As-Is Approximate Computing

MITALI SONI, Google
ASMITA PAL and JOSHUA SAN MIGUEL, University of Wisconsin-Madison

Although approximate computing promises better performance for applications allowing marginal errors, dearth of hardware support and lack of run-time accuracy guarantees makes it difficult to adopt. We present As-Is, an Anytime Speculative Interruptible System that takes an approximate program and executes it with time-proportional approximations. That is, an approximate version of the program output is generated early and is gradually refined over time, thus providing the run-time guarantee of eventually reaching 100% accuracy. The novelty of our As-Is architecture is in its ability to conceptually marry approximate computing and speculative computing. We show how existing innovations in speculative architectures can be repurposed for anytime, best-effort approximation, facilitating the design efforts and overheads needed for approximate hardware support. As-Is provides a platform for real-time constraints and interactive users to interrupt programs early and accept their current approximate results as is. 100% accuracy is always guaranteed if more time can be spared. Our evaluations demonstrate favorable performance-accuracy tradeoffs for a range of approximate applications.

CCS Concepts: • **Computer systems organization → Special purpose systems;**

Additional Key Words and Phrases: Approximate computing

ACM Reference format:
Mitali Soni, Asmita Pal, and Joshua San Miguel. 2022. As-Is Approximate Computing. *ACM Trans. Arch. Code Optim.* 20, 1, Article 3 (November 2022), 26 pages.
https://doi.org/10.1145/3559761

1 INTRODUCTION

Approximate computing is an emerging paradigm shown to be effective for real-time applications and interactive services [15, 26, 39]. Such applications often employ algorithms in machine learning and computer vision and process noisy sensor data and multimedia, all of which can tolerate inaccuracies in output [15, 16, 24, 25, 47]. For such applications, approximate computing can trade acceptable quality loss for improved performance.

Many datacenter workloads are interactive and approximate in nature [26]. However, different users and new inputs make each running instance of the same program unique. Under such scenarios, meeting the **quality-of-service** (**QoS**), i.e., latency) expectations while ensuring **quality-of-results** (**QoR**), i.e., accuracy) becomes challenging. Our proposed approach of approximate

Mitali Soni completed this work while at University of Wisconsin-Madison.
Authors' addresses: M. Soni, Google 1600 Amphitheatre Parkway, Mountain View, CA 94043; email: mitali.soni@wisc.edu; A. Pal and J. San Miguel, University of Wisconsin-Madison, 1415 Engineering Drive (Engineering Hall), Room 3627 Madison, WI 53706; emails: asmita.pal@wisc.edu, jsanmiguel@wisc.edu.

1544-3566/2022/11-ART3 $15.00
https://doi.org/10.1145/3559761

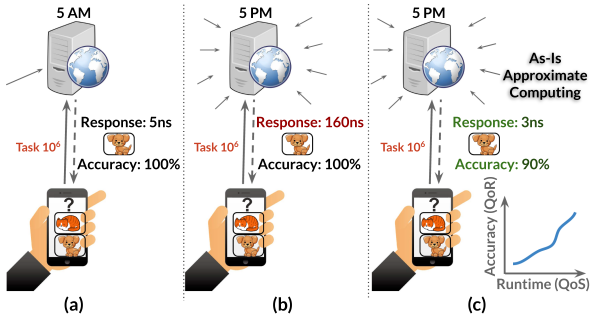

Fig. 1. QoS vs. QoR tradeoff in a datacenter setting. (a) Highest QoS and QoR under light server load. (b) Low QoS but high QoR under heavy server load. (c) High QoS and high QoR with anytime approximate computing even in heavily loaded server.

computing presents an opportunity to improve the efficiency of datacenter workloads by enabling an execution model where QoR scales directly proportional to QoS as shown in Figure 1.

Challenges of Approximate Computing. Despite the potential of this paradigm, it has not been widely adopted in current systems due to the following conventional wisdom.

— *No run-time guarantees on output accuracy.* State-of-the-art approximation techniques lack the ability to dynamically enforce guarantees on accuracy while executing. The convention is to (1) employ offline or online accuracy models [4, 14] or (2) present experimental results that demonstrate empirically low error [4, 25, 47]. Unfortunately, these approaches are still unable to guarantee acceptability of all application outputs on-the-fly. Furthermore, it is unclear if this problem can ever be solved since the notion of acceptability is subjective and impossible to safely quantify.

— *Too big a departure from commodity systems.* To achieve efficiency gains, approximation techniques often impose significant changes to the underlying hardware, requiring fundamentally new designs for compute [13, 14, 48] and storage [20, 37]. The challenge is that different approximations have varying impacts on different applications, in terms of both performance and accuracy. Intuitively, there is no one-size-fits-all solution. It is hard to justify investing significant chip real estate and verification efforts in an approximation mechanism that only helps some applications and potentially compromises the correctness of others.

As-Is Computing. We propose *As-Is*, an *Anytime Speculative Interruptible System* that maps approximate computing to speculative computing, overcoming the key challenges above. Conceptually, we marry the theoretical properties of anytime automata [39] with the practical benefits of speculative parallel architectures [21]. The former allows us to take any approximate application and execute it in a way that produces time-proportional approximations. That is, an approximate version of the program output is generated early and is gradually refined over time, thus providing the run-time guarantee of eventually reaching 100% accuracy. The latter allows us to rework existing innovations in speculative architectures for the purpose of approximation. This lets us implement approximations that work generally on a wide range of applications without having to commit to risky hardware changes.

Contributions. In this article, we offer the following:

— We propose *As-Is*, a novel architecture that takes an approximate program and executes it with time-proportional approximations. This allows a latency constraint or interactive user

Table 1. Comparison with Contemporary Approaches

Related Work	General Purpose	Time-Proportional Approximation	Interruptible
KickStarter [27, 52]	X	✓	X
Brainiac [10, 16]	X	✓	X
Samoyed [35]	✓	X	✓
Truffle [13], Enerj [14, 48]	✓	X	X
Aloe [22]	✓	✓	X
As-Is	✓	✓	✓

to interrupt the program early and accept the current approximate result *as is*. 100% accuracy is always guaranteed if more time can be spared.
— We present the unique challenges of building such a system that marries approximation and speculation. We show how together they can turn a typical speculative multiprocessor into a best-effort, approximate multiprocessor, with low additional hardware complexity.
— We evaluate As-Is and demonstrate favorable-accuracy tradeoffs on a range of approximate programs from machine learning, computer vision, graph analytics, and image processing.

2 BACKGROUND

We discuss the background concepts that are necessary to realize the marriage between approximate computing and speculative computing: anytime automata and ordered irregular parallelism.

2.1 Approximate Computing

Owing to the performance and energy gains from approximation, various approximation frameworks have been explored in the literature. The state-of-the-art general-purpose approximation frameworks are often harder to adopt since they lack execution time proportional gains in accuracy and the ability to interrupt the execution once the accuracy expectations have been met [13, 14, 48]. Moreover, these frameworks often rely on accuracy recovery mechanisms in the event that the output is not adequately accurate [22]. Even though few specialized approaches enable time-proportional approximation and support interruptibility, they are limited to specific algorithms which are being accelerated with approximation and cannot be extended to benefit various different workloads [10, 16, 27, 35, 52]. Table 1 compares the proposed As-Is system against relevant prior work. Kickstarter leverages iterative computation for graph algorithms and calculates intermediate values for subset of vertices. However, unlike As-Is they do not explore other benchmarks or even guarantee interruptibility of the application [52]. Brainiac also supports time proportional approximation using neural accelerators; however, their implementation does not claim that approximate outputs can be retrieved at any time. As-Is on the other hand employs a best-effort approach which allows retrieval of both precise and intermediate outputs [16]. In contrast, Samoyed is able to support an intermittent system by adopting smaller operations which can complete, when a larger operation requires more energy than a device can offer. However, it does not employ any approximation to achieve this, unlike As-Is [35]. Truffle exhibits high energy savings by supporting a low voltage knob for approximate calculations, but this does not guarantee that it will be able to give outputs with varying degrees of accuracy based on how long the system is allowed to run. As-Is is able to take care of the latter scenario along with energy savings [13]. Aloe is able to effectively demonstrate program reliability, but if the program is partially executed, as in for a shorter time, the checkers will consider any intermediate outputs as unacceptable [22].

2.2 The Anytime Automaton

The Anytime automaton [39] leverages anytime algorithms [11, 19, 29] that have been widely used in decision planning and artificial intelligence, to tackle key challenges in approximate computing. In the anytime automaton, increasingly accurate approximate versions of the final output are computed and used to eventually produce the final precise output. A key feature that anytime automaton provides is program interruptibility. At any point, if the program is interrupted, the automaton simply produces a *best-effort* or *as-is output* based on previously executed computations. This feature is useful since the application can now be stopped as soon as the output quality meets user expectations, saving both runtime as well as energy. In cases where the output quality falls below expectations, the user simply needs to let the application run longer.

The anytime automaton computation model provides a mechanism to extract significant parallelism from seemingly sequential applications (details of applications in Section 4.2). It accomplishes this by breaking down an application into a pipeline of fine-grained anytime computation tasks that can execute in parallel.

2.3 Ordered Irregular Parallelism

An ordered irregular parallel application consists of tasks with the following properties [21]. First, tasks follow a total or partial order. Second, tasks are dynamically created and creation order is different from execution order. Third, tasks may have data dependencies that are not known *a priori*. Ordered irregular parallel algorithms are common in data mining [50], graph analytics [18, 43], and event-driven systems [41].

An anytime automaton can also be considered as an ordered irregular parallel application. This is because in anytime automaton, each computation stage A is broken down into N tasks (i.e., A_1, A_2, …, A_N), where N is input-dependent and may not be known *a priori*. The parent-child relationships between tasks are known, but the data dependencies between tasks are unknown in advance. Each task is also assigned a timestamp and may create and enqueue child tasks with any timestamp equal to or greater than its own, leading to a total order among tasks.

2.4 Anytime Execution Model

Figure 2(a) shows an example program broken down into two computation stages S and f. We will use the same example throughout the article. The stage S samples all the pixels of an input image and builds a histogram of pixel intensities. The stage f takes a histogram as input and computes the arithmetic average of all the pixels. The stage S is made anytime with bit-reverse sampling and anytime stage f leverages reduced precision approximation. The stage S is broken down into three tasks and stage f is broken down into two tasks (Figure 2(b)). An execution schedule of tasks is shown in Figure 2(c).

3 AS-IS COMPUTING

In this section, we present the As-Is system, that maps anytime approximate computing to speculative multiprocessing. As-Is is a system-level realization of the anytime automaton theoretical model [39] and provides a practical implementation of task-level speculation for ordered irregular parallel applications.

As-Is benefits from the theoretical properties guaranteed by anytime automaton [39]. It is also inspired by prior works in conventional speculative computing [21]. However, prior works are not well suited for anytime computing because of the following challenges, which As-Is aims to address:

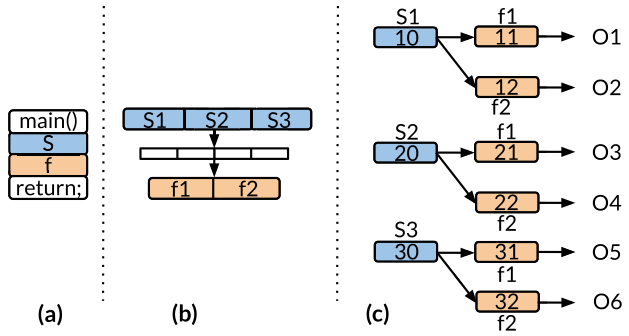

Fig. 2. Overview of anytime execution model. (a) Approximate program with anytime stages S and f. (b) Dataflow with three tasks for stage S and two tasks for stage f. (c) Dataflow execution of μTasks (described in Section 3).

(1) No programming model specification. As-Is provides a task-based programming model for expressing an application as an anytime automaton.
(2) No means to exploit the parallelism exposed. We recognize that we can maximize and extract the anytime parallelism by framing it as an ordered irregular speculative parallel application.
(3) No mapping to general-purpose hardware. As-Is builds a multiprocessor architecture and run-time system to efficiently execute anytime applications and handle task creation, scheduling, execution, and retirement.
(4) No support for best-effort execution. The As-Is system maps the theoretical iterative and diffusive stages to physical mandatory and best-effort tasks that are recognized and supported by the underlying architecture.
(5) No data dependence handling and conflict resolution. This work also provides a formal understanding of all possible true and false data dependencies among various tasks as well as outlines the conflict detection and resolution mechanisms needed to ensure that the eventual precise output is guaranteed.
(6) No support for explicit forms of approximation. As-Is augments the conventional speculative computing substrate with support for approximation techniques such as random sampling, bit-reverse sampling, and reduced precision arithmetic.

We first present the As-Is programming model that translates an application into an anytime automaton. The next section outlines the As-Is runtime system and provides details on task management for creation, scheduling, execution and retirement of μTasks. We then describe architectural support for approximation, conflict detection, and resolution. The section concludes with a walkthrough example.

3.1 As-Is Programming Model

As-Is uses a task-based programming model to declare programs in the form of anytime automata. Programs are composed of computation stages arranged in a feed-forward dataflow, as shown in Figure 2(a). Explicit buffers are used to pass information between parent and child stages (Figure 2(b)). The programmer specifies M computation stages: $S_0 \ldots S_{M-1}$, in a high-level language such as C, along with per-stage information regarding the input buffer location, output buffer location, and any applicable approximation techniques (e.g., random input sampling and reduced precision). We discuss supported approximations later in Section 3.2.3. An example annotated program for the anytime model in Figure 2 is shown in Listing 1.

```
// computation_stage_type(approximation_method, num_of_tasks, child_stage)
#pragma asis diffusive(outsampling, 3, f)
void s(int utask, void **in, void **out) {
    ...
}
#pragma asis diffusive(precision, 2) {
void f(int utask, void **in, void *out)
    END();
}
int main() {
    START(s);
}
```

Listing 1. Example As-Is program snippet for anytime execution model in Figure 2.

Computation Stages. There are three types of computation stages that can be declared.

— *Precise:* A precise stage applies no approximations and is executed exactly as defined.
— *Iterative:* An iterative anytime stage performs its computation in multiple iterations, each iteration with higher accuracy than the previous. Iterations are independent of one another; each iteration overwrites the previous iteration's (now stale) result in the output buffer upon completion.
— *Diffusive:* A diffusive anytime stage performs its computation in multiple steps, each step building on the accumulated result to bring accuracy closer to 100%. Steps may be dependent on one another and successful execution of each step is necessary for 100% accuracy.

As per the nomenclature in numerical methods, diffusive stages in As-Is are mapped to iterative stages where repeated state updates improve the solution with some convergence rate. However, we use the above terminology to remain consistent with the prior work this article is based on [39].

Computation Tasks. Each anytime computation stage S_i is decomposed into N tasks: $T_{Si,1}$... $T_{Si,N}$. These tasks are either iterations of an iterative stage or steps of a diffusive stage. The output of an iterative stage is 100% accurate upon completion of task $T_{Si,N}$, while the output of a diffusive stage is 100% accurate upon completion of all N tasks. In the Figure 2(b) example, anytime stages S and f are executed as three and two tasks, respectively.

The number of tasks per stage can either be programmer-specified or dynamically selected at run-time. In our implementation, we profiled our applications and manually specified the number of tasks. Automating this process is left for future work.

Pipeline. The feed-forward composition of tasks can be executed as a pipeline as shown in Figure 3. There are two types of pipeline interactions listed below. Our As-Is run-time system can infer the appropriate pipeline depending on the types of parent and child stages.

— *Asynchronous:* An asynchronous pipeline can be formed between any type of parent and child stages given that the stages can execute concurrently and independently. Each parent task spawns a new instance of the child stage, which implies multiple (redundant) instances of each child task. Parent and child tasks can execute concurrently, independent of one another.
— *Synchronous:* A synchronous pipeline can be formed only if (1) the parent stage is diffusive, and (2) the child's computation is distributive over the parent's computation. Parent and child tasks can execute concurrently but must synchronize to produce the 100% accuracy and to reduce the redundant work performed in the asynchronous pipeline.

Example. Figure 3 illustrates these concepts in practice, for the example program in Listing 1. Figure 3(a) shows the baseline sequential execution of the program. The first stage `sampling` is

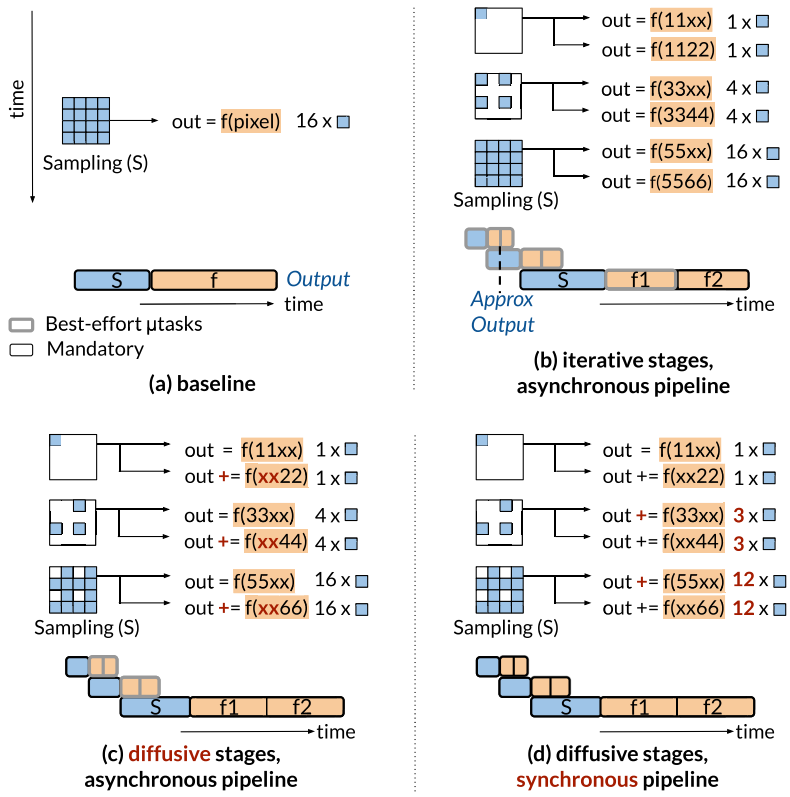

Fig. 3. Varying configurations of anytime stages and pipelines for the example program in Listing 1. The top shows the computations performed by the two stages over time (vertical axis), while the bottom shows how the μTasks are scheduled for pipelined execution (time is horizontal axis). Best-effort μTasks are shown with a light border.

composed of three tasks, while the second stage `f` is composed of two tasks. With iterative stages in an asynchronous pipeline (Figure 3(b)), each later task potentially recomputes results that were previously computed and overwrites the current output with a more accurate version. `f(11xx)` represents the output of the first task of stage `f` after processing the output of its parent task i.e., first task of stage `S`, whereas `f(1122)` represents the second and final output generated by the stage `f` over the same output from the first task of stage `S`.

With diffusive stages in an asynchronous pipeline, child stage `f` processes different versions of outputs produced by parent stage `S` as opposed to merely updates of them. However, with diffusive stages in a synchronous pipeline (Figure 3(d)), each later task builds directly on top of previously computed results, making the cumulative output more accurate.

3.2 As-Is Runtime System and Task Management

3.2.1 Definitions. We first define key terms, which will be described in detail in the following sections.

— *μTask:* A unique instance of a task. For example, in Figure 3, the second stage `f` is composed of two tasks for each of its parent's three tasks, yielding 6 μTasks.

— *Mandatory μTask:* A μTask whose result is necessary to obtain 100% accuracy in the final program output.
— *Best-Effort μTask:* A μTask whose result is not needed to obtain 100% accuracy in the final program output.
— *Complete:* A μTask completes when it finishes producing its output and schedules its child μTasks for execution.
— *Retire:* A μTask retires when it has completed and is the oldest non-retired μTask in the system.
— *Abort:* A μTask aborts (and re-executes if mandatory) when its data accesses conflict with another μTask that takes priority.

As described in previous sections, computation stages in the program are approximated in an anytime manner. This leads to multiple μTasks, each contributing towards higher accuracy in the program output. The subsequent subsections detail the lifetime of each μTask.

3.2.2 μTask Creation. Each μTask is uniquely defined by an M-digit μID, where M is the number of computation stages. This is shown in Figure 2(c). A μTask's μID is set such that it satisfies all of the following:

— μID is greater than its parent's μID.
— μID is greater than its predecessor's μID (e.g., S_2's predecessor is S_1).
— μID is less than the μIDs of all its ancestors' successors (e.g., the successors to μTask 12's ancestors are 21 and 22).

μIDs are assigned at the time of μTask creation. They specify the total order in which μTasks appear to execute such that the program output increases in accuracy over time and is guaranteed to reach 100%. Similar to timestamps in ordered irregular speculation [21], μIDs are used to resolve all data conflicts among best-effort and mandatory tasks. The details of data dependencies and conflict resolution are described later in Sections 3.2.6 and 3.3.4, respectively.

To schedule μTasks on available cores, the As-Is run-time system employs a priority queue that schedules tasks based on their μID.

3.2.3 μTask Execution. μTasks with lower μIDs are scheduled for execution whenever there is an available core. All μTasks may run concurrently and any data dependencies among them are resolved dynamically.

3.2.4 μTask Completion and Retirement. Due to the concurrent execution of μTasks, a μTask with a higher μID may complete before one with a lower μID. Such a μTask resides in the task queue (Figure 5, discussed is 3.3) until it is the lowest μID in the system, upon which it retires.

3.2.5 Best-effort and Mandatory μTasks. The As-Is architecture leverages speculative execution to perform its approximations and extract pipeline parallelism inherent to the anytime automaton execution model. In conventional architectures, speculative tasks are squashed, rolled back, and re-executed when a mis-speculation is detected. However, in As-Is, not all μTasks are mandatory. That is, some μTasks are executed only for the purpose of producing an approximate intermediate result before the final output is generated. Such tasks may be best-effort, implying that they do not contribute to the 100%-accurate output and thus do not need to be re-executed if aborted.

Formal Definitions. Formally, given a task $T_{Si,j}$, which is the jth task in stage S_i (consisting of N tasks in total), this task is defined to be *best-effort* if S_i is iterative and $j < N$. Consequently, a μTask instance of $T_{Si,j}$ is best-effort if either (1) task $T_{Si,j}$ is best-effort, or (2) its parent μTask is

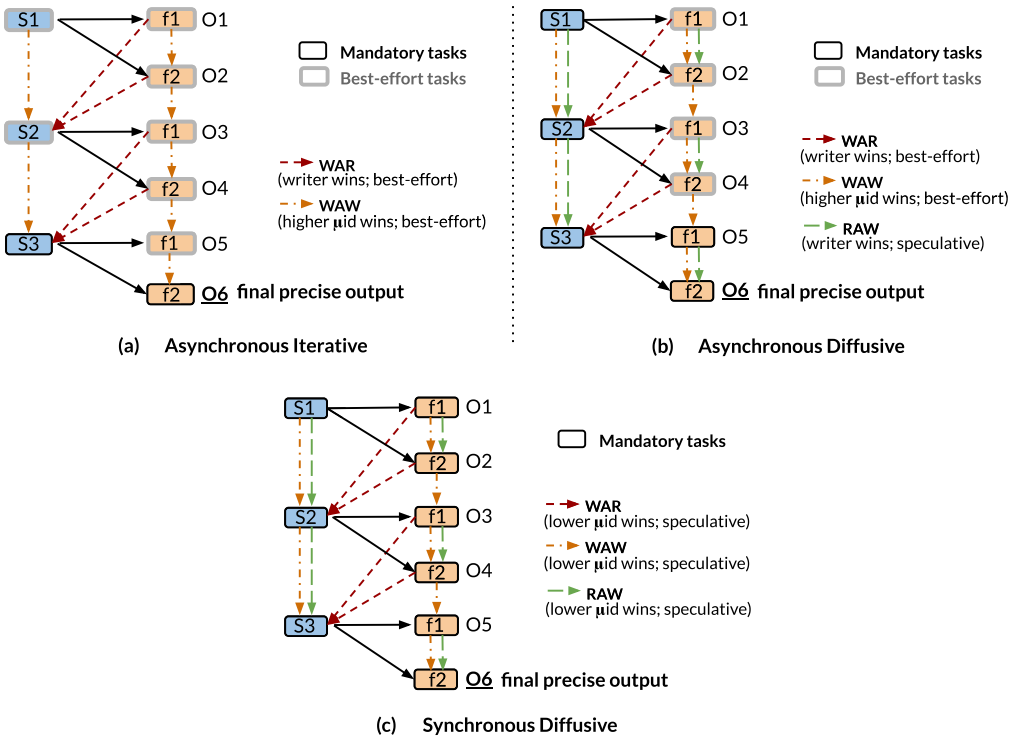

Fig. 4. Illustration of which μTasks are best-effort or mandatory (critical) and which data hazards are possible (and how they are resolved), depending on the stage-pipeline organization: (a) iterative stages in asynchronous pipeline; (b) diffusive stages in asynchronous pipeline; (c) diffusive stages in synchronous pipeline.

best-effort. All other μTasks are *mandatory*. Figure 4 shows which μTasks are best-effort depending on different stage-pipeline organizations.

Practical Implications. As-Is executes all tasks speculating that there are no data dependencies between tasks. If a dependence violation for a task is detected, the task is said to have been mis-speculated. Best-effort μTasks mitigate penalties of mis-speculation for intermediate approximate outputs. Upon a data conflict, a best-effort μTask is aborted without any re-execution. Other μTasks that are descendants of the aborting μTask are also terminated; descendants are found by walking the task queue and identifying any μTasks that have the same most significant digits in their μIDs as the aborting μTask. Aborted best-effort μTasks may reduce the number of intermediate outputs produced, but computations for the final precise output remain unaltered. Successful execution of best-effort μTasks is desirable but not critical to the program execution. In contrast to best-effort μTasks, a mandatory μTask is considered critical for the application since each mandatory task contributes to the final precise output. This is demonstrated by the example in Figure 3. Thus mandatory μTasks are always re-executed upon a conflict. Since mandatory μTasks are critical to the execution, they are strictly ordered in completion; whereas best-effort μTasks can complete in any order.

3.2.6 μTask Data Dependencies. We describe in detail how data dependencies arise for the different possible stage-pipeline configurations, shown in Figure 4.

Iterative Stages, Asynchronous Pipeline. In this case, each μTask reads whatever data is currently available in its input buffer. As a result, there are no dependencies between concurrently executing μTasks of the same stage. In the example shown in Figure 4(a), only μTasks $S_3 \rightarrow f_2$ are needed for the precise (i.e., 100%-accurate) program output. All other μTasks do not fall on this critical-accuracy path and are thus best-effort. To guarantee that the final precise output is eventually reached in this case, mandatory μTasks need to win all of their data conflicts. Since the output buffer is shared by all the μTasks in a computation stage, the highest μID at every computation stage wins all write-after-write (*WAW*) conflicts. The writer μTask among producer-consumer stages wins a write-after-read (*WAR*) conflict as the reader task is expected to be operating on older stale data at the time of update. Since all stages are iterative, μTasks within the same stage do not depend on each other's results; thus read-after-write (*RAW*) dependencies do not exist.

Diffusive Stages, Asynchronous Pipeline. With diffusive stages, μTasks use the previously computed output and refine them for more accurate outputs. Thus with the diffusive model, the final μTask of every computation stage needs to complete for the final precise output. In this case, the critical-accuracy path consists of μTasks $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow f_1 \rightarrow f_2$ as shown in Figure 4(b), each of which is treated as a mandatory task. False dependencies (*WAW* and *WAR*) are handled as previously discussed for an asynchronous pipeline. True (*RAW*) dependencies may now exist. If S_2 depends on S_1's output, and a conflict arises during their concurrent execution, then S_2 must be aborted and re-executed.

Diffusive Stages, Synchronous Pipeline. In a synchronous pipeline, all μTasks compute a portion of the final precise output and are essential for correct execution. Thus all μTasks are mandatory as depicted in Figure 4(c). On every conflict, μTasks with a lower μID take priority, and higher μIDs are re-executed. Even though the synchronous pipeline resembles a conventional task-speculative architecture, As-Is is able to produce time-proportional approximate outputs and can be stopped as soon as the intermediate program output is deemed acceptable.

3.3 As-Is Architecture

This section describes the As-Is architectural components and how they operate in detail.

3.3.1 Overview. Figure 5 shows the organization of the As-Is architecture. As-Is is built as a cache-coherent shared-memory multiprocessor. Any data dependencies between tasks are detected and resolved dynamically, inspired by task-level speculation [21]. Though we implement a homogeneous multiprocessor, As-Is can support heterogeneous cores, which can allow for flexibility in where and how approximations are applied. All-in-all, this organization enables time-proportional approximations. For latency-critical or user-interactive applications, the program may be interrupted early, accepting the current approximate result as is. However, if accuracy becomes a concern depending on application and system variations, 100% accuracy is still guaranteed eventually; i.e., As-Is can always operate as a traditional speculative multiprocessor.

3.3.2 Task Queue. All μTasks dispatched for execution are allocated an entry in the task queue (Figure 5) to track their progress. The task queue can be small; our implementation uses a task queue size of 32 entries. A larger task queue can be easily supported for more processor cores. All μTasks may run concurrently; any data dependencies among them are resolved by the conflict resolution units (CRs in Figure 5), discussed later. Note that during execution, the task queue entries are not modified; they are only needed during μTask creation and completion. When a μTask completes, it updates its task queue entry and pushes its output data to memory. This is because we opt for a lazy versioning scheme, detailed in later sections. Even after completion, a μTask may

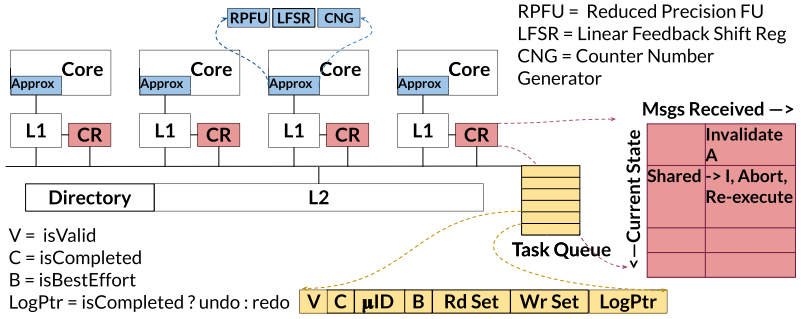

Fig. 5. As-Is architecture.

still be aborted due to a data conflict. Upon retirement, the task queue entry is removed, and any log due to versioning is cleared.

3.3.3 Approximation. We implement hardware support for three fundamental approximation primitives that can be applied to an anytime stage:

— *Input sampling with a pseudo-random permutation:* Instead of fetching and operating on input data elements sequentially, a **linear-feedback shift register** (**LFSR**) is used to process them in a pseudo-random order. This is useful for many computations that perform reductions or collect statistics of their input data. The input data elements are viewed as part of a probability distribution and are processed in an unbiased manner via the LFSR. Each processor core is equipped with an LFSR counter coupled to the register file, acting in lieu of the loop iterator register when input sampling is enabled. μTasks of an input-sampled stage use the same LFSR seed values, and each μTask processes exponentially more elements than its predecessor.

— *Output sampling with a bit-reverse (tree) permutation:* For sampling data elements of the output, we implement logic to permute bits of the loop iterator register in a bit-reverse manner. This effectively produces the output elements such that the output gradually increases in resolution. An example is shown in Figure 3 for a 4×4 dataset. Naturally, each μTask in an output-sampled stage processes exponentially more elements than its predecessor.

— *Bit sampling (i.e., reduced precision):* We support low-precision units for long-latency integer arithmetic operations (e.g., multiplication), drawing from prior work on subword-level approximation [15]. The most-significant bits are processed first, yielding an early estimate of the final value.

Though other approximation techniques can be supported, we focus our efforts on these three primitives since they are the most general and cover a wide range of applications. Furthermore, they require only minor changes to the microarchitecture. The Approx unit in Figure 5 consists of three subunits—(1) **Reduced precision functional unit** (**RPFU**) to support reduced precision approximation, (2) LFSR for pseudo random input sampling, and (3) **Counter number generator** (**CNG**) to enable bit-reverse output sampling.

3.3.4 Conflict Detection and Resolution. **Data Restructuring.** When applying input or output sampling, data elements are processed in a non-sequential order that breaks spatial locality. To address this, As-Is supports restructuring the datasets such that the elements are prearranged in memory in the same order as the corresponding permutation (i.e., LFSR or tree). This is

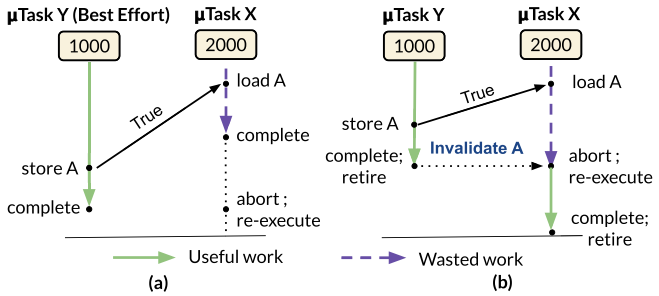

Fig. 6. As-Is conflict resolution in the presence of true dependencies.

straightforward since these permutations are deterministic and statically known. Thus, As-Is is able to retain the inherent spatial locality of the original program.

This section details how true and false data dependencies are handled. While executing, all μTasks maintain a read set and a write set (implemented via bloom filters in the conflict resolution units, Figure 5) that track all memory addresses that are accessed via load and store instructions, respectively. Note that conflict detection operates at a cache block granularity to simplify the hardware implementation and piggyback on cache coherence messages. As-Is employs a lazy versioning and conflict resolution scheme, where all store values are buffered in a redo log and pushed to memory only upon μTask completion. We opt for a lazy scheme since it simplifies how false dependencies are handled, unique to our asynchronous and synchronous pipeline organizations. Upon completion, since it is still possible for a best-effort μTask to be aborted prior to retirement, an undo log is atomically created while pushing the store values to memory. The read and write sets are also copied from the μTask's local conflict resolution unit to its task queue entry so that other μTasks can check it for conflicts.

True Dependencies. We outline how two tasks Y and X, with μIDs of 1,000 and 2,000, respectively, execute on our architecture in the presence of a *RAW* dependence between them. There are three possible scenarios that can occur at run-time:

(1) If X loads the dependent memory location A after Y completes, there would be no hazard as X would have read the data value written by Y. This is ensured via traditional cache coherence.
(2) If X loads A and completes before Y completes (which is only possible if Y is best-effort) as shown in Figure 6(a) when Y completes and retires, it walks the task queue and compares its read and write sets with all completed μTasks with higher μIDs. If such a conflicting μTask X is found, it is aborted and re-executed.
(3) If X loads A and completes after Y as shown in Figure 6(b), it would receive a coherence invalidation from Y while executing. We augment coherence messages to include the μIDs of the requesting μTasks. This is then used by the local conflict resolution unit to catch the conflict and abort (and re-execute) X.

False Dependencies. We outline how the two tasks from our previous discussion, Y and X, execute in the presence of false dependencies between them. There are four possible scenarios:

(1) If X completes after Y completes, there would be no hazard as task X would overwrite the dependent memory location A after Y has accessed it, as shown in Figure 7(a).
(2) If Y is mandatory (i.e., executing speculatively) then no other μTask with higher μID is allowed to complete before Y since all μTasks are strictly ordered with mandatory μTasks. Thus this scenario shown in Figure 7(b) is impossible.

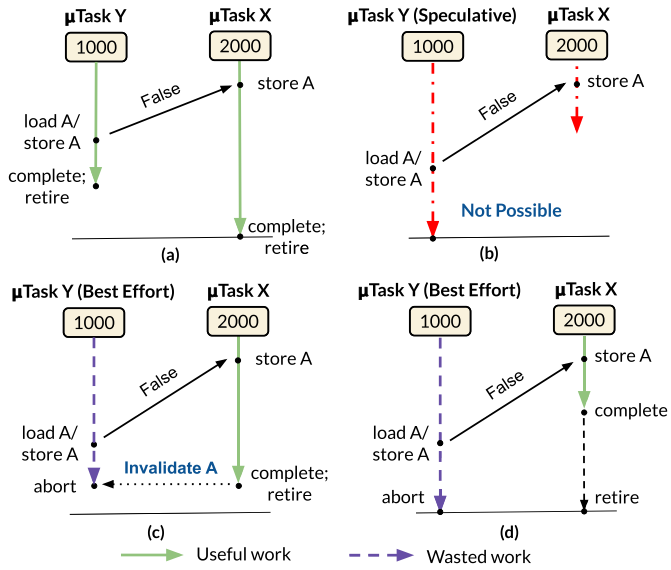

Fig. 7. As-Is conflict resolution in the presence of false dependencies.

(3) If X completes before best-effort Y as shown in Figure 7(c) when X pushes its store values, an invalidation request is sent to the core executing Y. Since Y is best-effort, it would simply be aborted by the conflict resolution unit without re-execution.
(4) If X completes before best-effort Y has even accessed A, when Y attempts to complete, it observes the conflict with X (a higher μID) at the task queue and aborts itself. This is shown in Figure 7(d).

The CR unit in Figure 5 consists of a finite state machine that responds to incoming coherence messages based on a block's current state. The CR unit is presented as a look-up table with a sample entry for the example in Figure 6(b)).

3.4 Walk-through Example

In this section, we present an example of Histogram equalization (`histeq`) execution on the As-Is system as depicted in Figure 8. (`histeq`) is commonly used to enhance the contrast of an image by scaling its pixel intensities. In this example, there are three computation stages—A, B, and C. Stage A is a diffusive stage and uses input sampling with pseudo-random permutations. Stage B is a precise stage without any approximations. Stage C is a diffusive stage that employs output sampling with bit-reverse permutation. Stage A has two tasks A_1 and A_2, stage B has one task and stage C has two tasks C_1 and C_2. Even though stage B has a single task, it has two μtasks since there are two parent tasks A_1 and A_2. Similarly, stage C has four μtasks. As the μtasks execute, they produce intermediate output O_1, O_2, and O_3 with increasing accuracy which translates to improved contrast for (`histeq`). Precise output O4 is produced in the end.

4 METHODOLOGY

This section describes our experimental configuration and applications.

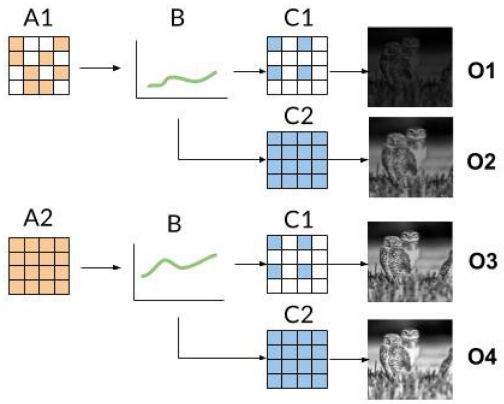

Fig. 8. Execution of histogram equalization benchmark on the As-Is architecture.

4.1 Experimental Setup

We simulate the proposed As-Is architecture on a custom cycle-level, event-driven microarchitectural simulator based on Pin [34]. The simulator models simple x86-based in-order cores running at 2 GHz. We simulate a two-level cache hierarchy where each core has a 32 kB private, write-through L1 cache, and all cores share a unified 2 MB L2 cache. In addition to traditional microarchitectural components, the simulator models As-Is overheads such as conflict detection, mis-speculation penalties of aborted tasks, and re-execution of aborted tasks in detail. For power analysis, we use McPAT's [32] in-order core and memory hierarchy models at 32 nm logic.

4.2 Applications

We evaluate As-Is on a range of approximate applications from PERFECT [5], AxBench [53], and SNAP [28]. PERFECT contains kernels from the embedded computing domain whereas AxBench is an approximate computing benchmark suite. SNAP is used for analyzing the scope of approximation in large graphs. We focus on six widely used kernels from diverse application domains which can benefit from approximate computing. We divide each benchmark into computation stages and apply appropriate approximation techniques per computation stage. Each anytime computation stage is further split into μTasks at the time of execution. Table 2 lists all the benchmarks along with the configuration used in the evaluation. We use **normalized root mean square error** (**NRMSE**) for measuring the error of intermediate approximate outputs. We compute accuracy (i.e., QoR) as `(1- NRMSE)` of the approximate output against the precise output.

The following approximate techniques in our evaluation:

— Input data sampling with random permutation.
— Output data sampling with bit-reverse or tree permutation.
— Reduced fixed-point precision.

Sampling techniques are amenable to anytime computing as data elements can be sampled with progressively decreasing granularity. For unordered data, to avoid bias in the memory ordering, data is sampled in a random manner. In our experiments, we use a simple LFSR implementation for random number generation. When the position of data is significant such as in images and videos, sampling data in bit-reverse or tree permutation can improve the accuracy of intermediate outputs. Reduced fixed point precision techniques can also be made anytime by computing data bits in the order of significant (most significant bit to least significant bit).

Table 2. As-Is Configuration Parameters for Different Applications

	Computation Stage Parameters				
Application	**Stage**	**Is Anytime ?**	**# μTasks**	**Computation Mode**	**Approximation Technique**
Histogram Equalization (`histeq`)	Stage 1	Yes	2, 4, 8	Diffusive	Random Input Sampling
	Stage 2	No	1	Precise	-
	Stage 3	Yes	2	Diffusive	Tree Output Sampling
2D Convolution (`2dconv`)	Stage 1	Yes	8, 16, 32	Diffusive	Reduced Precision, Tree Output Sampling
Discrete Wavelet Transform (`dwt`)	Stage 1	Yes	8, 16, 32	Diffusive	Tree Output Sampling
3* K-Means Clustering(`kmeans`)	Stage 1	Yes	2, 4, 8	Diffusive	Random Input Sampling
	Stage 2	No	1	Precise	-
	Stage 3	Yes	2	Diffusive	Tree Output Sampling
Debayering (`debayer`)	Stage 1	Yes	8,16,32	Diffusive	Tree Output Sampling
Betweenness Centrality (`centrality`)	Stage 1	Yes	8,16,32	Diffusive	Random Input Sampling
PageRank (`pagerank`)	Stage 1	Yes	8,16,32	Diffusive	Random Input Sampling
	Stage 2	No	1	Precise	-

2dconv. 2d convolution is used in computer vision, signal processing, and machine learning. Each output pixel is computed by the average of dot product sum of input pixels and the filter. We retain all the computations of 2dconv in a single stage anytime automaton. 2dconv benefits from both tree sampling as well as reduced fixed point precision.

histeq. Histogram equalization improves image contrast by calculating the cumulative distribution of pixel intensities and is commonly used for thermal, satellite, and x-ray images. The anytime automaton of this benchmark consists of three three computation stages—Stage 1 processes all the input pixels, Stage 2 calculates the scaling factor of each pixel for better contrast, and Stage 3 builds the final image.

dwt. Discrete wavelet transform is employed in data compression. We construct an automaton with a single stage that performs a discretely-sampled wavelet transform on an image. We evaluate the accuracy of this kernel by executing inverse transform precisely and comparing it to the precise output.

kmeans. K-means clustering partitions pixels of an image to different clusters based on its distance from the nearest mean and used in data mining applications. In this three stage automaton, the first stage computes the cluster centroids and assigns pixels to clusters based on their Euclidean distances, the second stage computes the average centroid values and the last stage samples the pixels and re-assigns them to a clusters.

debayer. Debayering algorithm takes an undersampled image from an image sensor and overlays a color filter array, to create a complete color image. A single-stage automaton suffices the computational complexity involved in the interpolation needed to re-create the complete image. To accommodate the scope of multiple μTasks, we apply a tree-based sampling.

centrality. Betweenness centrality is a metric based on shortest paths between nodes, and used extensively to analyze social networks. It takes an unweighted graph and uses the Brandes algorithm for calculating betweenness centrality which is the fraction of shortest paths that pass through n [28].

pagerank. Pagerank is used by search engines to evaluate the importance of a web page for deciding the order of search results. The pages act as nodes and the hyperlinks as the edges of a graph.

In our experiments, each benchmark is run with variable number of μTasks per anytime computation stage. The runtime numbers for these experiments are normalized with respect to the baseline system that executes each computation stage as a single μTask.

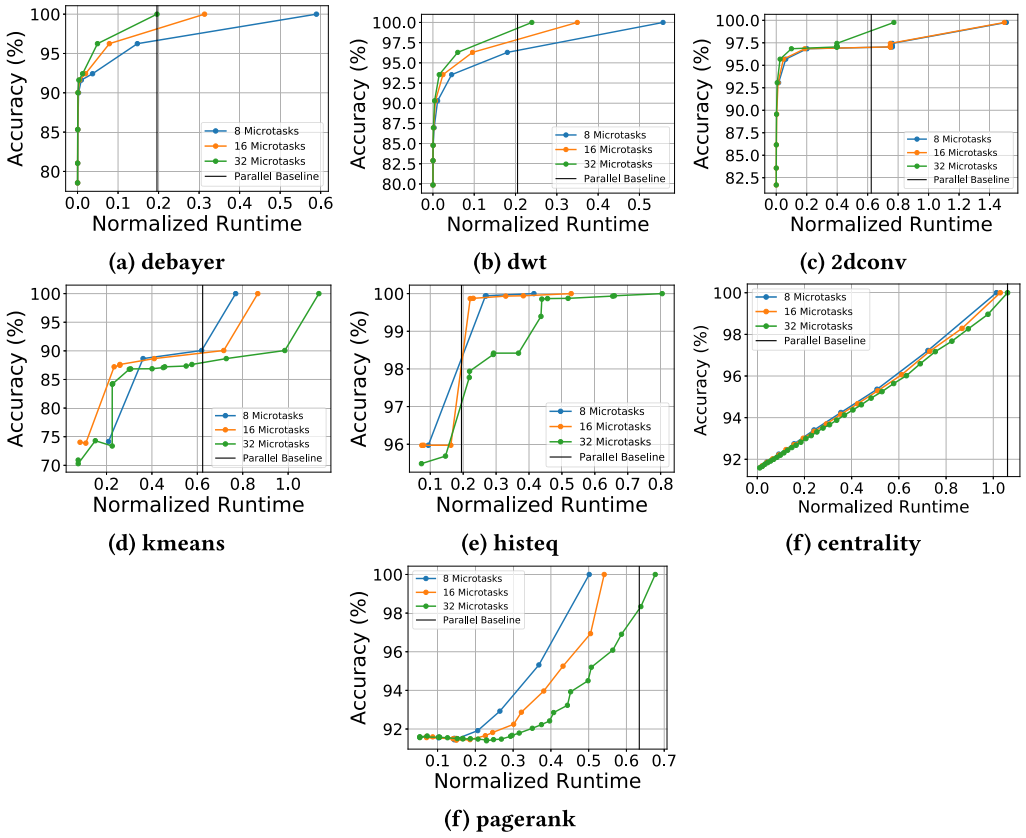

Fig. 9. Runtime-accuracy profiles for all benchmarks for μtasks = (8, 16, 32) on 4 processors.

5 EVALUATION

We evaluate the As-Is system for program output accuracy, runtime benefits, and power consumption. We present our results, analysis, and design considerations in this section.

5.1 Runtime-Accuracy Tradeoffs

In this section, we present the performance-accuracy tradeoffs enabled by our As-Is system. Figure 9, shows runtime-accuracy profiles for all the benchmarks on three configurations of μtasks each: 8, 16, and 32 on a 4 processor system. The y-axis represents accuracy in terms of (1- NRMSE) and the x-axis is run time normalized to the sequential execution of the same program. For a fair comparison, we also indicate the execution time of the parallel implementation of the same program using pthreads with 32 threads and 4 processors.

For all the applications, the As-Is system produces several outputs of high accuracy much earlier than the parallel baseline. In debayer, dwt, and 2dconv outputs with over 90% accuracy are produced as early as 1% of the sequential execution time as seen in the runtime-accuracy profiles in Figure 9(a), (b), and (c). These applications are embarrassingly parallel and thus, the precise output is reached faster with increasing number of μtasks with many tasks executing concurrently and probability of hazards with random sampling being low.

It is worth noting that 2dconv takes longer to reach the precise output. This is because 2dconv employs reduced precision approximation which helps in saving only the latency of multiplies and

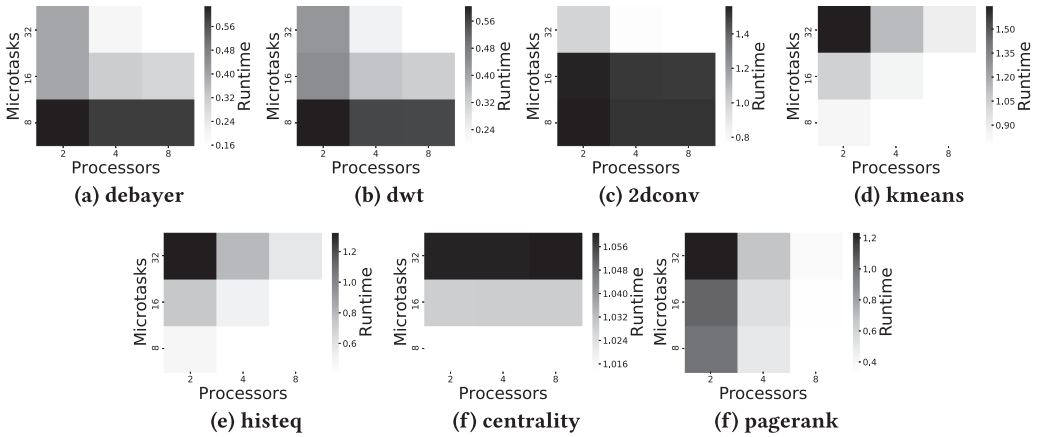

Fig. 10. Time taken to achieve 100% accuracy for different numbers of μtasks = (8, 16, 32) and processors = (2, 4, 8).

in our experiments, multiplies take only a small fraction of the run time. Even though reduced precision is one of the heavily studied approximation techniques, we find its utility is not as effective for the applications we studied for the following two reasons: (1) Reduced precision only seems to work for applications where the latency of arithmetic instructions dominates the execution time. (2) Reducing the precision creates RAW hazards among the tasks which prolong the time to reach the precise output.

K-means and histeq consist of three computation stages and perform many redundant computations to generate intermediate approximate outputs. For very large unweighted graphs, the centrality or pagerank computed after processing a small subset of nodes is highly representative and thus, both metrics start from a fairly high accuracy. However, all tasks end up updating each node's centrality leading to more conflicts thus, resulting in almost sequential run time. In contrast, pagerank is embarrassingly parallel and takes 60% time to reach the precise output. Conflicts in pagerank primarily occur between μtasks from two stages, unlike centrality.

5.2 Sensitivity Study: μTasks and Processors

We now look at the trends observed in the runtime-accuracy profiles further by showing how long it takes to achieve 100% accuracy as we vary the number of μTasks and processors in Figure 10. More processors present more opportunities to exploit parallelism. As a result, increasing the number of processors reduces the execution time for almost all the applications. Similarly, increasing number of μTasks allows many tasks to execute concurrently. Thus, increasing the μTasks from 8 to 32 in embarrassingly parallel applications like debayer, dwt, and 2dconv reduce the run time as seen in Figure 10(a), (b), (c), and (f).

Though the first stage of pagerank is embarrassingly parallel, pagerank has a second stage that is non-anytime. Thus, higher number of μTasks in the first stage lead to more redundant executions of the non-anytime second stage. This is the general trend for all benchmarks that have a non-anytime stage, i.e., pagerank, histeq, and kmeans. With many tasks trying to access similar data, the conflict rate of tasks increases. Thus, for these benchmarks, the runtime to achieve 100% accuracy increases with the number of μTasks due to increased conflicts and redundant execution. As all the μTasks in centrality update the same data, the runtime is less affected by increase in number of available processors. Centrality is more prone to hazards with increase in μTasks as all the tasks modify the same centrality table.

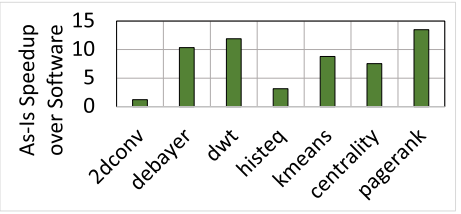

Fig. 11. Speedup achieved by As-Is over software implementation for 95% accuracy.

5.3 As-Is and Software Only Implementation

We compare As-Is with a software only implementation, shown in Figure 11 for 8 μTasks and 4-processor configuration. As-Is reaches 95% accuracy 1.22× −11.89× faster than the software implementation, due to significant software overheads. As expected, the applications that take longer to achieve 95% accuracy see significant benefits from As-Is over the benchmarks that achieve high accuracy within the first few μTasks.

5.4 As-Is and Conventional Speculative Architectures

Figure 12 presents a comparison of As-Is hardware and software components with the closest relevant speculative architecture [21] for 32 μTasks and 4 processor configurations. We analyze the speedup from As-Is system by comparing the runtime-accuracy profiles for three different configurations: (1) As-Is hardware and software (2) As-Is software with baseline speculative hardware and (3) baseline application with baseline speculative hardware.

The As-Is software component breaks down the application in μTasks that generate increasingly accurate outputs whereas the baseline software application would generate only one final output. In As-Is, the potential conflicts among tasks are statically predetermined due to the nature of our synchronous and asynchronous pipelines. As-Is leverages the knowledge of potential conflicts to utilize the system better by letting tasks abort without re-executing them while trying to produce best-effort approximate outputs. However, a baseline speculative system would instead treat all the tasks as mandatory and force them to finish by re-executing them on every conflict. Moreover, in As-Is, producer-consumer relationships between stages are known *a priori*. This allows As-Is to deal with the false dependencies introduced because of the shared buffer between the stages by opting for lazy versioning and conflict detection, in contrast to the eager versioning and eager conflict detection of the baseline speculative system.

For all the benchmarks except centrality, As-Is achieves highly accurate outputs earlier than the other configurations. Centrality has lots of hazards and faster conflict resolution from eager conflict detection and versioning speeds it up on the baseline architecture.

5.5 Benefits from Approximation vs. Speculation

Figure 13 presents the distribution of benefits from approximation alone and approximation along with speculation. For most benchmarks, unlocking the available parallelism with speculation gives significant benefits than just employing approximation techniques. However, there are two exceptions to these trends: centrality and histeq. Because of the partitioning of the histogram buckets there are lots of read and write conflicts in building the histogram and the earlier tasks get aborted very frequently upon parallelization. Similarly, centrality is also hurt by speculation because of lots of collisions between tasks.

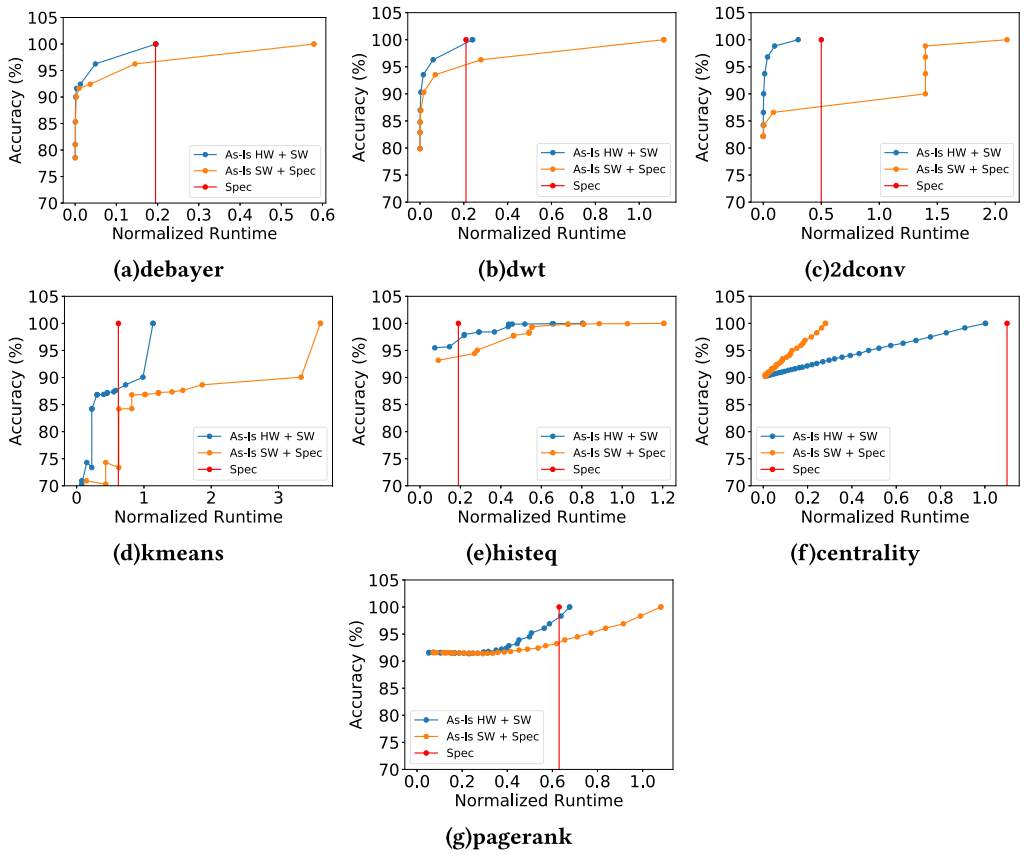

Fig. 12. Comparison between software and hardware components of As-Is and conventional speculative architecture.

5.6 Sensitivity to Cache Block Size

Figure 14 presents the impact of cache block size on As-Is system performance. We study the performance of debayer application with cache block sizes of 4 B, 16 B, and 64 B. Large cache blocks reduce the number of cold misses. However, they also introduce false sharing of data. Our results indicate that the locality benefits of a larger cache block outweigh the resulting false sharing and increased conflicts among tasks.

5.7 Mean and Variance Across Different Inputs

The results reported so far correspond to a single input image or graph. However, to account for variability across different inputs, we run all the benchmarks across four different inputs and mean and standard deviation between runtimes to reach 90% accuracy scaled to the sequential baseline on a 32 μTasks and 4 processor system is reported in Figure 15.

5.8 Area Overheads

As-Is adds three additional hardware overheads: (1) 32 entry task queue (0.34 mm^2) (2) 16 kB dual port SRAM bloom filters (0.05 mm^2) and (3) 7 registers for sampling approximation (0.0007 mm^2). We use CACTI [33] to compute these areas at 32 nm ITRS-HP technology node. Overall these structures consume 4.4% additional area over the baseline speculative system [21].

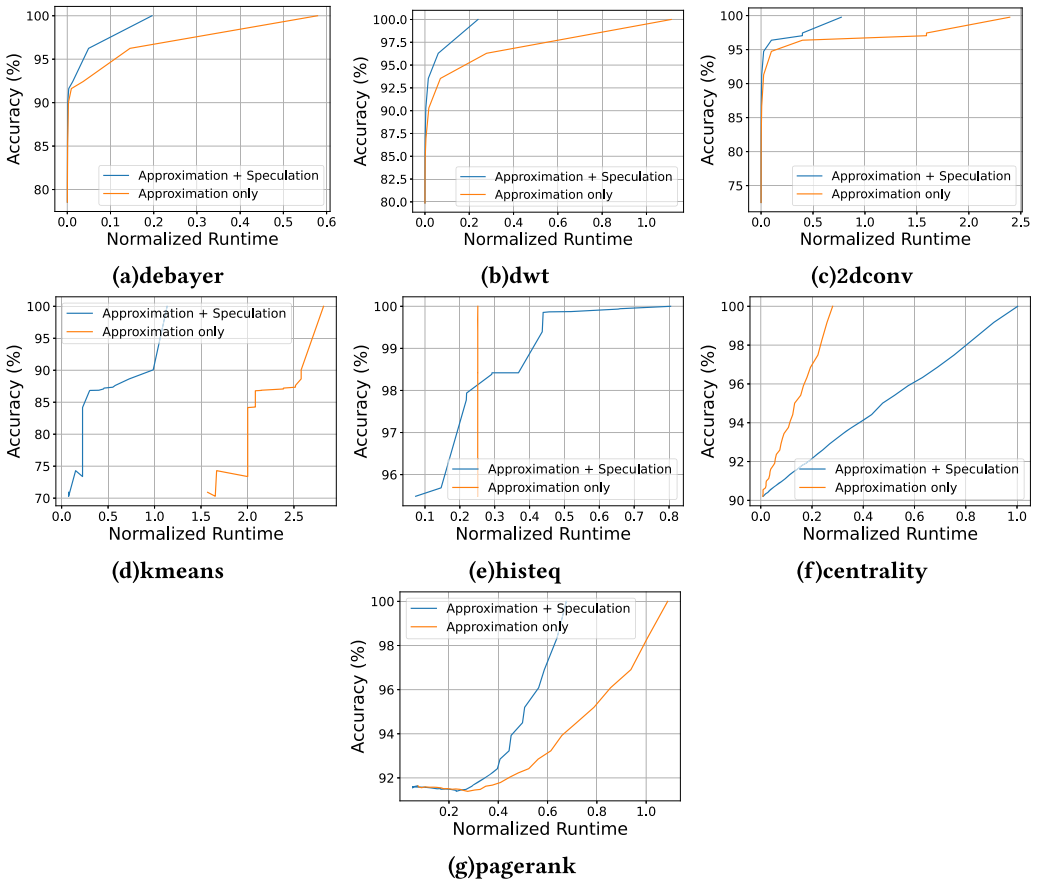

Fig. 13. Comparison between benefits from approximation only and speculation along with approximation for μTasks = 32 on 4 processors.

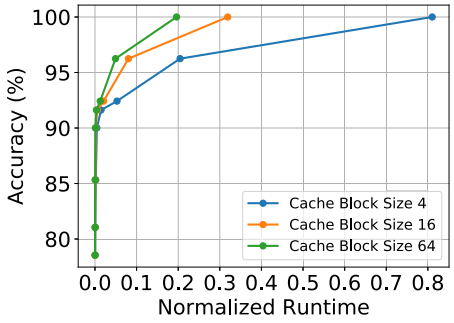

Fig. 14. Sensitivity to Cache Block sizes for debayer.

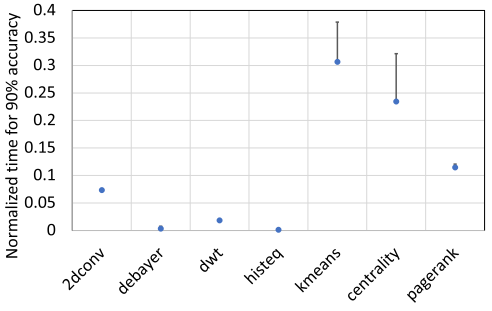

Fig. 15. Mean and standard deviation of runtime to reach 90% accuracy for different inputs across all the benchmarks.

Table 3. Average Power Consumption for 32 μtasks on 4 Processors with Breakdown for Additional Overheads

Benchmark	Runtime Power	Bloom Filter	Task Queue	Approximation	Logic Overhead	Total Power
2dconv	1.641534808	0.018709115	4.24045E-08	0	0.0069083	1.667152266
debayer	1.887845233	0.001436962	4.45023E-08	0	0	1.88928224
dwt	1.235862257	0.00118626	4.97517E-08	0	0	1.237048567
histeq	1.048868066	0.001406349	2.6365E-08	3.33101E-08	0	1.050274474
kmeans	2.284551799	0.008963389	3.67173E-08	4.63893E-08	0	2.293515271
centrality	1.050264271	0.062085359	5.3236E-08	6.72593E-08	0	1.112349751
pagerank	1.119802284	4.89692E-05	4.77938E-08	6.03836E-08	0	1.119851361

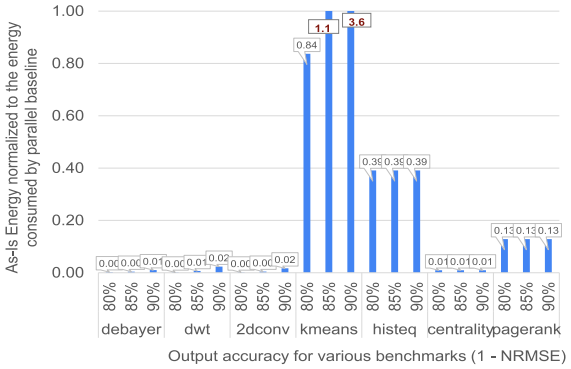

Fig. 16. Energy Consumption on As-Is to achieve 90%, 85%, and 80% accuracy normalized to parallel baseline.

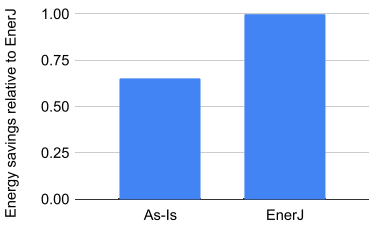

Fig. 17. Energy consumption on As-Is and EnerJ for 2dconv.

5.9 Power Overheads

The power overheads of the As-Is system are presented in Table 3. The y-axis is the average power to reach 100% accuracy on As-Is normalized to the average power of the pthreads implementation. In As-Is, the benchmarks have 32 μTasks in total and the pthreads implementation employs 32 threads. Both configurations are run on a 4 processor system.

Kmeans, debayer and 2dconv incur high power overheads. In fact, kmeans takes 2.2× power to reach 100% accuracy due to increased runtime and conflicts with a larger number of μtasks as observed in Figures 9 and 10. However, power consumption of histeq, dwt, centrality, and pagerank is at par with the baseline.

5.10 Energy Savings

As-Is does enable significant energy savings for earlier approximate outputs. The fraction of energy spent on As-Is to achieve 80%, 85%, and 90% accuracy normalized to the energy consumed by parallel baseline is reported in Figure 16, given 32 μTasks and a 4-processor configuration. Even though the kmeans power is 2.3× that of parallel baseline as shown in Table 3 and the runtime to reach 100% accuracy is also 2× that of parallel baseline as shown in Figure 9, As-Is is able to produce 80% accurate output with only 0.84× energy of parallel baseline.

5.11 Comparison with State-of-the-art

In Figure 17 we compare As-Is with other state-of-the-art approximate computing approaches like EnerJ [48]. For 2dconv benchmark with 4 bit reduced precision, EnerJ is able to achieve 97.04% accuracy and to reach the same level of accuracy, As-Is consumes 0.65× energy when compared to Enerj.

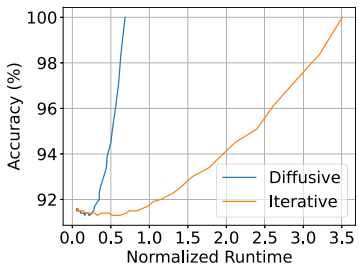

Fig. 18. Diffusive vs. iterative computation stage for pagerank.

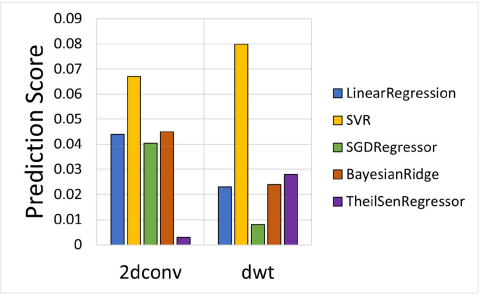

Fig. 19. Dynamic Accuracy using regression models.

5.12 Iterative vs. Diffusive Stages

Since diffusive tasks build on top of the previous output whereas iterative tasks overwrite the output buffer, all the diffusive computation stages can be expressed as iterative stages. Thus, from the efficiency perspective, we opted to implement stages as diffusive instead of iterative whenever possible to minimize the redundant computations. However, we perform a case study to compare diffusive and iterative implementations of the single-stage pagerank automaton in Figure 18 for 32 μTasks and 4-processor configuration. With iterative stages even though it takes much longer for the precise output to be generated, outputs with high accuracy are still generated within 5% of the baseline runtime.

5.13 Dynamic Accuracy Evaluation

As-Is being an interruptible system, it would be hard to provide a dynamic accuracy mechanism that automatically infers when it is appropriate to terminate the program, given a desired user accuracy threshold. We train several regression-based models in order to gauge the runtime for a programmer-specified accuracy.

5.13.1 Implementation. For this case study, we look into **2dconv** and **dwt** operating on 32 μtasks and 4 processors. The training data is gathered by collecting runtime-accuracy numbers across 10 different images and the test dataset comprises of one image. Each image generates 16 data points on the runtime-accuracy curve for 2dconv and 6–8 for dwt. We train the following models in order to gauge the runtime required to achieve a desired user-specified accuracy: (1) Linear Regression, (2) **Support Vector Regression** (**SVR**), (3) SGD Regressor, (4) Bayesian Ridge, and (5) TheilSen Regressor.

5.13.2 Evaluation. To evaluate our models, we use a custom score metric for interruptible execution. The score reflects a difference in the expected value over the predictor value by a certain model.

When the predicted runtime is greater than the expected runtime (as defined by the user-specified accuracy), the difference between the two is the model score. In this case, there is no accuracy deficit, so the score reflects how close the execution is to the ideal latency. On the other hand, if the predicted runtime is shorter, there is a chance of accuracy loss. The score reflects this loss since the model terminates execution earlier than ideal. The scoring is summarized by Equation (1). The score is computed by accumulating all root mean squared values of individual

runtimes. A lower score indicates better prediction by the model.

$$Expected_{Runtime} = \text{Actual Runtime(User-Specified Accuracy)}$$
$$Score = \begin{cases} \Delta Runtime, & \text{if } Predicted_{Runtime} > Expected_{Runtime} \\ \Delta Accuracy, & \text{else} \end{cases} \quad (1)$$

The scores of all test points are accumulated (via root mean square) and shown in Figure 19 Out of all the regression models, TheilSen Regressor seems to have the best fit for $2dconv$; however, SGDRegressor performs best for dwt. Across these two applications, SGD regressor has the best fit, given the scores. Though given the low discrepancy in scores, any of our evaluated predictor models would be effective.

5.14 Limitations Discussion

Currently, As-Is imposes programmer effort in porting over any application, breaking it into anytime computing stages and annotating it with the relevant information. With follow-on work on compilers, much of this effort can be avoided or reduced. However, this effort is currently outside the scope of this article. Our evaluation is based on simulations and even though this is common for research works in the architecture community, we understand that the benefits of the As-Is platform may look different on a real system. We believe that the current evaluation is highly reflective of the performance and energy savings on a realized As-Is system. We have not done a comprehensive search of the application space, particularly bigger and end-to-end applications, but we expect them to only strengthen our evaluation.

6 RELATED WORK

Approximate Computing. Approximate computing allows trading off accuracy and reliability for better performance and energy efficiency and has been widely explored in the prior work [3, 4, 13, 14, 16, 25, 38, 40, 46–48]. Reduced precision [23], loop perforation [49], load value approximation [38], memoization [1, 44, 45] are some hardware-based approximation techniques explored. These approaches provide several different mechanisms for programmers to specify approximation opportunities at a fine granularity like instructions, variables, functions and patterns. Several techniques also employ neural networks to estimate approximable code regions [2, 12, 14, 16, 17, 25, 31, 42, 51, 54]. In contrast, As-Is attempts to uncover benefits of approximation at a coarse task level granularity and exploits accuracy guarantees and interruptibility of anytime execution.

Best Effort Computing. Best-effort computing has been studied in the context of parallel computing [6], data mining [8, 36], and hardware transactional memory [7, 9, 30]. However, As-Is focuses on marrying best-effort computing with approximation. Moreover, in approaches like [6], best-effort and mandatory task execution is required to be isolated in both hardware and software, whereas the As-Is system can support both of these computations on the same substrate and relies on the underlying architecture to resolve data dependencies. Earlier techniques also rely on programmers to annotate best-effort computations; however, the As-Is runtime system can automatically classify μTasks based on the computation stage-pipeline configuration.

Intermittent Computing. As-Is like task-level programming model is also leveraged by research in intermittent computing for recoverability [15, 35]. However, their focus has been on developing software-based techniques to ensure forward progress within tight energy constraints without much emphasis on the performance of the underlying system. However, As-Is is built with the goal of extracting performance at an acceptable accuracy loss aided by speculation to uncover the

parallelism present in anytime computing. Even though software approaches may enable anytime-like computing models, optimizations at the hardware level are integral for system performance.

7 CONCLUSION

We propose As-Is, an *Anytime Speculative Interruptible System* that supports architectural realization of anytime approximate computing with speculative computing. The As-Is task-based programming model expresses an application as an anytime automaton. The theoretical anytime stages are formalized into physical mandatory and best-effort μTasks which are created, scheduled, executed, resolved and retired by the underlying architecture and runtime system. Our results indicate favourable runtime-vs.-accuracy tradeoffs over a wide range of benchmarks.

REFERENCES

[1] C. Alvarez, J. Corbal, and M. Valero. 2005. Fuzzy memoization for floating-point multimedia applications. *IIEEE Transactions on Computers* 54, 7 (2005), 922–927. DOI: https://doi.org/10.1109/TC.2005.119

[2] Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. 2014. General-purpose code acceleration with limited-precision analog computation. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture*. 505–516. DOI: https://doi.org/10.1109/ISCA.2014.6853213

[3] Mohammad Ashraful Anam, Paul N. Whatmough, and Yiannis Andreopoulos. 2013. Precision-energy-throughput scaling of generic matrix multiplication and discrete convolution kernels via linear projections. In *Proceedings of the 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*. 21–30. DOI: https://doi.org/10.1109/ESTIMedia.2013.6704499

[4] Woongki Baek and Trishul M. Chilimbi. 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[5] Kevin Barker, Thomas Benson, Dan Campbell, David Ediger, Roberto Gioiosa, Adolfy Hoisie, Darren Kerbyson, Joseph Manzano, Andres Marquez, Leon Song, Nathan Tallent, and Antonino Tumeo. 2013. *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. Pacific Northwest National Laboratory and Georgia Tech Research Institute. Retrieved from http://hpc.pnnl.gov/projects/PERFECT/.

[6] Srimat T. Chakradhar and Anand Raghunathan. 2010. Best-effort computing: Re-thinking parallel software and hardware. In *Proceedings of the 47th Design Automation Conference*. 865–870.

[7] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. 2009. Rock: A high-performance sparc CMT processor. *IEEE Micro* 29, 2 (2009), 6–16.

[8] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference 2013*. 113:1–113:9.

[9] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. 2011. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 39–52.

[10] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. 2017. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 561–574.

[11] Thomas Dean and Mark Boddy. 1988. An analysis of time-dependent planning. In *Proceedings of the 7th AAAI National Conference on Artificial Intelligence*. AAAI, 49–54. Retrieved from http://dl.acm.org/citation.cfm?id=2887965.2887974.

[12] Schuyler Eldridge, Florian Raudies, David Zou, and Ajay Joshi. 2014. Neural network-based accelerators for transcendental function approximation. In *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*. Association for Computing Machinery, 169–174. DOI: https://doi.org/10.1145/2591513.2591534

[13] 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*. 301–312.

[14] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2015. Neural acceleration for general-purpose approximate programs. *Communications of the ACM* 58, 1 (2015), 105–115.

[15] Karthik Ganesan, Joshua San Miguel, and Natalie Enright Jerger. 2019. The what's next intermittent computing architecture. In *Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture*.

[16] Beayna Grigorian, Nazanin Farahpour, and Glenn Reinman. 2015. BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture*. 615–626.

[17] Beayna Grigorian and Glenn Reinman. 2014. Dynamically adaptive and reliable approximate computing using light-weight error analysis. In *Proceedings of the 2014 NASA/ESA Conference on Adaptive Hardware and Systems*. 248–255. DOI: https://doi.org/10.1109/AHS.2014.6880184

[18] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*. ACM, 3–12. DOI: https://doi.org/10.1145/1941553.1941557

[19] Eric J. Horvitz. 1987. Reasoning about beliefs and actions under computational resource constraints. In *Proceedings of the 3rd Conference on Uncertainty in Artificial Intelligence*. AUAI, 429–447.

[20] A. Jain, P. Hill, S. 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 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*. 1–13.

[21] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. 2016. Unlocking ordered parallelism with the swarm architecture. *IEEE Micro* 36, 3 (2016), 105–117. DOI: https://doi.org/10.1109/MM.2016.12

[22] Keyur Joshi, Vimuth Fernando, and Sasa Misailovic. 2020. Aloe: Verifying reliability of approximate programs in the presence of recovery mechanisms. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 56–67.

[23] Patrick Judd, Jorge Albericio, Tayler H. Hetherington, Tor M. Aamodt, Natalie D. Enright Jerger, Raquel Urtasun, and Andreas Moshovos. 2015. Reduced-precision strategies for bounded memory in deep neural nets. arXiv:1511.05236. Retrieved from https://arxiv.org/abs/1511.05236.

[24] Patrick Judd, Jorge Albericio, and Andreas Moshovos. 2017. Stripes: Bit-serial deep neural network computing. *Computer Architecture Letters* 16, 1 (2017), 80–83. DOI: https://doi.org/10.1109/LCA.2016.2597140

[25] Daya Shanker Khudia, Babak Zamirai, Mehrzad Samadi, and Scott A. Mahlke. 2015. Rumba: An online quality management system for approximate computing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 554–566.

[26] N. Kulkarni, F. Qi, and C. Delimitrou. 2018. Leveraging approximation to improve datacenter resource efficiency. *IEEE Computer Architecture Letters* 17, 2 (2018), 171–174.

[27] Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian Neamtiu. 2016. Efficient processing of large graphs via input reduction. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. 245–257.

[28] Jure Leskovec and Rok Sosič. 2016. SNAP: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology* 8, 1 (2016), 1.

[29] Victor R. Lesser, Jasmina Pavlin, and Edmund Durfee. 1988. Approximate processing in real-time problem solving. *AI Magazine* 9, 1 (1988), 49.

[30] Yossi Lev and Jan-Willem Maessen. 2008. Split hardware transactions: True nesting of transactions using best-effort hardware transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 197–206.

[31] Boxun Li, Peng Gu, Yi Shan, Yu Wang, Yiran Chen, and Huazhong Yang. 2015. RRAM-based analog approximate computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 12 (2015), 1905–1917. DOI: https://doi.org/10.1109/TCAD.2015.2445741

[32] 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*.

[33] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *Proceedings of the International Conference on Computer-Aided Design*. IEEE.

[34] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 190–200. DOI: https://doi.org/10.1145/1065010.1065034

[35] Kiwan Maeng and Brandon Lucia. 2019. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[36] Jiayuan Meng, Srimat T. Chakradhar, and Anand Raghunathan. 2009. Best-effort parallel execution framework for Recognition and mining applications. In *Proceedings of the 23rd IEEE International Symposium on Parallel and Distributed Processing*. 1–12.

[37] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger. 2015. Doppelgänger: A cache for approximate computing. In *Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture*. 50–61.

[38] J. S. Miguel, M. Badr, and N. E. Jerger. 2014. Load value approximation. In *Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*.

[39] Joshua San Miguel and Natalie Enright Jerger. 2016. The anytime automaton. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE, 545–557. DOI: https://doi.org/10.1109/ISCA.2016.54

[40] 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 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2014*. 309–328.

[41] Jayadev Misra. 1986. Distributed discrete-event simulation. *ACM Computing Surveys* 18, 1 (1986), 39–65.

[42] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. 2015. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015*. 603–614.

[43] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 12–25. DOI: https://doi.org/10.1145/1993498.1993501

[44] Abbas Rahimi, Luca Benini, and Rajesh K. Gupta. 2013. Spatial memoization: Concurrent instruction reuse to correct timing errors in SIMD architectures. *IEEE Transactions on Circuits and Systems II: Express Briefs* 60, 12 (2013), 847–851. DOI: https://doi.org/10.1109/TCSII.2013.2281934

[45] Michael Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. 2015. Monitoring and debugging the quality of results in approximate programs. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 399–411. DOI: https://doi.org/10.1145/2694344.2694365

[46] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based approximation for data parallel applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[47] 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*.

[48] 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*. 164–174.

[49] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin C. Rinard. 2011. Managing performance vs. accuracy tradeoffs with loop perforation. In *Proceedings of the SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and ESEC'11: 13th European Software Engineering Conference*. 124–134.

[50] Pang-Ning Tan. 2018. *Introduction to Data Mining*. Pearson Education India.

[51] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. 2014. AxNN: Energy-efficient neuromorphic systems using approximate computing. In *Proceedings of the 2014 IEEE/ACM International Symposium on Low Power Electronics and Design*. 27–32. DOI: https://doi.org/10.1145/2627369.2627613

[52] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 237–251. DOI: https://doi.org/10.1145/3037697.3037748

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

[54] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. 2015. ApproxANN: An approximate computing framework for artificial neural network. In *Proceedings of the 2015 Design, Automation Test in Europe Conference Exhibition*. 701–706.

Received 4 March 2021; revised 4 August 2022; accepted 15 August 2022