Appears in the Proceedings of 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)

Architecture Support for Disciplined Approximate Programming

Hadi Esmaeilzadeh Adrian Sampson Luis Ceze
University of Washington
Department of Computer Science & Engineering
{hadianeh,asampson,luisceze}@cs.washington.edu

Doug Burger
Microsoft Research
dburger@microsoft.com

Abstract

Disciplined approximate programming lets programmers declare which parts of a program can be computed approximately and consequently at a lower energy cost. The compiler proves statically that all approximate computation is properly isolated from precise computation. The hardware is then free to selectively apply approximate storage and approximate computation with no need to perform dynamic correctness checks.

In this paper, we propose an efficient mapping of disciplined approximate programming onto hardware. We describe an ISA extension that provides approximate operations and storage, which give the hardware freedom to save energy at the cost of accuracy. We then propose Truffle, a microarchitecture design that efficiently supports the ISA extensions. The basis of our design is dual-voltage operation, with a high voltage for precise operations and a low voltage for approximate operations. The key aspect of the microarchitecture is its dependence on the instruction stream to determine when to use the low voltage. We evaluate the power savings potential of in-order and out-of-order Truffle configurations and explore the resulting quality of service degradation. We evaluate several applications and demonstrate energy savings up to 43%.

Categories and Subject Descriptors C.1.3 [*Other Architecture Styles*]; C.0 [*Computer Systems Organization*]: Hardware/software interfaces

General Terms Design, Performance

Keywords Architecture, disciplined approximate computation, power-aware computing, energy

1. Introduction

Energy consumption is a first-class concern in computer systems design. Potential benefits go beyond reduced power demands in servers and longer battery life in mobile devices; reducing power consumption is becoming a requirement due to limits of device scaling in what is termed the *dark silicon* problem [4, 11].

Prior work has made significant progress in various aspects of energy efficiency. Hardware optimizations include power gating, voltage and frequency scaling, and sub-threshold operation with error correction [10]. Software efforts have explored managing energy as an explicit resource [30], shutting down unnecessary hardware, and energy-aware compiler optimizations [28]. Raising energy concerns to the programming model can enable a new space of energy savings opportunities.

Trading off quality of service is one technique for reducing energy usage. Allowing computation to be approximate can lead to significant energy savings because it alleviates the "correctness tax" imposed by the wide safety margins on typical designs. Indeed, prior work has investigated hardware and algorithmic techniques for approximate execution [2, 12, 14, 16]. Most applications amenable to energy–accuracy trade-offs (e.g., vision, machine learning, data analysis, games, etc.) have approximate components, where energy savings are possible, and precise components, whose correctness is critical for application invariants [19, 23].

Recent work has explored language-level techniques to assist programmers in identifying *soft slices*, the parts of programs that may be safely subjected to approximate computation [23]. The hardware is free to use approximate storage and computation for the soft slices without performing dynamic safety checks. Broadly, we advocate co-designing hardware support for approximation with an associated programming model to enable new energy-efficiency improvements while preserving programmability.

In this paper, we explore how to map disciplined approximate programming models down to an approximation-aware microarchitecture. Our architecture proposal includes an ISA extension, which allows a compiler to convey what can be approximated, along with microarchitectural extensions to typical in-order and out-of-order processors that implement the ISA. Our microarchitecture proposal relies on a dual voltage supply for SRAM arrays and logic: a high V_{dd} (leading to accurate operation) and a low V_{dd} (leading to approximate but lower-power operation). We discuss the implementation of the core structures using dual-voltage primitives as well as dynamic, instruction-based control of the voltage supply. We evaluate the energy consumption and quality-of-service degradation of our proposal using a variety of benchmarks including a game engine, a raytracer, and scientific algorithms.

The remainder of this paper is organized as follows. Section 2 describes our ISA proposal and its support for an efficient microarchitecture via tight coupling with static compiler guarantees. Section 3 then explains microarchitecture implementation alternatives. Section 4 follows up with a detailed description of our dual-voltage microarchitecture and how the instruction stream can control dynamic voltage selection. Section 5 is our evaluation of power savings and quality-of-service degradation. Sections 6 and 7 discuss related work and conclude.

2. An ISA for Disciplined Approximate Computation

With disciplined approximate programming, a program is decomposed into two components: one that runs *precisely*, with the typical semantics of conventional computers, and another that runs *approximately*, carrying no guarantees but only an expectation of best-effort computation. Many applications, such as media processing and machine learning algorithms, can operate reliably even when errors can occur in portions of them [8, 9, 18, 23, 27]. Floating-point data, for example, is by nature imprecise, so many FP-heavy applications have inherent tolerance to error. An architecture supporting disciplined approximation can take advantage of relaxed precision requirements to expose errors that would otherwise need to be prevented or corrected at the cost of energy. By specifying the portion of the application that is tolerant to error, the programmer gives the architecture permission to expose faults when running that part of the program. We propose an ISA that enables a compiler to communicate where approximation is allowed.

Our ISA design is defined by two guiding principles: the ISA should provide an *abstract* notion of approximation by replacing guarantees with informal expectations; and the ISA may be *unsafe*, blindly trusting the programming language and compiler to enforce safety invariants statically.

Replacing Guarantees with Expectations ISAs normally provide formal *guarantees* for operations (e.g., an "add" instruction must produce the sum of its operands). Approximate operations, however, only carry an *expectation* that a certain operation will be carried out correctly; the result is left formally undefined. For example, an approximate "add" instruction might leave the contents of the output register formally undefined but specify an *expectation* that the result will approximate the sum of the operands. The compiler and programmer may not rely on any particular pattern or method of approximation. Informally, however, they may expect the approximate addition to be useful in "soft" computation that requires summation.

The lack of strict guarantees for approximate computation is essential for *abstract* approximation. Instructions do not specify which particular energy-saving techniques are used; they only specify where approximation may be applied. Consequently, a fully-precise computer is a valid implementation of an approximation-aware ISA. Compilers for such ISAs can, without modification, take advantage of new approximation techniques as they are implemented. By providing no guarantees for approximate computation, the ISA permits a full range of approximation techniques.

By leaving the kind and degree of approximation unspecified, an approximation-aware ISA could pose challenges for portability: different implementations of the ISA can provide different error distributions for approximate operations. To address this issue, implementations can allow software control of implementation parameters such as voltage (see Section 3). Profiling and tuning mechanisms could then discover optimal settings for these hardware parameters at application deployment time. This tuning would allow a single application to run at the same level of quality across widely varying approximation implementations.

Responsibilities of the Language and Compiler An architecture supporting approximate computing requires collaboration from the rest of the system stack. Namely, the architecture relegates concerns of *safety and programmability* to the language and compiler. In order to be usable, approximate computation must be exposed to the programmer in a way that reduces the chance of catastrophic failures and other unexpected consequences. These concerns, while important, can be relegated to the compiler, programming language, and software-engineering tools.

EnerJ [23] is a programming language supporting disciplined approximation. Using a type system, EnerJ provides a static *non-interference* guarantee, ensuring that the approximate part of a program cannot affect the precise portion. In effect, EnerJ enforces separation between the error-tolerant and error-sensitive parts of a program, identifying and isolating the parts that may be subject to relaxed execution semantics. This strict separation brings safety and predictability to programming with approximation. Because it is static, the non-interference guarantee requires no runtime checking, freeing the ISA (and its implementation) from any need for safety checks that would themselves impose overheads in performance, energy, and design complexity.

In this paper, we assume that a language like EnerJ is used to provide safety guarantees *statically* to the programmer. The ISA must only expose approximation as an option to the compiler: it does not provide dynamic invariant checks, error recovery, or any other support for programmability. The ISA is thus unsafe *per se*. If used incorrectly, the ISA can produce unexpected results. It is tightly coupled with the compiler and trusts generated code to observe certain invariants. This dependence on static enforcement is essential to the design of a simple microarchitecture that does not waste energy in providing approximation.

2.1 Requirements for the ISA

An ISA extension for disciplined approximate programming consists of new instruction variants that leave certain aspects of their behavior undefined. These new instructions must strike a balance between energy savings and usability: they must create optimization opportunities through strategic use of undefined behavior but not be so undefined that their results could be catastrophic. Namely, approximate instructions should leave certain data values undefined but maintain predictable control flow, exception handling, and other bookkeeping.

To support a usable programming model similar to EnerJ, an approximation-aware ISA should exhibit the following properties:

- Approximate computation must be controllable at an *instruction granularity*. Approximation is most useful when it can be interleaved with precise computation. For example, a loop variable increment should likely be precise while an arithmetic operation in the body of the loop may be approximate. For this reason, it must be possible to mark individual instructions as either approximate or precise.
- The ISA must support *approximate storage*. The compiler should be able to instruct the ISA to store data approximately or precisely in registers, caches, and main memory.
- It must be possible to *transition* data between approximate and precise storage. (In EnerJ, approximate-to-precise movement is only permitted using an explicit programmer "endorsement," but this annotation has no runtime effect.) For full flexibility, the ISA must permit programs to use precise data approximately and vice-versa.
- Precise instructions, where approximation is not explicitly enabled, must carry traditional semantic guarantees. The effects of approximation must be constrained to where it is requested by the compiler.
- Approximation must be confined to predictable areas. For example, address computation for memory accesses must always be precise; approximate store instructions should not be allowed to write to arbitrary memory locations. Approximate instructions must not carry semantics so relaxed that they cannot be used.

Table 1. ISA extensions for disciplined approximate programming. These instructions are based on the Alpha instruction set.

Group	Approximate Instruction
Integer load/store	LDx.a, STx.a
Integer arithmetic	ADD.a, CMPEQ.a, CMPLT.a, CMPLE.a, MUL.a, SUB.a
Logical and shift	AND.a, NAND.a, OR.a, XNOR.a NOR.a, XOR, CMOV.a, SLL.a, SRA.a, SRL.a
Floating point load/store	LDF.a, STF.a
Floating point operation	ADDF.a, CMPF.x, DIVF.a, MULF.a, SQRTF.a, SUBF.a, MOV.a, CMOV.a, MOVFI.a, MOVIF.a

2.2 ISA Extensions for Approximation

Table 1 summarizes the approximate instructions that we propose adding to a conventional architecture. Without loss of generality, we assume an underlying Alpha ISA [1].

Approximate Operations The extended ISA provides approximate versions of all integer arithmetic, floating-point arithmetic, and bitwise operation instructions provided by the original ISA. These instructions have the same form as their precise equivalents but carry no guarantees about their output values. The approximate instructions instead carry the informal expectation of approximate adherence to the original instructions' behavior. For example, ADD.a takes two arguments and produces one output, but the ISA makes no promises about what that output will be. The instruction may be expected to typically perform addition but the programmer and compiler may not rely on any consistent output behavior.

Approximate Registers Each register in the architecture is, at any point in time, in either *precise mode* or *approximate mode*. When a register is in approximate mode, reads are not guaranteed to obtain the exact value last written, but there is an expectation that the value is likely the same.

The compiler does not explicitly set register modes. Instead, the precision of a register is implicitly defined based on the precision of the last instruction that wrote to it. In other words, a precise operation makes its destination register precise while an approximate operation puts its destination register into approximate mode.

While register precision modes are set implicitly, the precision of operand accesses must be declared explicitly. Every instruction that takes register operands is extended to include an extra bit per operand specifying the operand's precision. This makes precision level information available to the microarchitecture *a priori*, drastically simplifying the implementation of dual-voltage registers (see Section 4.1). It does not place a significant burden on the compiler as the compiler must statically determine registers' precision anyway. If the correspondence between register modes and operand accesses is violated, the value is undefined (see below).

With this design, data can move freely between approximate and precise registers. For example, a precise ADD instruction may use an approximate register as an operand; this transition corresponds to an *endorsement* in the EnerJ language. The opposite transition, in which precise data is used in approximate computation, is also permitted and frequently occurs in EnerJ programs.

Approximate Loads, Stores, and Caching The ISA defines a *granularity of approximation* at which the architecture supports setting the precision of cached memory. In practice, this granularity will likely correspond to the smallest cache line size in the processor's cache hierarchy.[1] For example, if an architecture has 16-byte L1 cache lines and supports varying the precision of every cache line, then it defines the approximation granularity to be 16 bytes. Each region of memory aligned to the granularity of approximation (hereafter called an *approximation line* for brevity) is in either approximate or precise mode at any given time.

An approximation line's precision mode is implicitly controlled by the precision of the loads and stores that access it. In particular, the ISA guarantees reliable data storage for precise accesses if, for every load from line x, the preceding store to line x has the same precision. (That is, after a precise store to x, only precise loads may be issued to x until the next store to x.) For the compiler and memory allocator, this amounts to ensuring that precise and approximate data never occupy the same line. Memory allocation and object layout must be adapted to group approximate data to line-aligned boundaries. Statically determining each line's precision is trivial for any language with sufficiently strong static properties. In EnerJ specifically, a type soundness theorem implies that the precision of every variable is known statically for every access.

The ISA requires this pattern of consistent accesses in order to simplify the implementation of approximation-aware caching. Specifically, it allows the following simple cache-approximation policy: a line's precision is set by misses and writes but is not affected by read hits. During read hits, the cache can assume that the precision of the line matches the precision of the access.

Approximate loads and stores may read and write arbitrary values to memory. Accordingly, precise stores always write data reliably, but a precise load only reads data reliably when it accesses a line in precise mode. However, any store (approximate or precise) can only affect the address it is given: address calculation and indexing are never approximated.

Approximate Main Memory While this paper focuses on approximation in the core, main memory (DRAM modules) may also support approximate storage. The refresh rate of DRAM cells, for example, may be reduced so that data is no longer stored reliably [19]. However, memory approximation is entirely decoupled from the above notion of approximate caching and load/store precision. This way, an approximation-aware processor can be used even with fully-precise memory. Furthermore, memory modules are likely to support a different granularity for approximation from caches—DRAM lines, for instance, typically range from hundreds of bytes to several kilobytes. Keeping memory approximation distinct from cache approximation decouples the memory from specific architecture parameters.

The program controls main-memory approximation *explicitly*, using either a special instruction or a system call to manipulate the precision of memory regions. Loads and stores are oblivious to the memory's precision; the compiler is responsible for enforcing a correspondence. When main-memory approximation is available, the operating system may store precision levels in the page table to preserve the settings across page evictions and page faults.

Preservation of Precision In several cases, where the ISA supports both precise and approximate operation, it relies on the compiler to treat certain data consistently as one or the other. For example, when a register is in approximate mode, all instructions that use that register as an operand must mark that operand as approximate. Relying on this correspondence simplifies the implementation of approximate SRAM arrays (Section 4.1).

The ISA does not enforce precision correspondence. No exception is raised if it is violated. Instead, as with many other situations in the ISA, the resulting value from any inconsistent operation is left undefined. Unlike other approximate operations, however, the *informal expectation* in these situations is also weaker: precise reads from approximate-mode registers, for example, should be expected to return random data. The compiler should avoid these situations even when performing approximate computation.

[1] Note that architects generally avoid tying cache line size to the ISA. However, we believe that in cases of strong co-design between architecture and compiler such as ours, it is acceptable to do so.

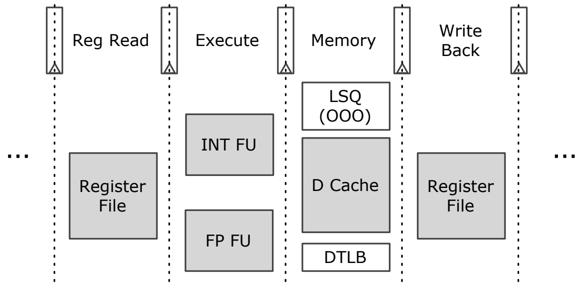


Figure 1. The data movement/processing plane of the processor pipeline. Approximation-aware structures are shaded. The instruction control plane stages (Fetch and Decode, as well as Rename, Issue, Schedule, and Commit in the OOO pipeline) are not shown.

These situations constitute violations of precision correspondence:

- A register in approximate mode is used as a precise operand to an instruction. (Note that a precise instruction can use an approximate operand; the operand must then be declared as approximate.)
- Conversely, a precise-mode register is used as an approximate operand.
- An approximate load (e.g., LDW.a) is issued to an address in an approximation line that is in precise mode.
- Conversely, a precise load is issued to an approximate-mode line.

In every case, these are situations where undefined behavior is already present due to approximation, so the ISA's formal guarantees are not affected by this choice. Consistency of precision only constitutes a recommendation to the compiler that these situations be avoided. These weak semantics keep the microarchitecture simple by alleviating the need for precision state checks (see Section 4.1).

3. Design Space

As discussed above, approximate instructions may produce arbitrary results. For example, ADD.a may place any value in its destination register. However, approximate instructions still have defined semantics: ADD.a cannot modify any register other than its output register; it cannot raise a divide-by-zero exception; it cannot jump to an arbitrary address. These guarantees are necessary to make our approximation-aware ISA usable.

Our microarchitecture must carefully distinguish between structures that can have relaxed correctness and those for which reliable operation is always required. Specifically, all fetched instructions need to be decoded precisely and their target and source register indices need to be identified without error. However, the content of those registers may be approximate and the functional units operating on the data may operate approximately. Similarly, memory addresses must be error-free, but the data retrieved from the memory system can be incorrect when that data is marked as approximate. Consequently, we divide the microarchitecture into two distinct planes: the *instruction control plane* and the *data movement/processing plane*. As depicted in Figure 1, the data movement/processing plane comprises the register file, data caches, load store queue, functional units, and bypass network. The instruction control plane comprises the units that fetch, decode, and perform necessary bookkeeping for in-flight instructions. The instruction control plane is kept precise, while the data movement/processing plane can be approximate for approximate instructions. Since the data plane needs to behave precisely or approximately depending on the instruction being carried out, the microarchitecture needs to do some bookkeeping to determine when a structure can behave approximately.

This paper explores voltage reduction as a technique for saving energy. (Other techniques, such as aggressive timing closure or reducing data width, are orthogonal.) Each frequency level (f_{max}) is associated with a minimum voltage (V_{min}) and lowering the voltage beyond that may cause timing errors. Lowering the voltage reduces energy consumption quadratically when the frequency is kept constant. However, we cannot lower the voltage of the whole processor as this would cause errors in structures that need to behave precisely. This leads to the core question of how to provide precise behavior in the microarchitecture.

One way to provide precise behavior, which we explore in this paper, is to run critical structures at a safe voltage. Alternatively, error correction mechanisms could be used for critical structures and disabled for the data movement/processing plane while executing approximate instructions. This way, the penalty of error checking would not be incurred for approximate operations. However, if V_{dd} were low enough to make approximation pay off, many expensive recoveries would be required during precise operations. For this reason, we propose using two voltage lines: one for precise operation and one for approximate operation. Below we describe two alternative designs for a dual-voltage microarchitecture.

Unchecked Dual-Voltage Design Our dual-voltage microarchitecture, called Truffle, needs to guarantee that (1) the instruction control remains precise at all times, and (2) the data processing plane structures lower precision only when processing approximate instructions. Truffle has two voltages: a nominal, reliable level, referred to as $V_{dd}H$, and a lower level, called $V_{dd}L$, which may lead to timing errors. All structures in the instruction control plane are supplied $V_{dd}H$, and, depending on the instruction being executed, the structures in the data processing plane are dynamically supplied $V_{dd}H$ or $V_{dd}L$. The detailed design of a dual-voltage data processing plane, including the register file, data cache, functional units, and bypass network, is discussed in the next section.

Checked Dual Voltage Design The energy savings potential of Truffle is limited by the proportion of power used by structures that operate precisely at $V_{dd}H$. Therefore, reducing the energy consumption of precise operations will lead to higher relative impact of using approximation. This leads to another possible design point, which lowers the voltage of the instruction control plane beyond $V_{dd}H$ but not as aggressively as $V_{dd}L$ and employs error correction to guarantee correct behavior using an approach similar to Razor [10]. We refer to this conservative level of voltage as $V_{dd}L_{high}$. The data plane also operates at $V_{dd}L_{high}$ when running the precise instructions and $V_{dd}L$ when running the approximate instructions. Since the instruction control plane operates at the reliable voltage level, the components in this plane need to be checked and corrected in the case of any errors. The same checking applies to the data movement/processing plane while running precise instructions. While this is an interesting design point, we focus on the unchecked dual-voltage design due to its lower complexity.

Selecting $V_{dd}L$ In the simplest case, $V_{dd}L$ can be set statically at chip manufacture and test time. However, the accuracy of approximate operations depends directly on this value. Therefore, a natural option is to allow $V_{dd}L$ to be set dynamically depending on the QoS expectations of the application. Fine-grained voltage adjustments can be performed after fabrication using off-chip voltage regulators as in the Intel SCC [13] or by on-chip voltage regulators as proposed by Kim et al. [15]. Off-chip voltage regulators have a high latency while on-chip voltage regulators provide lower-latency, fine-grained voltage scaling. Depending on the latency requirement of the application and the type of regulator available, $V_{dd}L$ can be selected at deployment time or during execution. Future work should explore software-engineering tools that assist in selecting per-application voltage levels.

4. Truffle: A Dual-Voltage Microarchitecture for Disciplined Approximation

This section describes Truffle in detail. We start with the design of a dual-voltage SRAM structure, which is an essential building block for microarchitectural components such as register files and data caches. Next, we discuss the design of dual-voltage multiplexers and level shifters. We then address the design of structures in both in-order and OOO Truffle cores, emphasizing how the voltage is selected dynamically depending on the precision level of each instruction. Finally, we catalog the microarchitectural overheads imposed by Truffle's dual-voltage design.

4.1 Dual-Voltage SRAM Array

We propose dual-voltage SRAM arrays, or DV-SRAMs, which can hold precise and approximate data simultaneously. Like its single-voltage counterpart, a DV-SRAM array is composed of mats. A mat, as depicted in Figure 2, is a self-contained memory structure composed of four identical subarrays and associated predecoding logic that is shared among the subarrays. The data-in/-out and address lines typically enter the mat in the middle. The predecoded signals are forked off from the middle to the subarrays. Figure 2 also illustrates a circuit diagram for a single subarray, which is a two-dimensional array of single-bit SRAM cells arranged in rows and columns. The highlighted signals correspond to a read access, which determines the critical path of the array.

For any read or write to a DV-SRAM array, the address goes through the predecoding logic and produces the one-hot **`rowSelect`** signals along with the **`columnSelect`** signals for the column multiplexers. During read accesses, when a row is activated by its **`rowSelect`** signal, the value stored in each selected SRAM cell is transferred over two complementary **`bitline`** wires to the sense amplifiers. Meanwhile, the **`bitlines`** have been precharged to a certain V_{dd}. Each sense amplifier senses the resulting swing on the **`bitlines`** and generates the subarray output. The inputs and outputs of the sense amplifiers may be multiplexed depending on the array layout. The sense amplifiers drive the **`dataOut`** of the mat for a read access, while **`dataIn`** drives the **`bitlines`** during a write access.

To be able to store both approximate and precise data, we divide the data array logic into two parts: the indexing logic and the data storage/retrieval logic. To avoid potential corruption of precise data, the indexing logic *always* needs to be precise, even when manipulating approximate data. The indexing logic includes the address lines to the mats, the predecoding/decoding logic (row and column decoders), and the **`rowSelect`** and **`columnSelect`** drivers. The data storage/retrieval logic, in contrast, needs to alternate between precise and approximate mode. Data storage/retrieval includes the precharge/equalizing logic, SRAM cells, bitline multiplexers, sense amplifiers, **`dataOut`** drivers and multiplexers, and **`dataIn`** drivers. For approximate accesses, this set of logic blocks operates at $V_{dd}L$.

In each subarray of a mat, a row of bits (SRAM cells) is either at high voltage ($V_{dd}H$) or low voltage ($V_{dd}L$). The precision column in Figure 2, which is a single-bit column operating at $V_{dd}H$, stores the voltage state of each row. The precision column is composed of 8-transistor cells: 6-transistor SRAM cells each augmented by two PMOS transistors. In each row, the output of the precision column drives the power lines of the entire row. This makes it possible to *route only one power line* to the data rows as well as the prechargers and sense amplifiers. This way, the extra $V_{dd}L$ power line is only routed to the precision column in each subarray and is distributed to the rows through the precision cells, which significantly reduces the overhead of running two voltage lines.

For a read access, the **`bitlines`** need to be precharged to the same voltage level as the row being accessed. Similarly, the sense amplifiers need to operate at the same voltage level as the subarray row being accessed. A **`precision`** signal is added to the address bits and routed to all the mats. The **`precision`** signal presets the voltage levels of the precharge and equalizing logic and the sense amplifiers before the address is decoded and the **`rowSelect`** and **`columnSelect`** signals are generated. This voltage presetting ensures that the sense amplifiers are ready when the selected row puts its values on the **`bitlines`** during a read. Furthermore, during a write, the value in the precision column is set based on the **`precision`** signal.

Since the precision cell is selected at the same time as the data cells, the state stored in the precision column cannot be used to set the appropriate voltage levels in the sense amplifiers and prechargers. Therefore, the precision information needs to be determined *before* the read access starts in the subarrays. This is why our ISA extensions (Section 2) include a precision flag for each source operand: these flags, along with the precision levels of instructions themselves, are used to set the **`precision`** signal when accessing registers and caches.

4.2 Voltage Level Shifting and Multiplexing

Several structures in the Truffle microarchitecture must be able to deal with both precise and approximate data. For this reason, our design includes *dual-voltage multiplexers*, which can multiplex signals of differing voltage, and *voltage level shifters*, which enable data movement between the high- and low-voltage domains.

Figure 3 illustrates the transistor-level design of the single-bit, one-input dual-voltage multiplexers (DV-Mux) as well as the high-to-low (H2L) and low-to-high (L2H) level shifters. The select line of the multiplexer and its associated inverter operate at $V_{dd}H$, while the input data lines can swing from 0 V to either $V_{dd}L$ or $V_{dd}H$. The L2H level shifter is a conventional differential low-to-high level shifter and the H2L level shifter is constructed from two back-to-back inverters, one operating at $V_{dd}H$ and the other operating at $V_{dd}L$. In addition to the **`input`** signal, the level shifters take an extra input, **`precision`**, to identify the voltage level of **`input`** and disengage the level-shifting logic to prevent unnecessary power consumption. For example, in the L2H level shifter, when **`precision`** is 0, **`input`** is precise (0 V or $V_{dd}H$) and does not require any level shifting, (**`output`** ← **`input`**). However, when **`precision`** is 1, the level shifter is engaged and generates the **`output`** with the required voltage level.

4.3 Truffle's Microarchitectural Constructs

This section describes the Truffle pipeline stage-by-stage, considering both in-order and out-of-order implementations. The out-of-order design is based on the Alpha 21264 [3] and uses a tag-and-index register renaming approach. We highlight whether a pipeline stage belongs to the instruction control plane or the data movement/processing plane. Importantly, we discuss how the voltage is dynamically selected in the structures that support approximation.

Fetch (OOO/in-order) [Instruction Control Plane] The fetch stage belongs to the instruction control plane and is identical to a regular OOO/in-order fetch stage. All the components of this stage, including the branch predictor, instruction cache, and ITLB, are ordinary, single-voltage structures. Approximate instructions are fetched exactly the same way as precise instructions.

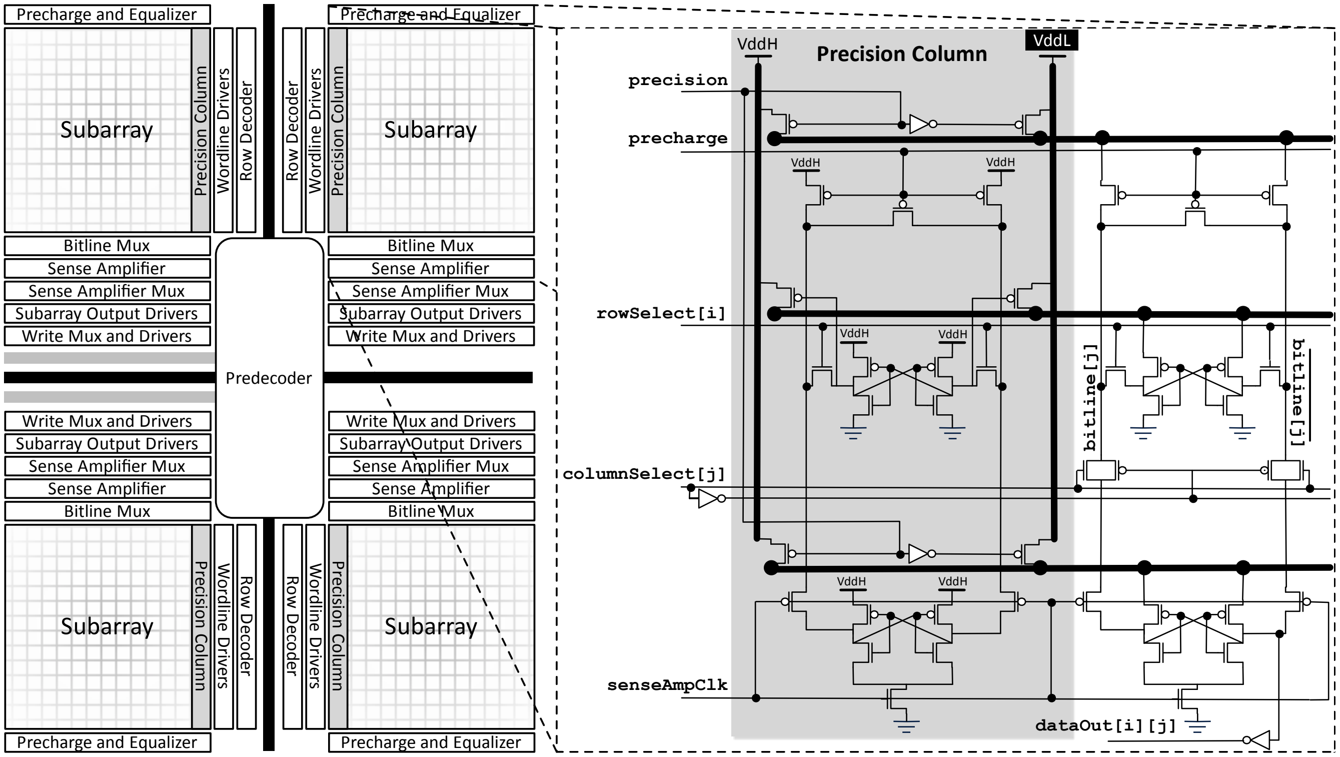


Figure 2. Dual-voltage mat, consisting of four identical dual-voltage subarrays, and partial transistor-level design of the subarrays and the *precision column*, which is shaded. The power lines during a read access are shown in bold.

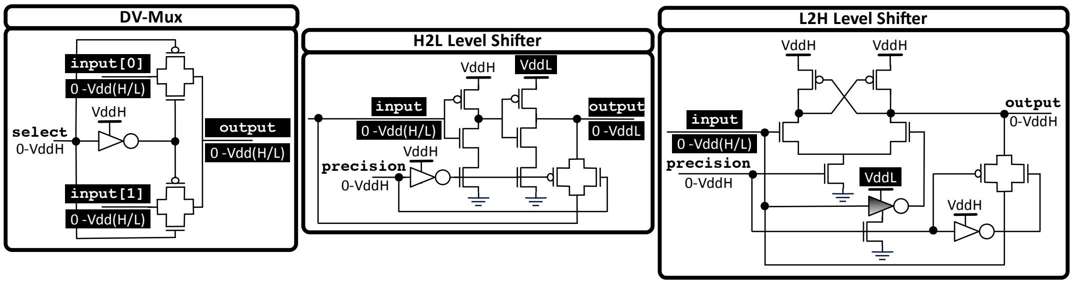


Figure 3. Transistor-level design of dual-voltage multiplexer (DV-Mux) and high-to-low (H2L) and low-to-high (L2H) level shifters.

Decode (OOO/in-order) [Instruction Control Plane] The instruction decoding logic needs to distinguish between approximate and precise instructions. The decode stage passes along one extra bit indicating the precision level of the decoded instruction.

In addition, based on the instruction, the decoder generates precision bits to accompany the indices for each register read or written. These register precision bits will be used when accessing the dual-voltage register file as discussed in Section 4.1. The precision levels of the source registers are extracted from the operand precision flags while the precision of the destination register corresponds to the precision of the instruction. For load and store instructions, the address computation must always be performed precisely, even when the data being loaded or stored is approximate. For approximate load and store instructions, the registers used for address calculation are always precise while the data register is always approximate. Recall that the microarchitecture does not check that precise registers are always used precisely and approximate registers are used approximately. This correspondence is enforced by the compiler and encoded in the instructions, simplifying the design and reducing the overheads of mixed-precision computation.

Rename (OOO) [Instruction Control Plane] For the OOO design, in which the register renaming logic generates the physical register indices/tags, the physical register tags are coupled with the register precision bits passed from the decode stage. The rest of the register renaming logic is the same as in the base design.

Issue (OOO) [Instruction Control Plane] The slots in the issue logic need to store the register index precision bits as well as the physical register indices. That is, each physical tag entry in an issue slot is extended by one bit. The issue slots also need to store the bit indicating the precision of the instruction. When an approximate load instruction is issued, it gets a slot in the load/store queue. The address entry of each slot in the load/store queue is extended by one extra bit indicating whether the access is approximate or precise. The precision bit is coupled with the address in order to control the precision level of the data cache access (as described above in Section 4.1).

Schedule (OOO) [Instruction Control Plane] As will be discussed in the execution stage, there are separate functional units for approximate and precise computation in Truffle. The approximate functional units act as *shadows* of their precise counterparts. The issue width of the processor is *not* extended due to the extra approximate functional units and no complexity is added to the scheduling logic to accommodate them. The coupled approximate/precise functional units appear to the scheduler as a single unit. For example, if an approximate floating-point operation is scheduled to issue, the precise and approximate FPUs are both considered busy. The bit indicating the precision level of the instruction is used to enable either the approximate or precise functional unit exclusively.

Register Read (OOO/in-order) [Data Movement/Processing Plane] The data movement/processing plane starts at the register read stage. As depicted in Figure 4, the pipeline registers are divided into two sets starting at this stage: one operating at $V_{dd}L$ (Approx Data Pipe Reg) and the other operating at $V_{dd}H$ (Precise Data Pipe Reg + Control). The approximate pipeline register holds approximate data. The precise pipeline register contains control information passed along from previous stages and precise program data. The outputs of the precise and approximate pipeline registers are multiplexed through a DV-Mux as needed.

The dual-voltage register file (physical in the OOO design and architectural in the in-order design) is made up of DV-SRAM arrays. Each register can be dynamically set as approximate or precise. While the precision levels of the general-purpose registers are determined dynamically, the pipeline registers are hardwired to their respective voltages to avoid voltage level changes when running approximate and precise instructions back-to-back.

The ISA allows precise instructions to use approximate registers as operands and vice-versa. To carry out such instructions with minimal error, the voltage levels of the operands need to be changed to match the voltage level of the operation. In the register read stage, the level shifters illustrated in Figure 4 convert the voltage level of values read from the register file. For example, if `reg1` is precise and used in an approximate operation, its voltage level is shifted from high to low through an H2L level shifter before being written to an approximate pipeline register. The precision level of `reg1`, denoted by `reg1Precision`, is passed to the level shifter to avoid unnecessary level shifting in the case that `reg1` is already approximate. Similarly, a low-to-high (L2H) level shifter is used to adjust the voltage level of values written to the precise pipeline register. Note that only *one* of the approximate or precise data pipeline registers is enabled based on the precision of the instruction.

Execute (OOO/in-order) [Data Movement/Processing Plane] As shown in Figure 4, all the functional units are duplicated in the execution stage. Half of them are hardwired to $V_{dd}H$, while the other half operate at $V_{dd}L$ as *shadows*. That is, the scheduler does not distinguish between a shadow approximate FU and its precise counterpart. Only the instruction precision bit controls whether the results are taken from the precise or approximate FU. Low-voltage functional units are connected to a low-voltage pipeline register. The outputs of the approximate FUs connect to the *same* broadcast network as their precise counterparts through a dual-voltage multiplexer driven by the approximate and precise pipeline register pair at the end of the execution stage.

The inputs of functional units may also be driven by the bypass network. Level shifters at the inputs of the functional units adjust the level of the broadcasted input data using the broadcasted precision bit. Only a single-bit precision signal is added to the broadcast network. Because the output of each FU pair is multiplexed, the extra FUs do not increase the size of the broadcast network beyond adding this single-bit precision line. While the data bypass network alternates between $V_{dd}L$ and $V_{dd}H$, the tag forwarding network in the OOO design always works at the high voltage level since it carries necessarily-precise register indexing information.

To avoid unnecessary dynamic power consumption in the functional units, the input to one functional unit is kept constant (by not writing to its input pipeline register) when its opposite-precision counterpart is being used.

An alternative design could use dual-voltage functional units and change the voltage depending on the instruction being executed. This would save area and static power but require a more complex scheduler design that can set the appropriate voltage level in the functional unit before delivering the operands to it. Since functional units can be large and draw a considerable amount of current while switching, a voltage-selecting transistor for a dual-voltage functional unit needs to be sized such that it can provide the required drive. Such transistors tend to consume significant power. Another possibility is lowering the FU voltage based on phase prediction. When the application enters an approximate phase, the voltage level of the functional unit is lowered. Phase prediction requires extra power and complicates the microarchitecture design. The main objective in the design of Truffle is to keep the microarchitectural changes to a minimum and avoid complexity in the instruction control and bookkeeping. Furthermore, since programs consist of a mixture of precise and approximate instructions, it is not obvious that phase-based voltage level adjustments can provide benefit that compensates for the phase prediction overhead. Additionally, static partitioning can help tolerate process variation by using defective functional units for approximate computations.

Memory (OOO/in-order) [Data Movement/Processing Plane] As previously discussed, the address field in each slot of the load/store queue is extended by one bit that indicates whether the address location is precise or approximate. The data array portion of the data cache is a DV-SRAM array, while the tag array is an ordinary single-voltage SRAM structure. The approximation granularity in the data cache is a cache line: one extra bit is stored in the tag array which identifies the precision level of each cache line. The extra bit in the load/store queue is used as the precision signal when accessing the DV-SRAM data array. The miss buffer, fill buffer, prefetch buffer, and write back buffers all operate at $V_{dd}H$. The DTLB also requires no changes.

The precision level of a cache line is determined by the load or store instruction that fills the line. If an approximate access misses in the cache, the line is fetched as approximate. Similarly, a precise miss fills the cache line as precise. Subsequent write hits also affect line precision: when a store instruction modifies the data in a cache line, it also modifies the line's precision level (see Section 2).

This paper focuses on the Truffle core and does not present a detailed design for approximation-aware lower levels of cache or main memory. The L1 cache described here could work with an unmodified, fully-precise L2 cache, but a number of options are available for the design of an approximation-aware L2. If the L2 has the same line size as the L1, then an identical strategy can be applied. However, L2 lines are often larger than L1 lines. In that case, one option is to control L2 precision at a sub-line granularity: if the L2 line size is n times the L1 line size, then the L2 has n precision columns. An alternative design could pair the lower-level caches with main memory, setting the precision of L2 lines based on the explicitly-controlled main-memory precision levels. These non-core design decisions are orthogonal to the core Truffle design and are an avenue for future work.

Write Back (OOO/in-order) [Data Movement/Processing Plane] The write back value comes from the approximate or precise

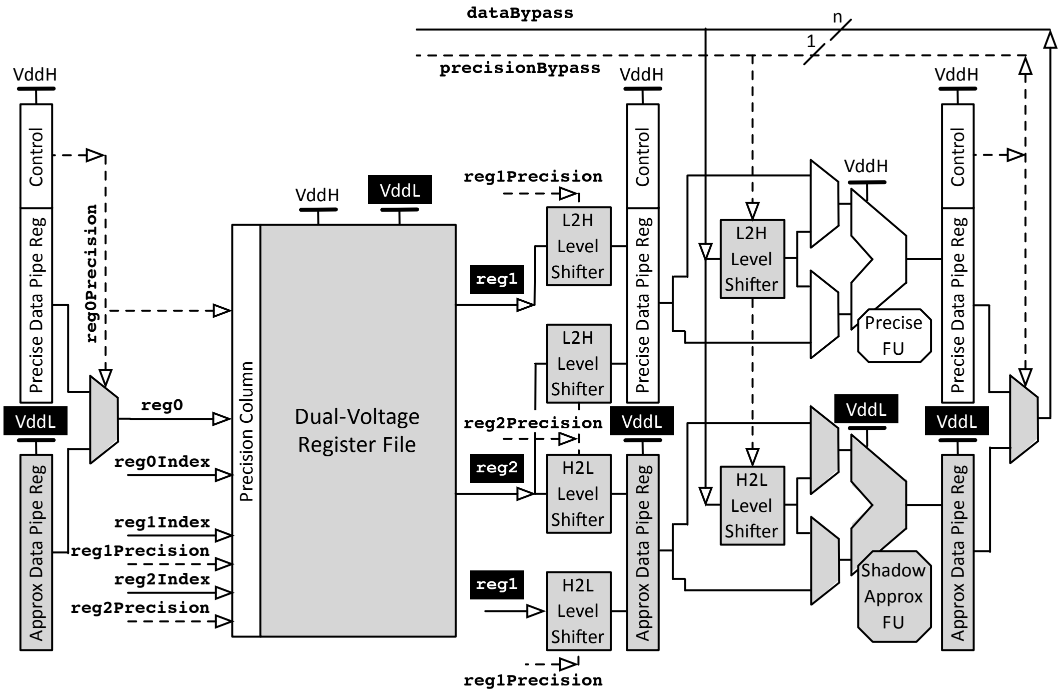


Figure 4. The register and execution stages in the Truffle pipeline along with the bypass network. The DV-Muxes and other approximation-aware units are shaded. The single-bit precision signals in the bypass network and in each stage are dashed lines.

pipeline register. As shown in Figure 4, the dual-voltage multiplexer driving **reg0** forwards the write back value to the data bypass network. The precision bit accompanying the write back value (**reg0Precision**) is also forwarded over the bypass network's precision line.

Commit (OOO) [Instruction Control Plane] The commit stage does not require any special changes and the reorder buffer slots do not need to store any extra information for the approximate instructions. The only consideration is that, during rollbacks due to mispredictions or exceptions, the precision state of the registers (one bit per register) needs to be restored. Another option is to restore all the registers as precise, ignoring the precision level of the registers during rollback.

4.4 Microarchitectural Overheads in Truffle

In this section, we briefly consider the microarchitectural overheads imposed by routing two power lines to structures in Truffle's data movement/processing plane. First, the data part of the pipeline registers is duplicated to store approximate data. An extra level of dual-voltage multiplexing is added after these pipeline registers to provide the data with the correct voltage for the next stage. The DV-SRAM arrays, including the register file and data array in the data cache, store one precision bit per row. In addition, a one-bit precision signal is added to each read/write port of the DV-SRAM array and routed to each subarray along with the $V_{dd}L$ power line. The tag for each data cache line is also augmented by a bit storing the precision state of the line. However, the tag array itself is an ordinary single-voltage SRAM structure. To adjust the voltage level of the operands accessed from the dual-voltage register file, one set of H2L and one set of L2H level shifters is added for each read port of the register file. Similarly, in the execution stage, one set of each level shifter type is added per operand from the data bypass network. The data bypass network is also augmented with a single-bit precision signal per operand. In addition, each entry in the issue queue is extended by one bit preserving the precision level of the instruction. Each physical tag entry in an issue slot is also extended by one precision bit. Similarly, the slots in the load/store queue are augmented by one extra bit indicating whether the access is approximate or precise. The pipeline registers also need to store the precision level of the operands and instructions.

5. Experimental Results

Our goals in evaluating Truffle are to determine the energy savings brought by disciplined approximation, characterize where the energy goes, and understand the QoS implications for software.

5.1 Evaluation Setup

We extended CACTI [21] to model the dual-voltage SRAM structure we propose and incorporated the resulting models into McPAT [17]. We modeled Truffle at the 65 nm technology node in the context of both in-order and out-of-order (based on the 21264 [3]) designs. Table 2 shows the detailed microarchitectural parameters.

Disciplined approximate computation represents a trade-off between energy consumption and application output quality. Therefore, we evaluate each benchmark for two criteria: energy savings and sensitivity to error. For the former, we collect statistics from the benchmarks' execution as parameters to the McPAT-based power model; for the latter, we inject errors into the execution and measure the consequent degradation in output quality.

For both tasks, we use a source-to-source transformation that instruments the program for statistics collection and error injection. Statistics collected for power modeling include variable, field, and array accesses, basic blocks (for branches), and arithmetic and logical operators. A cache simulator is used to distinguish cache hits and misses; the register file is simulated as a small, fully-associative cache. For error injection, each potential injection point (approximate operations and approximate memory accesses) is in-

Table 2. Microarchitectural parameters.

Parameter	OOO Truffle	In-order Truffle
Fetch/Decode Width	4/4	2/2
Issue/Commit Width	6/4	—/—
INT ALUs/FPUs	4/2	1/1
INT Mult/Div Units	1	1
Approximate INT ALUs/FPUs	4/2	1/1
Approximate INT Mult/Div Units	1	1
INT/FP Issue Window Size	20/15	—
ROB Entries	80	—
INT/FP Architectural Registers	32/32	32/32
INT/FP Physical Registers	80/72	—
Load/Store Queue Size	32/32	—
ITLB	128	64
I Cache Size	64 Kbyte	16 Kbyte
Line Width/Associativity	32/2	16/4
DTLB	128	64
D Cache Size	64 Kbyte	32 Kbyte
Line Width/Associativity	16/2	16/4
Branch Predictor	Tournament	Tournament

Table 3. List of benchmarks.

Application	Description	Type
fft	SciMark2 benchmark: scientific kernels	FP
sor		FP
mc		FP
smm		FP
lu		FP
zxing	Bar code decoder for mobile phones	FP/integer
jmeint	jMonkeyEngine game framework: triangle intersection kernel	FP
imagefill	ImageJ raster image processing application: flood-filling kernel	integer
raytracer	3D image renderer	FP

tercepted; each bit in the resulting value is flipped according to a per-component probability before being returned to the program.

Benchmarks We examine nine benchmark programs written in the EnerJ language, which is an extension to Java [23]. The applications are the same programs that were evaluated in [23]: existing Java programs hand-annotated with approximate type qualifiers that distinguish their approximate parts. Five of the benchmarks come from the SciMark2 suite. ZXing is a multi-format bar code recognizer developed for Android smartphones. jMonkeyEngine is a game development engine; we examine the framework's triangle-intersection algorithm used for collision detection. ImageJ is a library and application for raster image manipulation; we examine its flood-filler algorithm. Finally, we examine a simple 3D raytracer.

For each program, an application-specific *quality-of-service metric* is defined in order to quantify the loss in output quality caused by hardware approximation. For most of the applications, the metric is the root-mean-square error of the output vector, matrix, or pixel array. For jmeint, the jMonkeyEngine triangle-intersection algorithm, the metric is the proportion of incorrect intersection decisions. Similarly, for the zxing bar code recognizer, it is the proportion of unsuccessful decodings of a sample QR code image.

5.2 Unchecked Truffle Microarchitecture

Figure 5 presents the energy savings achieved in the core and L1 cache for the unchecked dual-voltage Truffle microarchitecture in both OOO and in-order configurations. In both designs, $V_{dd}H = 1.5$ V and $V_{dd}L$ takes values that are 50%, 62.5%, 75%, and 87.5% of $V_{dd}H$. The frequency is set constant at 1666 MHz. The Truffle

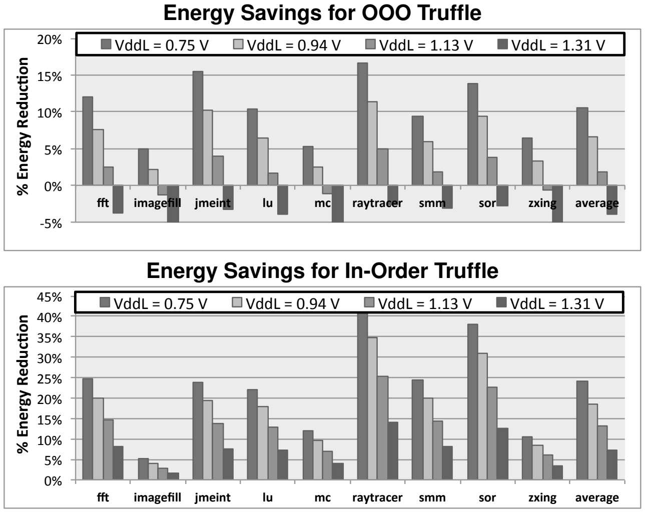


Figure 5. Percent energy reduction with unchecked OOO and in-order Truffle designs for various $V_{dd}L$ voltages.

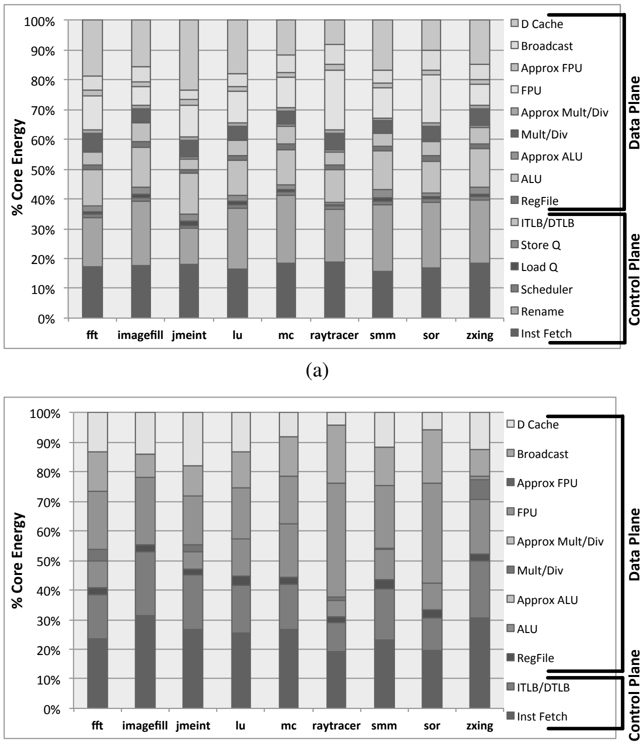
cores include DV-SRAM arrays and extra approximate functional units. The baseline for the reported energy savings is the same core operating at the reliable voltage level of 1.5 V and 1666 MHz *without* the extra functional units or dual-voltage register files and data cache. Our model assumes that Truffle's microarchitectural additions do not prolong the critical path of the base design.

Depending on the configuration, voltage, and application, Truffle ranges from increasing energy by 5% to saving 43%. For the in-order configuration, all voltage levels lead to energy savings; the OOO design shows energy savings when $V_{dd}L$ is less than 75% of $V_{dd}H$. Our results suggest Truffle exhibits a "break even" point at which its energy savings outweigh its overheads.

The difference between the energy savings in OOO and in-order Truffle cores stems from the fact that the instruction control plane in the OOO core accounts for a much larger portion of total energy than in the in-order core. Recall that the instruction control plane in the OOO core includes instruction fetch and decode, register renaming, instruction issue and scheduling, load and store queues, and DTLB/ITLB, whereas in the in-order setting it includes only instruction fetch and decode and the TLBs. Since approximation helps reduce energy consumption in the data movement/processing plane only, the impact of Truffle in in-order cores is much higher. Furthermore, the OOO Truffle core is an aggressive four-wide multiple-issue processor whereas the in-order Truffle core is two-wide. Anything that can reduce the energy consumption of the instruction control plane indirectly helps increase Truffle's impact.

Figure 6 depicts the energy breakdown between different microarchitectural components in the OOO and in-order Truffle cores when $V_{dd}L = V_{dd}H$ (i.e., with fully-precise computation). Among the benchmarks, imagefill shows similar benefits for both designs. For this benchmark, 42% and 47% of the energy is consumed in the data movement/processing plane of the OOO and in-order Truffle cores, respectively. On the other hand, raytracer shows the largest difference in energy reduction between the two designs; here, the data movement/processing plane consumes 71% of the energy in the OOO core but just 50% in the in-order core. In summary, the simpler the instruction control plane in the processor, the higher the potential for energy savings with Truffle.

In addition to the design style of the Truffle core, the energy savings are dependent on the proportion of approximate computation in the execution. Figure 7 shows the percentage of approximate dynamic instructions along with the percentage of approxi-



(a)

(b)

Figure 6. Percent energy consumed by different microarchitectural components in the (a) OOO and (b) in-order Truffle.

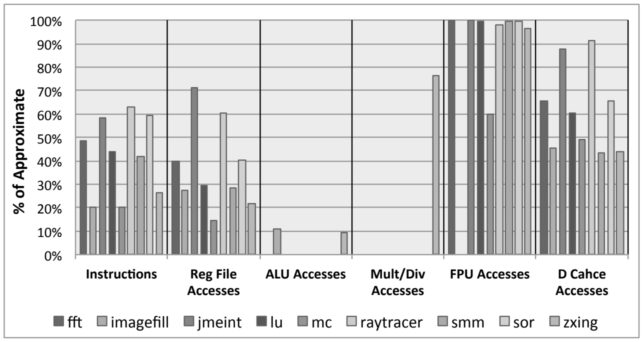


Figure 7. Percentage of approximate events.

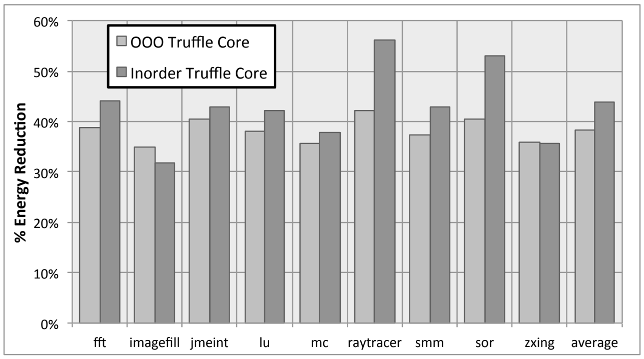


Figure 8. Percent energy reduction *potential* for checked in-order and OOO Truffle designs with $V_{dd}L = 0.75$ V.

mate ALU, multiply/divide, and floating point operations as well as the percentage of approximate data cache accesses. Among the benchmarks, imagefill has the lowest percentage of approximate instructions (20%) and no approximate floating point or multiplication operations—only 11% of its integer operations are approximate. As a fully integer application, it exhibits no opportunity for floating-point approximation, and its approximate integer operations are dwarfed by precise control-flow operations. imagefill also has a low ratio of approximate data cache accesses, 45%. These characteristics result in low potential for Truffle, about 5% energy savings for $V_{dd}L = 0.75$ V. Conversely, raytracer shows the highest ratio of approximate instructions in the group. Nearly all (98%) of its floating point operations are approximate. In addition, raytracer has the highest ratio of approximate data cache accesses in the benchmark set, 91%, which makes it benefit the most from Truffle. The high rate of floating-point approximation is characteristic of the FP-dominated benchmarks we examined: for many applications, more than 90% of the FP operations are approximate. This is commensurate with the inherently approximate nature of FP representations. Furthermore, for many benchmarks, FP data constitutes the application's error-resilient data plane while integers dominate its error-sensitive control plane.

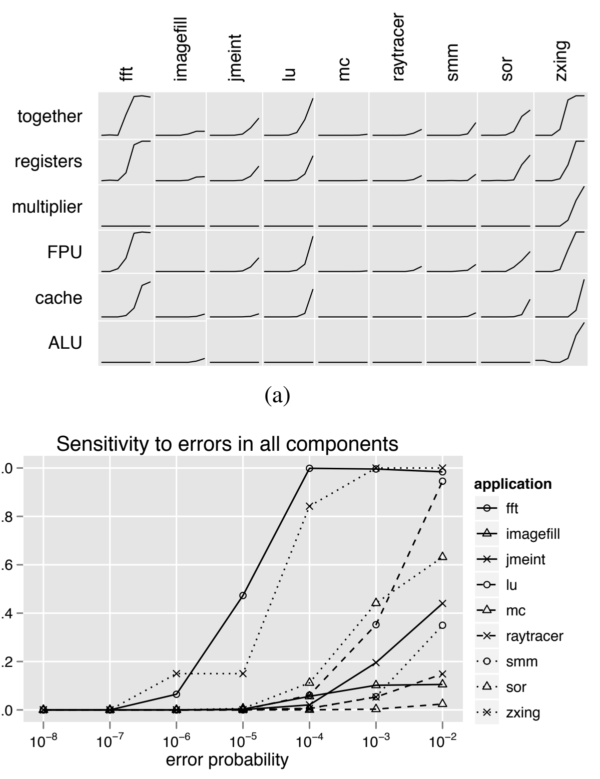
These results show that, as the proportion of approximate computation increases, the energy reductions from the Truffle microarchitecture also increase. Furthermore, some applications leave certain microarchitectural components unexercised, suggesting that higher error rates may be tolerable in those components. For example, none of the benchmarks except imagefill exercise the approximate integer ALU, and the approximate multiply/divide unit is not exercised at all. As a result, higher error rates in those components may be tolerable. The results also support the utility of application-specific $V_{dd}L$ settings, since each of the benchmarks exercise each approximate component differently.

Overall, these results show that disciplined approximation has great potential to enable low-power microarchitectures. Also, as expected, the simpler the microarchitecture, the higher the energy savings potential.

Overheads We modified McPAT and CACTI to model the overheads of Truffle as described in Section 4.4. The energy across all the benchmarks increases by at most 2% when the applications are compiled with no approximate instructions. The approximate functional units are power-gated when there are no approximate instructions in flight. The energy increase is due to the extra precision state per cache line and per register along with other microarchitectural changes. CACTI models show an increase of 3% in register file area due to the precision column and a 1% increase in the area of the level-1 data cache. The extra approximate functional units also contribute to the area overhead of Truffle.

5.3 Opportunities in a Checked Design

As discussed above, reducing energy consumption of the instruction control plane (and the energy used in *precise* instructions) can increase the overall impact of Truffle. Section 3 outlines a design that uses a sub-critical voltage and error detection/correction for the microarchitectural structures that need to behave precisely. We now present a simple limit study of the potential of such a design. Figure 8 presents the energy savings potential when the voltage level of the instruction control plane is reduced to 1.2 V, beyond the reliable voltage level of $V_{min} = 1.5$ V, and $V_{dd}L = 0.75$ V. The results show only the ideal case in which there is no penalty associated with error checking and correction in the precise computation. As illustrated, the gap in energy savings potential between the OOO



(a)

(b)

Figure 9. Application sensitivity to circuit-level errors. Each cell in (a) has the same axes as (b): application QoS degradation is related to architectural error probability (on a log scale). The grid (a) shows applications' sensitivity to errors in each component in isolation; the row labeled "together" corresponds to experiments in which the error probability for all components is the same. The plot (b) shows these "together" configurations in more detail. The output error is averaged over 20 replications.

and in-order designs is significantly reduced. In one case, imagefill, the checked OOO Truffle core shows higher potential compared to the checked in-order Truffle core. In this benchmark, the energy consumption of the instruction control plane is more dominant in the OOO design and thus lower voltage for that plane is more effective than in the in-order design. Note that in an actual design, energy savings will be restricted by the error rates in the instruction control plane and the rate at which the precise instructions fail, triggering error recovery. The overhead of the error-checking structures will further limit the savings.

5.4 Error Propagation from Circuits to Applications

We now present a study of application QoS degradation as we inject errors in each of the microarchitectural structures that support approximate behavior. The actual pattern of errors caused by voltage reduction is highly design-dependent. Modeling the error distributions of approximate hardware is likely to involve guesswork; the most convincing evaluation of error rates would come from experiments with real Truffle hardware. For the present evaluation, we thoroughly explore a space of error rates in order to characterize the range of possibilities for the impact of approximation.

Figure 9 shows each benchmark's sensitivity to circuit-level errors in each microarchitectural component. Some applications are significantly sensitive to error injection in most components (fft, for example); others show very little degradation (imagefill, raytracer, mc, smm). Errors in some components tend to cause more application-level errors than others—for example, errors in the integer functional units (ALU and multiplier) only cause output degradation in the benchmarks with significant approximate integer computation (imagefill and zxing).

The variability in application sensitivity highlights again the utility of using a tunable $V_{dd}L$ to customize the architecture's error rate on a per-application basis (see Section 3). Most applications exhibit a critical error rate at which the application's output quality drops precipitously—for example, in Figure 9(b), fft exhibits low output error when all components have error probability 10^{-6} but significant degradation occurs at probability 10^{-5}. A software-controllable $V_{dd}L$ could allow each application to run at its lowest allowable power while maintaining acceptable output quality.

In general, the benchmarks do not exhibit drastically different sensitivities to errors in different components. A given benchmark that is sensitive to errors in the register file, for example, is also likely to be sensitive to errors in the cache and functional units.

6. Related Work

A significant amount of prior work has proposed hardware that compromises on execution correctness for benefits in performance, energy consumption, and yield. ERSA proposes collaboration between discrete reliable and unreliable cores for executing error-resilient applications [16]. Stochastic processors encapsulate another proposal for variable-accuracy functional units [22]. Probabilistic CMOS (PCMOS) proposes to use the probability of low-voltage transistor switching as a source of randomness for special randomized algorithms [5]. Finally, algorithmic noise-tolerance (ANT) proposes approximation in the context of digital signal processing [12]. Our proposed dual-voltage design, in contrast, supports fine-grained, single-core approximation that leverages language support for explicit approximation in general-purpose applications. It does not require manual offloading of code to co-processors and permits fully-precise execution on the same core as low-power approximate instructions. Truffle extends general-purpose CPUs; it is not a special-purpose coprocessor.

Relax is a compiler/architecture system for suppressing hardware fault recovery in certain regions of code, exposing these errors to the application [9]. A Truffle-like architecture supports approximation at a single-instruction granularity, exposes approximation in storage elements, and guarantees precise control flow even when executing approximate code. In addition, Truffle goes further and elides fault *detection* as well as recovery where it is not needed.

Razor and related techniques also use voltage underscaling for energy reduction but use error recovery to hide errors from the application [10, 14]. Disciplined approximate computation can enable energy savings beyond those allowed by correctness-preserving optimizations.

Broadly, the key difference between Truffle and prior work is that Truffle was co-designed with language support. Specifically, relying on *disciplined approximation* with strong static guarantees offered by the compiler and language features enables an efficient and simple design. Static guarantees also lead to strong safety properties that significantly improve programmability.

The error-tolerant property of certain applications is supported by a number of surveys of application-level sensitivity to circuit-level errors [8, 18, 27]. Truffle is a microarchitectural technique for exploiting this application property to achieve energy savings.

Dual-voltage designs are not the only way to implement low-power approximate computation. Fuzzy memoization [2] and bit-width reduction [26], for example, are orthogonal techniques for approximating floating-point operations. Imprecise integer logic blocks have also been designed [20]. An approximation-aware processor could combine dual-voltage design with these other techniques.

Previous work has also explored dual-V_{dd} designs for power optimization in fully-precise computers [6, 29]. Truffle's instruction-controlled voltage changes make it fundamentally different from these previous techniques.

Truffle resembles architectures that incorporate information flow tracking for security [7, 24, 25]. In that work, the hardware enforces information flow invariants dynamically based on tags provided by the application or operating system. With Truffle, the *compiler* provides the information flow invariant, freeing the architecture from costly dynamic checking.

7. Conclusion

Disciplined approximate programming is an effective and usable technique for trading off superfluous correctness guarantees for energy savings. Dual-voltage microarchitectures can realize these energy savings by providing both approximate and precise computation to be controlled at a fine grain by the compiler. We propose an ISA that simplifies the hardware by relying on the compiler to provide certain invariants statically, eliminating the need for checking or recovery at run time. We describe a high-level microarchitecture that supports interleaved high- and low-voltage operations and a detailed design for a dual-voltage SRAM array that implements approximation-aware caches and registers. We model the power of our proposed dual-voltage microarchitecture and evaluate its energy consumption in the context of a variety of error-tolerant benchmark applications. Experimental results show energy savings up to 43%; under reasonable assumptions, these benchmarks exhibit low or negligible degradation in output quality.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. We also thank Sied Mehdi Fakhraie, Jacob Nelson, Behnam Robatmili, and the members of the Sampa group for their feedback on the manuscript. This work was supported in part by NSF grant CCF-1016495, a Sloan Research Fellowship, and gifts from Microsoft and Google.

References

[1] *Alpha Architecture Handbook, Version 3*. Digital Equipment Corporation, 1996.

[2] C. Alvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.*, 54(7), 2005.

[3] S. Y. Borkar and A. A. Chien. The Aplha 21264 Microprocessor. *MICRO*, 1999.

[4] S. Y. Borkar and A. A. Chien. The future of microprocessors. *CACM*, 54, May 2011.

[5] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee. Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology. In *DATE*, 2006.

[6] C. Chen, A. Srivastava, and M. Sarrafzadeh. On gate level power optimization using dual-supply voltages. *IEEE Trans. VLSI Syst.*, 9, 2001.

[7] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *ISCA*, 2007.

[8] M. de Kruijf and K. Sankaralingam. Exploring the synergy of emerging workloads and silicon reliability trends. In *SELSE*, 2009.

[9] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *ISCA*, 2010.

[10] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO*, 2003.

[11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[12] R. Hegde and N. R. Shanbhag. Energy-efficient signal processing via algorithmic noise-tolerance. In *ISLPED*, 1999.

[13] J. Howard et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC*, 2010.

[14] A. Kahng, S. Kang, R. Kumar, and J. Sartori. Designing a processor from the ground up to allow voltage/reliability tradeoffs. In *HPCA*, 2010.

[15] W. Kim, D. Brooks, and G.-Y. Wei. A fully-integrated 3-level DC/DC converter for nanosecond-scale DVS with fast shunt regulation. In *ISSCC*, 2011.

[16] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra. ERSA: Error resilient system architecture for probabilistic applications. In *DATE*, 2010.

[17] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[18] X. Li and D. Yeung. Exploiting soft computing for increased fault tolerance. In *ASGI*, 2006.

[19] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving refresh-power in mobile devices through critical data partitioning. In *ASPLOS*, 2011.

[20] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas. Bio-inspired imprecise computational blocks for efficient VLSI implementation of soft-computing applications. *Trans. Cir. Sys. Part I*, 57, 2010.

[21] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *MICRO*, 2007.

[22] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones. Scalable stochastic processors. In *DATE*, 2010.

[23] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[24] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.

[25] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *ASPLOS*, 2009.

[26] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Trans. VLSI Syst.*, 8(3), 2000.

[27] V. Wong and M. Horowitz. Soft error resilience of probabilistic inference applications. In *SELSE*, 2006.

[28] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *MICRO*, 2005.

[29] C. Yeh, Y.-S. Kang, S.-J. Shieh, and J.-S. Wang. Layout techniques supporting the use of dual supply voltages for cell-based designs. In *DAC*, 1999.

[30] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *ASPLOS*, 2002.