, 2012, pp. 33–38.

[47] A. Marongiu, P. Burgio, and L. Benini, "Fast and lightweight support for nested parallelism on cluster-based embedded many-cores," in *Design, Automat. Test Eur. Conf. Exhibit.*, Mar. 2012, pp. 105–110.

[48] P. Burgio, G. Tagliavini, A. Marongiu, and L. Benini, "Enabling fine-grained openMP tasking on tightly-coupled shared memory clusters," in *Design, Automat. Test Eur. Conf. Exhibit.*, Mar. 2013, pp. 1504–1509.

[49] S. Agathos, V. Dimakopoulos, A. Mourelis, and A. Papadogiannakis, "Deploying openMP on an embedded multicore accelerator," in *Proc. Int. Conf. Embed. Comput. Syst.: Architect., Model., Simulat.*, Jul. 2013, pp. 180–187.
[50] D. Bortolotti *et al.*, "VirtualSoC: A full-system simulation environment for massively parallel heterogeneous system-on-chip," in *IPDPS Workshops*, 2013, pp. 2182–2187.
[51] Leon3 [Online]. Available: http://www.gaisler.com/cms/
[52] P. Hoang and J. Rabaey, "Scheduling of DSP programs onto multiprocessors for maximum throughput," *IEEE Trans. Signal Process.*, vol. 41, no. 6, pp. 2225–2235, Jun. 1993.
[53] V. K. P. M. Lee and W. Liu, "A mapping methodology for designing software task pipelines for embedded signal processing," *Parallel Distribut. Process.*, pp. 937–944, 1998.
[54] A. Moreno *et al.*, "Load balancing in homogeneous pipeline based applications," *Parallel Comput.*, vol. 38, no. 3, pp. 125–139, 2012 [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001566

Abbas Rahimi (S'10) received the B.S. degree in computer engineering from the School of Electrical and Computer Engineering at the University of Tehran, Tehran, Iran, in March 2010. He is currently a fourth year Ph.D. candidate in the Department of Computer Science and Engineering at the University of California, San Diego, La Jolla, CA, USA.

Since June 2010, he has also been with the Microelectronic Group at the University of Bologna, Bologna, Italy and the Integrated Systems Laboratory at the Swiss Federal Institute of Technology Zurich, Zurich, Switzerland. His research interests are in the resilient system design, design for robustness, and high-performance on-chip interconnections. In this area, he has published more than 20 papers in peer-reviewed international journals and conferences.

Mr. Rahimi received the Best Paper Candidate at 50th IEEE/ACM Design Automation Conference.

Daniele Cesarini received the B.S. degree in computer engineering from the University of Bologna, Bologna, Italy, in 2010, where he is currently an M.S. degree student in the Department of Electrical, Electronic and Information Engineering.

He joined to the Micrel Lab in 2013 and works in variability-aware environment for multiprocessors system on chip. His research interests include programming models, compilers, and languages support for parallel computing.

Andrea Marongiu (M'04) received the M.S. degree in electronic engineering from the University of Cagliari, Cagliari, Italy, in 2006, and the Ph.D. degree in electronic engineering from the University of Bologna, Bologna, Italy, in 2010.

He currently is a postdoc researcher at the Department of Electrical, Electronic and Information Engineering, University of Bologna, Bologna, Italy. He also holds a postdoc position at ETHZ, Zurich, Switzerland. His research interests concern parallel programming model and architecture design in the single-chip multiprocessors domain, with special emphasis on compilation for heterogeneous architectures, efficient usage of on-chip memory hierarchies, and SoC virtualization.

Rajesh K. Gupta (F'04) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Kanpur, Kalyanpur, India, in 1984, the M.S. degree in electrical engineering and computer science from the University of California, Berkeley, CA, USA, in 1986, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 1994.

He is a Professor of computer science and engineering at the University of California, San Diego (UCSD), La Jolla, CA, USA, and holds the Qualcomm endowed Chair. He directs the smart buildings/smart grids task force at UCSD in his role as Associate Director for the California Institute for Telecommunications and Information Technology (CalIT2). His recent contributions include SystemC modeling and SPARK parallelizing high-level synthesis, both of which are publicly available and have been incorporated into industrial practice. Earlier, he led or co-led DARPA-sponsored efforts under the Data Intensive Systems (DIS) and Power Aware Computing and Communications (PACC) programs that demonstrated architectural adaptation and compiler optimizations in building high-performance and energy-efficient system architectures. He currently leads the National Science Foundation Expedition on Variability.

Luca Benini (F'07) is Full Professor at the University of Bologna, Bologna, Italy, and is the Chair of Digital Integrated Circuits and Systems at ETHZ. He has served as Chief Architect for the Platform2012/STHORM project in STmicroelectronics, Grenoble in the period 2009–2013. He has held visiting and consulting researcher positions at EPFL, IMEC, Hewlett-Packard Laboratories, and Stanford University. His research interests are in energy-efficient system design and multi-core SoC design. He is also active in the area of energy-efficient smart sensors and sensor networks for biomedical and ambient intelligence applications. He has published more than 700 papers in peer-reviewed international journals and conferences, four books and several book chapters.

Dr. Benini is a member of the Academia Europaea.