A Variability-Aware OpenMP Environment for Efficient Execution of Accuracy-Configurable Computation on Shared-FPU Processor Clusters

Abbas Rahimi[†], Andrea Marongiu[‡], Rajesh K. Gupta[†], Luca Benini[‡]

[†]Department of Computer Science and Engineering, UC San Diego, La Jolla, CA 92093, USA
[‡]Dipartimento di Ingegneria dell'Energia Elettrica e dell'Informazione, Università di Bologna, 40136 Bologna, Italy
{abbas, gupta}@cs.ucsd.edu, {a.marongiu, luca.benini}@unibo.it

ABSTRACT

We propose a tightly-coupled, multi-core cluster architecture with shared, variation-tolerant, and accuracy-reconfigurable floating-point units (FPUs). The resilient shared-FPUs dynamically characterize FP pipeline vulnerability (FPV) and expose it as metadata to a software scheduler for reducing the cost of error correction. To further reduce this cost, our programming and runtime environment also supports *controlled* approximate computation through a combination of design-time and runtime techniques. We provide OpenMP extensions (as custom directives) for FP computations to specify parts of a program that can be executed approximately. We use a profiling technique to identify tolerable error significance and error rate thresholds in error-tolerant image processing applications. This information guides an application-driven hardware FPU synthesis and optimization design flow to generate efficient FPUs. At runtime, the scheduler utilizes FPV metadata and *promotes* FPUs to accurate mode, or *demotes* them to approximate mode depending upon the code region requirements. We demonstrate the effectiveness of our approach (in terms of energy savings) on a 16-core tightly-coupled cluster with eight shared-FPUs for both error-tolerant and general-purpose error-intolerant applications.

Categories and Subject Descriptors

B.8.0 [Performance and Reliability]: General

General Terms

Reliability, Design, Performance, Algorithms.

Keywords

PVT variability, timing error, floating-point, resilient, approximation, OpenMP, multi-core.

1. INTRODUCTION

Variability is a growing challenge in microelectronic designs [1],[2]. Static process variations manifest themselves as die-to-die and within-die variations. Die-to-die variations affect all computing units on a die equally, whereas within-die variations induce different characteristics for each computing unit. Dynamic variations are caused by the operating conditions. 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 [3]. Variations may prevent a circuit from meeting timing constraints thus resulting in timing errors. IC designers commonly use guardbands on the operating frequency or voltage to ensure error-free operation for the worst-case variations. Given the increasing costs of guardbands, it is important to make a design inherently resilient to errors and variations. In this paper, we focus on resiliency to timing errors caused by variations.

Resilient designs typically employ *in situ* or replica circuit sensors to detect the variability-induced timing error in both logic and memory blocks. For logic, error-detection sequential (EDS) [4] circuit sensors have been employed, while an 8T SRAM arrays utilized a tunable replica bits [5]. A common strategy is to detect variability-induced delays using data that arrives shortly after the relevant clock edge and flagging it as a timing error. On detection, the timing failures are corrected by replaying the errant operation with a larger guardband through various adaptation techniques. For instance, a resilient 45-nm integer-scalar core [6] places EDS within 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) initially flushes the pipeline to resolve any complex bypass register issues. To ensure error recovery, the ECU supports two separate techniques: instruction replay at half frequency, and multiple-issue instruction replay at the same frequency. These techniques impose a latency of up to 28 extra recovery cycles per error with an energy overhead of 26nJ for the resilient 7-stage integer core [6].

The cost of these recovery mechanisms is high in the face of frequent timing errors in aggressive voltage down-scaling and near-threshold computation [7] in an attempt to save power. This cost is exacerbated in floating-point (FP) pipelined architectures because FP pipelines typically have high latency, e.g., up to 32 cycles to execute depending upon the type and precision on an ARM Cortex-A9 [8], and higher energy-per-instruction costs than their integer counterparts. Further, deeper pipelines induce higher pipeline latency and higher cost of recovery through flushing and replaying. These energy-hungry high-latency pipelines are prone to inefficiencies under timing errors because the number of recovery cycles per error is increased at least linearly with the pipeline length. More importantly, FP pipelines are often shared among cores due to their large area and power cost. For instance, the AMD Bulldozer architecture shares a floating-point unit (FPU) between a dual-clustered integer core, with four pipelines [9]. UltraSPARC T1 also has a shared-FPU between eight cores. This makes the cost of recovery even more pronounced for a cluster of tightly-coupled processors utilizing shared resources.

978-1-4799-1417-3/13/$31.00

1.1 Contributions

Our goal is to reduce the cost of a resilient FP environment which is dominated by the error correction. Tolerance to error in execution is often a property of the application: some applications, or their parts, are tolerant to errors (notably, media processing applications), while some other parts must be executed exactly as specified. We either explicitly ignore the timing errors − if possible − in a fully *controlled* manner to avoid undefined behavior of programs; or we try to reduce the frequency of timing errors by assigning computations to appropriate pipelines with lower vulnerability. Accordingly, this paper makes three contributions:

1. We propose a set of accuracy-reconfigurable FPUs that are resistant to variation-induced timing errors and shared among tightly-coupled processors in a cluster. This resilient shared-FPUs architecture supports online timing error detection, correction, and characterization. We introduce the notion of FP pipeline vulnerability (**`FPV`**), captured as metadata, to expose variability and its effects to a software scheduler for reducing the cost of error correction. A runtime ranking scheduler utilizes the **`FPV`** metadata to identify the most suitable FPUs for the required computation accuracy for the minimum timing error rate.

2. Using the notions of *approximate* and *accurate* computations, we describe a compiler and architecture environment to use approximate computations in a user- or algorithmically-controlled fashion. This is achieved via design-time profiling, synthesis, and optimization in conjunction with runtime characterization techniques. This approach eliminates the cost of error correction for specific annotated approximate regions of code if and only if the propagated error significance and error rate meet application-specific constraints on quality of output. For *error-tolerant* applications our OpenMP extensions specify parts of a program that can be executed approximately, thus providing a new degree of scheduling flexibility and error resilience. At design-time, code regions are profiled to identify acceptable error significance and error rate. This information drives synthesis of an application-driven hardware FPU. At runtime, as different sequences of OpenMP directives are dynamically encountered during program execution, the scheduler *promotes* FPUs to accurate mode, or *demotes* them to approximate mode depending upon the code region requirements. Section 3 and Section 4 cover these details.

3. Our approach enables efficient execution of finely interleaved approximate and accurate operations enforced by various computational accuracy demands within and across applications. We demonstrate the effectiveness of our approach on a 16-core tightly-coupled cluster in the presence of timing errors. For general-purpose error-intolerant application, our approach reduces the recovery cycles that yield an average energy saving of 22% (and up to 28%), compared to the worst-case design. For error-tolerant image processing applications with annotated approximate directives, 36% energy saving is achieved while maintaining acceptable quality degradation. In Section 5, we present experimental results followed by conclusions in Section 6.

2. RELATED WORK

Characterization and use of variability-affected execution of instructions is an active area of research. Rahimi *et al.* have considered execution vulnerability at the instruction level [10], across a sequence of instructions [11] to expose variability and its effects to the software stack. Another technique is to use procedure-level vulnerability [12] 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. An extension to the OpenMP 3.0 tasking programming model is also proposed to dynamically characterize task-level vulnerability in a shared memory processor clusters [13]. Here, the runtime system matches different characteristics of each variability-affected core to various levels of vulnerability of tasks. A configurable-accuracy integer adder is proposed in [14], where the error correction module is power-gated during approximate operations. By contrast, this paper focuses on energy-hungry FP operations.

To ensure practical use, approximate computation must be controllable at the granularity of instructions [15] given the difficulties in identifying large blocks of 'error-tolerant' instructions. This requires interleaved execution of approximate and precise computations. Error resilient system architecture (ERSA) [16] isolates execution of control-intensive tasks (on super reliable cores) from execution of data-intensive tasks (on relaxed reliability cores). ERSA is suited for applications consisting of a set of coarse-grained isolated tasks that can be expressed entirely with approximate computation. However, in case of a fine-grain interleaving of accurate and approximate instructions, ERSA migration costs of over thousand cycles [17] are simply too high to be useful. Further, ERSA does not support any standard parallel programming model of execution.

EnerJ [18] is a programming language supporting *disciplined approximation* that lets programmers declare which parts of a program can be computed approximately to save computational effort (and power). A program is decomposed into two components: one that runs precisely, and another that runs approximately, carrying no guarantees on the output of computation. Green [19] also trades off quality of service for improvements in energy consumption, while providing statistical quality of service guarantees. Truffle [15], a dual-voltage microarchitecture design, supports mapping of disciplined approximate EnerJ programs through ISA extensions. It applies a high voltage for precise operations and a low voltage for approximate operations. Truffle relies on the programming language to provide safety guarantees *statically* to the programmer. Truffle does not provide dynamic invariant checks, and error recovery that yield to an *unsafe* ISA. Furthermore, EnerJ and Truffle impose excessive guardbands on the precise conventional computation due to lack of full resiliency supports. They also target single-core machines. We consider an OpenMP environment, as the de facto standard for parallel programming on shared memory multi-cores systems, to support both resiliency and configurability for accuracy. Moreover, we guarantee a controlled, thus *safe*, approximation computation leveraging both design-time and runtime techniques.

3. CONTROLLED APPROXIMATION

Approximate computation leverages the inherent tolerance of some (type of) applications within certain error bounds that are acceptable to the end application. Two metrics have been previously proposed to quantify tolerance to errors [24]: error rate and error significance. The error rate is the percentage of cycles in which the computed value of a FP operation is different from the correct value. The error significance is the numerical difference between the correct and the computed results.

Disciplined approximated programming allows programmers to identify parts of a program for approximate computation [15]. This is commonly found in applications in vision, machine learning, data analysis, and computer games. Conceptually, such pro-

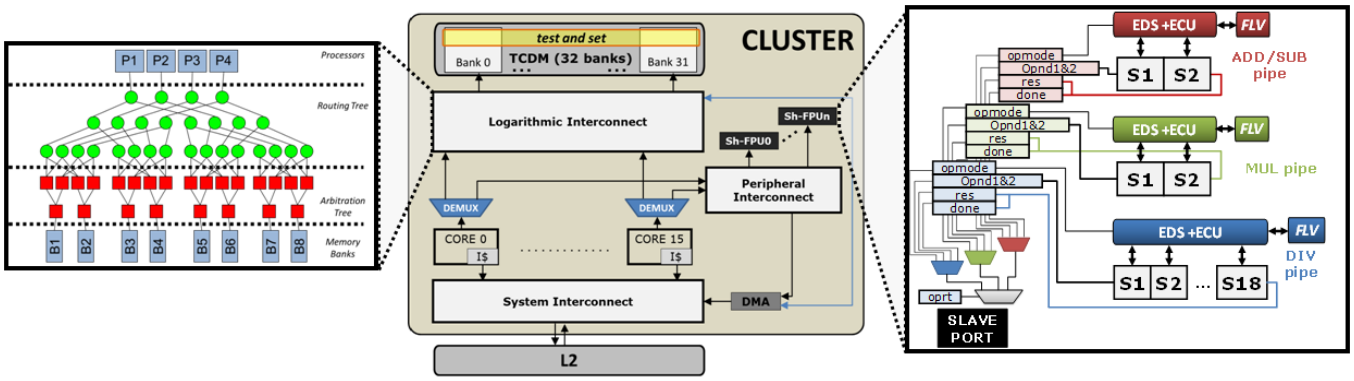

Figure 1: Variability-aware cluster architecture with shared-FPUs.

grams have a vector of 'elastic outputs' than a singular correct answer. Within the range of acceptable outputs, the program can still appear to execute correctly from the user's perspective [15],[16],[18] even if the individual computations are not exact. Programs with elastic outputs have application-dependent fidelity metrics, such as peak signal to noise ratio (PSNR), associated with them to characterize the quality of the computational result. The degradation of output quality for such applications is acceptable if the fidelity metrics satisfy a certain threshold. For example, in multimedia applications the quality of the output can be degraded but acceptable within the constraints of PSNR $\geq$ 30dB [25].

The timing error must be controllable because it could occur anytime and anywhere in the circuit. Therefore, three conditions must be satisfied to ensure that it is safe *not* to correct a timing error when approximating the associated computation:

i. The error significance is controllable and below a given threshold;

ii. The error rate is controllable and below a given error rate threshold;

iii. There is a region of the program that can produce an acceptable fidelity metric by tolerating the uncorrected, thus propagated, errors with the above-mentioned properties.

These conditions can be satisfied either through a set of profiling phases, or a set of threshold values specified by a domain *expert* via application knowledge. As we will detail in Section 5.1.2, the output information of our profiling phase is a set of threshold values that guarantee an acceptable fidelity metric. Any timing error greater than the set of thresholds triggers the recovery mechanism during the approximate operation to avoid unacceptable accuracy and undefined program behavior (e.g., in case of data-dependent control-flow), therefore guaranteeing a *safe* approximate computation.

In Section 4, we describe how we use these rules in OpenMP environment to ensure that approximate computations always deliver the required accuracy, and how they can be used for efficient hardware FPU synthesis and optimizations.

4. VARIABILITY-AWARE OPENMP ENVIRONMENT

4.1 Accuracy-Configurable Architecture

We now describe the architectural details of the proposed processing cluster with variation-tolerant accuracy-reconfigurable shared-FPUs, shown in Figure 1. The architecture is inspired by the tightly-coupled *clusters* in STMicroelectronics P2012 [20] 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 [21]. 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 (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 processors and memories (see the left part of Figure 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 (based on the AHB bus in this work). Since the TCDM has a small size (256KB) the software must explicitly orchestrate continuous data transfers from L2 to L1, to ensure locality of computations. 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.

We extend this baseline cluster architecture with our resilient shared-FPUs. Similar to the DMA, our FPU design is also controlled via memory-mapped registers, accessible through a slave port on the peripheral interconnect. The designed FPU is based on 32-bit single precision, compatible with the IEEE standard 754[1] [22], and supports addition (ADD), subtraction (SUB), multiplication (MUL), division (DIV). As shown in the rightmost part of Figure 1, the FPU has three pipeline blocks which work in parallel. The first pipeline has two stages and handles ADD and SUB operations, the second pipeline also has a latency of two cycles for MUL operation. The third pipeline has 18 stages to manipulate DIV operation. Each pipeline's inputs and outputs are retrieved from a minimal register file (one register file per pipeline to allow for parallel execution). A common write-only **`optr`** register encodes the targeted operation, and is used to select the target pipe-

[1] In this standard, each 32 bit FP number contains 1 sign bit, 8 bits as exponent, and 23 bits as fraction. The standard provides special representation for exception cases, such as infinity, not a number (NaN), etc.

line. For each pipeline there is a write-only **opmode** register that determines whether the current operation is accurate or approximate. The next registers are also write-only operand registers (**opnd1** and **opnd2**) that contain the input operands. The **res** register is read-only and stores the output of the pipeline. Finally, the last register (**done**) is also read-only and is used for synchronization with the processor, as it holds a signal that notifies pipeline execution completion.

Every pipeline block has two dynamically reconfigurable operating modes: (i) accurate, and (ii) approximate. To ensure 100% timing correctness in the accurate mode, every pipeline uses the EDS circuit sensors as well as the ECU to detect and correct any timing error due to static and dynamic delay variations [6]. Note that the area overhead of EDS and ECU is negligible (3.8% area overhead [6]). During accurate operation if a timing error is detected, the EDS circuits prevent pipeline from writing results to **res** register and thus avoid corrupting the architectural state. To recover the errant operation without changing the clock frequency, the ECU employs a multiple-issue operation replay mechanism. Prior to replaying the errant operation, the ECU initially flushes the pipeline, and reissues the errant operation multiple (M) times. The ECU sets the number of replica operations equals the number of corresponding pipeline stages (M=2 for ADD/SUB/MUL and M=18 for DIV) to ensure the register inputs for each pipeline stage are set to the appropriate value, thus guaranteeing correct execution of the valid operation (M-th operation). This recovery technique allows entire components of the cluster work at same frequency (with memories at a 180° phase shift) therefore avoiding the cost of inter-clock domain synchronization that can significantly increase communication latency. *However, this recovery mechanism incurs the energy overhead.*

In the approximate mode, the pipeline simply disables the EDS circuit sensors on the less significant N bits of the fraction where N is reprogrammable through a memory-mapped register. The sign and the exponent bits are always protected by EDS. This allows the pipeline to ignore any timing error below the less significant N bits of the fraction and save on the recovery cost. While other configurable-accuracy integer block implementations [14] power gate the error correction unit during the approximate operations, for FP pipelines with complex circuit topology, we only disable the error detection circuits partially on N bits of the fraction. This enables the FP pipeline for executing the subsequent accurate or approximate software blocks without any problem in power retention. Further, this ensures that the error significance threshold is always met, but limits the use of the recovery mechanism to those cases where the error is present on the most significant bits. To keep focus on the FPU architecture, we assume that the scalar integer cores and the memory components are resilient, for instance by utilizing the error detection and correction mechanisms [6], and tunable replica bits [5].

4.1.1 Floating-point Pipeline Vulnerability

To characterize vulnerability of every FP pipeline to the variability-induced timing error, we propose the notion of FP pipeline vulnerability (**FPV**) as a metadata. The **FPV** metadata is defined as the percentage of cycles in which a timing error occurs on the pipeline reported by the EDS sensors. To compute **FPV**, the ECU dynamically characterizes this per-pipeline metric over a programmable sampling period. The characterized **FPV** of each pipeline is visible to the software through memory-mapped registers. Thus, the runtime software scheduler leverages this characterized information for better utilization of FP pipelines, for example, it can assign fewer operations to a pipeline with higher **FPV** metadata. The runtime software scheduler can also *demote* a pipeline to the approximate mode.

We leverage this dynamic reconfiguration capability to allow the runtime scheduler to perform on-line selection of best FP pipeline candidates. This allows us to match oncoming program requests for accurate or approximate FP computation. The granularity at which a FP pipeline is configured for accurate/approximate operating mode is that of a software block, annotated by the programmer through specific language constructs (directives), as we explain in the following section. We consider eight shared FPUs integrated in our cluster. Since the number of FPUs is smaller than the number of processors, we describe our scheduling scheme in Section 4.3.

4.2 OpenMP Compiler Extension

Recently the programming model has been explored as a means to enable new opportunities for power savings [18]. The disciplined approximated programming allows programmers to identify regions of code that may be subjected to approximate computation, and are consequently tolerant to energy–accuracy trade-offs [15],[18],[19]. Applied to our architecture, FPUs under device variability are subject to timing errors, which require energy- and performance- expensive techniques to be corrected. However, the correctness of the result could be traded-off for reduced energy if the programmer took responsibility for indicating which program parts could tolerate errors as an approximation of the expected result (e.g., lower than a given error significance threshold). We provide two custom directives to OpenMP to identify approximate or accurate computations with an arbitrary granularity determined by the size of the structured block enclosed by the two custom directives:

```
#pragma omp accurate
  structured-block

#pragma omp approximate [clause]
  structured-block
```

The `approximate` directive allows the programmer to specify the tolerated error for the specific computation through an additional *clause*:

```
error_significance_threshold (<value N>)
```

The error is specified as the least significant N bits of the fraction. By default, if the programmer does not specify an error significance threshold it is assumed zero-tolerance (i.e., the `approximate` directive behaves as the `accurate`). By using this clause the `approximate` structured blocks have deterministic *fully-predictive* semantics: the maximum error significance for every FP instruction of the structured block is *bound* below the less significant N bits of the fraction. Moreover, any `approximate` instruction cannot modify any register other than its own **res** and **done** registers.

To show how the compiler transforms a region of code annotated with these directives, let us consider the code snippet for *Gaussian* smoothing filter [14],[23] in Figure 2. Here, the programmer has indicated the whole **parallel** block as an accurate computation, with the exception of the FP multiplication and accumulation of the input data. These two operations are annotated for the approximate computation with a tolerance threshold of less signif-

```
#pragma omp parallel
{
 #pragma omp accurate
 #pragma omp for
 for (i=K/2; i <(IMG_M-K/2); ++i) {
 // iterate over image
  for (j=K/2; j <(IMG_N-K/2); ++j) {
   float sum = 0;
   int ii, jj;
   for (ii =-K/2; ii<=K/2; ++ii) {
   // iterate over kernel
    for (jj = -K/2; jj <= K/2; ++jj) {
     float data = in[i+ii][j+jj];
     float coef = coeffs[ii+K/2][jj+K/2];
     float result;
     #pragma omp approximate  \
                error_significance_threshold(20)
       {
        result = data * coef;
         sum += result;
       }
    }
   }
   out[i][j]=sum/scale;
  }
 }
}
```

Figure 2: Code snippet for *Gaussian* filter utilizing OpenMP variability-aware directives.

icant 20 bits of the fraction derived from profiling phases in Section 5.1.2. The compiler transforms the **approximate** block as follows:

```
int ID = GOMP_resolve_FP (GOMP_APPROX, 20);
GOMP_FP (ID, data, coeff, GOMP_MUL, &result);
int ID = GOMP_resolve_FP (GOMP_APPROX, 20);
GOMP_FP (ID, sum, result, GOMP_ADD, &sum);
```

The first instruction generated inside a translated approximate block is a call to the **GOMP_resolve_FP** runtime library function. This API implements the variation-tolerant scheduling algorithm, described in the following section. It takes two integer parameters as inputs. The first describes the target operating mode of the FP pipeline, approximate or accurate, and the second one contains the error significance threshold value, extracted from the **error_significance_threshold** clause. As a result, this function returns a unique identifier number (ID) for the FP pipeline block. From this point, the FP pipeline will be associated to the processor that has invoked this function. The scheduler internally marks this FP pipeline resource as busy, so that no new upcoming requests could consider it for execution. Once a link to a physical FP is set, it is configured for the desired mode. The compiler also transforms statements containing a FP operation into a call to the **GOMP_FP** runtime library function. Within this function we actually program the target shared-FPU.

```
int GOMP_FP (int id, float op1, float op2,
                enum operation, float* dest)
```

A FPU programming sequence consists of three writes and two reads (not considering polling) into the memory-mapped FPU register file (see rightmost part of Figure 1). The first parameter of **GOMP_FP** is used to resolve the address of the target register file. Parameters two and three are the operands of the FP operation, while parameter four specifies which operation has to be executed. Parameter five points to the storage (variable) into which the result from the FPU is read. Before reading this output the processor polls on the **done** register to check that FPU has produced the result.

A similar transformation process is applied to **accurate** blocks, with the only difference that the **GOMP_resolve_FPU** function will be invoked with **GOMP_ACCURATE** and **"0"** as input parameters.

4.3 Runtime Support and *FPV* Utilization

The runtime library is a software layer that lies between the variation-tolerant shared-FPU architecture and the compiler-transformed OpenMP application. The goal of our variation-aware scheduler is to inspect the status of the FPUs and allocate them to **approximate** and **accurate** software blocks in such a way to reduce the overall cost of timing error correction. This is accomplished in a two-fold manner: (i) the variation-aware scheduler reduces the number of recovery cycles for **accurate** blocks by favoring utilization of FPUs with a lower **FPV**, thus lower error rate and energy; (ii) the variation-aware scheduler further reduces the cost of error correction by deliberately propagating the error toward application, thus excluding the correction cost. The latter guarantees the quality of service for **approximate** blocks by demoting FPUs to the approximate mode for ignoring errors that match the tolerance expressed via the **error_significance_threshold** clause.

To allow for quick selection of best suited devices for the accuracy target at hand, our scheduler ranks all the individual pipelines based on their **FPV**. For every type of FP operations (ADD, SUB, MUL, DIV), the scheduler reads the corresponding characterized **FPV**, and then sorts all the pipelines by increasing **FPV** across FPUs. The sorted list is maintained in the shared TCDM, to make it visible to all the cores and accessible with low latency. The **FPV** for every FPU could be statically pre-determined (e.g., during a profile run), but in general when the program starts such information may not be available. In this case FPUs are simply scheduled with a round-robin policy, but our system performs online characterization in the background to dynamically collect **FPV** signatures for every FPU.

Once this information is available in the sorted lists, the scheduler can optimize FPU allocation for **accurate** software blocks. This is implemented in the **GOMP_resolve_FP** function.

```
GOMP_resolve_FP (int opmode, int thresh)
```

Within this function, once the operating mode has been determined (the **opmode** parameter), as a first step the scheduler locks the sorted list structure, to prevent inconsistencies due to concurrently executing **accurate** or **approximate** blocks, then it traverses the list, starting from the head, until it finds an available pipeline. Once the target FP pipeline has been identified, it is configured to the desired **opmode** on-the-fly, and its ID is returned to the application for offloading the consecutive FP instruction. This configurability partially enables/disables the error detection on the less significant *N* bits of the fraction determined through the **error_significance_threshold** clause. Consequently, the FP pipeline is able to detect and correct any timing error if it is reconfigured for the accurate mode; on the other hand, in the approximate mode the FP pipeline ignores any timing error on the less significant *N* bits of the fraction. Using

these sorted lists, for every type of FP operations the ranking algorithm tries to highly utilize those pipelines with a lower **FPV** (and rarely allocate operations to the pipelines at the end of list), thus the aggregate recovery cycles for execution of FP operations will be reduced. Figure 3 illustrates the ranking (RANK) algorithm.

When handling requests for approximate FPU resources, the pipeline selection phase can have an additional check to assure efficiency of approximate executions. If a FP pipeline displays a high error rate, i.e., a **FPV** close to one, it might not be a suitable candidate for the approximation execution, mainly because there is a high probability that a timing error could also happen in the more significant bits. In this case, the FP pipeline enforces the cost of recovery which wipes out the benefit of the relaxed approximate execution. To avoid this situation, the scheduler can selectively *virtualize K* number of FP pipelines (with a low **FPV**) among all available FP pipelines, for every type of operations. In this reactive technique, two (or more) OpenMP-visible *virtual* FP instructions must share a single physical FP. This is implemented by determining the end point of the sorted list through specifying the error rate threshold. When the error rate threshold is specified the scheduler limits its search for the approximate operations until a certain element of the sorted list, e.g., in Figure 3 until *K*-th pipeline. As soon as the scheduler finds a pipeline which has a higher **FPV** than the error rate threshold, it marks it as the virtual end point of the list for the approximate operations. Therefore, for the following approximate requests, the scheduler starts from the start point of the sorted list, and traverses down toward the virtual end point of the corresponding sorted list for finding a free pipeline. However, this virtualization technique limits the available parallelism discussed in the Section 5.

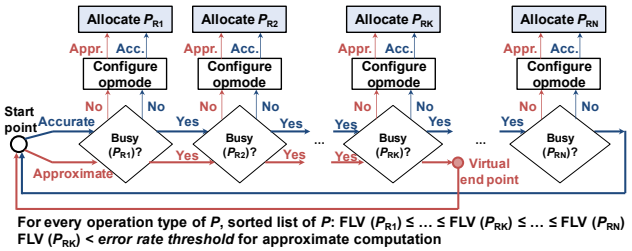

Figure 3. RANK scheduling based FPV of FP pipelines.

4.4 Application-Driven Hardware FPU Synthesis and Optimization

In the earlier sections, we describe the three essential components of our variability-aware OpenMP environment: the language directive extensions, the compiler and runtime support, and the accuracy-configurable architecture. In this section, we introduce an optional yet effective methodology to generate efficient hardware FPU. The design flow should be done by choosing a threshold that is acceptable on a wide class of application, and if an application cannot tolerate this type of inaccuracy, the runtime system must reconfigure architecture to the accurate mode. We couple the proposed methodology with the application tolerable error analysis presented in Section 3. As we have mentioned earlier, the output information of the profiling phase is two threshold values, i.e., the error significance threshold and the error rate threshold, that guarantee the acceptable fidelity metric (in our case: PSNR ≥ 30dB). This information is utilized during design-time flow for synthesis and optimization of hardware FPUs; Figure 4 illustrates the proposed methodology.

The error significance threshold indicates that any timing error below the bit position of e.g., *N* can be ignored since it will not induce large deviations from the corrected value. This means for the approximate computation the only important parts are the bit positions higher than *N* since any timing error on these bits have to be corrected to guarantee the acceptable fidelity metric. Therefore, an efficient FPU for the approximate mode should eliminate the possibility of any timing error on the high order bits, while relaxing this constraint on the low order bits. At the same time they should not be too relaxed, to avoid the generation of many errors that have to be recovered in the accurate mode. Consequently, a set of tight timing constraints is generated to guide the hardware synthesis and optimization flow for providing fast paths connected to the high order bits (thus the lower delay, and the lower probability of timing errors). The synthesis CAD tool meets these constraints by utilizing fast leaky standard cells (low-V_{TH}) for the paths with the tight timing constraint, while utilizing the regular and slow standard cells (regular-V_{TH} and high-V_{TH}) for the rest of paths. As a result, the new generated hardware FPU will experience a lower probability of the timing error on the bit positions higher than *N*, at the power expense of higher leaky cells.

We have applied the proposed methodology to optimize the netlist of the shared-FPUs. The approximation-aware timing constraints try to deliver fast paths connected to bit position of `20` up to `32`. As a result, the optimized shared-FPUs experience lower timing error rate; compared to the non-optimized shared-FPUs, the total recovery cycles are reduced by 46% and 27% in the accurate and approximate modes, respectively. On the other hand, the total power overhead of the optimized shared-FPUs is 16% in comparison with the non-optimized shared-FPUs (19% overhead in leakage power). However, this power overhead is highly compensated because the optimized shared-FPUs spend smaller number of clock cycles to compute the same amount of work. Experimental results in Section 5.1.3 quantify the energy benefit of this proposed methodology.

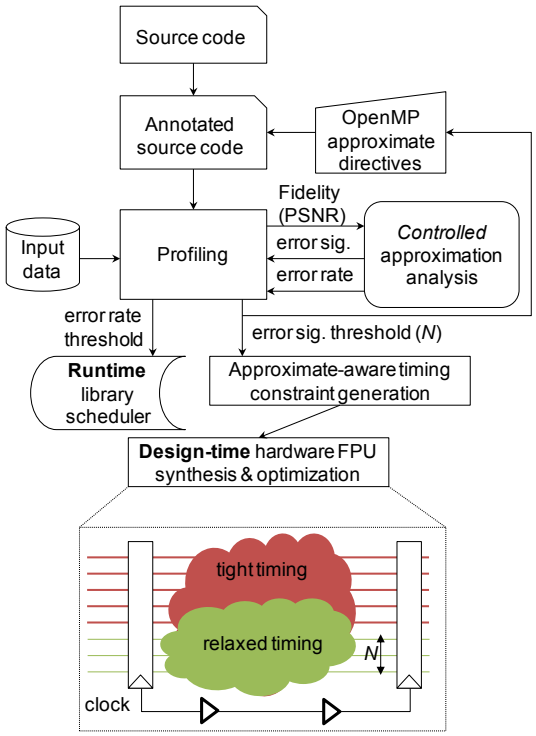

Figure 4: Methodology for application-driven hardware FPU synthesis and optimization.

The proposed optimization methodology is based on either designer knowledge (provided from a domain expert), or static profiling (derived from the fidelity metric and error analysis). We should note that the static profiling is a common technique for approximate computation analysis [19],[23]. However, our methodology takes advantage of the maximum allowable error significance at design-time, while the error detection and correction circuits embedded in FPUs are responsible to dynamically handle any non-maskable timing error.

5. EXPERIMENTAL RESULTS

We demonstrate our approach on an OpenMP-enabled SystemC-based virtual platform for on-chip multi-core shared-memory clusters with hardware accelerators [27]. Table I summarizes the architectural parameters. A cycle-accurate SystemC model of the shared-FPUs is also integrated to the virtual platform, which enables the variability-affected emulation. To accurately emulate the low-level device variability on the virtual platform, we have integrated the variability-induced error models at the level of individual FP pipelines using the instruction-level vulnerability characterization methodology presented in [10]. The RTL description of shared-FPUs are generated and optimized by FloPoCo [28], an arithmetic FP core generator of synthesizable VHDL. Then, the shared-FPUs have been synthesized for TSMC 45nm technology, the general purpose process. The front-end flow with multi V_{TH} cells has been performed using *Synopsys Design Compiler* with the topographical features, while *Synopsys IC Compiler* has been used for the back-end. The design has been typically optimized for timing to meet the signoff frequency of 625MHz at (SS/0.81V/125°C).

Next, we have analyzed the delay variability of the shared-FPUs under process and temperature variations. First, to observe the effect of static process variation on the eight shared-FPUs, we have analyzed how the critical paths of each pipeline are affected due to within-die and die-to-die process parameters variation. Therefore, the various pipelines within the FPUs experience different variability-induced delay and thus display various error rate. During the sign-off stage, we have injected process variation in the shared-FPUs using the variation-aware timing analysis engine of *Synopsys PrimeTime VX* [29]. It utilizes process parameters and distributions of 45nm variation-aware TSMC libraries [30] derived from first-level process parameters by principal component analysis. Second, to observe the effects of temperature variations, we employ voltage-temperature scaling feature of *Synopsys PrimeTime* to analyze the delay and power variations under temperature fluctuations. Finally, the variation-induced delay is back-annotated to the post-layout simulation to quantify the error rate of individual pipelines. For every back-annotated variation scenarios, the FP pipelines are characterized with a representative random set of 10^7 inputs, automatically generated by FloPoCo. Finally, these error rate models are integrated to the corresponding modules in the SystemC virtual platform to emulate variability.

Table I. Architectural parameters of shared-FPUs cluster.

ARM v6 core	16	TCDM banks	16
I$ size(per core)	16KB	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
Shared-FPUs	8	FP ADD latency	2
FP MUL latency	2	FP DIV latency	18

5.1 Error-tolerant Applications

In this section we evaluate the effectiveness of the proposed variability-aware OpenMP environment under the process variability for the error-tolerant image processing applications. For benchmark, we consider two widely-used image processing applications as the approximate programs: *Gaussian* smoothing filter [14],[23], and *Sobel* edge detection algorithm [26].

5.1.1 Execution without Approximation Directives

For the first experiments, we marked the entire program for accurate computation (`#pragma omp accurate`), representative of what a non-expert programmer would achieve without application profiling, tuning, and code annotation. Later, we show how these applications can benefit from the approximate code annotation. We have compared the proposed ranking scheduling (RANK) with the baseline round-robin scheduling (RR) in terms of FP energy and total execution time. The RR algorithm assigns the FP operations to the pipelines in the order they become available, while RANK utilizes the sorted list structure of the **`FPV`**. Figure 5 shows the shared-FPU energy and total execution time for the target applications for RANK normalized to the baseline RR algorithm. Each bar (or point) indicates the normalized shared-FPUs energy (or the total execution time) for a set of different input sizes.

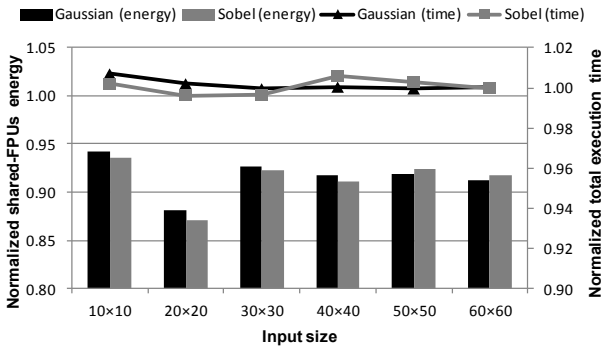

Figure 5. Energy and execution time of RANK scheduling (normalized to RR) for accurate *Gaussian* and *Sobel* filters.

As shown, the RANK algorithm achieves up to 12% lower energy for the shared-FPU compared to RR algorithm, while the maximum timing penalty is less than 1%. This energy saving is achieved by leveraging the characterized **`FPV`** metadata and the sorted list data structure that enable high utilization of those pipelines that display lower error rates. Consequently, it reduces the total recovery cycles, and energy. Moreover, the total timing overhead of the RANK is minimal, and the overhead for sorting and searching among eight shared-FPUs is highly amortized. These low cost features are accomplished through the advantages of fast TCDM, carefully placing the key data structures in TCDM, and the low-latency logarithmic interconnection.

5.1.2 Profiling Error-tolerant Applications

In this section we present the profiling phases for producing useful threshold information to enable approximate computation. We analyze the manifestation of a range of error significance and error rate on the PSNR of the two image processing applications. We have annotated the approximable regions of the application codes using the proposed OpenMP custom directives (the code snippet for the *Gaussian* filter is shown in Figure 2). The annotated approximate regions of both applications are only composed of FP addition and multiplication operations. We quantify how much

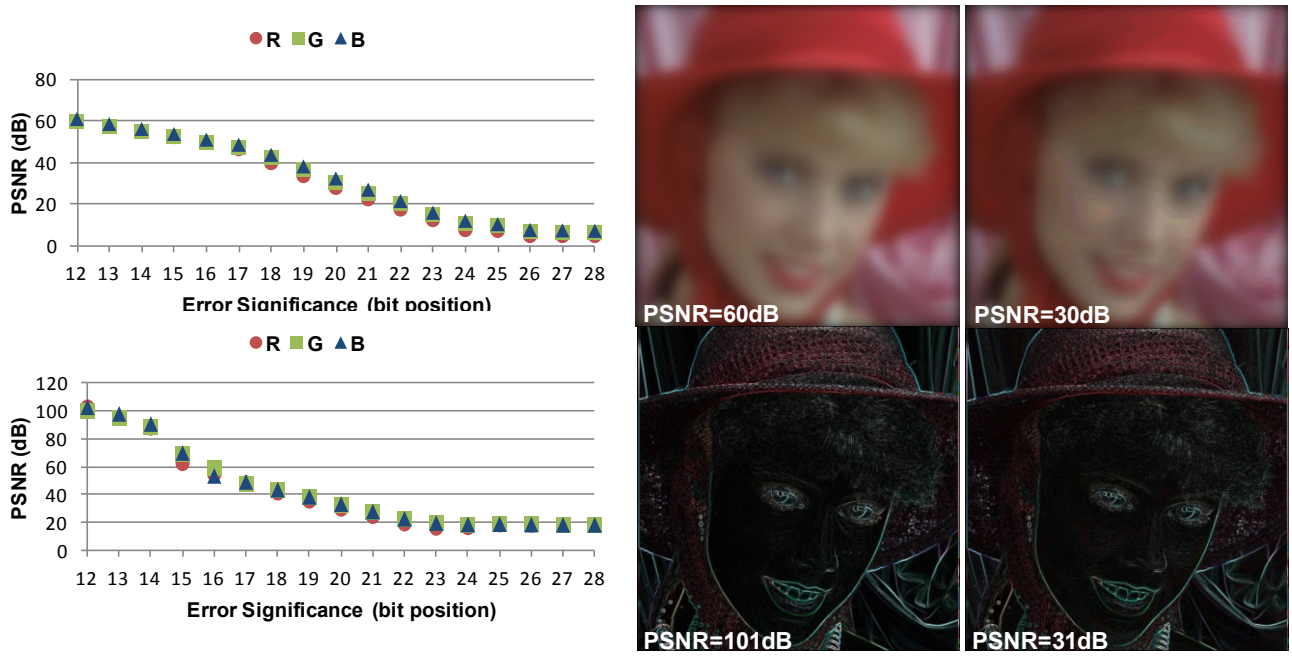

Figure 6: PSNR degradation as a function of error significance: a) for *Gaussian* filter (top); b) for *Sobel* filter (bottom).

error significance can be tolerated in these approximate regions, given a maximum error rate. To do so, we have profiled the annotated approximate regions of the programs. In a series of profiling, we have monotonically increased the error significance by injecting the timing errors as random multiple-bit toggling up to a certain bit position of the FP single precision representation. The position of multiple-bit toggling is varied from `1` to `28`, for a wide range of `1%` error rate to `100%`.

Figure 6 illustrates results for the error rate of `100%`, i.e., every addition and multiplication operation in the FP approximate regions has an errant output depending up on the injected error significance. Figure 6.a shows the PSNR degradation of output image of the *Gaussian* filter as a function of the error significance. As shown, the three channels of RGB color space, experience similar PSNR degradations by increasing the error significance. Figure 6.b also illustrates the similar trend for the *Sobel* filter. The rightmost part of Figure 6 shows that this degradation of the quality is acceptable from the user's perspective. In summary, the output information of these profiling indicates that for a given error rate of `{100%, 50%, 25%}` if the timing error lies within the bit position of `0` to `{20, 21, 22}` of the fraction part, these two applications can tolerate the timing error by delivering a PSNR of greater than 30dB. This information is essential not only during runtime to intentionally ignore the tolerable timing errors, but also for efficient hardware FPU synthesis and optimizations, detailed in the following section.

Therefore, for the approximate regions of these applications, we have set the error rate threshold to `100%`, and the error significance threshold to `20` to maintain the acceptable PSNR. By setting the threshold of the error rate to `100%`, during the runtime execution of the approximate regions all FPUs can be utilized. This is important in data-parallelized image applications where there is enough parallelism, and especially so when the number of FPUs is lower than the number of the cores and any time-multiplexing might incur performance degradation.

5.1.3 Execution with Approximation Directives

Now, let us quantify the benefit of the approximate computation using the information of the profiling. Since the RANK scheduling algorithm surpasses the baseline RR algorithm, for the rest of results we have used the RANK algorithm. We have repeated the experiments in Section 5.1.1, but for two variants of the applications code. In the first version, the programs are entirely composed of the accurate FP operations, and the in the second version the programs utilize the approximate ADD and MUL operations in the annotated regions of code.

Figure 7 shows the total shared-FPUs energy for these two versions of the programs with different input sizes. The first group of bars shows the energy of the shared-FPUs for the accurate programs, while the second group of bars refers to the approximate programs. For example, with an input size of 60×60, the shared-FPUs consume 3.5µJ (or 4.6µJ) for the accurate *Gaussian* (or *Sobel*) program, while execution of the approximate version of the program reduces the energy to 2.8µJ (or 3.5µJ), achieving 24% (or 30%) energy saving. This energy saving is achieved by ignoring the timing error within the bit position of `0` to `20` of the fraction part. The next two bars show the energy of an optimized hardware implementation of the shared-FPUs, discussed in the following.

To generate the efficient FPUs suitable for these applications we leveraged the hardware FPU synthesis and optimization methodology proposed in Section 4.4. Therefore, the application-driven timing constraints guide the CAD flow to selectively optimize timing of the desired paths. Figure 7 also shows the energy differences between the non-optimized and optimized FPUs in the two operating modes. On average, compared to the non-optimized shared-FPUs, the optimized shared-FPUs achieves 25% and 7% lower energy for the accurate and approximate modes, respectively. Overall, utilization of the annotated programs with the approximate directives on top of the optimized shared-FPUs achieves an average energy saving of 36%.

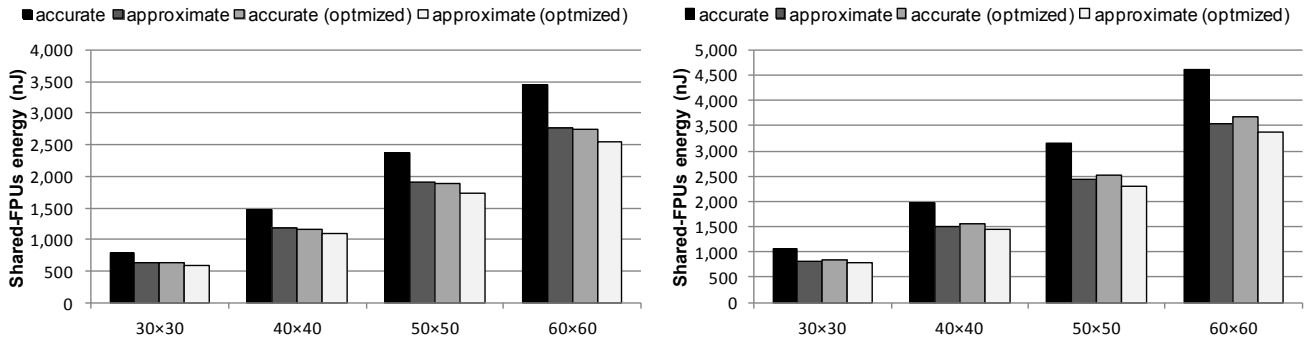

Figure 7: FP energy of accurate and approximate programs for non-optimized and optimized hardware shared-FPUs.

5.2 Error-intolerant Applications

Using the concept of configurable accuracy as discussed earlier, we now show that the proposed variability-aware OpenMP environment not only facilitates efficient execution of the approximate programs, but also reduces the cost of recovery for the error-intolerant general-purpose applications. We have evaluated the effectiveness of our proposed approach in the presence of process variability under operating temperature fluctuations for five error-intolerant applications: three widely used 2-D computational kernels (matrix multiplication, matrix addition with scalar multiplication, and DCT), Monte Carlo kernel, and image conversion kernel (HSV2RGB).

Figure 8 shows the shared-FPUs energy saving of these application compared to the worst-case design. For these experiments, we consider 25% voltage overdesigned for the baseline FPUs which can guarantee their error-free operations [15]. On average 22% (and up to 28%) energy saving is achieved at the operating temperature of 125°C, thanks to allocating the FP operations to the appropriate pipelines. As shown, this saving is consistent (20%-22% on average) across a wide temperature range (ΔT=125°C), thanks to the online **FPV** metadata characterization which reflects the latest variations, thus enabling the scheduler to react accordingly. The lower temperature leads to a higher delay in the low-voltage region of nanometer CMOS technologies [31], thus the higher error rate and the more energy for recovery. Please note that after having the ranked pipelines tables on TCDM, we rarely need to re-execute the sorting algorithm unless we sense a temperature fluctuation which has a slow timing-constant.

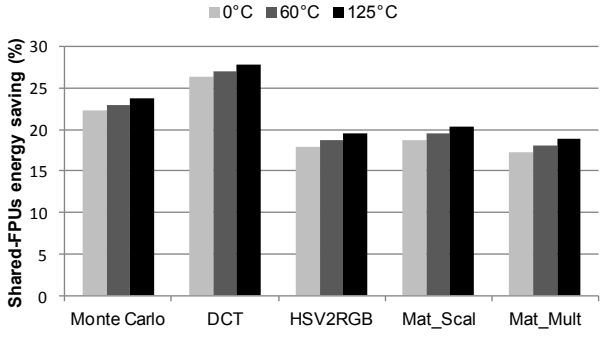

Figure 8. Shared-FPUs energy saving for the error-intolerant applications compared to the worst-case design.

5.3 Comparison with Truffle

We compare our proposed environment with Truffle. Truffle, as a single core architecture, duplicates all the functional units in the execution stage. Half of them are hardwired to Vdd_{High} (to execute the accurate operations), while the other half operate at Vdd_{Low} (to execute the approximate operations). To have an iso-area comparison with Truffle, as it is suggested in their paper, we assume that Truffle uses dual-voltage FPUs and changes the voltage depending on the instruction being executed. This would also save the static power. To have a fair comparison, we also assume that Truffle employs a fast Vdd-hopping technique to switch between Vdd_{High} and Vdd_{Low}. Among the Vdd-hopping implementation techniques [32]-[34], Beigne *et al.* propose a Vdd-hopping unit with voltage transitions less than 100ns [32]. Kim *et al.* also propose fast on-chip voltage regulators with transitions time of 15ns−20ns [34], thus we consider this transition time and optimistically augment a latency of 10-cycle for switching FPUs between the accurate and approximate modes. We apply Truffle limitation to our virtual platform cluster to quantify its energy.

For comparison, we consider two application scenarios: (i) once the cluster is executing only one approximate application; (ii) simultaneous execution of one approximate application with one accurate application. In the former scenario, entire 16 cores of the cluster cooperatively execute one of the approximate image applications, while in the latter scenario, eight cores execute the approximate *Gaussian* filter and the other eight core execute the accurate matrix multiplication, simultaneously. Figure 9 compares the shared-FPUs energy of Truffle with our proposed approach when executing the above two scenarios. As shown, our proposed approach surpasses Truffle in the both applications scenarios. In the former scenario, on average, our approach saves 20% more energy compared to Truffle by reducing the conservative voltage overdesigned for the accurate part of filters application. For the mixed scenario of the applications, our approach saves 36% more energy, since Truffle highly faces with the overhead of frequent switching between the accurate and approximate modes which is imposed by *interference* of the accurate and approximate operations resulting from the concurrent execution of *Gaussian* and matrix multiplication applications.

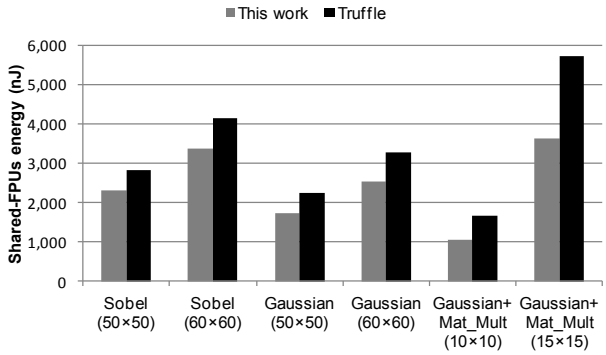

Figure 9. Energy comparison with Truffle: (i) only approximate ; (ii) concurrent approximate and accurate applications.

6. CONCLUSION

We propose an OpenMP programming environment that is resilient to variability-induced timing errors and suitable for fine-grained interleaved approximate and accurate computation on shared-FPUs processor clusters. This is orchestrated through a vertical abstraction of circuit-level variations into a high-level parallel software execution. The OpenMP extensions help a programmer specify accurate and approximate FP parts of a program. The underlying architecture features a set of shared-FPUs with two *sensing* and *actuation* primitives; every FPU dynamically senses the timing errors, characterizes its own **`FPV`** metadata, and can be configured to operate in the approximate or accurate modes. The runtime scheduler utilizes the sensed **`FPV`** metadata, and parsimoniously actuates depending upon the code region requirements on the computational accuracy. These three components in the proposed environment support a controlled approximation computation through various design-time phases (applications profiling, and FPU synthesis & optimization) in combination with runtime sensing and actuation primitives. Either the environment deliberately ignores the otherwise expensive timing error correction in a fully controlled manner, or it tries to reduce the frequency of timing errors.

For general-purpose error-intolerant applications, our approach reduces energy up to 28%, across a wide temperature range (ΔT=125°C), compared to the worst-case design. For error-tolerant image processing applications with the annotated approximate directives, on average, 36% energy saving is achieved while maintaining the PSNR > 30dB. In comparison with the state-of-the-art architecture [15], our approach saves 36% more energy when executing finely interleaved mixture of FP operations.

7. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation's Variability Expedition in Computing under award n. 1029783, Virtical GA n. 288574, and ERC-AdG MultiTherman GA n. 291125.

8. REFERENCES

[1] P. Gupta, et al., "Underdesigned and Opportunistic Computing in Presence of Hardware Variability," *IEEE Trans. on CAD of Integrated Circuits and Systems* 32(1) (2012), pp. 489-499.

[2] J. Henkel, et al., "Design and architectures for dependable embedded systems," *Proc. ACM/IEEE CODES+ISSS*, 2011, pp. 69-78.

[3] ITRS [Online]. Available: http://public.itrs.net

[4] K.A. Bowman, et al., "Energy-Efficient and Metastability-Immune Resilient Circuits for Dynamic Variation Tolerance," *IEEE Journal of Solid-State Circuits* 44(1) (2009), pp. 49-63.

[5] A. Raychowdhury, et al., "Tunable Replica Bits for Dynamic Variation Tolerance in 8T SRAM Arrays," *IEEE Journal of Solid-State Circuits* 46(4) (2011), pp.797-805.

[6] K.A. Bowman, et al., "A 45 nm Resilient Microprocessor Core for Dynamic Variation Tolerance," *IEEE Journal of Solid-State Circuits* 46(1) (2011), pp. 194-208.

[7] M.R. Kakoee, I. Loi, L. Benini, "Variation-Tolerant Architecture for Ultra Low Power Shared-L1 Processor Clusters," *IEEE Trans. on Circuits and Systems II* 59(12) (2012), pp.927-931.

[8] Technical Reference Manual, ARM Cortex-A9, rev.: r2p2.

[9] AMD "Bulldozer" Core Technology [online]. Available: http://www.sgi.com/partners/technology/downloads/ADM_Bulldozer_Core_Technology.pdf

[10] A. Rahimi, L. Benini, R. K. Gupta, "Analysis of Instruction-level Vulnerability to Dynamic Voltage and Temperature Variations," *Proc. ACM/IEEE DATE*, 2012, pp. 1102-1105.

[11] A. Rahimi, L. Benini, R. K. Gupta, "Application-Adaptive Guardbanding to Mitigate Static and Dynamic Variability," *IEEE Transactions on Computers*, 2013.

[12] A. Rahimi, L. Benini, R. K. Gupta, "Procedure hopping: a low overhead solution to mitigate variability in shared-L1 processor clusters," *Proc. ACM/IEEE ISLPED*, 2012, pp. 415-420.

[13] A. Rahimi, A. Marongiu, P. Burgio, R. K. Gupta, L. Benini, "Variation-tolerant OpenMP Tasking on Tightly-coupled Processor Clusters," *Proc. ACM/IEEE DATE*, 2013, pp. 541-546.

[14] A. B. Kahng, S. Kang, "Accuracy-Configurable Adder for Approximate Arithmetic Designs," *Proc. ACM/IEEE DAC*, 2012, pp. 820-825.

[15] H. Esmaeilzadeh, A. Sampson, L. Ceze, D. Burger, "Architecture Support for Disciplined Approximate Programming," *Proc. ACM ASPLOS*, 2012, pp. 301-312.

[16] L. Leem, et al., "ERSA: Error Resilient System Architecture for probabilistic applications," *Proc. ACM/IEEE DATE*, 2010, pp. 1560-1565.

[17] 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 Journal of Solid-State Circuits* 46(1) (2011), pp. 184-193.

[18] A. Sampson, et al., "EnerJ: Approximate data types for safe and general low-power computation," *Proc. ACM PLDI*, 2011, pp. 164-174.

[19] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," *Proc. ACM PLDI*, 2010, pp. 198-209.

[20] L. Benini, E. Flamand, D. Fuin, D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," Proc. *ACM/IEEE DATE*, 2012, pp. 983-987.

[21] A. Rahimi, I. Loi, M.R. Kakoee, L. Benini, "A Fully-Synthesizable Single-Cycle Interconnection Network for Shared- L1 Processor Clusters," *Proc. ACM/IEEE DATE*, 2011, pp. 1-6.

[22] IEEE Computer Society (1985), IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std 754, 1985.

[23] M. S. Lau, et al. "Energy-Aware Probabilistic Multiplier: Design and Analysis", *Proc. ACM/IEEE CASES*, 2009, pp. 281-290.

[24] M. A. Breuer, "Intelligible Test Techniques to Support Error-Tolerance", *Proc. Asian Test Symp.*, 2004, pp. 386-393.

[25] Barni, Mauro, "Document and image compression," CRC Press, May 2006, pp. 168-169.

[26] H. Esmaeilzadeh, A. Sampson, L. Ceze, D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," *Proc. ACM/IEEE MICRO*, 2012, pp. 449-460.

[27] P. Burgio, et al., "OpenMP-based synergistic parallelization and HW acceleration for on-chip multi-core shared-memory clusters," *Proc. ACM/IEEE DSD*, 2012, pp. 751-758.

[28] FloPoCo [Online]. Available: http://flopoco.gforge.inria.fr/

[29] PrimeTime® VX User Guide, June 2011.

[30] TSMC 45nm standard cell library release note, Nov. 2009.

[31] R. Kumar, V. Kursun, "Reversed Temperature-Dependent Propagation Delay Characteristics in Nanometer CMOS Circuits," *IEEE Transactions on Circuits and Systems* 53(10) (2006), pp.1078-1082.

[32] E. Beigne, et al., "An Asynchronous Power Aware and Adaptive NoC Based Circuit," *IEEE J. of Solid-State Circuits* 44(4) (2009).

[33] A. Rahimi, M. E. Salehi, S. Mohammadi, S. M. Fakhraie, "Low-energy GALS NoC with FIFO-monitoring dynamic voltage scaling," *Microelectronics Journal* 42(6) (2011), pp. 889-896.

[34] W. Kim, D.M. Brooks, G. Wei, "A fully-integrated 3-level DC/DC converter for nanosecond-scale DVS with fast shunt regulation," Proc. *IEEE ISSCC*, 2011, pp.268-270.