The What's Next Intermittent Computing Architecture

Karthik Ganesan*, Joshua San Miguel† and Natalie Enright Jerger*
*University of Toronto, †University of Wisconsin-Madison
karthik.ganesan@mail.utoronto.ca, jsanmiguel@wisc.edu, enright@ece.utoronto.ca

***Abstract*—Energy-harvesting devices operate under extremely tight energy constraints. Ensuring forward progress under frequent power outages is paramount. Applications running on these devices are typically amenable to approximation, offering new opportunities to provide better forward progress between power outages. We propose What's Next (*WN*), a set of anytime approximation techniques for energy harvesting: subword pipelining, subword vectorization and skim points. Skim points fundamentally decouple the checkpoint location from the recovery location upon a power outage. Ultimately, WN transforms processing on energy-harvesting devices from all-or-nothing to as-is computing. We enable an approximate (yet acceptable) result sooner and proceed to the next task when power is restored rather than resume processing from a checkpoint to yield the perfect output. WN yields speedups of 2.26x and 3.02x on non-volatile and checkpoint-based volatile processors, while still producing high-quality outputs.**

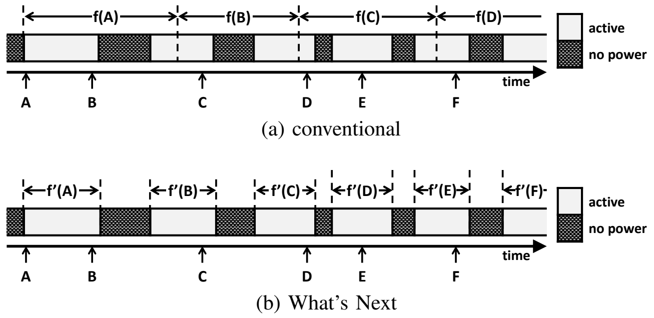
***Keywords*-energy harvesting; intermittent computing; approximate computing;**

I. Introduction

Energy-harvesting devices are an emerging class of embedded systems that eschew batteries by running directly off of energy gathered from the environment. These devices are powered from sources such as solar, WiFi, RF and motion-based energy [35]. Ultra-low-power processors running on harvested energy open up new and exciting applications in fields such as medical devices [15], computer vision [39], environmental sensing [52], remote sensing [39], and wildlife tracking and monitoring [47]. Typical energy-harvesting applications exhibit several common characteristics that the What's Next (*WN*) architecture targets:

- Subject to non-trivial checkpointing, recovery and re-execution overheads due to frequent power outages [18].
- Amenable to approximation.
- Contain many word-granularity integer and fixed-point operations that are costly in ultra-low-power processors.

Harvested energy sources only produce sufficient energy to power these devices for up to a few milliseconds at a time [42]. As a result, energy-harvesting devices operate under the *intermittent computing* paradigm; they incur frequent power outages. To enable forward progress despite frequent power outages, these systems use non-volatile processors or periodically checkpoint to non-volatile memory [42]. This allows the processing of a single input to span multiple power cycles. However, when new input data arrives, the system must choose to either continue processing old data or discard it and move on to processing new data.



(a) conventional

(b) What's Next

Figure 1: Overview of application (`f`) executing on an energy-harvesting system. With conventional execution, each input (`A` to `F`) is processed to completion. With WN, the application (`f′`) returns an acceptable approximate result per input without running to completion, accelerating forward progress.

The goal of WN is to provide a partial answer when energy is scarce but offer the flexibility to refine that answer if more energy is available.[1] To this end, we extend the anytime automaton model [45] which provides early approximate versions of an application's output; as the application continues to run, it refines the output towards the precise result. In particular, the anytime automaton provides *interruptibility* and *flexibility*. With *interruptibility*, processing can be halted (e.g., by a power outage) while still providing a valid, approximate output. This is in contrast to a more traditional, *all-or-nothing* computing approach. As intermittently-powered devices are frequently interrupted due to power outages, the interruptibility of the anytime model marries well with this domain. *Flexibility* is also key; if greater accuracy is required, an application can run longer (expending greater energy) to produce a more accurate result.

Consider the example application f in Figure 1. In a conventional energy-harvesting system (Figure 1a), the device resumes processing each input after a power outage until the final precise result is achieved. As a result, input F arrives while the device is still processing input D. Using the WN technique (Figure 1b), we work on input A while power remains. When a power outage occurs, an acceptable result is already available for A and we can begin processing the next available input when power returns. As a result, we

[1]"I understood the point... when I ask what's next, it means I'm ready to move on to other things, so what's next?" – Jed Bartlet, The West Wing

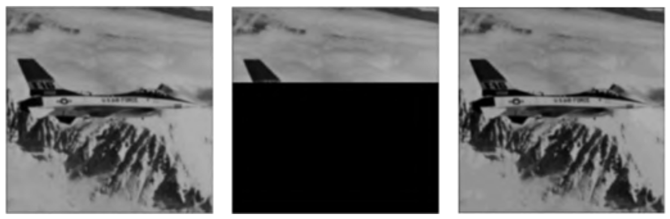

(a) baseline (100% runtime) (b) baseline (50% runtime) (c) WN (50% runtime)

Figure 2: Conv2d output: baseline and subword pipelining

start processing input F soon after it arrives and achieve greater forward progress across all input samples. The output quality is best-effort (i.e., we take the approximate result *as-is* when forced to power down) but forward progress improves substantially. As many energy-harvesting applications are naturally amenable to approximation, this trade-off is an effective and appropriate one.

To adapt the anytime paradigm to intermittent computing systems, we propose hardware and software changes to process data at *subword* rather than *word* granularity. Instead of processing an entire data word at a time, we split this word into subwords and process subwords from most to least significant.[2] Processing each subword generates an approximate result. We implement two subword-based modifications: *subword pipelining*, which decomposes high-latency instructions (e.g., multiplication) into smaller subword operations and *subword vectorization*, which fuses low-latency instructions (e.g., add, load, store) such that the most significant subwords of different data elements are processed in parallel. These techniques transform word-based (i.e., all-or-nothing) computing where the entire word must be processed to generate a result to an iterative model (i.e., as-is) where an approximate result is available after the first subword and the results improve with each subsequent subword we process.

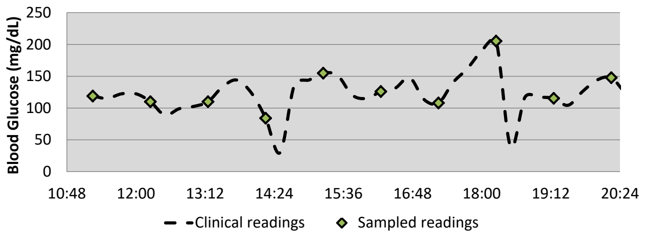
Finally, we propose *skim points*. In existing checkpointing schemes, a backup operation saves the current program counter to non-volatile memory. Then during a restore, the system resumes at the saved PC and continues executing. Skim points decouple the backup PC from the restore PC to allow processing to skip the remaining subwords if an acceptable result is available and start processing new data.

II. Motivation

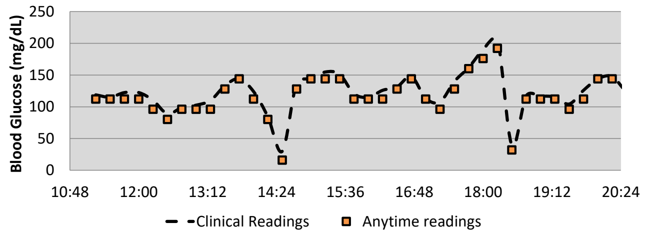
This section presents two examples to demonstrate the drawbacks of current intermittent computing systems that WN anytime processing overcomes.

How can anytime processing generate acceptable results earlier? Intermittent computing systems frequently do not have sufficient energy to fully process input data in a single power on cycle. When processing is interrupted, the current result is likely to be incomplete. In such situations, an approximate result can serve as a facsimile for the complete precise output. To demonstrate this, we compare the output of an image processing kernel from an anytime vs. a precise implementation (Figure 2). Figure 2b shows the output when the precise implementation losses power halfway through processing the image. Clearly the result is incomplete and processing must continue during the next active period to produce an acceptable output. Yet with anytime processing and the same total power-on time, we can compute a result for the entire image (Figure 2c). This result is both complete and of acceptable quality. If a higher quality output is needed, the system can run for longer, improving quality over time.

[2]We use subword sizes of 4 and 8 in our study.



(a) Readings produced via input sampling



(b) Readings produced via anytime processing

Figure 3: Comparison of blood glucose readings for input sampling and anytime processing compared to clinical data

How does anytime processing compare to input sampling? Energy-harvesting devices that operate on continuous input signals can sample those inputs if the processor is not able to keep up. Sampling with precise computation is less desirable than generating approximate outputs for a larger number of inputs. A precise implementation that drops samples to keep up with the rate of incoming data risks losing important information contained in the dropped samples. WN avoids this by generating an approximate result for all samples so that critical information is not missed.

Consider blood glucose monitoring for patients suffering from diabetes. Regular monitoring is required to detect dangerously low levels promptly so corrective action can be taken. Energy-harvesting, wearable monitoring devices are being developed to meet this need [15]. However, these devices must carefully balance their stringent energy supply with collecting readings as often as possible. A sampling-based approach collects readings less frequently due to frequent power outages, increasing the chances of missing critical events. Anytime processing reduces the energy spent processing each reading, resulting in more readings being processed overall (with a small loss in accuracy).

In Figure 3, the dotted line shows the data collected in a clinical setting at 15 minute intervals over a 10-hour period [10] with two dips in blood glucose at 14:30 and 18:30. These dips indicate periods of critically low blood glucose (values below 50 mg/dL). Quickly and correctly identifying these dips is critical for the successful management of diabetes. The input sampling technique in Figure 3a produces precise results but misses both critical events due to infrequent sampling. In contrast, if we process only the 4 most significant bits of each sample (Figure 3b), we catch both critical events with an average error of only 7.5%. This error is within $\pm 20\%$ error range required by international standards for glucose monitoring [21], making these results valid even after processing just the first subword. The resulting energy savings allow us to process more readings and as a result, catch both critical events. However, in cases where there is sufficient energy to continue processing the same data, our technique offers the advantage of improving accuracy over time. As detailed in the next section, WN easily trades off accuracy for frequency of samples based on the needs and constraints of each application and system.

III. What's Next Intermittent Computing Architecture

Building upon the observations in Section II, we propose the What's Next Intermittent Computing Architecture, which consists of three new mechanisms:

1) *Anytime subword pipelining* for long-latency integer and fixed-point operations (Section III-A);
2) *Anytime subword vectorization* for short-latency integer and fixed-point operations (Section III-B);
3) *Skim points* for committing approximate results upon a power outage (Section III-C).

Specifically, we introduce microarchitectural techniques (and corresponding language and compiler support) for prioritizing the computation of the most significant subwords. Less significant subwords are processed incrementally, improving quality over time. If power is lost before processing all subwords, the intermediate result serves as an approximation of the precise result; the application can move on using skim points. WN brings forth an anytime intermittent computing paradigm: *if sufficient energy is available, work towards the precise answer; if not, accept the current approximation and move on to processing the next sample.*

A. Anytime Subword Pipelining

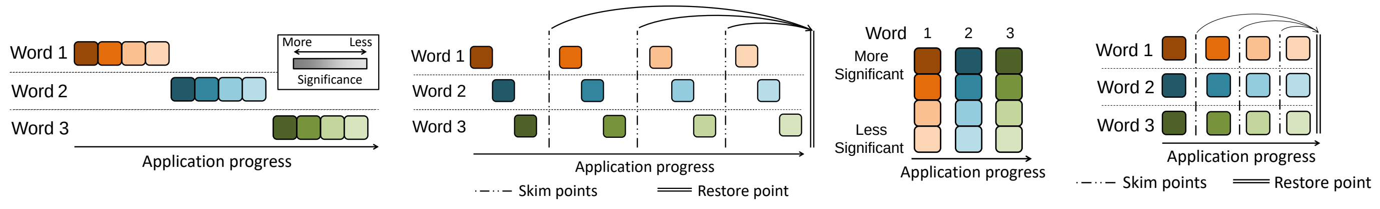
Anytime subword pipelining (SWP) breaks long-latency operations (e.g., fixed-point multiplication) into subword stages, starting with the most significant subword. The ultra-low power processors used in energy harvesting [18] such as the ARM M0+, do not have a hardware multiplier [3]. A 16×16 multiply is carried out iteratively, taking 16 cycles to complete. Our approach splits each word into smaller subwords to be processed from most to least significant. Instead of processing each word to completion as shown in Figure 4a (words 1 to 3), the most significant subwords from multiple data elements can be processed in a pipelined fashion (Figure 4b). Coupled with skim points (Section III-C), the application can now skip to the end of processing the current input data if a power outage is encountered. Otherwise, the entire pipeline runs to completion and is guaranteed to produce the precise result. To support anytime subword pipelining in WN, we take a hardware-software co-design approach and address three challenges.

Which computations are candidates for SWP? We generalize candidates as any operation or function `f` that satisfies the following properties: 1) incurs long latency, and 2) is distributive over addition. The first property is not necessary for correctness but is desirable to reap performance gains. There is no benefit to breaking a single-cycle operation (e.g., addition) into multiple stages; this would merely increase its latency and yield slowdown. A candidate operation must have a latency greater than a single cycle to see a speedup. The second property is a requirement for guaranteeing that the precise result is reached once all subwords have been processed. Figure 5 shows the transformation of a long-latency function `f` by the compiler into smaller subword computations. Assuming that the two properties listed above are satisfied, the input to `f` (e.g., `a`) can always be broken into subwords and processed in stages as long as they are accumulated into the same location (e.g., `x`). Splitting `f` into shorter stages allows the individual subwords to be processed in less time.

What microarchitectural support is needed? To support subword pipelining, we implement subword variants of long-latency operations. For some long-latency instruction `OP`, we introduce new instructions: `<OP>_ASP<BITS>`, where `BITS` specifies the subword size. This is shown in Listing 2, which is the assembly code for the example program in Listing 1 after enabling anytime SWP. `MUL_ASP8` (Line 5) is a variant of the `MUL` instruction that instead performs a 16×8-bit multiplication. The third parameter specifies the location of the 8-bit subword of the second operand; for example, the most significant 8-bit subword of a 16-bit operand is at position 1, as shown in Line 5. As a result, multiplication is performed in shorter stages (e.g., 8 cycles per `MUL_ASP`). Note that we use the `LDRB` instruction (Line 4), which loads just a single byte from memory instead of the `LDR` operation, which loads an entire word.

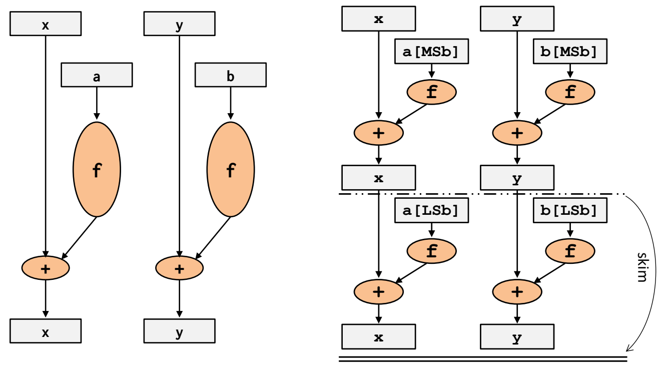
We find that the multiply operation is the predominant candidate in our applications.[3] As the M0+ processor we target supports a 32-bit datapath, we use a 16x16 multiplier as our full precision case. Since the ARM M0+ core uses an iterative multiplier [3], one bit of the operand is multiplied each cycle. N cycles are required to obtain the result of

[3]Although not found in our benchmarks, more complex operations such as floating point, square root and trigonometric functions are also candidates for SWP. For example, trigonometric operations are commonly used in wirelessly powered biomedical applications [40].



(a) conventional (b) subword pipelining (c) conventional (d) subword vectorization

Figure 4: Anytime subword pipelining and vectorization for long-latency and short-latency operations.



(a) before (b) after

Figure 5: IR transformation for anytime SWP.

Listing 1: Example source code for 8-bit anytime SWP.

```
#pragma asp input(A, 8);   // (array, bits)
#pragma asp output(X);     // (array)
for (i = 0; i < N; i++)
  X[i] += A[i] * F[i];
```

multiplying by an N-bit subword. We modify the 16×16-bit multiply operation to support multiplication with a single subword operand at a time. This straightforward change allows us to support multiplication by subword sizes of less than 16 bits and reap performance gains, since multiplying by a subword is faster than multiplying by the full word. We support two subword granularities—4 and 8—and implement two new instructions: `MUL_ASP4` and `MUL_ASP8`, which incur 4 and 8 cycles, respectively. We explore smaller subwords in Section V-E.

How much programmer intervention is required? Since software support for anytime SWP is implemented entirely in the compiler's intermediate representation (IR), minimal changes are required in the source code. As shown in Listing 1, we only need programmer declarations (i.e., `asp` pragmas) for input and output memory locations that are amenable to approximation. For each input, the programmer specifies the subword size (8 bits in this example). Computations in the IR that satisfy these two properties are transformed to enable subword pipelining automatically (Figure 5). In this example, the candidate computation `f` is a multiplication with some array element `F[i]`.

Algorithm 1 provides a high-level description of the compiler pass needed for SWP. If the operation's input and

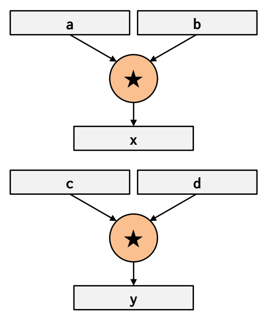
Listing 2: Example assembly code for 8-bit anytime SWP.

```
LOOP_MSb:
  LDR R3, [R0, #0]     @ X[i]
  LDR R4, [R1, #0]     @ F[i]
  LDRB R5, [R2, #1]    @ A[i][MSb]
  MUL_ASP8 R4, R5, #1 @ X += F * A
  ADD R3, R4
  STR R3, [R0, #0]
  ...                  @ ++i < N
  B LOOP_MSb
SKM END
...
LOOP_LSb:
  LDR R3, [R0, #0]     @ X[i]
  LDR R4, [R1, #0]     @ F[i]
  LDRB R5, [R2, #0]    @ A[i][LSb]
  MUL_ASP8 R4, R5, #0 @ X += F * A
  ADD R3, R4
  STR R3, [R0, #0]
  ...                  @ ++i < N
  B LOOP_LSb
END:
```

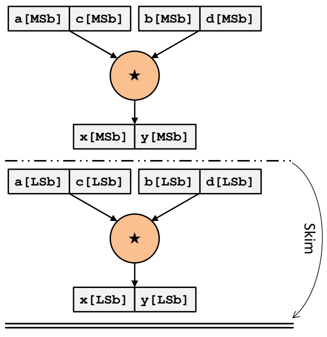
```
foreach long-latency operation do
    if operands annotated by pragma asp then
        Obtain subword size from pragma directive;
        Call loop fission pass on loop containing operation;
        Modify long-latency operation with anytime
          equivalent in each loop instance;
end
```

Algorithm 1: Compiler pass for SWP

output operands are annotated with a pragma `asp` directive (i.e., `A` & `X` in Listing 1), the operation is marked as a target for SWP. Once a target operation is determined, the loop encompassing this operation is split into multiple loops (using a simple loop fission compiler pass [2]). The loop is split twice for the 8-bit case and 4 times for the 4-bit case. The long-latency operation in each loop instance is modified with its anytime equivalent. For example, full-precision `MUL` operations are changed to anytime `MUL_ASP` operations, along with the third parameter to indicate which subword should be processed in that operation. Minimal code size increases occur due to multiple loops with SWP operations. For the largest benchmark in our suite, the code size only increases by 1KB from the precise 16-bit case to the anytime 4-bit case, allowing SWP to be used even on extremely small



(a) before



(b) after

Figure 6: IR transformation for anytime SWV.

energy-harvesting devices with limited on-board memory.

Memoization. SWP allows additional optimizations that can further boost its gains; we now discuss how to combine SWP with memoization. Memoization [27] reduces the overhead of repeated executions of long-latency instructions [49], [8]. Results from prior executions are stored in a direct-mapped table. When an anytime multiplication instruction is encountered, we first check the table. If a matching entry is found, the result is returned in a single cycle (as opposed to the 4, 8 or 16 cycles taken for 4-bit, 8-bit or precise multiply operations, respectively).

Zero Skipping. In addition to memoization, we also support zero skipping [1]. As multiplications with zero are seen more often than with any other value, we exclude any multiplications where either input is zero from being memoized. If either multiplier input is zero, we return an output of zero in a single cycle. We investigate the speedup of memoization and zero skipping for multiply instructions in Section V-E.

B. Anytime Subword Vectorization

Anytime subword vectorization (SWV) transposes short-latency operations (e.g., addition) so the most significant subwords are processed first. In contrast to anytime SWP where a single data element is split into subwords, in anytime SWV, subwords from multiple data elements are processed in parallel. Thus, we generate an approximate result for several words in a single cycle, thereby gaining a speedup over the baseline implementation. The target operations are bitwise operations such as AND, OR, NOT, XOR etc. as well as addition and subtraction. In a conventional processor, words are operated on one at a time, as shown in Figure 4c (words 1 to 3). We transpose the subwords (Figure 4d), operating on multiple words in parallel and processing subwords in decreasing order of significance. This enables execution on the most impactful bits first. Skim points (Section III-C) can again be used to jump ahead if there is insufficient energy; otherwise, we continue improving the approximation quality. The precise result is guaranteed when all subwords of all data elements have been processed. To support anytime subword vectorization in WN, we take a similar approach to SWP.

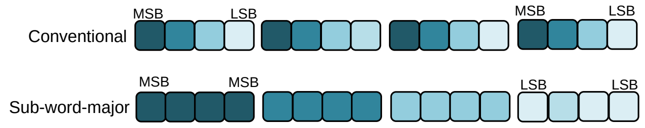


Figure 7: Subword-major compared to conventional row-major layout in memory for both input and output.

Listing 3: Example source code for 8-bit anytime SWV.

```
#pragma asv input(A, 8);  // (array, bits)
#pragma asv input(B, 8);  // (array, bits)
#pragma asv output(X, 8); // (array, bits)
for (i = 0; i < N; i++)
  X[i] = A[i] + B[i];
```

Which computations are candidates for subword vectorization? We generalize candidates as any computation that employs an operator ★ (Figure 6) that is element-wise on the binary expansion of its operands. That is, ★ must satisfy the following:

$$\mathtt{a} \bigstar \mathtt{b} \;=\; \big(\mathtt{a}_0 \bigstar \mathtt{b}_0\big) \times 2^0 + ... + \big(\mathtt{a}_{n-1} \bigstar \mathtt{b}_{n-1}\big) \times 2^{n-1}$$

where $\mathtt{a}_0 ... \mathtt{a}_{n-1}$ and $\mathtt{b}_0 ... \mathtt{b}_{n-1}$ are the bits of operands `a` and `b`. We can only vectorize operations where both operands can be decomposed into subwords, as opposed to just one operand in subword pipelining.

What microarchitectural support is needed? We introduce a set of new instructions: `<OP>_ASV<BITS>`, where `OP` specifies some short-latency operation (★) and `BITS` specifies the subword size. This is shown in Listing 4, which is the assembly code for the example program in Listing 3 after enabling anytime SWV for addition. `ADD_ASV8` (Line 4) is a variant of the 32-bit `ADD` instruction that instead performs four 8-bit additions. Four elements of `X` are computed in a single parallel operation. WN supports anytime SWV for logical (bitwise-or, bitwise-and, exclusive-or), memory (load, store) and arithmetic operations (fixed-point addition).

How much programmer intervention is required? As with pipelining, anytime SWV only requires programmer declarations (i.e., `asv` pragmas) for approximate input and output data, shown in Listing 3. The subword size needs to be specified for both the inputs and outputs (8 bits in the example). In this example, ★ is addition, which is the most common operator for vectorization in the applications we studied. For the rest of this section, we focus on addition but the techniques discussed also apply to other operations.

Figure 6 shows the compiler transformation once a candidate instruction with operator ★ is identified. We employ a similar compiler pass for SWV as the one for SWP shown in Algorithm 1. To support SWV, we transpose data words into subword-major order in memory as shown in Figure 7. As we target energy-harvesting devices that likely obtain inputs from sensors, it is a simple matter of transposing the data being received from the sensor to support subword-major ordering. The inputs and outputs can simply be statically

Listing 4: Example assembly code for 8-bit anytime SWV.

```
LOOP_MSb:
 LDR R3, [R0, #0]     @ A[i:i+3][MSb]
 LDR R4, [R1, #0]     @ B[i:i+3][MSb]
 ADD_ASV8 R3, R4      @ X = A + B
 STR R3, [R2, #0]     @ X[i:i+3][MSb]
 ...                  @ i += 4; i < N
 B LOOP_MSb
SKM END
...
LOOP_LSb:
 LDR R3, [R0, #0]     @ A[i:i+3][LSb]
 LDR R4, [R1, #0]     @ B[i:i+3][LSb]
 ADD_ASV8 R3, R4      @ X = A + B
 STR R3, [R2, #0]     @ X[i:i+3][LSb]
 ...                  @ i += 4; i < N
 B LOOP_LSb
END:
```

encoded in subword-major order and stay that way for the duration of the application. Though it is possible to transpose the data elements back to row-major order afterwards, we find that this is not necessary for most applications, since 1) subword-major ordering is deterministic and can be done statically, and 2) locality of the memory ordering is typically not a concern since energy-harvesting devices tend to have no caches (or only small caches, if any).

Provisioned addition. Subword vectorization of logical and memory operations does not require any new instructions nor changes to hardware; the compiler can simply use their full-precision equivalents (e.g., performing four 8-bit exclusive-or operations can be done via one 32-bit operation). Vectorized addition, however, cannot use 32-bit adders since carry bits must not propagate across different subwords. Compared to traditional vector instructions that expand the datapath to support parallel operations (such as Intel AVX [20]), we repurpose existing hardware to achieve vectorization at a smaller subword granularity.[4] We take an existing 32-bit adder unit and reconfigure it with straightforward modifications to support 4-bit and 8-bit parallel subword operations, while still maintaining support for full 32-bit additions. At intervals of every four (1-bit) full adders, a mux is inserted into the carry chain, as shown in Figure 8. When a subword vectorization instruction is encountered, the muxes pass zeroes into the appropriate carry-in bits. For example, with `ADD_ASV8`, the muxes between bits 7 and 8, bits 15 and 16 and bits 23 and 24 pass in zeroes, thereby allowing the adder to perform four independent additions: adding bits 0-7, bits 8-15, bits 15-23 and bits 23-31, respectively. While inserting these muxes increases the overall latency of the adder, we expect this to have no impact on processor frequency. We evaluate this in Section IV.

Unlike with traditional vector instructions, subword vectorization for addition needs to consider the possibility of

[4]Vector operations for 8-bit data elements exist, such as multimedia extensions [26], [28]; the key difference is that anytime subword vectorization computes on full-precision (32-bit) data elements, one subword at a time.

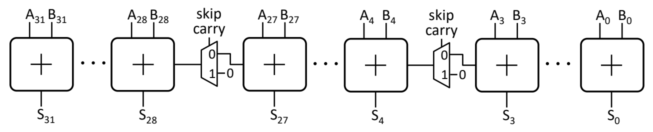


Figure 8: Design of 32-bit adder with SWV support. Muxes are placed after every 4 full adders for a total of 7 Muxes.

losing carry-out bits between subwords. For example, in a conventional 32-bit addition, any carry-out when adding the 8-bit least significant subwords are propagated to the next subword and eventually accumulated into the 32-bit sum. However, with anytime subword vectorization, two 8-bit subword operands (e.g., `A[i][LSb]` and `B[i][LSb]` in Listing 4) are summed into a memory location that is only 8 bits in size (e.g., `X[i][LSb]`). Since the subwords are computed separately (i.e., `X[i][MSb]` is computed much earlier than `X[i][LSb]`), any intermediate carry-out bits are lost. This is the baseline support that we provide, where `ADD_ASV4` and `ADD_ASV8` are used for 4-bit and 8-bit subword additions, respectively. We refer to this as *unprovisioned* addition.

For applications where dropping carry-out bits is harmful to approximation quality, we also support *provisioned* subword vectorization. The programmer merely adds a third parameter to the pragmas in Listing 3: `#pragma asv input(A, 8, provisioned)`. With provisioned vectorization, subwords are allocated double the bits to include carry-out bits in the sum. In this case, `ADD_ASV8` and `ADD_ASV16` are used for 4-bit and 8-bit subword additions, respectively. Since the sum and the carry bits can both be accommodated, there is no overflow and we are able to always reach the precise result in the end with no loss of information. This approach is unique to our anytime WN architecture; conventional vectorization [26], [28] does not need to support provisioned addition as it assumes the sum to be the same bit-width as the operands. We compare unprovisioned and provisioned addition in Section V-E.

C. Skim Points

Checkpointing ensures forward progress on intermittently powered systems in the face of frequent power outages. Checkpointing has been implemented with ultra-low power processors such as the MSP430 [32], [42] and the ARM M0+ [18]. These processors consist of traditional volatile memory elements such as SRAM, which lose data on a power loss. Thus they are paired up with non-volatile memory such as Flash [42] or FRAM [32] to allow the system to save state prior to a power outage. Other processors incorporate non-volatile memory such as FRAM directly into the processor pipeline [35]. These processors, designed specifically for energy harvesting, automatically save state as they proceed. After power is restored, they resume processing immediately without needing to restore state from main memory.

Our WN architecture processes more samples than a conventional energy-harvesting system, by trading off quality for runtime using SWP and SWV. If an approximate result is acceptable, we process more input samples and avoid missing any, which is likely when the precise implementation is unable to keep up with the rate of incoming samples and must spend more time computing the precise result. Accepting the approximate results allows us to then *bypass* the rest of the processing for the current set of data and move on to the new data.

To support this, we introduce *skim points*, a mechanism for decoupling the checkpoint location from the restore location. We implement a new `SKM` instruction—shown in Listings 2 and 4—which indicates that an acceptable quality level has been reached. Upon encountering a `SKM` instruction, the processor saves the target address in a dedicated SKM non-volatile register. The system then continues processing the current input and performs a regular backup upon the next power outage. When the system resumes from the power outage however, it first checks the SKM non-volatile register to see if a skim point was set. If the register was set, the system jumps to the target address and begins processing the next input. As shown in line 8 of Listing 4, `SKM` instructions are statically inserted in the program by the compiler (or optionally by the user) after processing all data elements for each subword. The first skim point is placed at the earliest point where some approximate output is available, which is generally after the most significant subwords are processed.

IV. Methodology

We apply WN to both volatile processor systems [6], [42], which rely on checkpointing to ensure forward progress and non-volatile processor systems, which incorporate non-volatile memory elements such as FRAM directly in the processor pipeline [35]. For the checkpoint-based volatile processor, we implement a version of Clank [18], which uses a writeback buffer to track idempotency violations and maintain consistent memory state across multiple active periods. For our non-volatile processor, we implement the backup-every-cycle policy [35].

Simulation infrastructure. We use a cycle-accurate ARM M0+ CPU [3] simulator [17]. This CPU, targeted towards ultra-low power domains such as IoT and energy harvesting, contains a 2-stage pipeline, no branch predictor or caches and implements an iterative multiplier that takes 16 cycles to compute a 16×16 product. The ARM M0+ processor supports a maximum operating frequency of 48MHz [3]. However, due to the tight energy budgets of energy-harvesting devices, processors typically run at much lower frequencies [35]. We opt for a 24MHz operating frequency [18].

Simulation methodology. Our simulator takes as input 1-kHz voltage traces captured from a Wi-Fi source [13]. We model a 10μF capacitor as our energy storage [42]. Our simulator assumes a constant energy per instruction. This is consistent with our own hardware validation done using a TI MSP430 CPU, commonly used for energy-harvesting [22], [32], [42]. Our measurements show that the energy per instruction on the MSP430 is also constant [46]. The energy cost of all instructions (including any additional instructions due to WN) are faithfully accounted for in our simulations. To model intermittent execution, each application is invoked 3 times on 9 different voltage traces. We present the median runtime and error from all the runs in our results.

Benchmarks. Table I lists the kernels evaluated; these kernels are typically used in energy-harvesting applications [23], [39] Each program is written in C and compiled using GCC 5.2. These kernels span a broad range of application domains and execution times to allow us to explore the diverse areas energy-harvesting devices are deployed in. These applications originally use floating point operations; we converted these to fixed-point, keeping the error between the two to under 1%. `Conv2d`, `MatMul` and `Var` use 16-bit fixed point values while `Home`, `NetMotion` and `MatAdd` use 32-bit values. We also show the percentage of dynamic instructions per benchmark that are amenable to WN.

Error metric. We use Normalized Root Mean Square Error (NRMSE) as our quality metric [11], [51]. Since acceptable quality is inherently subjective and application specific, we present error curves to demonstrate the runtime-quality trade-offs as opposed to a choosing a fixed quality target per application which may or may not be applicable in all cases.

V. Evaluation

Our experimental results explore trade-offs in application runtime and output quality in our WN architecture. First, we show how subword pipelining and subword vectorization yield decreasing output error over time (Section V-A). Then we demonstrate the efficacy of our approach for a checkpoint-based volatile processor (Section V-B) and a non-volatile processor (Section V-C). We provide an overview of the power and area impact of WN (Section V-D) and detail several case studies that explore the design space of WN (Section V-E).

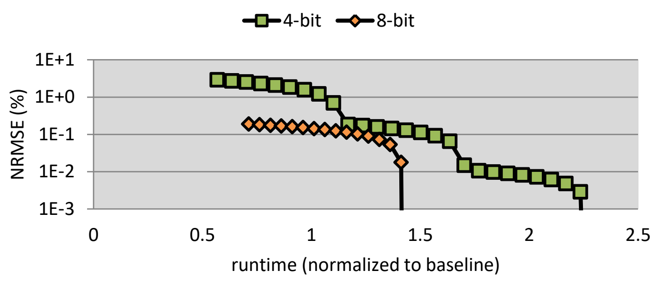
A. Runtime-Quality Trade-off

Figure 9 shows the runtime-quality curves for our applications when applying anytime WN techniques. Each figure shows the curves for 4-bit and 8-bit subwords. Runtime (x-axis) is normalized to the conventional precise execution. The y-axis effectively shows the error in output if the application was halted by a power outage at that moment. For SWV, we use *provisioned* addition.

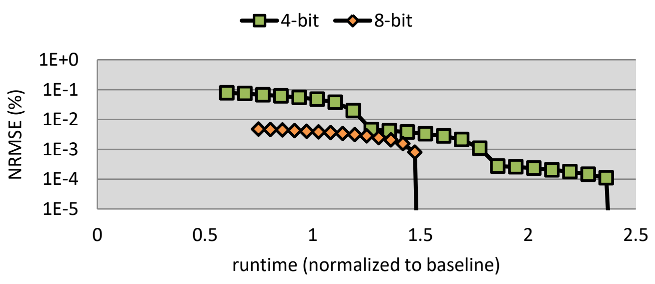
In all cases, quality improves as the application progresses, until the final precise output is reached for all benchmarks. Also, an approximate (yet acceptable) output is available early, allowing the application to be terminated early. However, our WN techniques incur runtime overhead to reach the precise output. This is due to the presence of other instructions that are not amenable to subword pipelining nor subword

Table I: Benchmark descriptions

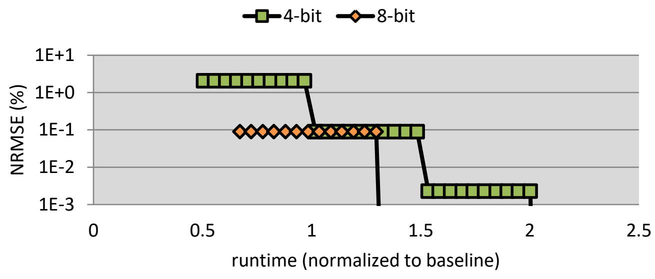
Benchmark	Area	Description	Insn %	Runtime	SWP	SWV
2D Convolution (Conv2d)	Image Processing	9×9 Gaussian filter applied on a 128×128 grayscale image.	10.49%	1487ms	✓	
Matrix Multiply (MatMul)	Data processing	Multiplication of two 64×64 Matrices	8.84%	298ms	✓	
Matrix Addition (MatAdd)		Addition of two 64×64 Matrices	8.94%	131ms		✓
Home Monitoring (Home)	Environmental Sensing	Periodic calculation of average conditions (e.g., temperature, humidity)	23.19%	30ms		✓
Data Logging (Var)		Calculates variance on data gathered from sensors	12.26%	32ms	✓	
Location Tracking (NetMotion)		Wildlife location tracking; calculates net movement over period of time	17.93%	47ms		✓



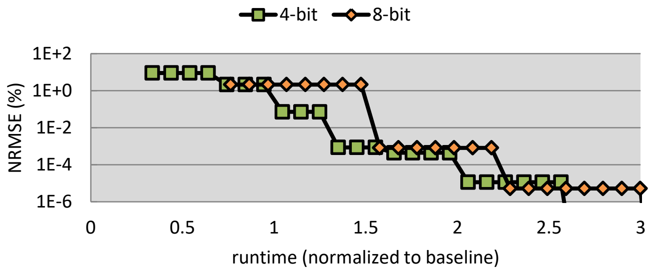
(a) Runtime-quality trade-off of Conv2d with SWP.



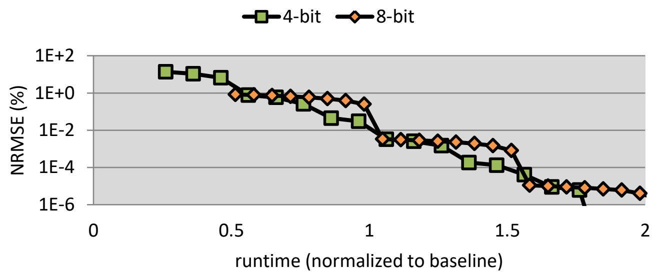
(b) Runtime-quality trade-off of MatMul with SWP.



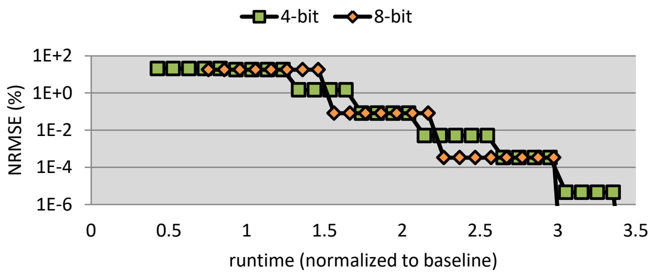
(c) Runtime-quality trade-off of Var with SWP.



(d) Runtime-quality trade-off of Home with SWV.



(e) Runtime-quality trade-off of MatAdd with SWV.



(f) Runtime-quality trade-off of NetMotion with SWV.

Figure 9: Runtime-quality trade-off curves.

vectorization and must be re-executed several times due to the iterative nature of our anytime WN techniques (e.g., conditional branches, address computation). On devices with frequent power outages, the benefit of producing outputs early and enabling interruptible execution outweighs the cost of longer runtime to the precise result. In the case of `MatAdd` (Figure 9e), with 8-bit SWV, an approximate result is available at only half of the baseline runtime. Terminating the application at that point would effectively yield a 2× speedup, with a error of just 0.8%.

Subword granularity controls the trade-off between how early an approximate output is available and how late the precise output is obtained. With smaller granularities (e.g., 4 bits), it generally takes longer for the application to produce the precise output. The advantage, however, is that an approximate output is available earlier (e.g., 2× earlier for 4-bit than 8-bit `MatAdd`, shown in Figure 9e). If the higher error (13.6% in 4-bit `MatAdd`) is tolerable based on the application, this yields a significant speed-up over the baseline.

For some applications, quality does not increase smoothly but rather increases in steps: `Var` (Figure 9c), `Home` (Figure 9d) and `NetMotion` (Figure 9f). Here, SWP and SWV are employed on reduction computations. Since registers are typically used to accumulate results, the output in non-volatile memory is unchanged until the entire reduction is finished (i.e., when the application finally stores the accumulated register value into memory). For instance in `Var`, several readings from sensors are summed up to calculate the variance. Thus, the output quality only increases once the final variance value is calculated and written to memory. A smoother quality curve could be obtained by modifying the benchmarks to update memory more frequently. However, this would incur additional overhead as the final

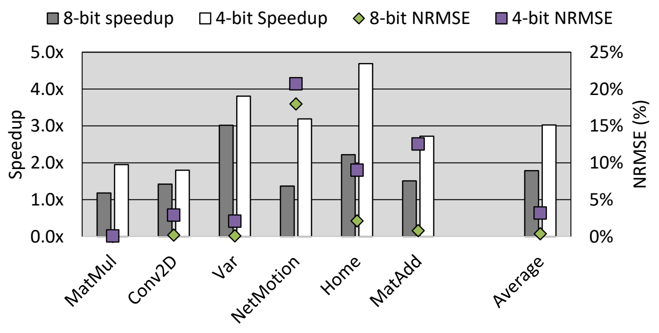


Figure 10: Speedup and quality (median NRMSE) of all benchmarks on volatile processor (with checkpointing)

variance calculation would have to be redone each time the value is written to memory.

Although only 10.53% of instructions (on average) are targeted by SWP (Table I) these long latency operations (i.e., multiplies) constitute a substantial portion of the total execution cycles for these applications. Thus, speeding up multiplies significantly improves overall program runtime. Similarly, SWV targets 16.68% of instructions on average. By effectively vectorizing targeted instructions, we increase throughout substantially and therefore application speedup.

B. *Checkpoint-Based Volatile Processor*

In checkpoint-based volatile processor systems, checkpoints are invoked periodically to back up the processor state. With Clank, this checkpointing is caused either by an idempotency violation or by a periodic watchdog interrupt. After a power outage, the processor state is recovered from the most recent checkpoint. This forces the application to re-execute any instructions after the checkpoint that were executed before the outage. In WN, skim points are inserted to indicate that an acceptable level of quality has been reached for the current input. Upon resuming from a power outage, the application can skip to the end of the current task and take its result as-is, producing an approximate output.

Figure 10 shows the median speedup and quality for all samples. By employing SWP and SWV, we observe considerable speedups, while still yielding high quality outputs. WN yields average speedups of 1.78× and 3.02× with errors of 0.36% and 3.17% for 8-bit and 4-bit subwords. In the best 8-bit case, `Var` improves performance by 3.02× with an error of 2%. In the best 4-bit case, `Home` achieves a speedup of 4.69× with an error of 9%. Our skim points eliminate the high overhead of re-executing instructions from the last checkpoint in volatile processor systems.

8-bit subword pipelining and subword vectorization generally produce higher quality results than the 4-bit techniques while 4-bit subword pipelining and subword vectorization yield higher speedups. This is expected since using 4-bit subwords allows for the output to be available earlier. However, as discussed in Section V-A, this comes at a cost of higher runtime overhead to reach the precise result. As a result, 4-bit techniques incur greater overhead compared to the precise case than the 8-bit versions.

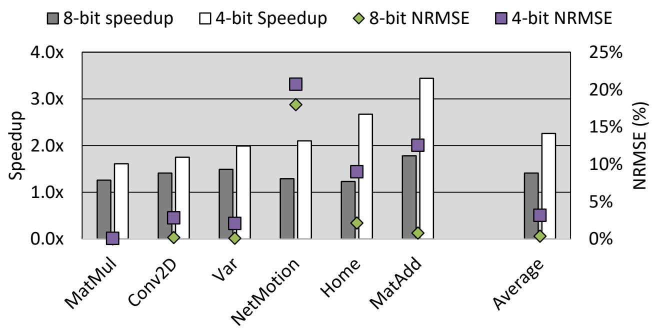


Figure 11: Speedup and quality (median NRMSE) for all benchmarks on NVP while varying the active power period.

C. *Non-Volatile Processor*

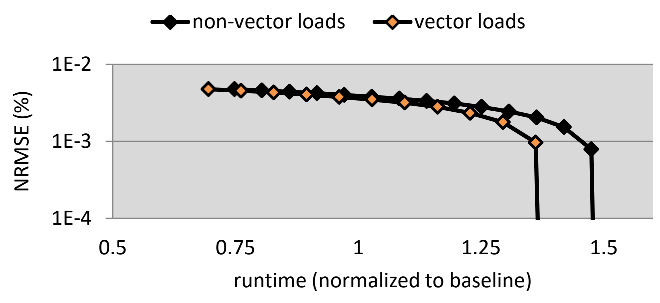
In the section, we evaluate the runtime-quality trade-off of our anytime approximations in the presence of frequent power outages, assuming a non-volatile processor (NVP). In a non-volatile processor, the processor's current state (e.g., program counter, register file) is backed up every cycle. Effectively, the current progress of the application is automatically checkpointed when power is lost. Similar to the checkpoint-based processor system, skim points allow the application to skip to the end of the current task and take its current result in non-volatile memory *as-is*, thereby producing an approximate output. Note that these results are not meant to compare non-volatile processors to checkpoint-based volatile processors but rather to show that WN provides benefits on both.

Figure 11 shows the speed-up and error for all benchmarks on NVP. The overall trend for NVP closely matches the results for checkpoint-based volatile processors (Section V-B). Our WN techniques yield average speedups of 1.41× and 2.26× with 8-bit and 4-bit subwords. In the best case, `MatAdd` improves performance by 1.78× and 3.44×, with output quality errors of 0.01% and 12.5% for 8-bit and 4-bit.

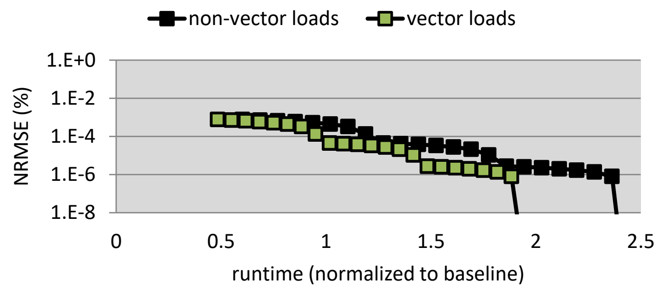
In general, we observe higher speedups with a checkpoint-based volatile processor than a non-volatile processor. This is expected as the checkpoint-based processor incurs greater re-execution overhead than the non-volatile processor, which saves state automatically. Thus, WN, which produces an approximate output sooner, avoids much of the re-execution overhead incurred by checkpoint-based systems leading to greater speedups on these systems.

D. *Area and Power Analysis*

For synthesis, we use the Synopsys Design Compiler Version N-2017.09. As energy-harvesting devices are typically manufactured at older technologies (e.g., 90nm [50]), we use TSMC's 65nm (nominal) process technology [31]. For area and power estimation, we use Cadence Innovus v16.22-s071 and Mentor Graphics ModelSim SE 10.4c.



(a) 8-bit



(b) 4-bit

Figure 12: Runtime-quality trade-off of MatMul with and without subword vectorization on load instructions.

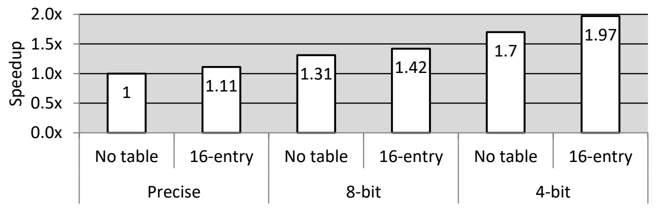


Figure 13: Speedup of Conv2d (when earliest available input is taken) with and without memoization and zero skipping. Results are normalized to the precise case, with no memoization or zero skipping.

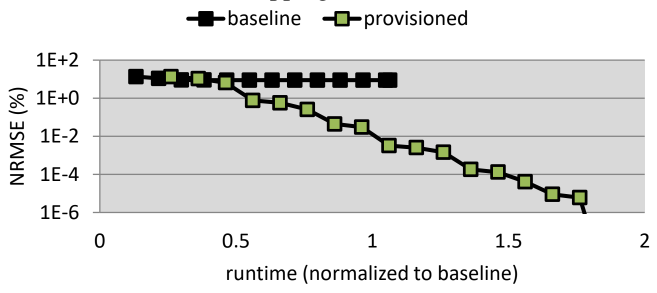


Figure 14: Runtime-quality trade-off of MatAdd with and without provisioned subword vectorization.

Using synthesis, we obtain an F_{max} of 1.12 GHz. As this is orders of magnitude above the 24MHz operating frequency of the CPU, the addition of the muxes has no impact on processor performance. The muxes in our carry chain incur an additional 0.02% area overhead, compared to a Cortex M0+ CPU implemented in 65nm [38]. The addition of the muxes increases the power consumption of the adder by 4%. Energy savings stem from the reduction in instructions executed, which we show in our prior performance results.

E. Design Exploration

In this section, we detail several case studies that explore the design space of WN.

Combining Vectorization and Pipelining. We have thus far focused on evaluating the impact of SWP and SWV separately. However, these two techniques are orthogonal and can be applied simultaneously to reap further gains. Figure 12 shows the runtime-quality curves for `MatMul` when applying both SWV and SWP. Specifically, the input data is transposed to subword-major order, enabling subword vectorization of the load instructions. Without SWV, each load instruction retrieves each data word in its entirety (i.e., all subwords) to perform a multiplication with SWP. This wastes memory bandwidth since only one of the subwords is needed for the pipelined multiplication. By vectorizing the loads, we better utilize bandwidth and improve performance. Applying load SWV to `MatMul` produces approximate outputs 1.08× and 1.24× earlier than without vectorization for 8-bit and 4-bit subwords.

Memoization. In this section, we explore the efficacy of memoization when used in conjunction with SWP. We employ a 16-entry direct-mapped lookup table that stores the results from prior multiplication operations.[5] The index into the table is the concatenation of the two least significant bits of both operands, while the lookup tag is the concatenation of the upper 14 bits of both operands. In the case of 16-bit memoization, we use all 28 tag bits. However, for 8-bit and 4-bit cases we only use 20 bits and 16 bits, respectively.

Figure 13 shows the results for `Conv2d` when the earliest available output is taken, with and without memoization and zero skipping enabled. The speedups are shown normalized to the precise case, with no memoization or zero skipping. Memoization and zero skipping further improve the speedup offered by SWP, from 1.7× to 1.97× for the 4-bit case and from 1.31× to 1.42× for the 8-bit case, while speeding up the precise case by 1.11×. The greater speedup for smaller subwords is expected as smaller subwords are more likely to be repeated often and therefore hit in the table, offering greater benefits for SWP compared to the precise case. Similarly, smaller subwords increase the chances for more zeros to be seen, making zero skipping more effective when used in conjunction with SWP. We use CACTI [30] and find that the 16-entry table only occupies 40.5% of the area of a 16x16 multiplier, making memoization a viable option for systems employing WN.

Provisioned Vectorization for Addition. Provisioned addition allocates extra space to alleviate overflow for computations that produce carry bits (i.e., addition and subtraction). Although the unprovisioned case yields an approximate output slightly earlier, its approximation quality sees little improvement over time and does not approach the precise

[5]We empirically determine that more entries only provides modest additional improvements at the cost of extra area.

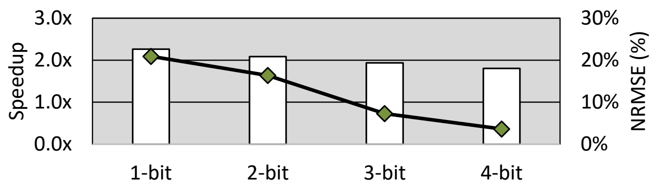


Figure 15: Speedup and error of Conv2d when earliest available output is taken with small subwords.

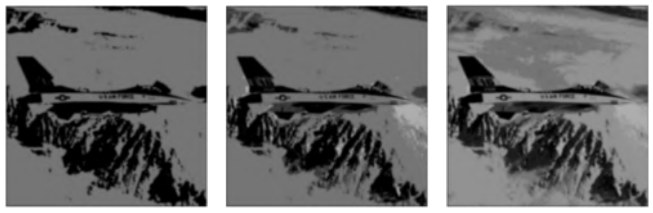

(a) 1-bit (b) 2-bit (c) 3-bit

Figure 16: Earliest available outputs of Conv2d with small subwords.

result, due to overflow as shown in Figure 14 (the provisioned case uses 8 bits per subword). The error for the unprovisioned case does not decrease when subsequent subwords are processed. The original 32-bit inputs in an application are not likely to overflow since they typically do not utilize all 32 bits. However, with subword vectorization, there is much higher risk of overflow in the subword results since less significant subwords are likely to fully utilize their bits. Using provisioned addition, we eliminate this issue, and ensure that we eventually reach the precise output as shown in Section V-A. Provisioned addition marries well with our goal of producing progressively higher quality results with each subsequent subword processed. While provisioning is an issue with operations such as addition and subtraction, bitwise operations such as AND, OR, NOT etc., can reap the full performance gains of our unprovisioned SWV.

Pipelining with Small Subwords. Though we have focused on 8-bit and 4-bit subwords, it is possible to use even smaller subwords. This section explores the use of 3-, 2- and 1-bit subwords for SWP. Figure 15 shows the speedup (relative to the baseline) and quality of `Conv2d` if the applications were terminated as soon as an approximate output is available (i.e., as soon as the most significant subword is processed in our experiments). Smaller subwords have higher error but yield greater speedups. Figure 16 visualizes the approximate outputs of `Conv2d` with 4-, 3-, 2- and 1-bit subword pipelining. Contrast these outputs to that of the baseline precise execution in Figure 2a. In a conventional processor, terminating the application halfway (i.e., 2× speedup) yields an unacceptable output (Figure 2b). However, by applying 1-bit subword pipelining, a complete output (Figure 16a) is achievable with a speedup of 2.26×.

Comparison to Input Sampling. The `Var` benchmark collects information from eight sensors and periodically calculates the variance of the dataset for logging. Figure 17

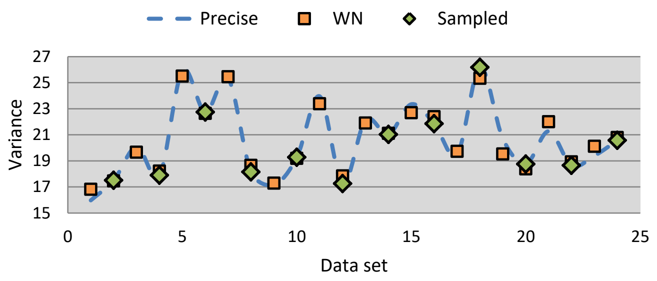


Figure 17: WN vs. input sampling for the Var benchmark.

shows a comparison of the results from this benchmark using WN vs. input sampling, for 24 data sets. Recall from Figure 9c that the `Var` benchmark takes half the time to calculate the first 4-bits of the result compared to the precise implementation. Thus, WN can process two samples for each sample that can be processed by the precise implementation, with the same energy budget. WN faithfully captures the peaks and troughs of the input signal, with only a 1.53% average error in the measured values.

F. Evaluation Summary

We present experimental results that demonstrate the efficacy of our WN mechanisms—subword pipelining, subword vectorization and skim points—on intermittent computing systems. Our applications achieve favorable runtime-quality curves that enable early availability of approximate (yet acceptable) outputs. Our WN techniques are effective in both non-volatile processors and checkpoint-based volatile processors, achieving average speedups of 1.41× (8-bit) and 2.26× (4-bit) in the former and 1.78× (8-bit) and 3.02× (4-bit) in the latter.

VI. Related Work

In this section, we highlight relevant work in three areas: techniques for energy-harvesting systems, anytime algorithms and reduced precision arithmetic.

Energy-Harvesting Devices. Several schemes have been proposed to allow energy-harvesting systems to make forward progress in the face of power outages [4], [5], [16]. Prominent techniques targeting volatile processors include Mementos [42], Hibernus [6], QuickRecall [22] and Clank [18]. There has also been much work in designing entirely non-volatile processors [33], [35], [41], [44].

Recent work [34] also addresses the *timeliness* of outputs on energy-harvesting systems. That work mines historical information from past inputs, retained in non-volatile memory, to improve the quality of those results in parallel with processing new data. Our work differs in several significant ways. We focus on processing the *current* data to an acceptable level of quality before moving onto the new data. As data in energy-harvesting systems is predominantly transmitted off-device, the technique proposed by Ma et al. adds additional overhead on the receiving server to keep track of multiple copies of a single output, each of possibly varying quality for the same input data. This also requires that any processing that depends on this output must be re-executed when a new output is received. Our work requires only minor changes to hardware; Ma et al. require significant changes such as SIMD units and buffers to store information from previous executions in order to improve on prior results.

Anytime Algorithms. Anytime algorithms produce an output whose accuracy increases over time and were initially developed for time-dependent planning and decision-making [9], [19], [29]. Anytime algorithms are most applicable in applications operating under real-time constraints where lower output quality is preferred to exceeding time limits. In contrast, we focus on the energy budget rather than the time constraints. Prior work considers the optimal scheduling policies [14], [48] and the error composition of anytime algorithms [53]. In computer architecture, there has been little work looking at anytime algorithms. Existing work includes the Anytime Automaton [45] on which this paper builds and work that explores porting contract anytime algorithms to GPUs and providing CUDA-enabled online quality control [36]. There has also been work that combines novel coding techniques with anytime principles to reduce the impact of stragglers in distributed computations [12].

Reduced-Precision Arithmetic. Reduced precision computation has been proposed across a number of domains [37], [43]. Reduced precision is used in deep neural networks to trade off accuracy for more efficient execution [24]. Bit-serial computation of reduced-precision operations can achieve even greater efficiency [25]. Bit-serial computation bears some similarity to our anytime subword computations; the key differences are 1) granularity and 2) our anytime approach is flexible, allowing computations to stop early. One of the key distinctions in our work is that we do not unilaterally reduce the precision of our computations; we take an anytime approach that *may* compute a reduced-precision result if the energy budget does not permit the full computation but also *may* produce the full result. Not all operands require the full-width precision of their datatype; dynamically recognizes operations that require less precision and performing these operations on a narrower bit-width saves power [7]; in contrast, we assume the full-width is needed and dynamically select how much of the data to process based on power available and desired quality.

VII. Conclusion

In this paper, we present novel techniques for anytime approximations on energy-harvesting devices. Specifically, we provide architectural support for subword pipelining and subword vectorization, allowing computations to dynamically process the most significant bits first. We then introduce skim points, which decouple the checkpoint locations from their recovery locations. Together, these mechanisms allow applications to produce an approximate (yet acceptable) output early. Then in a best-effort manner, the application can continue refining the results towards the precise output. When a power outage occurs, the application bypasses to the end and takes its current approximate result as-is, enabling greater forward progress. We observe speedups of 2.26× and 3.02× on non-volatile and checkpoint-based volatile processors, respectively, while still producing high-quality outputs.

Acknowledgments

The authors thank the anonymous reviewers and members of the NEJ group for their valuable feedback. We also thank Mostafa Mahmoud for his help with synthesis. This research has been funded in part by the Computing Hardware for Emerging Intelligent Sensory Applications (COHESA) project. COHESA is financed under the National Sciences and Engineering Research Council of Canada (NSERC) Strategic Networks grant number NETGP485577-15. This work is further supported by a Percy Edward Hart Early Career Professorship, an IBM PhD Fellowship, the University of Toronto and the University of Wisconsin-Madison.

References

[1] J. Albericio *et al.*, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Int. Sym. on Computer Architecture*, 2016.

[2] R. Allen and K. Kennedy, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann San Francisco, 2002, vol. 289.

[3] *ARM Cortex M0+ Technical Reference Manual*, ARM Technologies Ltd., https://bit.ly/2KZ1WGf.

[4] D. Balsamo *et al.*, "Graceful performance modulation for power-neutral transient computing systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 738–749, 2016.

[5] D. Balsamo *et al.*, "Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 1968–1980, 2016.

[6] D. Balsamo *et al.*, "Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems," *IEEE Embedded Systems Letters*, 2015.

[7] D. Brooks and M. Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," in *High-Performance Computer Architecture*, 1999, pp. 13–22.

[8] D. Citron, D. Feitelson, and L. Rudolph, "Accelerating multimedia processing by implementing memoing in multiplication and division units," in *Architectural Support for Programming Languages and Operating Systems*, 1998, pp. 252–261.

[9] T. L. Dean and M. Boddy, "An analysis of time-dependent planning," in *AAAI*, 1988.

[10] C. G. Enright *et al.*, "Modelling glycaemia in ICU patients: A dynamic Bayesian network approach," in *Int. Conf. on Biomedical Engineering Systems and Technologies*, 2010.

[11] H. Esmaeilzadeh *et al.*, "Architecture support for disciplined approximate programming," in *Architectural Support for Programming Languages and Operating Systems*, 2012.

[12] N. S. Ferdinand and S. C. Draper, "Anytime coding for distributed computation," in *IEEE Allerton conference on Communication, Control, and Computing*, 2016, pp. 954–960.

[13] M. Furlong *et al.*, "Realistic simulation for tiny batteryless sensors," in *International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, 2016.

[14] A. Garvey and V. Lesser, "Design-to-time real-time scheduling," *IEEE Transactions on Systems, Man, and Cybernetics*, 1993.

[15] T. N. Gia *et al.*, "IoT-based continuous glucose monitoring system: A feasibility study," in *Int. Conf. on Sustainable Energy Information Technology*, 2017.

[16] J. Hester, L. Sitanayah, and J. Sorber, "Tragedy of the Coulombs: Federating energy storage for tiny, intermittently-powered sensors," in *ACM Conference on Embedded Networked Sensor Systems*, 2015.

[17] M. Hicks, *Thumbulator: Cycle accurate ARMv6-m instruction set simulator.*, 2016, https://bit.ly/2RJX36A.

[18] M. Hicks, "Clank: Architectural Support for Intermittent Computation," in *Int. Sym. on Computer Architecture*, 2017.

[19] E. J. Horvitz, "Reasoning about beliefs and actions under computational resource constraints," in *Workshop on Uncertainty in Artificial Intelligence*, 1987.

[20] *Intel AVX-512*, Intel Corp., https://intel.ly/2SyYl4i.

[21] *ISO Accuracy standards for Diabetes Monitoring*, ISO, https://bit.ly/2dLMX03.

[22] H. Jayakumar, A. Raha, and V. Raghunathan, "QUICKRECALL: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers," in *Int. Conf. on Embedded Systems*, 2014, pp. 330–335.

[23] P. Juang *et al.*, "Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet," in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[24] P. Judd *et al.*, "Proteus: Exploiting numerical precision variability in deep neural networks," in *Int. Conf. on Supercomputing*, 2016.

[25] P. Judd *et al.*, "Stripes: Bit-serial deep neural network computing," in *IEEE/ACM Int. Sym. on Microarchitecture*, 2016, pp. 1–12.

[26] L. Kohn *et al.*, "The visual instruction set (VIS) in ultrasparc," in *Compcon'95. 'Technologies for the Information Superhighway', Digest of Papers.* IEEE, 1995, pp. 462–469.

[27] R. Lee, "Memo functions and machine learning," *Nature*, vol. 218, pp. 19–22, 1968.

[28] R. Lee, "Subword parallelism with MAX-2," *IEEE Micro*, vol. 16, no. 4, pp. 51–59, 1996.

[29] V. Lesser, J. Pavlin, and E. Durfee, "Approximate processing in real-time problem-solving," *AI Magazine*, 1988.

[30] S. Li *et al.*, "CACTI-P: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *Int. Conf. on Computer-Aided Design*, 2011, pp. 694–701.

[31] Y. Liu *et al.*, "A 65nm reram-enabled nonvolatile processor with 6x reduction in restore time and 4x higher clock frequency," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, 2016.

[32] B. Lucia and B. Ransford, "A Simpler, Safer Programming and Execution Model for Intermittent Systems," in *Int. Conf. on Programming Language Design and Implementation*, 2015.

[35] K. Ma *et al.*, "Architecture exploration for ambient energy harvesting nonvolatile processors," in *Proceedings of the Int. Sym. on High Performance Computer Architecture*, 2015.

[33] K. Ma *et al.*, "NEOFog: Nonvolatility-exploiting optimizations for fog computing," in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2018.

[34] K. Ma *et al.*, "Incidental computing on IoT nonvolatile processors," in *Int. Sym. on Microarchitecture*, 2017.

[36] R. Mangharam and A. A. Saba, "Anytime Algorithms for GPU Architectures," in *RTSS*, 2011.

[37] T. Moreau *et al.*, "Approximating to the last bit," in *Workshop on Approximate Computing*, 2016.

[38] J. Myers *et al.*, "An 80nw retention 11.7pj/cycle active subthreshold arm cortex-m0+ subsystem in 65nm cmos for wsn applications," *IEEE International Solid-State Circuits Conference*, 2015.

[39] S. Naderiparizi *et al.*, "WISPCam: A battery-free RFID camera," in *IEEE Int. Conf. on RFID*, 2015.

[40] G. O'Leary *et al.*, "Nurip: Neural interface processor for brain-state classification and programmable-waveform neurostimulation," *IEEE Journal of Solid-State Circuits*, 2018.

[41] M. Qazi, A. Amerasekera, and A. P. Chandrakasan, "A 3.4-pJ FeRAM-enabled D flip-flop in 0.13-m CMOS for nonvolatile processing in digital systems," in *IEEE Journal of Solid-State Circuits*, vol. 49, 2014, pp. 202–211.

[42] B. Ransford, J. Sorber, and K. Fu, "Mementos: system support for long-running computation on RFID-scale devices," in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[43] C. Rubio-Gonzalez *et al.*, "Precimonious: Tuning assistant for floating-point precision," in *Int. Conf. High Performance Computing, Networking, Storage and Analysis*, 2013.

[44] N. Sakimura *et al.*, "A 90nm 20MHz fully nonvolatile microcontroller for standby-power-critical applications," in *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, 2014.

[45] J. San Miguel and N. Enright Jerger, "The Anytime Automaton," in *Int. Sym. on Computer Architecture*, 2016.

[46] J. San Miguel *et al.*, "The EH Model: Early Design Space Exploration of Intermittent Processor Architectures," in *Int. Sym. on Microarchitecture*, 2018.

[47] M. W. Shafer and E. Morgan, "Energy Harvesting for Marine-Wildlife Monitoring," in *ASME Conference on Smart Materials, Adaptive Structures and Intelligent Systems*, 2014.

[48] W.-K. Shih, J. W. S. Liu, and J.-Y. Chung, "Fast algorithms for scheduling imprecise computations," in *RTSS*, 1989.

[49] A. Sodani and G. Sohi, "Dynamic Instruction Reuse," in *Int. Sym. on Computer Architecture*, 1997.

[50] *Microcontroller Advances from STMicroelectronics Extend Performance Leadership for Smarter Technology Everywhere*, ST Microelectronics, https://bit.ly/2PofKuy.

[51] A. Yazdanbakhsh *et al.*, "RFVP: Rollback-free value prediction with safe-to-approximate loads," *ACM Transactions on Architecture and Code Optimization*, 2016.

[52] F. Zamora-Martinez *et al.*, "On-line learning of indoor temperature forecasting models towards energy efficiency," *Energy and Buildings*, vol. 83, pp. 162–172, 2014.

[53] S. Zilberstein, "Operational Rationality through Compilation of Anytime Algorithms," Ph.D. dissertation, Technion - Israel Institute of Technology, 1982.