SHASTA: Synergic HW-SW Architecture for Spatio-temporal Approximation

GOKUL SUBRAMANIAN RAVI, JOSHUA SAN MIGUEL, and MIKKO LIPASTI, University of Wisconsin Madison, Electrical and Computer Engineering

A key requirement for efficient general purpose approximate computing is an amalgamation of flexible hardware design and intelligent application tuning, which together can leverage the appropriate amount of approximation that the applications engender and reap the best efficiency gains from them. To achieve this, we have identified three important features to build better general-purpose cross-layer approximation systems: ① individual per-operation ("spatio-temporally fine-grained") approximation, ② hardware-cognizant application tuning for approximation, ③ systemwide approximation-synergy.

We build an efficient general purpose approximation system called SHASTA: Synergic HW-SW Architecture for Spatio-Temporal Approximation, to achieve these goals.[1] First, in terms of hardware, SHASTA approximates both compute and memory—SHASTA proposes (a) a form of timing approximation called Slack-control Approximation, which controls the computation timing of each approximation operation and (b) a Dynamic Pre-L1 Load Approximation mechanism to approximate loads prior to cache access. These hardware mechanisms are designed to achieve fine-grained spatio-temporally diverse approximation. Next, SHASTA proposes a Hardware-cognizant Approximation Tuning mechanism to tune an application's approximation to achieve the optimum execution efficiency under the prescribed error tolerance. The tuning mechanism is implemented atop a gradient descent algorithm and, thus, the application's approximation is tuned along the steepest error vs. execution efficiency gradient. Finally, SHASTA is designed with a full-system perspective, which achieves Synergic benefits across its optimizations, building a closer-to-ideal general purpose approximation system.

SHASTA is implemented on top of an OOO core and achieves mean speedups/energy savings of 20%–40% over a non-approximate baseline for greater than 90% accuracy—these benefits are substantial for applications executing on a traditional general purpose processing system. SHASTA can be tuned to specific accuracy constraints and execution metrics and is quantitatively shown to achieve 2–15× higher benefits, in terms of performance and energy, compared to prior work.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Hardware** → **Logic circuits**; • **Computing methodologies** → *Machine learning*;

Additional Key Words and Phrases: Approximate computing, general purpose systems, timing approximation, load approximation, approximation tuning, gradient descent

[1]New article, not an extension of a conference paper.

Authors' addresses: G. S. Ravi, J. S. Miguel, and M. Lipasti, University of Wisconsin Madison, Electrical and Computer Engineering, 1415 Engineering Dr, Madison, WI 53706; emails: {gravi, jsanmiguel}@wisc.edu, mikko@engr.wisc.edu.

1544-3566/2020/09-ART25
https://doi.org/10.1145/3412375

ACM Reference format:
Gokul Subramanian Ravi, Joshua San Miguel, and Mikko Lipasti. 2020. SHASTA: Synergic HW-SW Architecture for Spatio-temporal Approximation. *ACM Trans. Archit. Code Optim.* 17, 4, Article 25 (September 2020), 26 pages.
https://doi.org/10.1145/3412375

1 INTRODUCTION

Workloads from several prevalent and emerging application domains, such as vision, machine learning, and data analytics, possess the ability to produce outputs of acceptable quality in the presence of inexactness or approximations in a large fraction of their underlying computations [40]. Allowing computations to be approximate can lead to significant improvements in processor efficiency, because it alleviates the "correctness tax" [10] imposed by always accurate systems. This "correctness tax" can take multiple forms. In software, this could refer to overzealous functional accuracy or excessive loop iterations whose execution has low impact on the application's accuracy [35]. In hardware, it could refer to the higher power/latency of accurate compute operations or the effectively lower throughput stemming from full bit-width memory operations.

The mainstream adoption of approximate computing and its use in a broader range of applications is predicated upon the creation of programmable platforms for approximate computing [40]. The key requirement for efficient general purpose approximate computing is an amalgamation of (i) general purpose hardware designs flexible enough to leverage any amount/type of approximation that an application engenders and (ii) tuning mechanisms intelligent enough to reap the optimum efficiency gains (in terms of performance and energy) given a hardware design, an application, and a resiliency specification. In this regard there are many existing limitations and opportunities in prior proposals, both in terms of general purpose hardware designs as well as in terms of approximation tuning mechanisms (which are discussed in Section 2 and Section 8). In this article, we propose SHASTA, a cross-layer solution for building optimal General Purpose Approximation Systems (GPAS), i.e., approximation systems suited to a wide range of general purpose error-tolerant applications, each with unique and potentially fine-grained approximation characteristics. SHASTA tackles the opportunities discussed above to achieve significant benefits in execution efficiency. We classify the goals for building optimal GPAS and how SHASTA addresses them into three broad categories below.

1.1 Hardware Enabling Spatio-temporal Diversity at Fine Granularity

Challenge: An ideal GPAS should provide the flexibility to control each executing operation uniquely, as accurate or approximate. Moreover, each approximate computation should be ideally allowed to have its own individual/unique amount of approximation. This approximation diversity can be thought of as two components—spatial and temporal. Spatial diversity in approximation implies that each static approximate operation (i.e., every approximate variable in an approximate application's code) should be allowed unique approximation control. Temporal diversity in approximation implies every dynamic instance of static approximation operations (for example, every iteration of an approximate variable) should be allowed different approximations, i.e., the approximation applied to the static operation should be allowed to evolve over time (say, across the loop iterations). Further, the system should be able to dynamically reconfigure these approximation settings with low overhead. While such flexibility is easier to explore in software, fine granularities and spatio-temporal diversity of approximation is more challenging in hardware.

SHASTA's hardware approximation: For compute approximation, SHASTA proposes a new variant of Timing approximation called *Slack-control Approximation*. *Slack-control Approximation* is inspired by REDSOC [32], a previously proposed clock-cycle slack recycling that targets

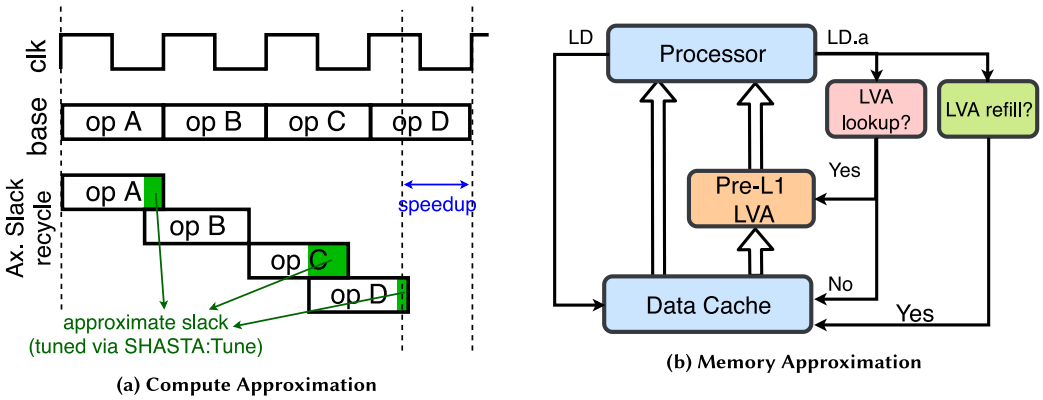


Fig. 1. Hardware approximation.

accurate computing wherein the computation timing of operations is controlled at sub-clock-cycle granularities and integral clock cycles worth of latency are saved when sufficient slack accumulates over sequences of operations. SHASTA extends REDSOC ideas to approximate computing, performing more aggressive slack recycling and achieving a flexible approximation scheme that can be controlled on a per-clock-cycle and per-computation-unit basis, thus allowing fine granularity and spatio-temporal diversity. While mechanism specifics are detailed in Section 4.1, an overview of SHASTA's compute approximation is shown in Figure 1(a). The following points can be noted: First, approximation is achieved by reducing computation time, allowing a sequence of operations to be completed faster than the accurate baseline, resulting in speedup. Second, not all computations in the sequence have to be approximate. Third, different approximate computations are allowed varying degrees of approximation (hence, varying amounts of slack).

For memory approximation, SHASTA implements *Dynamic Pre-L1 Load Approximation*. Inspired by LVA [19] but going beyond, this technique approximates loads prior to L1 cache access by reading values out of a small approximator that is integrated close to the datapath. This results in latency and energy savings, an approximate value is used instead of having to read the data from the L1 cache, or worse from lower-level caches or memory. An overview is shown in Figure 1(b). The key advancement to prior work is the ability to perform fine-grained spatio-temporally diverse approximation. Our implementation enables the control of each unique load's approximation degree (how often the data in the approximator are refilled) and approximation confidence (how often the data are approximated, i.e., looked up in the approximator) independent of other loads. This allows more efficient use of load approximation—creating better fine-grained trade-offs between error and efficiency.

As highlighted above, both of these techniques enable fine-grained spatio-temporally diverse approximation suited to general purpose computing systems, which is largely unachievable in prior work. Compute and memory approximation are discussed in Section 4.

1.2 Automated Hardware-cognizant Approximation Tuning

Challenge: We call amount of approximation assigned to each approximation operation as the *approximation configuration* of the application. An ideal GPAS requires an intelligent tuning mechanism to identify any given application's optimum approximation configuration. This is especially important in systems allowing fine granularities of spatio-temporal approximation diversity wherein there is potentially considerable efficiency difference between optimal and sub-optimal configurations.

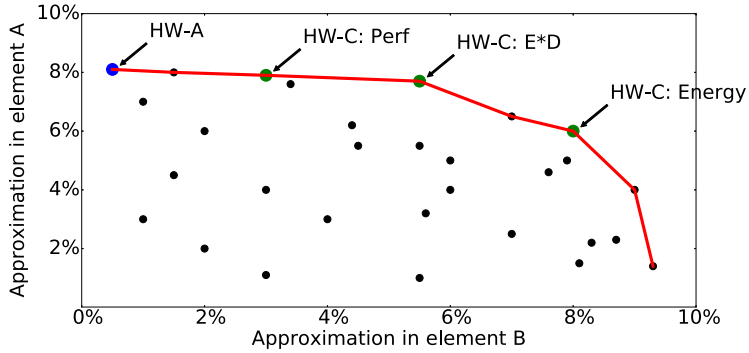


Fig. 2. Hardware-cognizant approximation tuning.

Figure 2 shows an example micro-application with two approximate elements. The data points on the scatter plot show various approximation configurations. The points falling on the red line are those configurations satisfying the application's specified error tolerance (for example, 90% accuracy). Among these, the blue dot (HW-A) is the configuration selected by a hardware-agnostic mechanism (akin to References [16, 20, 22, 40]), which optimizes towards application's error tolerance in a pre-specified order: in this example, first on element A and then on element B. This configuration can be sub-optimal, because the tuning mechanism is not taking into account the impact of this configuration on the hardware's execution efficiency.

SHASTA's approximation tuning: To our knowledge, in the GPAS domain, especially targeting fine-grained and diverse approximation, SHASTA is the first proposal to actively consider hardware-cognizance in approximation-tuning. Tuning is performed by dynamically evaluating approximation's actual effect on the hardware. The tuning mechanism, implemented over a gradient descent algorithm, iterates through a sequence of approximation configurations and evaluates the application error and hardware execution metric (e.g., performance/energy) for those configurations. Gradually it moves the approximation configuration of the application in the direction of the steepest efficiency-error gradient, i.e., one that achieves the best efficiency metric improvement for the lowest change in error, until the optimum configuration is reached. Figure 2 shows the approximation configurations chosen by the hardware-cognizant mechanism (HW-C) for different metrics: performance, energy, and energy-delay. Not only are the chosen configurations unique for each metric (and optimum for that metric), they are also different from the one chosen by the hardware-agnostic one. The tuning mechanism is discussed in detail in Section 3.

1.3 Synergic Optimization of Varied forms of Approximation

Challenge: An ideal GPAS should be able to employ multiple approximation techniques in conjunction. This is especially important in general purpose systems wherein the benefits obtained from any one form of approximation might be minimal for some applications. The multiple techniques might comprise, for instance, approximation in software, hardware compute, and hardware memory. For optimum approximation to be achieved, the hardware-cognizant tuning mechanism described earlier should be able to manage every fine-grained and diverse approximate computation of all the enabled approximation techniques, and the prescribed approximation configuration should be cognizant of the interactions between these various approximation techniques. While prior works have proposed systems supporting multiple forms of approximation [12, 14, 15, 28–30, 37, 40], these proposals, to the best of our knowledge, have mostly targeted application-specific systems or have insufficiently explored the intelligent management of approximate interactions,

which are very diverse and occur at very fine granularities. We discuss these works further in Section 2 and Section 8.

SHASTA's system support and synergic benefits: In SHASTA, the hardware and software proposals are built into a robust GPAS, aided by (a) an ISA with approximation extensions, (b) a compiler that takes approximation annotated applications to generate an executable utilizing these ISA extensions, and (c) a runtime that is run periodically over the lifetime of an application, which leverages the tuning mechanism to tune the application's approximation configuration in accordance with the application's execution characteristics. We show that this cross-layer approximation system is able to achieve better error-tolerant execution efficiency in comparison to prior work and moreover it is able to achieve synergy among the multiple approximation techniques. System details and synergic effects are discussed in Section 5.

1.4 Summary of SHASTA's Contributions

We summarize key takeaways from SHASTA's novel contributions towards General Purpose Approximation Systems (GPAS) below:

(1) Proposes a flexible and dynamic framework for fine granularity approximation on GPAS.
(2) Allows each candidate variable (across the program) to be approximated differently.
(3) Allows each dynamic instance of a candidate variable to be approximated differently at different loop iterations.
(4) Allows both compute approximation and memory approximation (and can support additional forms as well).
(5) Automates application tuning for approximation based on given error tolerance target.
(6) Application tuning takes hardware execution benefits into consideration.
(7) Tuning also looks at combined benefits of all forms of approximation on accuracy and cost savings.
(8) Tuning is efficiently implemented atop a gradient descent algorithm.

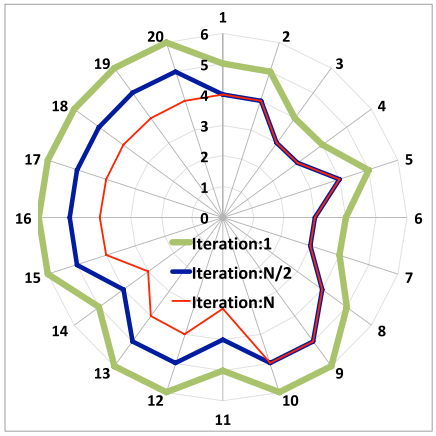
2 BACKGROUND AND MOTIVATION

2.1 Hardware Approximation with Spatio-temporal Diversity at Fine Granularities

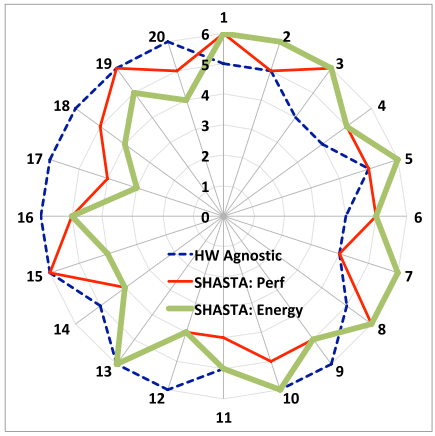
Prior advancements at the application, compiler, and ISA levels [6, 7, 21, 25, 36, 40] have enabled fine granularities and diverse approximation at the software level—these works are discussed in more detail in Section 8. Unfortunately, reaping the benefits of software-enabled fine granularity and spatio-temporally diverse approximation in hardware is complicated, especially when targeting GPAS.

Consider the code snippet in *Listing 1* that performs K-Means clustering under some specified error tolerance. The snippet shows one portion of the application that involves calculation of Euclidean distance. It involves three specific computations marked as approximate: subtraction, multiplication, and addition. All other operations are accurate—for instance, the for-loop induction variable i is not approximated to avoid compromising control flow. It is evident that approximate applications like *Listing 1* tend to contain a fine-grained mixture of accurate and approximate operations. Further, in *Listing 1* the spatial and temporal diversity in approximation is evident. Spatial diversity is enabled by a unique k_i for each approximate operation, which denotes the "level" of approximation for op_i. Temporal diversity is indicated by the $k[it]$, i.e., a static approximate operation can have different approximation "levels" over different iterations of the while-loop. In terms of diversity, while recent efforts have explored either one form, mostly in software [6, 21, 25] but some in hardware [40], no prior work has tackled an optimal amalgamation of both.

Spatio-temporal diversity is further illustrated in Figure 3(a) for a toy approximate application (figure details described in caption). The different radial values across the circumferential variables



(a) Spatio-Temporal Diversity



(b) HW Cognizant Tuning

Fig. 3. The approximation configuration for a toy application with 20 approximate elements, each with 7 different approximation levels. The circumferential axis denotes the variables, while the radial axis denotes the approximation level for that variable. The lower the radial value, the greater the approximation. Also, the first 10 approximate elements are compute operations while the last 10 are load operations.

```
@Ax(0.9) float kmeans(dataSet, k) {
  while(it < MAX || condition) {
    ----;
    ----;
    #Euclidean distance
    for (int i = 0; i < v1.length; ++i) {
      float d = @Ax(v1[i] - v2[i], k1[it]);
      sum = @Ax(sum + @Ax(d*d, k2[it]), k3[it]);
    }
    ----;
    ----;
    it++;
  }
  return centroids;
}
```

Listing 1. Approximation-enabled programs.

indicate spatial diversity. Further, the changing configuration over iterations of the application (moving from green to blue to red) indicates temporal diversity. The approximation across the variables increases from iteration 1 to iteration N/2, typical of clustering-style applications (e.g., K-Means) that settle close to appropriate clusters in early iterations. Moreover, the load operations become even more approximate from iteration N/2 to iteration N, indicative of saturating of stored memory values over time (thus allowing more approximation).

In light of applications with fine-grained intermingling of approximate and accurate operations such as the code in *Listing 1*, most prior approximate compute hardware for general-purpose applications employ dedicated functional units for performing both accurate and approximate operations concurrently. Resulting hardware challenges include resource duplication (a side effect being resource under-utilization), operation scheduling complications, design complexities [8, 9],

circuit overheads [9], and thus might offer control only at coarse granularities such as vector operations [40] or code blocks. Note that this still does not support diversity—to support diverse approximation in conjunction, these proposals would require even more duplicate resources (one for each approximate level), severely exacerbating hardware and management complexities. More details on prior proposals can be found in Section 8. Note: In this work, in the context of temporal diversity, we specifically tackle iterative applications, i.e., we allow a static operation's approximation to evolve from one iteration to the next.

Takeaway 1: *Existing hardware approximation solutions are less capable of exploiting the fine granularities and spatial or temporal approximation diversity that software can enable.*

2.2 HW-cognizant Approximation Tuning

An ideal approximation tuning mechanism needs to be able to perform the following:

① Identify the tolerable approximation for an application (in the Listing 1 Code snippet, this is 10%). ② Identify application elements, i.e., variables/computations that can be approximated (Code snippet: three compute operations and related loads). ③ Identify potential approximation configurations such that their assignments to application elements meet the application's error tolerance requirements. ④ Choose the right approximation configuration to maximize any specified execution metrics such as performance, energy efficiency, or power.

Solutions targeting the first three challenges are briefly discussed in Section 8. From a hardware-software co-design perspective, the fourth challenge is most significant. Most of the prior proposals that target general purpose computing perform tuning that is relatively "hardware-agnostic." This means that even though the overall goal of the approximation is to improve execution efficiency, the approximation tuning mechanism is mostly oblivious to the quantitative effects of the chosen approximation configuration on hardware execution (e.g., IPC/Energy). They ignore target hardware characteristics and only algorithmically maximize the element-wise approximation until the application is closest to its overall error tolerance [16, 22] or use only minimal statically obtained hardware metrics [20, 40]. More details about these proposals in Section 8.

While some solutions that perform system-cognizant approximation tuning have been proposed (with varying knowlege of the system-level effects of their particular approximations) [12, 14, 15, 28–30, 37], these solutions are mostly not designed for general purpose systems and further, are less suited to fine granularities of diverse approximation. More details on these solutions are discussed in Section 8. However, SHASTA targets GPAS and specifically focuses on fine-grained and diverse approximation, with potentially hundreds of approximate elements, which significantly complicates the implementation of hardware cognizance.

Figure 3(b) different approximation configurations chosen by tuning mechanisms. All the configurations are for an error tolerance of 10%. The figure shows an approximation configuration chosen by a hardware-agnostic) tuning mechanism in blue. It also shows the configurations selected by SHASTA in green and red. While the red configuration uses highest performance as the optimization metric, the green uses lowest energy.

The hardware-agnostic configuration is the result of a greedy optimization algorithm over the variables (as used in prior work)—it approximates variables 1–5 aggressively (which it tunes first), almost maxing out on the error tolerance, and thus is forced to be very conservative on later variables 5–20. Note that the above tuning is being performed simply one variable after the other, until the error tolerance is reached, irrespective of the execution efficiency impact. SHASTA's configurations, however, are achieved by hardware-execution-cognizant tuning over each variable—more aggressive approximations for variables that provide better improvements to execution efficiency. Note that the configurations generated by SHASTA are different when the execution metric changes. The energy-optimum scenario (green) tunes the memory variables (10–20) more

aggressively, because memory approximations not only reduce latency (for better performance) but also reduce memory access, which directly impacts energy.

Takeaway 2: *HW-agnostic tuning mechanisms return sub-optimal approximation configurations, because they are not tuned cognizant to hardware's execution metric or its characteristics.*

2.3 Synergic Approximation System

Finally, we stress the need for synergic capabilities of the approximation HW-SW framework. In *Listing 1* and Figure 3, two forms of approximation are being employed: compute approximation and memory/load approximation. In an ideal approximate platform: ① the different approximation techniques in the hardware stack should coexist independent of each other, ② the hardware-software interface should abstract away the particular technique of approximation used, and ③ the software stack should be able to tune all forms of approximation in synergy.

For example, while compute and load approximation techniques are independent, the tuning mechanism should be cognizant of their intersecting dataflow, i.e., the approximate loads might be consumed by the approximate compute, thus, the errors from each of those operations can constructively or destructively intersect with each other. In this example, we refer to constructive intersection as (a portion of) the error in the load operations being masked away by the approximation in the compute operation. Conversely, destructive interference would amplify the individual errors.

Takeaway 3: *In a system with multiple approximation techniques, the entire HW-SW stack should be designed to maximize system synergy.*

3 APPROXIMATION TUNING MECHANISM

The highlighting characteristic of SHASTA's approximation tuning mechanism is that it is a combination of (i) being suited to general purpose approximation, (ii) supporting fine granularities and diverse approximation, and (iii) most importantly, enabling HW-cognizant optimization (it obtains actual execution measurements from hardware, such as IPC or energy, by running the application with the input sample) when tuning the approximation configuration.

Note that the tuning mechanism is capable of providing unique approximation values for *every dynamic instance of every approximate operation in the application.* For example, if N static program operations are marked as approximate and these are iterated over M times, there are N*M dynamic operations for which potentially unique approximate values can be assigned by the tuning mechanism. This achieves the goal for fine-grained spatio-temporal approximation diversity. Remember that the forms of approximation might be different for different approximation operations: for example, in our use case the compute approximation focuses on clock-cycle slack while the memory approximation focuses on approximation loads. All types of approximation can be supported the tuning mechanism as long as each type clearly identifies how the approximation mechanism varies for each "level" of approximation. Pseudocode for the tuning mechanism atop an application is shown in Listing 2 and features of the mechanism are discussed below.

① Each tuning epoch involves tuning the application's approximation configuration with some characteristic sample input.

② The tuning epoch starts with capturing the golden accurate application output, i.e., with no approximation, and its corresponding golden hardware execution metric (e.g., IPC/energy, using hardware/software counters).

③ The tuning mechanism will then run multiple "outer-loop" iterations of the application, each performing a set of "inner-loop" iterations.

④ Every "outer-loop" iteration of the tuning mechanism starts with a current approximation configuration. The goal at the end of the iteration is to find the new approximation configuration

```
def eval_epoch(f,sample,AC,tol)
  """
  f() is a function for approximation
  sample is the input sample for tuning
  AC is previous epoch's approximation config
  tol is the error tolerance
  """
  # Calculate golden output, h/w efficiency metric
  Out_gold = f(sample,0)
  Eff_gold = HW(f(sample,0))

  # Outer-Loop: until convergence/tolerance
  while True:
    AC_prev = AC;
    AC = eval_AC(f, sample, AC_prev, Out_gold, Eff_gold)
    ΔAC = AC-AC_prev
    if ΔAC < ε || f(sample,AC) > tol:
      break
  return AC

def eval_AC(f, sample, A, Out_gold, Eff_gold):
  """
  Calculate grad to create new approx configuration
  """
  # Application error at current approximation
  Out = f(sample,A)
  Δerror = Out - Out_gold
  # Approx. Application's execution benefit
  Eff = HW(f(sample,A))
  Δexe = Eff - Eff_gold
  # Loss dependent on error and efficiency
  Loss = L(Δexe,Δerror)

  # Inner-Loop: Iterate over all approx. ops
  while not A.finished:
       # Perturb approximation of ith op in A (ai)
       A_i = modify(A, ai, δ)
       # Calculate application error at A_i
       Out_mod = f(sample,A_i)
       Δerror_mod = Out_mod - Out_gold
       # Calculate h/w metric at A_i
       Eff_mod = HW(f(sample,A_i))
       Δexe_mod = Eff_mod - Eff_gold
       # Loss at A_i
       Loss_mod = L(Δexe_mod,Δerror_mod)
       # Compute the partial derivative
       grad[ai] = (Loss_mod - Loss) / δ
       # Step to next approximation and reset current
       A.iternext()

  # Calculate the new approximation configuration
  A = A - step_size*grad
  return A
```

Listing 2. Gradient descent approximation tuning.

that is the steepest move (in terms of the efficiency vs. error gradient) from the current configuration.

⑤ To achieve this, the current approximation configuration is independently perturbed by δ in each dimension (i.e., each variable). Each resulting configuration is a "test" approximation configuration.

⑥ The application is run with each such "test" configuration over the course of tuning mechanism's "inner-loop." For each of these "inner-loop" iterations, the application output and its hardware efficiency metric are obtained and compared with the golden values to obtain a "Loss" value. The loss is directly proportional to the application error and inversely to the efficiency improvement.

⑦ The gradient of the Loss function along each approximation dimension is calculated by evaluating how much the loss function changed along each dimension's δ perturbation. This information is used to obtain the new approximation configuration (at the start of the next iteration) and this ends the current "outer-loop" iteration.

⑧ In the next "outer-loop" iteration, the approximations with the steepest gradients from prior move by "one" approximation level, while the other approximations proportionally move by fractional approximation levels. Fractional approximation levels are interpreted by the hardware as a dynamic percentage.

⑨ Tuning continues until convergence, i.e., when there is marginal change to the loss function/gradient on consecutive iterations of the "outer-loop." This ends the tuning epoch and the application will run with the final approximation configuration until the next tuning epoch.

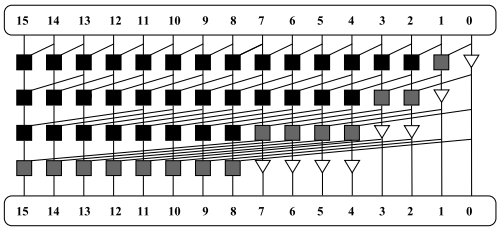
Overhead Evaluation: Table 1 (discussed in detail in Section 6) shows that the number of global iterations the gradient descent mechanism takes to convergence is low for our approximate applications. Moreover, these statistics are from cold starts wherein the mechanism is not in a tuned state prior to the training. In most scenarios, there is low change input characteristics from one epoch to the next, meaning tuning reaches optimum state for a new epoch in just 1 or 2 iterations. Thus, average-case tuning overheads are only 1–2 iterations of tuning per epoch and this greatly reduces the overhead of dynamically running the tuner. While cold start tuning overheads range from 1–30 ms, depending on the number of approximate variables, time to converge, training sample size, and so on, average case overheads are less than a ms. The table shows average case overheads with worst-case cold start overheads in parentheses.

4 DESIGN OF APPROXIMATION HARDWARE

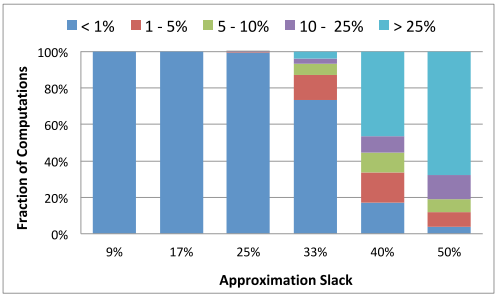
4.1 Compute Timing Approximation

Section 2 discussed the benefits and challenges of achieving fine-grained spatio-temporally diverse approximation. To achieve these goals SHASTA introduces a new form of Timing approximation called *Slack-Control Approximation* (SCA). SCA is inspired by REDSOC [32], a prior proposal from accurate computing, which identifies the unused portion of the clock cycle in ALU computations based on opcode and data-type and eliminates it by starting future computations early.

SCA extends this idea further by reducing an operation's compute time while risking higher computation error, but in a controlled manner. Other forms of timing approximation are discussed in Section 8. SCA is enabled by interpreting a computation's tolerable approximation as a clock cycle slack component: what we call *approximation slack*. Once approximation slack is identified, it is cut out (or "recycled") by the idea of per-operation slack recycling. Slack recycling is performed by enabling *transparent bypass paths* in a traditional synchronous execution pipeline. Slack recycling recycles the approximation slack in a "producer" operation by starting the execution of dependent "consumer" operations at the instant of completion of the producer operation, undeterred by



(a) 16-bit KS-Adder



(b) ADD: Timing Error Distribution

Fig. 4. Design for Slack-Control Approximation.

clock boundaries. Recycling approximation slack in this manner over multiple operations, executing on traditional functional units, allows acceleration of these compute sequences. This results in application speedup when such sequences lie on the critical path of execution.

In SHASTA, SCA is capable of fine-grained spatio-temporally diverse approximation by allowing the following: ① It can uniquely control each dynamic approximation execution's computation time individually, ② It allows same compute units to be used for both accurate and entire range of (timing) approximate compute, thus allowing fine-grained per-operation control without significant overheads, and ③ The computation time can be chosen from multiple discrete levels, every clock cycle, for every operation, depending on the amount of approximation the operation can endure.

4.1.1 Viewing Approximation as Cycle Slack. Consider the addition operation on a standard adder design (e.g., 16-bit Kogge Stone) as shown in Figure 4(a). We define the computation time of any pair of operands on this adder as the time required for all 16 bits of the output to settle at the correct values. This is dependent on (a) the two operands (i.e., the critical path that they trigger) and (b) the previous state of the output bits. For a given fixed state of operands, previous output, and operation voltage, it is intuitive that as the allowed computation time for this adder decreases, the potential for some of the output bits to be in an incorrect state increases. In other words, as the "approximation slack" increases, the approximation-error of the adder increases.

Prior work on circuit-level error analysis (B-HivE [38]) has shown that timing error as a function of voltage (for a set frequency) can be effectively modeled for different computations (e.g., add/multiply). We intuitively extend this to model computation approximation as a function of approximation slack. In actual chips, we expect this modeling to be performed statically at chip design time. Figure 4(b) shows the distribution of error magnitudes experienced by 1M random add computations for different approximation slack fractions. For addition operations, there are almost no error magnitudes greater than 1% across all random operations for 25% approximation slack. For 33% slack, more than 90% of operations have less than 5% error magnitude. This suggests (and is supported by results) that not only are average error rates low at reasonable approximation slack, but a majority of operations follow the same trends—meaning that tuning with reasonably characteristic sample inputs and running with inputs in the wild tend to show similar error-rates (even though we do not design to provide guarantees). Once the slack corresponding to different operations are estimated at design time, these values are stored in a slack look-up table (LUT).

Every approximation level corresponds to a different slack value. At the decode stage the instructions indicate the amount of approximation on their execution (set by the tuning mechanism). This number is used to look up the amount of approximate slack applicable for this computation, for

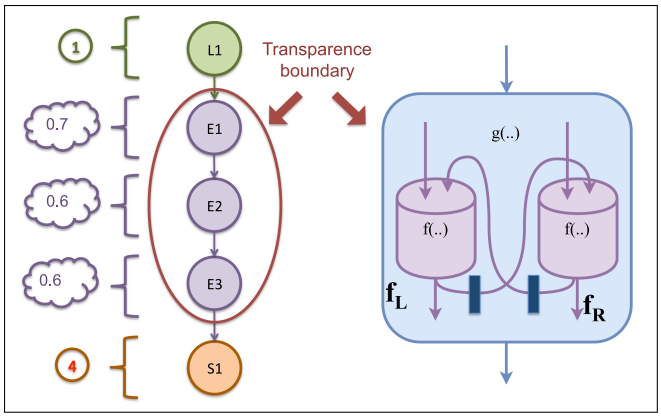


Fig. 5. Clock cycle slack recycling.

the particular level of approximation. In this work, we only approximate multiply, add and subtract operations. Details on approximation translation from software to hardware are discussed in Section 5.

4.1.2 Recycling Slack across Operations. Clock cycle slack in a producer operation is recycled by executing consumer operations immediately after the producer completes (i.e., and not on later cycles). This is achieved by enabling transparent dataflow among the compute units, achieved via intelligent flip-flop control between the execution units as is also pursued in prior work targeting accurate computing [32]. A transparent mode FF design is a simple implementation consisting of a standard FF but with a bypass path [13]. A mux at the end of the two paths can select the "opaque" stored FF value or the bypassed "transparent" value, based on an enable input. Note that this transparent datatflow is implemented only within data bypass network between execution units. Transparent mode is enabled in the bypass path between execution units whenever data are expected to flow through at non-clock boundaries (due to clock cycle slack).

A datagraph to illustrate the high-level impact of transparent dataflow for slack recycling is shown in Figure 5. The graph shows a sequence of dependent operations: 1 load, followed by 3 compute operations followed by a store. This illustration assumes all operations to be single-cycle operations. In a baseline synchronous design, the load finishes in cycle 1, the compute operations complete in cycles 2, 3, and 4, and the store occurs in cycle 5—thus, the entire dataflow graph would take 5 cycles to execute. The impact of transparent dataflow is as follows: operations that fall within the transparence boundary—that is, the compute operations—can flow transparently from one to the next without being limited by clock cycle boundaries. This enables approximate slack recycling. To showcase this, assume each of the 3 compute operations have some approximation slack—the computations only need 60% to 70% of the clock cycle. The load still completes in cycle 1 as before. The compute sequence of 3 operations E1-E2-E3 can execute transparently within the transparence boundary and complete in just 2 cycles—E1 executes from 1.0–1.7, E2 from 1.7–2.3, and E3 from 2.3–2.9. This allows the first dependent operation outside the transparent boundary—the synchronous store operation—to occur on cycle 4, rather than on cycle 5 in the baseline. Transparent dataflow can be achieved with just a pair of functional units that have bypass paths between them in at least one direction; for example, functional units FL and FR in Figure 5.

4.1.3 Support for OOO Core. Our target substrate for Slack-control Approximation is traditional out-of-order cores, thus, we adopt prior work [32] that proposed an OOO core design to recycle slack targeting accurate computing. In OOO cores, slack recycling requires modifications to the OOO scheduler, which are summarized below:

① **Eager Grandparent Wakeup:** Given a "grandparent-parent-child," and so on, sequence of dependent instructions, this mechanism allows speculative wakeup of a child instruction (from its reservation slot) based on its grandparent's tags. This allows the child instruction to be issued in parallel with its parent, thus enabling the parent computation's slack to be recycled.

② **Slack Tracking:** The cumulative slack over a sequence of operations is tracked—estimated based on the computation time of each operation. Tracking the cumulative slack is important because, when the total slack crosses integral clock cycles, synchronous operations (such as stores) can be performed on early clock cycles, resulting in speedup.

③ **Skewed Selection:** The OOO core's select logic is optimized to prioritize non-speculative operations over speculative (grandparent-awoken) operations, and this almost completely eliminates the erroneous expenditure of cycles and power under mispeculation.

Further details on the above components can be found in the original work [32].

4.2 Memory Load Approximation

Prior proposals for memory approximation already offer a suitable template for fine-grained spatio-temporally diverse approximation [16, 19]. But without a dynamic control mechanism to identify optimal approximation quantities throughout the application, these proposals perform poor exploration of fine granularities and diversity. SHASTA proposes a modified form of Load Value Approximation (LVA) [19], which enables more aggressive and dynamic use of the previously static approximation technique, enabling fine-grained spatio-temporal diversity and further improving its benefits in other ways.

4.2.1 LVA: Benefits and Limitations. LVA is motivated by the idea that applications suited to approximation often exhibit localized value similarity; they tend to reuse similar values. LVA proposes that for applications that can tolerate inexactness, the values associated with cache misses can be approximated. By approximating the load value on a cache miss, the processor can immediately proceed without waiting for the cache response.

Deployment of LVA is essentially controlled by two LVA-Control knobs—*approximation confidence* and *approximation degree*. Note: These knobs are as used in the original work and are paraphrased and explained below. *Approximation confidence* decides how often the data are approximated (based on past comparisons to some fixed error threshold), i.e., how often a value is returned from the approximator instead of a longer latency wait for the accurate value on a cache miss. This enables a trade-off between accuracy and latency of access. *Approximation degree* decides how often the data in the approximator is refilled with values from actual cache access (and thus retrained/refreshed). This effectively trades off accuracy for better energy efficiency in the memory hierarchy.

While latency and energy reduction is significant, there are limitations from LVA that our proposal exploits:

① First (and of most significance to SHASTA's fine granularity + diversity motivation), in LVA the approximation confidence's threshold value and approximation degree are static design time constants and uniform over all approximate loads—but this is not optimal. Among all loads that can be approximate, some approximate loads are less *influential* than others. While the errors from loads might sometimes be large, their effect on the application might be minimal (e.g., noise in

vision applications). Thus, deciding to approximate based on a particular load's error compared to a static threshold can have a detrimental impact to overall benefits. Also, it is a common characteristic among inputs that some approximate loads are more *stable* than others. Consider financial applications—it is often the case that some inputs are redundant or change very rarely. For example, in Blackscholes, a subset of the inputs takes on only four possible values, two of which occur over 98% of the time. Other inputs may change more frequently. Setting approximation degree to be constant results in higher error under aggressive settings and low benefit in conservative settings.

②Second, LVA invocations are rather infrequent, because LVA is only invoked on L1 cache misses and these misses are often low in many applications.

③ Third, in LVA every approximate load still performs all the minimum "tasks" that a traditional load performs—accessing the load-store queue, accessing the TLB for address translation, address generation computation, and accessing the L1 data cache.

4.2.2 Dynamic Pre-L1 Approximation. Our contribution to memory approximation is two-fold. First, we modify LVA to be dynamic—enabling it to tackle fine granularities and spatio-temporal diversity. This is achieved as follows: Each approximate load instruction is allowed a unique approximation confidence and a unique approximation degree, an optimum amount as determined by the tuning mechanism. These values are estimated based on application effects and not based on static error thresholds, and so on. The assigned values are unique per static approximate load as well as based on its iteration instance (if it belongs to a loop, thus exploiting temporal diversity). Therefore, the percentage of time that a static load instruction of some *N*th iteration looks up its approximate value in the approximator is tuned uniquely to each such load. Similarly, the percentage of time that the approximator is refilled with an accurate value from the cache is tuned uniquely to each approximate load. Thus "error vs. latency" and "error vs. energy" trade-offs are controlled in a disciplined manner, unique to each approximate load. The approximation information flow from the tuned application to the hardware is similar to the case of compute approximation and is detailed in Section 5.

Second, our memory approximation implements pre-L1 Approximation, which brings the approximator prior to the L1 cache. This allows the capability for invoking approximation on all loads and not just on L1 caches misses. Our design approximates loads early in the execution pipeline. Loads that are marked as approximate and that end up being approximated (based on approximation confidence) perform a *Pre-L1 LVA* lookup, as described earlier in Figure 1(b), and thus benefit from (a) lower latency than L1 access and (b) lower energy than traditional loads by skipping address generation, translation, dependence checks, and cache access. Implementing LVA prior to L1 cache access is key to some of the synergic gains in SHASTA—this is discussed in Section 5.

Figure 6 shows the general structure of our *Pre-L1 LVA* approximator table. The approximator consists of a simple instruction address–based hash that performs a lookup into a direct mapped approximator table. Each entry in the table only consists of a tag and a data block. The data block is essentially an approximate local history buffer (LHB)—storing some representation of the accurate reads (from cache) of the most recent approximate load values that match this entry's tag. In the figure, the LHB values (4.1, 3.9, and 4.0) are the accurate values of the three previous loads that matched this tag entry. An approximate value is then generated by employing some computation function f (we use average) on the values in the LHB.

4.3 Overhead Evaluation

Compute Approximation: The slack recycling implementation in an OOO core suffers reasonable overheads—an area overhead of 0.3% and an energy overhead of 0.8%. The approximation

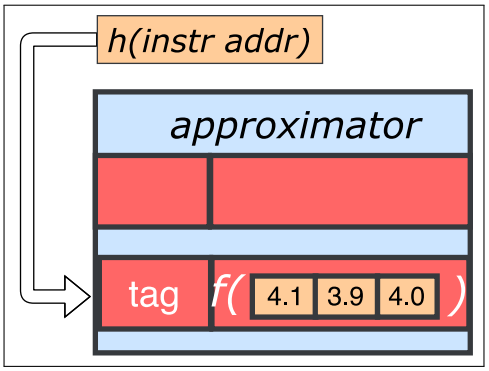


Fig. 6. Load approximator.

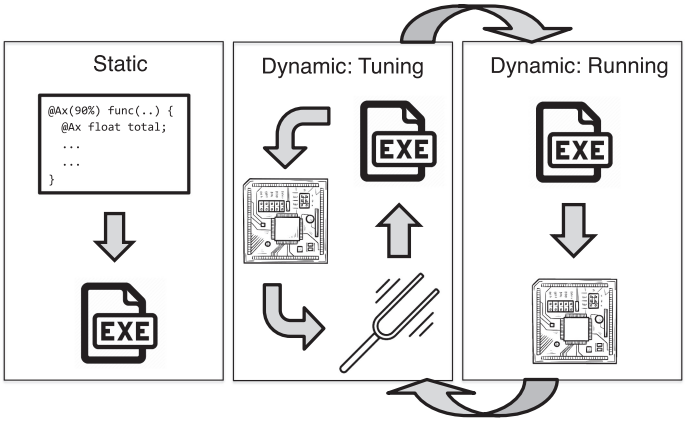


Fig. 7. SHASTA system overview.

slack LUT has area overhead of 0.52% and negligible access energy. All slack-related access and control occur in parallel with the core decode and scheduling logic—there is no increase in critical path latency.

Memory Approximation: Design space exploration results in the choice of 64 entries in the approximator—this results in an overhead of roughly 1KB of storage. Due to the small size of the approximator, access time is very low, pushing the loaded operand off the critical execution path.

5 SHASTA: SYSTEM AND SYNERGY

System Overview: Figure 7 illustrates the high-level overview of SHASTA showing the entire system flow from Ⓐ Static: Approximation specifications within the program and compiling to an approximation-enabled ISA, Ⓑ Dynamic Tuning: A tuning mechanism that iterates the application through multiple approximation configurations (along optimum gradient) on the target hardware, and Ⓒ Dynamic Running: program execution with tuned approximations on hardware.

PL/Compiler/Runtime Support: Programs are annotated as shown in *Listing 1*. The programmer annotates variables and computations that are amenable to approximation and specifies the target approximation for the application, as is done in prior work [7, 9, 36]. Among computations, this work currently only targets add, sub, mult operations for integer and floating point. Approximate load operations are inferred based on annotated variables (e.g., v1, v2, d, total in Listing 1).

Note that the tuning mechanism is agnostic to the fact there are multiple types of approximation (e.g., memory vs. compute). As long as the levels of approximation for each approximation-type are well defined, this tuning mechanism is suited to any form of approximation.

SHASTA's compiler takes the annotated application and generates an application executable that uses approximate instruction extensions to the ISA (we target ARM, but expect compatibility with other ISAs) for approximation. These instructions are {V}ADD.a, {V}SUB.a, {V}MUL.a, and {V}LDR.a and resemble traditional instructions apart from the approximation fields described below. It is important to note that the programmer does not have to specify the amount of approximation for each approximate operation—she only has to specify the approximation requirement at an application/function level and also specify the variables that can be approximated. The values themselves are automatically inferred by the tuning mechanism.

The approximate compute operations use a 3-bit value to indicate the amount of approximation slack that this operation should recycle. A value of 2 (on scale of 0 to 7) means the approximation slack is roughly 20% of the clock cycle (or computation time is 80% of the clock cycle). Approximate load instructions use two fields—the level of approximate degree (3-bit) and the level of approximate confidence (3-bit). The values of the degree/confidence are indicative of the percentage of time the load approximation–related action is performed. For example, an approximation confidence value of 5 (on the scale of 0 to 7) means that the load is looked up in the Pre-L1 LVA roughly 70% (i.e., 5/7) of the time. Note that the compiler does not generate any approximation amounts for the approximate instructions; that is done only by the tuning mechanism by running on the target hardware.

Finally, we enable the interaction between the application and the tuning mechanism via pragmas, as pursued in prior work [31, 39]. In the application, the programmer is expected to add pragmas prior to an approximable function's declaration (such as the K-Means function in *Listing 1*), providing its error tolerance, error metric, hardware efficiency metric, tuning granularity, and a pointer to some training input sample. This passes information to the runtime, which invokes the tuning mechanism (details in Section 3) at the appropriate tuning granularity (possibly every second, for $< 0.1\%$ overheads). The runtime also captures hardware measurements and provides information to the tuning mechanism.

Overall, the programmer burden only involves annotating approximating variables/computations and specifying approximate function pragmas. In our experience with our chosen approximate applications, this one-time overhead is very reasonable.

Synergy across HW approximation techniques: As discussed in previous sections, SHASTA performs approximations across both compute and memory. With SHASTA's intelligent tuning mechanism, the benefits from the synergy between compute and memory approximation is greater than combining individual benefits from memory-only/compute-only approximation (under same total error tolerance). This is because of the following:

① Tuning across memory+compute provides a larger tuning space (i.e., more approximate operations), increasing potential for better efficiency-error sweet-spots.

② When an approximate compute operation performs compute on already approximate memory operation ("an intersecting memory-compute approximation"), the resulting error is often less than effects of errors from "non-intersecting" memory and compute approximations. A portion of the error in the load operations is masked away by the approximation in the compute operation. Since approximation tuning involves actual running of the application on hardware, such effects are respected by the tuning control.

③ Slack-Control Approximation produces performance benefits by slack recycling over transparent chains of operations. Figure 4 showed that slack recycling only happens within a transparence boundary—one that is established by opaque memory operations. Approximating loads

Table 1. Approximate Applications

Application	Domain	Annotate	Itns	Dyn. Approx %	SHASTA TI	Greedy TI	Err 10%	Ovhd (ms)
Blackscholes	Finance	111	1	30.63%	9	71	9%	2.7 (24)
Mat Mul	ML	6	1	17.91%	7	30	4.5%	0.15 (1)
Inversek2j	Robotics	60	1	14.25%	10	101	11%	1.5 (15)
K-means	Mining	39	32	34.1%	22	130	9%	0.95 (21)
FFT	Sig Proc	60	1	11.25%	5	20	12.1%	1.4 (7)
Canneal	CAD	75	32	7.2%	14	120	8.5%	0.22 (26)
PageRank	Graph	25	32	30%	20	110	8.3%	1 (20)
MLP	ML	30	20	27%	18	80	5%	1.05 (19)

by skipping cache access removes this transparence boundary, which breaks chains—meaning that slack recycling can occur over longer chains.

While the importance of synergy has been discussed in prior work targeting application-specific systems (see Section 8), SHASTA enables synergy among multiple forms of approximation, suitable to GPAS. Further, SHASTA creates a platform to achieve synergy across diverse fine granularities of approximations across multiple approximation domains that, to our knowledge, is insufficiently explored in prior work.

In this work, SHASTA employs forms of approximation that requires the hardware to be capable of (a) slack recycling, i.e., have transparent paths in the data path, optimizations at scheduler and decode and (b) load value prediction/approximation tables, and so on. While these mechanisms allow SHASTA to be fine grained and diverse in its approximation, the SHASTA system of general purpose approximation could be employed to support other forms of approximation as well.

6 METHODOLOGY

Applications: We evaluate SHASTA across eight applications suitable to approximation from different domains—highlighting that SHASTA is a general purpose solution to approximation. The applications include those from the AxBench [41], Polybench, and Parsec [4] benchmark suites and others. Operations to approximate are identified by hand, based on guidelines established in prior work [19]. Applications are tuned to target three different accuracy levels of 99%, 95%, and 90%, as is common requirement for these applications. Note that they can be tuned for other accuracy levels as well. Accuracy measurement metrics for the AxBench applications are as defined in the benchmark suite [41]. Accuracy measurement of other applications are as performed in prior work [16, 19]. Applications are compiled for the ARM ISA to run on the Gem5 [5] architecture simulator.

Table 1 shows the different applications along with the number of approximate operations identified in the program code, the iterations of the application, as well as the resulting percentage of dynamic instructions that are approximate (Dynamic Approximation %). The table also shows the actual error estimated at test time (i.e., on the test inputs) when tuning for a 10% tolerance target on the training inputs of the applications. The tuning overheads in terms of tuning iterations (TI) are shown for the greedy algorithm and SHASTA's gradient descent algorithm—these are discussed in more detail later. The tuning overheads in time (ms) are shown in the last column (average/worst-case) and were discussed earlier—invoking the tuning mechanism once every second results in overheads of less than 0.1%.

Slack modeling: Slack is modeled via RTL design in Verilog and synthesis using the Synopsys Design compiler. We synthesize the execution pipeline stage with a 0.5 ns cycle time (i.e., 2 GHz)

Table 2. Processor Configuration

Parameter	Values
Frequency	2 GHZ
Front-end width	4
ROB/LSQ/RSE	80/32/64
ALU/SIMD/FP	4/2/2
L1/L2 Cache	64 kB/2 MB w/ prefetch
Approximator / LHB	64 / 4 entries

constraint. Our timing analysis agrees with estimations from prior work [38] as well as characterization via gate-level C-models. Considering the low effort, we expect state-of-the-art CAD tools to be capable of (or extendable to) such analysis. During processor execution, appropriate slack bucket is selected simply based on opcodes and operands: no dynamic timing analysis is involved.

Error modeling: Compute operations identified to be suitable for approximation are approximated via our approximation model build on top of the SoftInj error injection framework [38]. Memory operations suited to approximation are approximated by implementing the approximator described in Section 4.2 in software. Overall application error for a given approximation configuration is evaluated by natively running the application to completion and comparing with a known precise output.

Simulation: Application IPC for given approximation configuration is obtained via Gem5 [5] architectural simulation. We implement load approximation and slack recycling for out of order cores in Gem5. Power numbers are estimated in accordance to McPAT [17]. The processor configuration is described in Table 2.

Tuning: Application tuning is performed via wrapper functions written in C that run a numerical gradient descent tuning mechanism. The algorithm iterates until convergence. We also implement a hardware-agnostic greedy algorithm (for comparison) as described in multiple prior works such as Reference [16]. The algorithm finds a suitable approximation for the first variable while keeping others exact, then it fixes the approximation of first variable and moves on to the second variable while keeping the others exact, and so on—independent of hardware efficiency. The TI in Table 1 show the number of iterations before the tuning mechanism completes or converges for our proposed tuning mechanism via gradient descent and for a greedy mechanism (Greedy).

7 EVALUATION

7.1 Performance Speedup

Figure 8 shows the speedup obtained by SHASTA in comparison to (a) a traditional baseline without any approximation features, and prior proposals; (b) REDSOC [32], which performs slack recycling but only targeting accurate compute; and (c) LVA [19], which load approximates only on L1-misses and further, an optimization of prior work: (d) LVA-L1, which load approximates on L1-hits as well. We include LVA-L1 because, with our chosen processor configuration (L1 cache size, etc.), the applications we run see low miss rate and render traditional LVA ineffective. LVA mechanisms and SHASTA results are shown for a 10% error tolerance. Further, SHASTA is set to tune for the performance (IPC) metric. Speedup is a function of the fraction of dynamic approximations, influence of non-approximate instructions (e.g., their latencies), and application dataflow.

SHASTA clearly outperforms the competing mechanisms, achieving a mean speedup of 25% across the applications with a maximum speedup of 47% with K-Means, in comparison to the traditional baseline. Highest speedups are seen for K-Means, which has high error tolerance—this

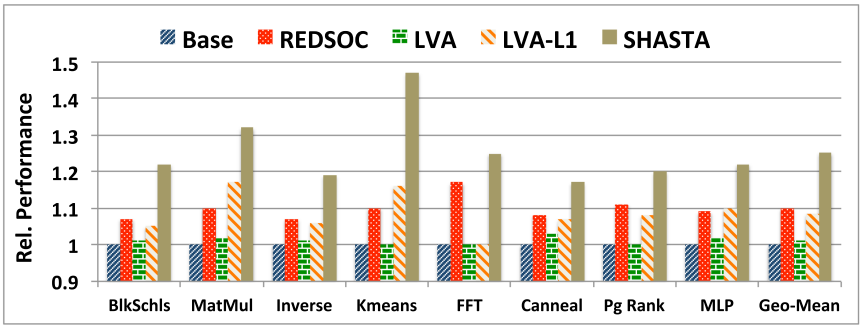


Fig. 8. Performance speedup.

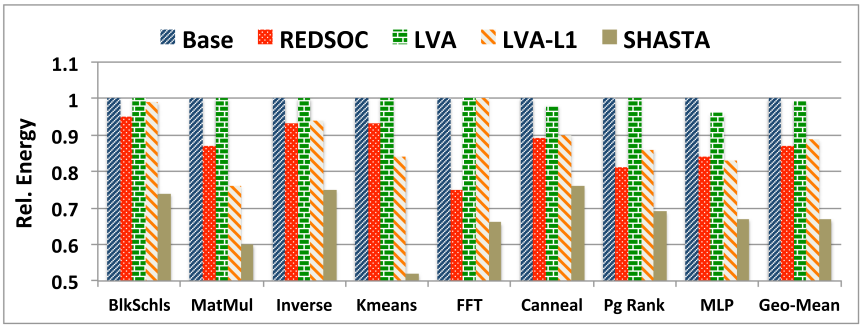


Fig. 9. Energy savings.

follows observations from prior work that discusses that very few bits of precision are required for reasonable accuracy in K-Means [16]. In comparison, REDSOC, LVA, LVA-L1, achieve speedups of 10%/2%/9%, respectively—clearly showing that achieving the design goals introduced earlier has a significant benefit to approximation systems.

7.2 Reduction in Energy Consumption

Figure 9 shows energy savings from SHASTA in comparison to the baseline, REDSOC, LVA, and LVA-L1, again for a error tolerance of 10%. Here, SHASTA is tuned for the energy metric. Trends from energy savings are similar to those observed in performance speedup. An interesting observation in tuned configurations is that tuning for energy reduction resulted in more aggressive memory approximation in comparison to tuning for performance. This intuitively makes sense, because while compute approximation directly benefits only performance, memory approximation benefits both performance (by latency reduction) and energy (by cache/memory access reduction).

We find memory approximation to be less beneficial in applications like Blackscholes (matching earlier observations [19]). LVA-L1 is of high benefit in a simple matrix multiply kernel (due to redundant as well as similar values) while traditional LVA is less useful, because the cache miss rate is negligible. SHASTA sees significant benefits across these kernels—in Blackscholes it is able to get significant benefits from compute approximation while it is able to leverage benefits from both pre-L1 LVA and compute approximation in matrix multiply, Mean energy savings for SHASTA are 33% with maximum reduction of 46% in K-Means. In comparison, other techniques see energy reduction in the range of 1% to 12%, clearly highlighting SHASTA's benefits.

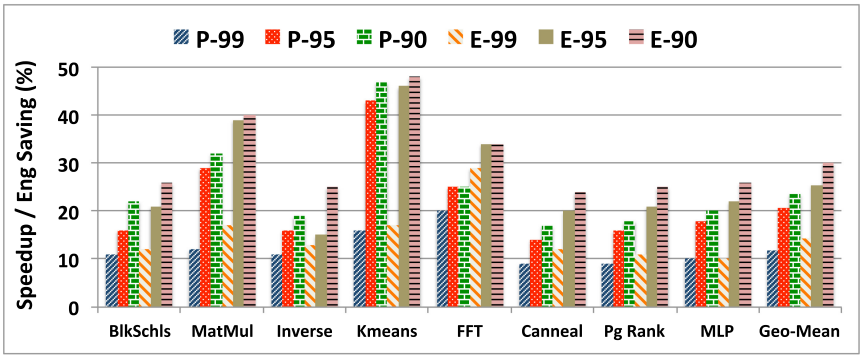


Fig. 10. Efficiency improvements at varying application accuracy.

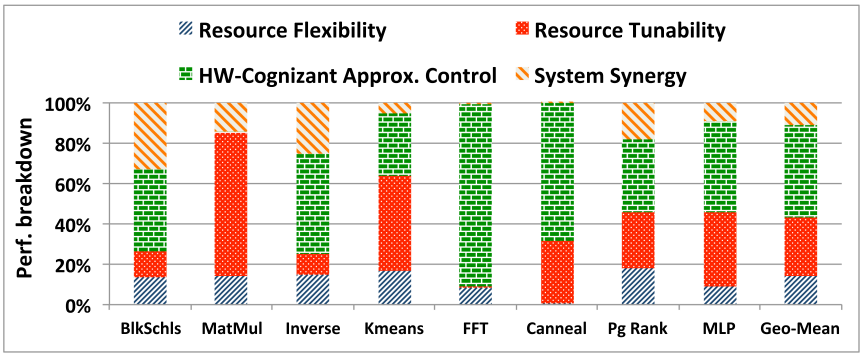


Fig. 11. Performance benefit breakdown.

7.3 Approximation Sweep

Figure 10 shows benefits from SHASTA (independent results for both speedup and energy reduction metrics) at different error tolerance levels. We sweep over accuracy requirements of 99%, 95%, and 90%. In the figure, "P" bars show performance speedup and "E" bars show energy savings. In almost all the applications, there is a clear monotonous relation between tolerated error and benefits from approximation—there is at least a gain of 10% in speedup/energy reduction when going from 99% accuracy to 90% accuracy.

K-Means sees a 5× difference in benefit between highest accuracy and lowest accuracy. This is because, in K-Means, 95% accuracy can be achieved with very low precision in computation but there is a significant increase in computation precision/correctness required for 99% accuracy.

Even at 99% accuracy, energy savings of 14% and speedup of 12% reflect the importance of approximate systems and their impact even under stringent accuracy requirements.

7.4 Breakdown of Benefits

Next, we breakdown the benefits of SHASTA in terms of the design goals. The analysis is shown in Figure 11, wherein we compare SHASTA to a baseline system without SHASTA's highlighting characteristics: (a) Spatio-temporal Diversity at Fine Granularity, (b) HW-cognizant Approximation Tuning and (c) Synergic benefits across approximation techniques. To measure the impact of each characteristic, we incrementally add each characteristic to the baseline system—to finally achieve SHASTA.

7.4.1 Spatio-temporal Diversity at Fine Granularity. We break this SHASTA benefit into two parts: (a) resource flexibility, i.e., every resource node in SHASTA can handle both accurate and approximate computations, and (b) resource tunability, i.e., the amount of approximation to be applied by each resource node can be uniquely controlled.

The benefit from resource flexibility is quantitatively observed by comparing SHASTA against the baseline, which provisions exactly one compute node for approximate-only compute. Thus, not more than 1 approximate compute operation can be processed in parallel. Figure 11 shows that SHASTA's resource flexibility contributed to 16% of its total benefits, with higher contributions in applications with more approximate ILP.

Next, Figure 11 shows that resource tunability contributes to 30% of SHASTA's overall benefits, obtained by comparing SHASTA against the baseline, which allows only fixed/static approximation. The approximation of the resource is tuned in accordance to the spatio-temporal approximation diversity in the application. Temporal diversity is especially important in iterative applications (K-Means, Canneal, Page Rank, and MLP) because the accuracy requirements vary across these iterations and need not be conservatively set by the worse-case. In the absence of spatial diversity, all static approximate operations will be tuned to the same approximation—we see this to be especially detrimental for simple applications such as matrix multiply, which have only a few static approximations but which can significantly influence the energy/performance of execution. Thus, spatial diversity contributes to almost 70% of its benefits from SHASTA.

7.4.2 HW-cognizant Approximation Tuning. To illustrate the benefits of hardware-cognizant tuning, we compare SHASTA's tuning mechanism against the baseline's greedy hardware-agnostic tuning. Benefits from intelligent tuning contribute to nearly 40% of SHASTA's benefits, as seen in Figure 11.

In applications such as K-Means, HW-agnostic tuning approximates compute variables further than ideal, thereby unable to sufficiently approximate memory variables that could provide significant energy savings. Somewhat similar are Blackscholes and Inversek2j. With a large number of approximate variables, the optimal compute approximations that produce best gradients for approximation (i.e., higher execution benefits but lower error) are deep into the program. A naive tuning mechanism hits the error-tolerance rate just within the first few approximate variables, never reaching the best-gradient variables. Iterative applications such as Canneal benefit greatly from the tuning mechanism, because specific iterations of the application (middle/late) provide best energy vs. error gradients and are found by the tuning mechanism.

7.4.3 Synergic Benefits. Finally, we quantify SHASTA's synergic benefits across the two approximation techniques used. We compare SHASTA against the baseline, in which compute approximation and memory approximation are tuned separately unbeknownst to each other. For this experimental baseline, we tune each of compute and memory for half of the total approximation (i.e., 5% error each), while SHASTA is tuned as a whole. The synergic benefits SHASTA obtains are reflected in Figure 11 and contribute to nearly 15% of the total benefits. Synergic benefits are particularly high in applications with higher error tolerance but more approximate variables such as Blackscholes and Inversek2j, wherein it is especially important to consider the relations between approximate memory operations being used by approximate compute operations.

In summary, it is evident that SHASTA's benefits are obtained from a combination of fine-grained spatio-temporal approximation capability, hardware-cognizant approximation tuning, as well as whole system synergy.

7.5 Discussion: More Comparisons

We make comparisons with some other prior works.

7.5.1 General Purpose Timing Approximation. We identify Truffle [9] to be the closest related prior work to SHASTA in terms of compute approximation—it focuses on GPAS and performs voltage scaling-based timing approximation. It also allows timing violations in the SRAM (as a form of memory approximation). We compare it with SHASTA in terms of SHASTA's characteristic features. Truffle is unable to achieve fine-grained latency reduction nor is it able to alternatively use the same resources for both approximate and accurate execution. Further, it does not support multiple levels of approximation and does not attempt to intelligently balance or tune different forms of approximation. Thus, it is unable to achieve fine-granularity spatio-temporally diverse approximation. Quantitatively, Truffle provides energy reduction of 2%–10% at non-definite application error. In comparison, SHASTA provides around 30+% energy reduction at well-defined 90% application accuracy.

7.5.2 Application-specific Approximation Systems. Prior works have proposed building approximation systems from the ground up that target a single application or domain. These systems often achieve synergy in approximation as well as perform system-aware approximation tuning. For example, Hashemi et al. [12] target biometric security systems, building an end-to-end flow from an input camera to the final iris encoding with intermediate approximate computational steps. Approximations at various stages are chosen based on their effects on the system, with the help of an RNN. While suitable to application-specific systems, they do not discuss extensions to general purpose approximation. Further, the approximation methods do not target fine granularities like in SHASTA. The use of an RNN at such fine granularities can cause explosive overheads.

Similarly, prior works from Raha et al. [28, 30] have also targeted Smart Camera Systems [28, 29] and Reduce-and-Rank kernels [30], again achieving synergy among different forms of approximation while focused on an application-specific subsystem. While also using a gradient-descent-style tuning mechanism (like SHASTA), these proposals again do not target general purpose approximation with fine granularities or spatio-temporal diversity.

7.5.3 Approximation at the Application Level. Approximation proposals from Hoffmann et al. [14, 15] have also analyzed trade-offs between accuracy and system energy, but at the application level. While suitable to many applications, these approximations and knobs are hand chosen from one application to the next [15] so are not easily malleable towards diverse general-purpose workloads. Apart from focusing on software-level solutions, these proposals do not target fine granularities or spatio-temporal diversity in their approximation methods. Moreover, in Reference [15], the approximation tuning searches through the entire configuration space, making it less suitable for a vast tuning space that is imposed by finer granularities and diversity of approximation. Further, the reinforcement learning approach proposed by Reference [14] is again less suited to the finer granularities, which is SHASTA's focus.

Similarly, Panyala et al. [25] perform application-specific approximation at the software level, targeting graph algorithms—they use techniques such as loop perforation, incomplete graph coloring, and synchronization. While loop perforation and synchronization are suited to general purpose iterative applications, in this work they are tuned specifically to PageRank and Community Detection. These methods are orthogonal to the approximation techniques in SHASTA and can be used in conjunction. Loop perforation is closest to the temporal diversity targeted by SHASTA. In their work, the benefits obtained by loop perforation alone for Community Detection are very limited—this is because of (a) the absence of spatial diversity, i.e., loop perforation eliminates entire iterations, and (b) absence of an intelligent tuning mechanism to select the right iterations for perforation/approximation. However, SHASTA shows that performing diverse fine-grained approximation with intelligent tuning has substantial efficiency benefits.

8 RELATED WORK

Hardware Approximation: Dedicated approximate units can run at low voltage [9], can be of specific low precision [8], or can perform temporally coarse-grained precision scaling via power/clock gating [40]. These solutions can incur significant design overheads, i.e., more functional units, dual (accurate/approximate) voltage rails, and scheduling logic overheads. Such proposals do not support multiple approximation levels concurrently, as this might require dedicated voltage rails (voltage scaling) or uniquely partially power-gated data paths (precision scaling) for each level of approximation supported. Thus, the employed approximation in hardware has to be kept conservative or runs the risk of hitting intolerable operation/application error. Timing approximation is a form of approximation that controls the computation timing [33]—specifically, it reduces computation timing, sacrificing accuracy for gains in execution efficiency. Timing approximation is usually achieved by under-volting [9] and is constrained in terms of its flexibility and ability to be reconfigured at fine granularities. Other forms of hardware-level approximation include, but are not restricted to, synthesis techniques [18, 24], self-healing designs [11], libraries for approximate-circuits–based design space exploration [23], and design methodologies for approximate CGRAs [2].

Software Approximation: Prior work such as EnerJ [36] enables programmers to annotate programs specifying operations or variables to be accurate or approximate via extensions to the programming language, compiler, and ISA. Recent advancements at application [6, 7] and ISA [40] levels have furthered software approximation capability beyond only a binary distinction between accurate and approximate regions [36] to allow for spatially diverse approximation control. Software approximation solutions have studied temporal approximation in the context of loop perforation, which skips a specific iteration subset of an iterative application [21, 25]. They have shown that iterative clustering algorithms like K-Means stabilize their solutions in an early fraction of the iterations (less than 1% change beyond 20% of the iterations).

Tuning for Approximation: It is generally assumed that the tolerable amount of approximation for an application is known as part of the application's or domain's specification. Moreover, it is usually assumed that the programmer can annotate the application to identify which elements can be approximate. Some software proposals have optimized this by requiring the programmers to only specify that a program can be approximate and automatically infer the operations and data that can be safely approximated [26, 34]. After elements for approximation and application error tolerance are identified by programmer/domain specification, prior software proposals have analyzed program-flow and input datasets to identify how much approximation each of the approximate elements can tolerate while maintaining the overall application error tolerance [6, 16, 20, 22, 26]. These satisfy the first three challenges identified in Section 2.2. Some approximation tuning proposals use greedy algorithms in random order over the approximate elements, maximizing approximation on one element before moving to the next [16, 22]. Alternative proposals order the approximate elements by energy per operation [20, 40] and then perform greedy tuning. While this is a reasonable heuristic for optimizing for power savings, it is not very useful for performance or energy consumption, since application performance is too dependent on dynamic application dataflow characteristics to be interpreted as a static per-operation metric. Finally, previous work has utilized gradient descent for application-specific approximation, to combine approximation of memory and compute, targeting Reduce and Rank applications [27]. Our work, however, utilizes gradient-descent-based tuning towards fine-grained approximation for GPAS.

System-level Approximation: The importance of maximizing the execution metric of interest while managing application error has also been stressed in prior work [37], though their focus is on applications whose error and hardware execution metrics can be completely modeled

mathematically, which can be challenging for complex applications with increasing approximate variables, especially when executing on diverse hardware platforms. Similarly, the need for hardware cognizance in approximation has been discussed in some system-level proposals [14, 15], but these works are not tuning approximation at an operation-level granularity, nor are they leveraging spatio-temporally diverse hardware approximation solutions. Further, application-specific approximation systems proposals [12, 28, 30] have build systems wherein each application-specific subsystem's approximation is managed intelligently, depending on the effects of the application on that subsystem. In most scenarios, these approximations are coarse-grained, implementing a very limited number of approximation knobs (often just one) per sub-system. Given an application's overall error tolerance, prior works targeting general purpose approximation [6, 16, 20, 22, 26] have proposed software-centric solutions (analyzing program-flow and input datasets) to obtain some feasible approximation configuration. These proposals perform tuning that is "hardware-agnostic" i.e., the tuning mechanism is oblivious to the effect of the chosen approximation configuration on hardware execution. Such solutions are sub-optimal, since they do not target a specific execution metric (e.g., performance or energy) and ignore features of the execution hardware. While some application-centric systems have analyzed the system as a whole to perform approximation tuning [1, 3, 12, 28, 30], these proposals are either not designed with a general purpose system in mind or they are unsuited to fine granularity and spatio-temporal diversity in approximation.

9 CONCLUSION

SHASTA proposes a novel hardware-software approach to designing efficient *General Purpose Approximation Systems*. It is able to improve hardware approximation capability achieving fine-grained spatio-temporally diverse approximation. At the same time, it adds hardware cognizance to approximation tuning to achieve the optimum execution efficiency under the prescribed error tolerance. Further, it achieves synergic benefits across optimizations, building a closer-to-ideal general purpose approximation system. Via qualitative and quantitative comparisons, we show that SHASTA is able to achieve considerably better benefits compared to prior work (2–15×) and can provide performance speedups or energy savings in the range of 20% to 40%, depending on the goals of the design, a rather significant improvement on top of traditional general purpose processor architectures.

REFERENCES

[1] M. Hanif, A. Marchisio, T. Arif, R. Hafiz, S. Rehman, and M. Shafique. 2018. X-DNNs: Systematic cross-layer approximations for energy-efficient deep neural networks. *J. Low Pow. Electron.* 14, 4 (2018).

[2] O. Akbari, M. Kamal, A. Afzali-Kusha, M. Pedram, and M. Shafique. 2018. Toward approximate computing for coarse-grained reconfigurable architectures. *IEEE Micro* 38, 6 (Nov. 2018), 63–72.

[3] T. Ayhan and M. Altun. 2019. Circuit aware approximate system design with case studies in image processing and neural networks. *IEEE Access* 7 (2019), 4726–4734.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7.

[6] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. 2015. Probability type inference for flexible approximate programming. In *Proceedings of the ACM SIGPLAN International Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*. ACM, New York, NY, 470–487.

[7] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'13)*. ACM, New York, NY, 33–52.

[8] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar. 2010. Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency. In *Proceedings of the Design Automation Conference*. 555–560.

[9] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Architecture support for disciplined approximate programming. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. ACM, New York, NY, 301–312.

[10] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE Computer Society, Washington, DC, 449–460.

[11] G. A. Gillani, M. A. Hanif, B. Verstoep, S. H. Gerez, M. Shafique, and A. B. J. Kokkeler. 2019. MACISH: Designing approximate MAC accelerators with internal-self-healing. *IEEE Access* 7 (2019), 77142–77160.

[12] S. Hashemi, H. Tann, F. Buttafuoco, and S. Reda. 2018. Approximate computing for biometric security systems: A case study on iris scanning. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'18)*. 319–324.

[13] E. L. Hill and M. H. Lipasti. 2006. Stall cycle redistribution in a transparent fetch pipeline. In *Proceedings of the 2006 International Symposium on Low Power Electronics and Design (ISLPED'06)*. 31–36.

[14] Henry Hoffmann. 2015. JouleGuard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 198–214.

[15] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic knobs for responsive power-aware computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. ACM, New York, NY, 199–212.

[16] A. Jain, P. Hill, S. C. Lin, M. Khan, M. E. Haque, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars. 2016. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–13.

[17] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. ACM, New York, NY, 469–480.

[18] Divya Mahajan, Kartik Ramkrishnan, Rudra Jariwala, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Anandhavel Nagendrakumar, Abbas Rahimi, Hadi Esmaeilzadeh, and Kia Bazargan. 2015. Axilog: Abstractions for approximate hardware design and reuse. *IEEE Micro* 35, 5 (2015), 16–30.

[19] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load value approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. IEEE Computer Society, Washington, DC, 127–139.

[20] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the ACM International Conference on Object-oriented Programming Systems Languages & Applications (OOPSLA'14)*. ACM, New York, NY, 309–328.

[21] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. 2010. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE'10)*. ACM, New York, NY, 25–34.

[22] Thierry Moreau, Felipe Augusto, Patrick Howe, Armin Alaghi, and Luis Ceze. 2017. *QAPPA: A Framework for Navigating Quality-energy Tradeoffs with Arbitrary Quantization*. Technical Report. University of Washington.

[23] Vojtech Mrazek, Muhammad Abdullah Hanif, Zdenek Vasicek, Lukas Sekanina, and Muhammad Shafique. 2019. AutoAx: An automatic design space exploration and circuit building methodology utilizing libraries of approximate components. In *Proceedings of the 56th Annual Design Automation Conference (DAC'19)*. Association for Computing Machinery, New York, NY.

[24] K. Nepal, Y. Li, R. I. Bahar, and S. Reda. 2014. ABACUS: A technique for automated behavioral synthesis of approximate computing circuits. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'14)*. 1–6.

[25] A. Panyala, O. Subasi, M. Halappanavar, A. Kalyanaraman, D. Chavarria-Miranda, and S. Krishnamoorthy. 2017. Approximate computing techniques for iterative graph algorithms. In *Proceedings of the IEEE 24th International Conference on High Performance Computing (HiPC'17)*. 23–32.

[26] Jongse Park, Xin Zhang, Kangqi Ni, Hadi Esmaeilzadeh, and Mayur Naik. 2014. *Expax: A Framework for Automating Approximate Programming*. Technical Report. Georgia Institute of Technology.

[27] A. Raha and V. Raghunathan. 2017. Synergistic approximation of computation and memory subsystems for error-resilient applications. *IEEE Embed. Syst. Lett.* 9, 1 (2017), 21–24.

[28] Arnab Raha and Vijay Raghunathan. 2017. Towards full-system energy-accuracy tradeoffs: A case study of an approximate smart camera system. In *Proceedings of the 54th Annual Design Automation Conference (DAC'17)*. Association for Computing Machinery, New York, NY.

[29] A. Raha and V. Raghunathan. 2018. Approximating beyond the processor: Exploring full-system energy-accuracy tradeoffs in a smart camera system. *IEEE Trans. Very Large Scale Integ. (VLSI) Syst.* 26, 12 (Dec. 2018), 2884–2897.

[30] A. Raha, S. Venkataramani, V. Raghunathan, and A. Raghunathan. 2017. Energy-efficient reduce-and-rank using input-adaptive approximations. *IEEE Trans. Very Large Scale Integ. (VLSI) Syst.* 25, 2 (Feb. 2017), 462–475.

[31] A. Rahimi, A. Marongiu, R. K. Gupta, and L. Benini. 2013. A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor clusters. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'13)*. 1–10.

[32] Gokul Subramanian Ravi and Mikko H. Lipasti. 2019. Recycling data slack in out of order cores. In *Proceedings of the IEEE 25th International Symposium on High Performance Computer Architecture (HPCA'19)*.

[33] Sherief Reda and Muhammad Shafique. 2018. *Approximate Circuits: Methodologies and CAD*. Springer.

[34] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. 2014. ASAC: Automatic sensitivity analysis for approximate computing. In *Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'14)*. ACM, New York, NY, 95–104.

[35] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. SAGE: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. ACM, New York, NY, 13–24.

[36] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, NY, 164–174.

[37] Xin Sui, Andrew Lenharth, Donald S. Fussell, and Keshav Pingali. 2016. Proactive control of approximate programs. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. ACM, New York, NY, 607–621.

[38] G. Tziantzioulis, A. M. Gok, S. M. Faisal, N. Hardavellas, S. Ogrenci-Memik, and S. Parthasarathy. 2015. b-HiVE: A bit-level history-based error model with value correlation for voltage-scaled integer and floating point units. In *Proceedings of the 52nd Annual Design Automation Conference (DAC'15)*. ACM, New York, NY.

[39] Vassilis Vassiliadis, Konstantinos Parasyris, Charalambos Chalios, Christos D. Antonopoulos, Spyros Lalis, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2015. A programming model and runtime system for significance-aware energy-efficient computing. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. ACM, New York, NY, 275–276.

[40] Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Quality programmable vector processors for approximate computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. ACM, New York, NY, 1–12.

[41] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. 2017. AxBench: A multiplatform benchmark suite for approximate computing. *IEEE Des. Test* 34, 2 (2017), 60–68.

Received October 2019; revised May 2020; accepted July 2020