AxMemo: Hardware-Compiler Co-Design for Approximate Code Memoization

Zhenhong Liu Amir Yazdanbakhsh[†*] Dong Kai Wang Hadi Esmaeilzadeh[‡] Nam Sung Kim

University of Illinois Urbana-Champaign [†]Google Brain [‡]University of California, San Diego

zliu118@illinois.edu ayazdan@google.com dwang47@illinois.edu hadi@eng.ucsd.edu nskim@illinois.edu

ABSTRACT

Historically, continuous improvements in general-purpose processors have fueled the economic success and growth of the IT industry. However, the diminishing benefits from transistor scaling and conventional optimization techniques necessitates moving beyond common practices. Approximate computing is one such unconventional technique that has shown promise in pushing the boundaries of general-purpose processing. This paper sets out to employ approximation for processors that are commonly used in cyber-physical domains and may become building blocks of Internet of Things. To this end, we propose AxMemo to exploit the computation redundancy that stems from data similarity in the inputs of code blocks. Such input behavior is prevalent in cyber-physical systems as they deal with real-world data that naturally harbors redundancy. Therefore, in contrast to existing memoization techniques that replace costly floating-point arithmetic operations with limited number of inputs, AxMemo focuses on memoizing blocks of code with potentially many inputs. As such, AxMemo aims to replace long sequences of instructions with a few hash and lookup operations. By reducing the number of dynamic instructions, AxMemo alleviates the von Neumann and execution overheads of passing instructions through the processor pipeline altogether. The challenge AxMemo facing is to provide low-cost hashing mechanisms that can generate rather unique signature for each multi-input combination. To address this challenge, we develop a novel use of Cyclic Redundancy Checking (CRC) to hash the inputs. To increase lookup table hit rate, AxMemo employs a two-level memoization lookup, which utilizes small dedicated SRAM and spare storage in the last level cache. These solutions enable AxMemo to efficiently memoize relatively large code regions with variable input sizes and types using the same underlying hardware. Our experiment shows that AxMemo offers 2.64× speedup and 2.58× energy reduction with mere 0.2% of quality loss averaged across ten benchmarks. These benefits come with an area overhead of just 2.1%.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322215

CCS CONCEPTS

• **Computer systems organization** → *Embedded hardware*;

KEYWORDS

Memoization; Approximate Computing; Hardware-Software Co-Design

ACM Reference Format:
Zhenhong Liu, Amir Yazdanbakhsh, Dong Kai Wang, Hadi Esmaeilzadeh, and Nam Sung Kim. 2019. AxMemo: Hardware-Compiler Co-Design for Approximate Code Memoization . In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3307650.3322215

1 INTRODUCTION

The economic cycle of the IT industry has long depended on delivering new capabilities in both devices and systems. The backbone of this cycle has been the continuous and unwavering improvements in general-purpose processors. As we enter the dark silicon era [6], the demising benefits from transistor scaling and conventional optimization technique can curtail this fruitful and innovative cycle. To avert such a scenario, the research community is exploring various avenues for continued improvements in different domains. Such improvements are more timely in the cyber-physical domain of applications as their demand for computing is increasing while the efficiency of microprocessors stagnate. Approximate computing is appropriate in this domain since the processing task involves noisy data and the output is either not necessarily unique or fully digital.

With the prevalence of low-power application scenarios such as Internet of Things (IoTs), improving mobile processor performance and energy efficiency is imperative. Modern CPUs spend rather large fraction of their time and energy in fetching, decoding and scheduling operations rather than executing them. Even for a double-precision fused multiply-add instruction, the energy spent on actual computation (execution unit) can be as low as 3% of the total energy of the instruction, from fetch to commit [9]. Proving approximation mechanisms that can eliminate the von Neumann overhead off the microprocessors can yield significant benefits.

One of the effective techniques for improving the energy efficiency is memoization. Memoization replaces operations with a series of lookup operations to previously

*This work has been done when the author was a PhD student at Georgia Institute of Technology.

recorded results table. This technique exploits the computation redundancy in the code for delivering higher speedup and energy efficiency. During the execution of an application, computation blocks may take the exact same inputs and generate the same outputs as previous instances of the computation. Such redundancy is because of either repetitive input patterns or the nature of the algorithm. Exact memoization has been explored at fine-grained instructions level [5, 28], a more coarse-grained function level [20, 34], and at task level granularity [3]. Except for the last work, the rest of these inspiring techniques do not exploit approximation. The technique in [3] is a pure software technique and does not explore the benefits of hardware support for hashing or reuse of the spare cache space for memoization. This paper is set out to bridge this gap by providing the hardware support for approximate memoization of relatively large blocks of code with multiple inputs/outputs. In order to replace a large code blocks with memoization (lookups) while preserving generality across various applications, we face two major challenges: (1) the hardware needs to handle a variety of input types and support memoization for different number of inputs in various applications, and (2) the memoization needs to maintain a high hit rate for the lookup operations in order to be beneficial, while with more inputs are added, the probability of having an exact match decreases. To address these challenges, AxMemo makes the following contributions:

(1) We develop a novel use of Cyclic Redundancy Checking (CRC) to hash the varying number of inputs using the same hardware unit.
(2) To increase hit rate, AxMemo employees a simple two-level memoization lookup, which utilizes a small dedicated SRAM and the spare storage in the last level caches.
(3) Evaluations with ten different benchmarks from various domains show that AxMemo provides 2.64× average speedup and 2.58× average energy reduction. These benefits come at the cost of 0.2% of average quality loss and merely 2.1% area overhead.

Building upon these contributions, we show that AxMemo efficiently memoizes relatively large code regions with variable input sizes and types using the same hardware mechanisms.

2 AXMEMO OVERVIEW

Before describing the memoization scheme of AxMemo, it is worth mentioning that AxMemo can be used only to memoize computation blocks equivalent to code sections that: 1) always produce the same results or return values given the same inputs, and 2) do not have any observable side effects, such as writing to a file. Computation blocks not satisfying either of these properties are ineligible for memoization.

Given a valid computation block, we transform the block into a branch structure. As shown in Fig. 1, we first look up the LUT using the inputs of the original computation block. If more than one computation blocks are memoized, we need

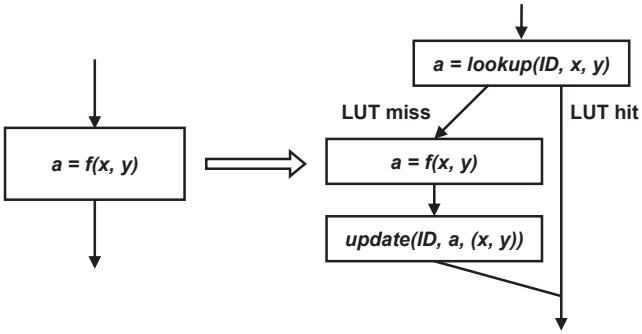


Figure 1: Control flow transformation of AxMemo. ID is the lookup table (LUT) ID.

to have multiple LUTs. To distinguish multiple LUTs, each LUT has a LUT ID. If there is a LUT hit, we copy the output of the LUT to the registers, skip the original computation and continue execution. Otherwise, we execute the original code and update the LUT, then continue execution. **Efficient lookup in hardware.** The transformation itself is simple and straightforward. However, as we discussed in the previous section, one of the biggest challenges is how to perform the lookup efficiently. The source of the inefficient lookup is the different numbers/sizes of inputs for different target computation blocks. Traditionally, memoization focuses on functions with one or two inputs, which is easy to handle in hardware by concatenating them. However, to generalize memoization, the different numbers/sizes of inputs make simply concatenating the inputs and using them as a lookup tag infeasible, especially when the computation block has many inputs. For instance, one of the target computation blocks in our benchmarks (Sobel) needs 9 floating-point numbers as inputs. Concatenating them means we need a tag of 36 bytes for each LUT entry. It not only requires significantly increase in the LUT capacity, but also causes other resource overhead, such as data movement and comparators. In the example of Sobel, we need be able to send more than 200 bits for one lookup and handle tag comparison for such large chunk of data, otherwise the hardware simply cannot exploit the memoization opportunity (such as previous work [34], which can handle memoization inputs of a total size up to 128-bit). On the other hand, such resources dedicated for memoization will be wasted for applications that only have a few inputs. Therefore, we believe a small and fixed size tag is the key to efficient lookup and memoization.

LUT hit rate. Having a high LUT hit rate is also crucial for memoization. To maintain high a hit rate for many inputs, we combine memoization with approximation. Approximation exploits the fault-tolerant characteristics of some applications. By relaxing the requirement of producing precise results and allowing an error bound in the output, applications in the field of image processing and computer vision for example, can achieve notable performance and/or energy efficiency benefits. Therefore, we allow similar inputs to be a match to increase hit rates in AxMemo.

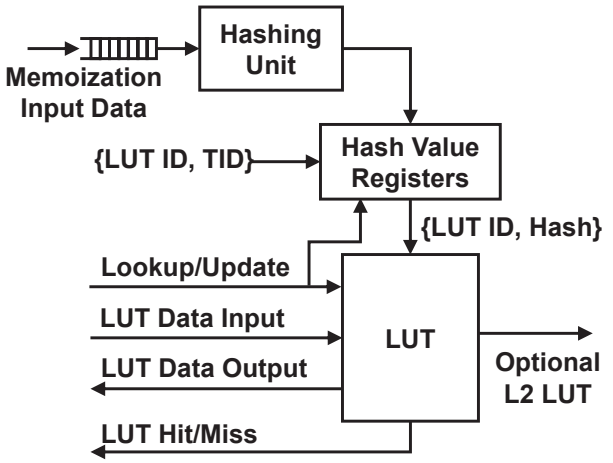


Figure 2: Overview of the memoization unit. Major components include hashing unit, hash value registers and LUT. TID denotes thread ID. Memoization inputs are the inputs of the computation block being replaced, such as x,y in Fig. 1.

3 MEMOIZATION UNIT

Most of the memoization operations are performed by the memoization unit. In AxMemo, the memoization units are private to each CPU core. Fig. 2 shows the overview of the memoization unit: the memoization unit mainly consists of Hashing Unit, Hash Value Registers and LUT. To efficiently perform the lookup with a fixed tag size, we hash the inputs of the original computation block, dubbed the *memoization inputs*, and use the hash value for the LUT tag. The hash values are first stored in Hash Value Registers. When a lookup or update request is received, the hash value is read from the Hash Value Registers using LUT ID (`LUT_ID`) and thread ID (`TID`) as an address to index an entry of a LUT. Combined with the `LUT_ID`, the hash value will be used to perform the lookup/update operations in the LUT. The LUT in the memoization unit is considered as L1 LUT, and we can also have an *optional* inclusive L2 LUT to improve hit rate.

3.1 Hash Key Generation and Memoization Approximation

Hash key generation. We propose to use the cyclic redundancy check (CRC) algorithm [18] for the hashing. CRC is a widely used error detection algorithm. An n-bit CRC algorithm can take an input of arbitrary size and generate an n-bit CRC value. There are many properties of CRC that make it suitable for memoization scheme:

(1) It does not need to have all the input data to start hashing. It takes a stream of inputs and "accumulates" them into the output. This is an important feature because it can help with hiding the latency of hash value calculation.
(2) Every bit of the inputs affects the CRC output, not like the sampling based algorithm such as the one used in [3].
(3) A hardware implementation of CRC is cheap.
(4) The CRC can work in many sizes: 16-bit CRC, 32-bit CRC, 64-bit CRC etc. We can use different CRC sizes for different designs.

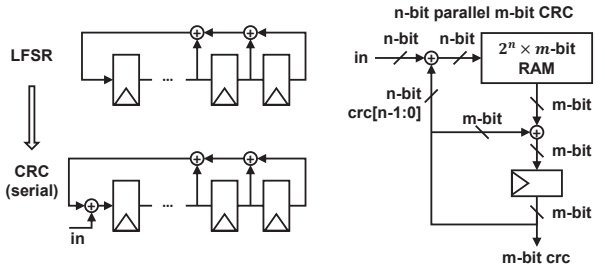


Figure 3: Implementations of cyclic redundancy checking (CRC) unit. The linear-feedback shift register (LFSR) unit and serial CRC unit are not showing actual configuration.

As shown in Fig. 3, a simple implementation of CRC is similar to linear-feedback shift register (LFSR). Compared with LFSR, however, CRC uses the XOR of the input bit and the feedback bit as the input to the first register, instead of using the feedback bit directly. A serial CRC implementation processes 1 bit of input every clock cycle. An alternative n-bit parallel implementation can process n bits per clock cycle, which reduce the latency to $1n$ of the serial version. The n-bit parallel implementation of m-bit CRC needs a $2^n \times m$-bit RAM. This small RAM stores constants that are required for the n-bit parallel implementation. Note that any hashing scheme incurs collision, i.e., two sets of inputs with different values generate the same hash value, but it is acceptable for approximable applications as long as it is not frequent.

Approximation for memoization. Even without considering hash collision, using a CRC value as a LUT tag still requires all the memoization inputs to be exact match to have a LUT hit. As we discussed in Section 2, we exploit the correlation between the similarity of the inputs and the similarity of the outputs and apply approximation to the inputs in AxMemo. The approximation can potentially increase the LUT hit rate, thus the performance of the application.

We use a simple approach to apply the approximation, truncating some least significant bits (LSBs) of the memoization inputs before sending them to the hashing unit. The level of approximation, i.e. the number of truncated bits, is programmer controllable for each variable. Mathematically, the truncation rounds the input down by a given relative precision (for floating-point variables) or absolute precision (for integer variables). Though we only evaluated truncation, more sophisticated approach can be applied since the approximation does not affect hashing unit.

In our experiment, we use the compiler tools to profile the applications with sample input set that is different from evaluation input set. Then we use the statistics to determine the number of bits to be truncated for each memoization input. Alternatively, we can use a dynamic approach. A certain percentage of the execution time can be allocated for profiling at runtime periodically. During the profiling phase, the memoization unit always returns miss to the processor even if there is a hit so we can use the computation results and the LUT output to calculate error and adjust the approximation level accordingly during the execution. See Section 5 for more detail.

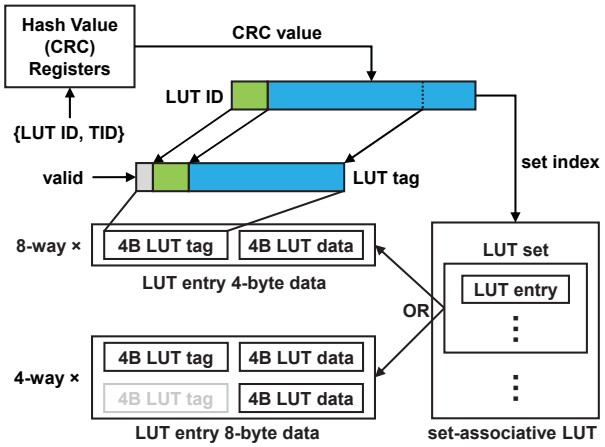


Figure 4: LUT organization similar to a set-associative cache. Each set can be configured to either 4-way or 8-way. The CRC value is combined with LUT ID and used for LUT tag.

3.2 Hash Value Registers

The Hash Value Registers (HVRs) are used to store the hash values and they are not just buffers for temporarily storing the CRC values that are ready. When the processor sends the memoization inputs of different LUTs to the memoization unit in an interleaved way, the HVRs store the intermediate results of CRC and serve as the hardware context of the CRC calculation. The HVRs are addressed by the LUT ID and the thread ID `{LUT_ID, TID}` as shown in Fig. 4.

Note that thread ID is the hardware thread ID used to identify the threads in processors support simultaneous multithreading (SMT). The number of needed CRC registers depends on the maximum number of allowed LUTs and maximum number of SMT threads. Supposing an architecture with up to 8 LUTs in one thread and 2 SMT threads, the CRC registers should contain at least $16 \times 32\text{-bit}$ registers for 32-bit CRC. For out-of-order processors, `{LUT_ID, TID}` is equivalent to the architectural name of the Hash Value Register. To support the instruction-level parallelism, more "physical" Hash Value Registers are needed and they should also be "renamed".

3.3 Lookup Table (LUT) Structure

The LUT has an organization similar to a normal set-associative cache, as depicted in Fig. 4. The LUT is composed of LUT entries organized in sets, which are equivalent to sets of cache lines. Each LUT entry has one LUT tag and one LUT data. Compared to a cache line, the LUT tag is equivalent to the address and the LUT data is equivalent to the data. The LUT data is 4-byte by default, and we can configure it to 8-byte by combining two LUT entries. The 8-byte LUT data is necessary to support 8-byte data types and can also be useful in the cases that the LUT logically has many outputs. In such cases, we can pack as many outputs into the 8-byte LUT data field as possible to reduce number of LUT accesses. Note that we may implement L1 LUT and *optional* L2 LUT differently. We use a dedicated SRAM array for L1 LUT while allocating part of last-level cache for L2 LUT. L1 LUT is limited to small sizes ($\leq 16\,\text{KB}$) and L2 LUT can use up to half of the last-level cache. Implementing a small separate L1 LUT can avoid interference with valuable L1 cache space and timing. On the other hand, using part of last-level cache for L2 LUT can avoid a large overhead to implement a large LUT in the baseline processor.

In AxMemo, one LUT set can be configured as either 8-way 4-byte LUT tags with 4-byte LUT data, or 4-way 4-byte LUT tags with 8-byte LUT data. In the latter case, the half of the LUT tags are not used. Since some lower bits of the CRC value `CRC` are used to index a set, we do not need to store the whole 32-bit `CRC` value in the LUT tag array. The upper bits of the LUT tag array is used for a valid bit and `LUT_ID`. Such configurations ensure there is enough space for 1-bit valid bit and 3-bit `LUT_ID`. With `LUT_ID` included in the LUT tag, we can store multiple logical LUTs in one unified LUT.

We chose the LUT entry size and associativity because such configuration allows one set of the LUT entries to just fit into a 64-byte last-level cache line, when LUT tag and LUT data are both considered as data for the cache. This allows us to use the last-level cache as L2 LUT most efficiently. For simplicity, we assign a fixed number of ways in the last-level cache to the L2 LUT in our experiment.

3.4 LUT Lookup and Update Operation

When a CRC value `CRC` is ready, the memoization unit can start a LUT lookup or update upon a request from the CPU. The CPU sends lookup/update requests along with the `LUT_ID` and its `TID` to the memoization unit.

LUT lookup operation. When receiving a lookup request, the memoization unit first checks if there is any pending CRC calculation for this LUT. Since our implementation only allows the lookup request to be sent after the last memoization input is sent to the memoization unit, we only need to check if there is data from this thread for the LUT waiting in the small input queue of the memoization unit. If there is any pending calculation, the memoization unit stalls the request until the calculation is completed. Then the lookup is performed in the LUT to find a LUT entry whose LUT tag matches with `{LUT_ID, CRC}`. A condition code, which is used for branch instructions, is also set based on lookup result, i.e. hit or miss. The condition code is used later by the program to determine whether or not the computation should be skipped.

Upon a LUT hit, the memoization unit returns the LUT data to the CPU register specified by the lookup request and the computation is skipped. Upon a LUT miss, the program executes the original computation and the memoization unit immediately starts to allocate an LUT entry for the update request and will update the LUT once the computation is done. When no invalid entry is available, the L1 LUT entries are invalidated (without L2 LUT) or evicted to L2 LUT (with L2 LUT) using the least recently used (LRU) policy. The same allocation policy is used for L2 LUT. Different from data in normal cache, the LUT entry will not be eventually

written back to main memory. The L2 LUT entry will always be invalidated when the it needs to be evicted.

LUT update operation. Once the original computation is done, the CPU will send an update request to the corresponding LUT entry. An update is very similar to a lookup: it first accesses the CRC register to get `CRC`, then it writes `LUT_ID, CRC` and the input data to the corresponding LUT entry and sets the valid bit. As mentioned earlier, the allocation of the LUT entry happens in parallel with the original computation, most update can perform the write immediately.

For multi-core processors, there is no coherence required for the LUTs, because the same LUT tag should always have the same LUT data without hash collision, which makes coherence unnecessary. If the same LUT tag has different LUT data in different LUT arrays, it means a collision occurred. In such a case, it is meaningless to force the LUTs to stay coherent since we cannot tell which data is more precise.

4 ISA DESIGN

To enable memoization, we need to extend the ARM-v8a ISA with the following five instructions. All of them can be encoded into 32-bit instructions:

(1) **`ld_crc`** `dst, [addr], LUT_ID, n`: This instruction loads memory content at `addr` to register `dst`. It also sends the loaded data (with the last `n` bits truncated) and the value of `LUT_ID` to the memoization unit. AxMemo compiler replaces the normal load with this instruction for the variables that are marked as input to the memoization region.

(2) **`reg_crc`** `src, LUT_ID, n`: This instruction reads a register `src`, truncates the last `n` bits and sends the truncated value to the LUT. The destination LUT is identified by the value of `LUT_ID`. In some benchmarks, such as FFT, all the inputs to the memoization are not load instructions. We include this instruction in AxMemo ISA to support such scenarios.

(3) **`lookup`** `dst, LUT_ID`: This instruction performs the LUT lookup in the memoization units. The target LUT is identified by the value of `LUT_ID`. It also sets the condition code for branch instructions based on the lookup result. If the access to the memoization unit is a hit, **`lookup`** writes the returned data to the destination register `dst`. We use **`lookup`** together with a normal branch instruction to skip the calculation, when the access to the memoization unit—with LUT identified by `LUT_ID` is a hit.

(4) **`update`** `src, LUT_ID`: This instruction sends the value of register `src` to memoization unit and insert it to the LUT. The LUT entry is allocated after a lookup miss. It uses `LUT_ID` as the identifier for LUT.

(5) **`invalidate`** `LUT_ID`: This instruction invalidates all the entries of the LUT, which is identified by `LUT_ID`. This instruction is only used at the end of the program execution, or when the program needs to reuse the LUT associated with `LUT_ID` for other logical LUT. **`invalidate`** is called infrequently (≈1% of total number of dynamic instructions) during the execution of the application and only consumed a few processor cycles because we use dedicated hardware for invalidating all LUT entries. (Section 2).

For **`ld_crc`** and **`reg_crc`**, the programmer may enable approximation by specifying a non-zero value for `n`. The value of `n` determines how many LSBs should be truncated. The programmer may disable approximation (truncating) by specifying a value of zero for `n`. All the instructions, which access the memoization unit, sends the target LUT (specified by instructions with `LUT_ID`) to the memoization unit as identifiers of the memoization request. To guarantee that all the input data are sent to the CRC unit in the same program order, the **`lookup`** must only be issued after all the input data are sent to the target CRC unit and stored in the LUT. To support this case, we impose a dependency on the **`ld_crc`**, **`reg_crc`**, and **`lookup`** instructions. Such a dependency is equivalent to that of reading a dummy register and then writing into the same dummy register (i.e. the dummy register is both the source register and the destination register). The imposed dependency makes these instructions to follow the exact same program order as defined in the program.

5 COMPILER SUPPORT FOR AXMEMO

Compiler-guided code analysis. To facilitate the use of AxMemo for generic programs, compilers and dynamic analysis tools are needed to identify suitable computation blocks as candidates for memoization. This task requires examining the program's dataflow characteristics and input patterns in a search for sections of code that are memoizable and yield promising speedup. Two key factors determine the effectiveness of AxMemo for such sections: 1. Execution time spent on them must be substantial enough to result in notable performance gains if memoized. 2. They must have few, approximable inputs to yield a justifiable hit rate and error bound. Both factors can be evaluated if we construct a dynamic data dependence graph (DDDG) of the program detailed at the instruction level with all intermediate input data recorded. With this motivation, we devise a compilation and analysis workflow shown in Fig. 5 to identify and memoize candidate computation blocks for AxMemo. ❶ We use LLVM-Tracer [24] to generate a dynamic LLVM intermediate representation (IR) trace of the program by executing it on a sample input set. Note that the sample input set and evaluation input set are disjoint input sets. ❷ We construct a DDDG from this trace using ALADDIN [25] with some modifications.

A DDDG $G = V, E$ is a directed acyclic graph whose vertices represent LLVM IR pseudo-instructions and edges represent data dependencies between vertices. For example, an edge $v \rightarrow w \in E$ indicates that the output of instruction v is used as an input operand to instruction w. Each vertex of the DDDG is weighted by its estimated latency. Note that LLVM IR allows an arbitrary number of registers hence the DDDG captures true dependencies of the program. An AxMemo

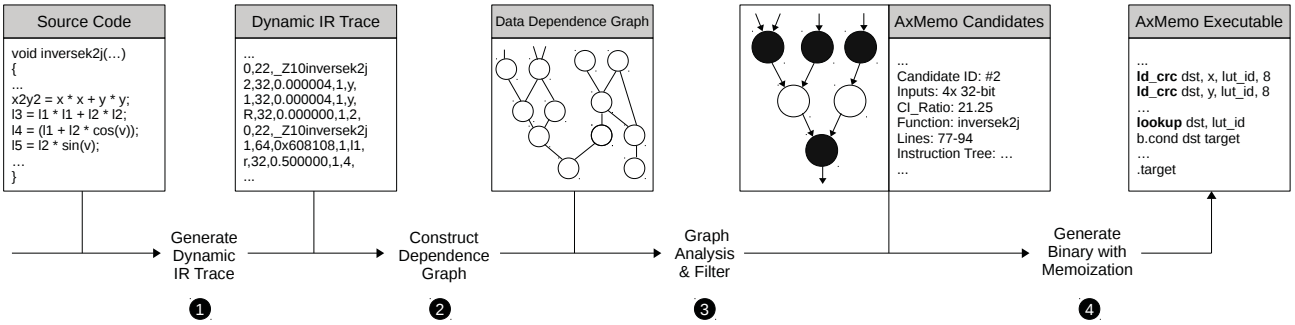


Figure 5: Compilation and analysis flow to identify optimal code sections for AxMemo

```
double normalize (
     double in,
     double min,
     double max,
     double min_n
     double max_n
)
{
   return (((in - min) / (max - min)) *
          (max_n - min_n)) + min_n;
}
```

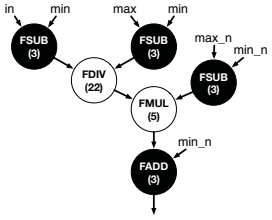


Figure 6: Example subgraph from the Dynamic Data Dependence Graph of Blackscholes. Black vertices are input/output vertices. The number in parenthesis indicates the weight (latency) of each vertex.

transformable candidate subgraph $S = (V_s, E_s)$ is a subgraph of the DDDG G that can be memoized with AxMemo without interfering original program flow. S has a set of input vertices $V_i \subset V_s$ and a set of output vertices $V_o \subset V_s$ satisfying:
(1) If $v \to w \in E$ where $v \in V \setminus V_s$ and $w \in V_s$, then $w \in V_i$
(2) If $v \to w \in E$ where $v \in V_s$ and $w \in V \setminus V_s$, then $v \in V_o$

In other words, S reflects a program block at the instruction granularity whose inputs are operands of vertices in V_i and outputs are results of vertices in V_o. Fig. 6 shows an example subgraph of the DDDG of the benchmark Blackscholes.

Recall that the potential speedup of an AxMemo transformed block depends on the execution time and input constraint factors detailed earlier. Thus, to capture the desirability of a candidate subgraph S, we define its Compute-to-Input ratio (CI_Ratio) as follows:

$$CI\_Ratio = \frac{\sum_{v \in V_s}(v \cdot \text{weight})}{\#\text{ of Inputs}} \quad (1)$$

A higher CI_Ratio means the corresponding code block likely needs more cycles to execute and/or has fewer inputs. Therefore, the higher this ratio is, the more execution cycles we can replace with a lookup and/or potentially higher hit rate we can expect due to fewer inputs. In step ❸, our analysis task simplifies to finding candidate subgraphs S in the DDDG G with a high CI_Ratio but not exceeding the number of inputs allowed by AxMemo. Our algorithm consists of running a directed breadth first search rooted at each vertex of the transpose of G. For each vertex $v \in V$, we find the AxMemo transformable subgraph S with v as the sole output vertex (i.e. $V_o = \{v\}$) having the highest CI_Ratio and add it to our candidate list if that ratio exceeds our predefined threshold. To aid this search, programmers

Table 1: The dynamic data dependence graph (DDDG) analysis of AxBench [31] and Rodinia [4] Benchmarks.

	Benchmarks	Total # of Dynamic Subgraphs	# of Unique Subgraphs	Compute/# of Inputs Ratio	Memoizaton Coverage
AxBench	Blackscholes	61,114	8	48.41	75.24%
	FFT	5,376	3	43.85	93.83%
	Inversek2j	840	4	38.13	67.91%
	Jmeint	516	4	9.87	53.10%
	JPEG	260	6	15.49	19.30%
	K-means	387	4	9.01	75.31%
	Sobel	32,288	2	23.81	35.32%
Rodinia	Hotspot	15,429	43	16.35	45.43%
	LavaMD	24,614	16	13.77	65.28%
	SRAD	110,003	2	21.58	45.20%

may specify specific functions for analysis rather than the entire program as sections involving system calls or I/O operations are non-memoizable. After the search process, we often identify more than 10^4 candidates depending on the application and its input size. This is no surprise as our graph is constructed from a dynamic trace, thus many subgraphs will have identical structure if they belong to a loop body or repeated function call. Therefore, in the last step of ❸ we filter out candidate subgraphs if they are structurally equivalent to or subsets of other candidates. This is done by comparing their static instruction IDs from compiled assembly, for example, a loop body iterated many times will generate the same subgraph with identical static instruction IDs. Finally, we merge the remaining subgraphs with high overlap to create larger subgraphs for better memoization efficiency.

Table 1 shows our analysis on applications from the AxBench and Rodinia benchmark suites. The first two columns respectively denote the total number of candidate subgraphs identified and the number of unique subgraphs after filtering in step ❸. The third column is the average CI_Ratio among all filtered candidate subgraphs. The last column, *Memoization Coverage*, is measured by $\frac{\sum(v_s \cdot \text{weight})}{\sum(v \cdot \text{weight})}$ where $v_s \in V_s$ are vertices belonging to candidate subgraphs and $v \in V$ are all vertices of the DDDG. In other words, coverage is the vertex weight ratio of candidate sections to the entire graph. This fraction gives an estimate of the potential computation time that can be eliminated by memoization.

These results predict that benchmarks such as Blackscholes and FFT, both boasting a high CI_Ratio and coverage, will achieve significant speedup if hit rates are respectable after truncation. An important caveat of memoization coverage is that it does not always directly translate to an upper

Table 2: Evaluated benchmarks. All data sets used are default and provided by the benchmark suite. Memoization input size denotes the total input size in bytes for each (logical) LUT. The number of tuples corresponds the number of memoized blocks of the benchmark. The actual LUT tag in hardware is generated by hashing.

	Benchmark	Domain	Description	Input Dataset	Memoization Input Size (bytes)	# of Truncated bits
AxBench	Blackscholes	Financial Analysis	Calculates the price of European-style options	200K options	24	0
	FFT	Signal Processing	Radix-2 Cooley-Turkey FFT	4,096 floating-point data points	4	0
	Inversek2j	Robotics	Calculates the coordinated of a two-joint arm	1.24 million pairs of angles	8	8
	Jmeint	3D-Gaming	Detects the intersection of two triangles	Coordinates of 145K pairs of triangles	36	6
	JPEG	Compression	Compresses an image using JPEG standard	512x512 pixel images	(16, 16)	(2, 7)
	K-means	Machine Learning	K-mean clustering on an image	512x512 pixel images	12	16
	Sobel	Image Processing	Applies Sobel filter on an RGB image	512x512 pixel images	36	16
Rodinia	Hotspot	Physics Simulation	Simulates the temperature of an IC chip	512x512 maps of power and temperature	16	8
	LavaMD	Molecular Dynamics	Simulates interaction of particles with charge	16x100 particles of random initial position	12	0
	SRAD	Medical Imaging	Image denoising	458x502 pixel medical images	24	18

bound on performance speedup. Our DDDG model weighs each pseudo-instruction individually by its cycle time, yet modern processors are able to execute multiple instructions concurrently with multiple functional units.

Code Generation. With the final candidate computation blocks for memoization selected, step ❹ begins by selecting the number bits of the inputs to be truncated such that we can achieve a high hit rate while keeping output error within a given bound. To do so, we profile applications by observing error rates when truncating inputs by different numbers of bits. For all benchmarks evaluated, we truncate bits while constraining output error to less than 0.1% (or 1% if the output is an image). After determining the number of truncated bits, AxMemo instructions are inserted into selected code sections of the assembly and recompiled.

6 EVALUATION

Benchmarks. Table 2 summarizes the benchmarks from AxBench [31] and Rodinia [4] that we evaluated. The benchmarks from AxBench cover a wide range of applications suitable for approximation, including financial analysis, signal processing, robotics, and machine learning. To further evaluate the benefits of AxMemo across other domains, we include three other benchmarks—in the domain of physics simulation, molecular dynamics, and medical imaging—from Rodinia [4]. The input datasets used for evaluation are provided directly by the benchmark suites. Column 5 in Table 2 lists the total size of memoization inputs in bytes for each benchmark. The large sizes of the memoization inputs demonstrate the necessity for using CRC values as LUT tags. The last column in Table 2 indicates the number of truncated bits per input for each benchmark. This number is selected based on compiler analysis and profiling to achieve the highest hit rate while satisfying output error constraints as described in Section 5. Our compiler analysis and experiment both show that 32-bit CRC is generally large enough to avoid collision. We use the complete AxMemo compilation workflow to generate memoization-enabled binaries for the evaluated benchmarks.

Quality metric and monitoring. Since we apply bit truncation on inputs and the hash function may result in collision, applications may suffer a degradation in output error. To assess output quality when the memoization is enabled, we use output error (Equation 2) defined as follows [3]:

$$E_r = \frac{\sum_i (\hat{X}_i - X_i)^2}{\sum_i X_i^2} \quad (2)$$

In the equation, X represents correct results from the unmodified source code and $\hat{X}$ represents results with AxMemo enabled. The output of Jmeint is a boolean value indicating whether two 3D triangles intersect. As such, we use misclassification rate (percentage of incorrect classifications). To ensure the output quality, we also implement quality monitoring scheme to prevent large output error. During the execution, every 1 out of 100 LUT hits is ignored. The LUT performs the lookup normally but returns a miss instead of hit. The LUT output is used to compare against the data sent by processor for updating the LUT. For each comparison, a simple relative error is calculated. For every 100 comparison, the statistics of the relative error is checked. If more than 10% of the relative errors are larger than 10%, the memoization is disabled.

6.1 Experimental Setup

Cycle-accurate simulator. All experiments in this section are assessed with the gem5 simulator [2] to evaluate the benefits of AxMemo. We use one of the gem5's default configuration which accurately models a high-performance in-order (HPI) ARM processor, as we target processors for low-power applications. The HPI processor, released by ARM, includes detailed configurations and timing parameters to model a modern in-order processor implementation using `ARM-v8a` ISA. Although we evaluate in-order processor, AxMemo can also be implemented in out-of-order processors as we explained in Section 3 and 4. Table 3 summarizes key microarchitectural parameters of the configuration used for AxMemo. We modified the gem5 simulator to include all proposed ISA extensions and additional hardware necessary for AxMemo. One memoization unit is appended to each core of the processor. Table 4 shows timing parameters for the proposed instructions. We extract these parameters from synthesis results shown in Table 5. The reported timing includes the 1-cycle overhead of reads/writes to the dummy

Table 3: Major microarchitectural parameters for the ARM high-performance in-order (HPI) processor using `ARM-v8a` ISA. Note that only 1MB of shared L2 cache is enabled since only one of the two cores is used in system emulation mode.

Number of Cores, Frequency	Two cores, 2GHz
Issue Width	Two, in-order
Number of Integer Units / Core	Two ALUs, One Multiplier, One Divider
Number of FP Units / Core	One
Number of Ld/St Units / Core	One
L1 Instruction Cache	32KB, 2-way set-associative, 1-cycle hit latency
L1 Data Cache	32KB, 4-way set-associative, 1-cycle hit latency
L2 Shared Cache	2MB, 16-way set-associative, 13-cycle hit latency
Memory Configuration	4GB, 1600MHz DDR3, two channels

Table 4: Timing parameters for AxMemo ISA extensions.

AxMemo Instruction	Latency
`ld_crc` `dst, [addr], LUT_ID, n`	One cycle for each byte of data. Does not stall CPU unless the input queue of the memoization unit is full.
`reg_crc` `src, LUT_ID, n`	One cycle for each byte of data. Does not stall CPU unless the input queue of the memoization unit is full.
`lookup` `dst, LUT_ID`	Two cycles for L1 LUT and 13 cycles for L2 LUT. Waits until the undergoing CRC operation finishes.
`update` `src, LUT_ID`	Two cycles.
`invalidate` `LUT_ID`	One cycle for each way in a set.

register, which is necessary to enforce program ordering for **`ld_crc`**, **`reg_crc`**, and **`lookup`** instructions (Section 4). We use `CLANG/LLVM 3.4` and `GCC 4.8` to compile, analyze, and generate benchmark binaries equipped with memoization. We also enable maximum compiler optimization for all applications prior to transforming memoized sections.

Hardware synthesis. We implement all proposed microarchitectural units, including the 32-bit CRC unit, Hash Value Registers (`16×32-bit`) and LUTs of various sizes in `Verilog`. The 8-bit parallel 32-bit CRC unit uses an iterative algorithm and processes 8 bits of the input each cycle. To match the throughput of the CRC unit with the most common case of a 4-byte input, we unrolled the 32-bit CRC unit four times and apply pipelining. We use the `Synopsys Design Compiler (K-2015.06-SP3-1)` with a `FreePDK 45 nm` technology model. To remain consistent with the process technology used to model other components, we properly scaled down synthesis results of the proposed microarchitectural units to `32 nm`. Since all synthesized hardware components have a latency smaller than `0.5 ns`, we do *not* need to reduce the baseline core clock frequency in our simulations (Table 3). The area overhead, energy consumption, and the timing parameters of the synthesized units are shown in Table 5. With the largest L1 LUT (`16 KB`), the added memoization units for all the cores consumes up to 0.166 mm^2 (2.08 %) area per the HPI processor with an estimated area of 7.97 mm^2, estimated using McPAT version 1.3 [10] also with `32 nm` technology (note that L2 LUT is partitioned from L2 cache). The quality monitoring unit uses comparison logic proposed in [13], the area (power) overhead is 16.8 μm^2 (7.47 μW) with a latency of 0.96 `ns`.

LUT hardware configurations. To better understand the benefits of AxMemo, we perform our experiments using various LUT configurations, all built from the same base design and a 32-bit CRC unit. We only vary the size of the LUTs and the number of levels of LUTs for each configuration. For the experiments using one level LUTs, their sizes

Table 5: Area, energy and timing analysis for `32 nm` technology node. CRC32 denotes 32-bit CRC. CRC32 unit shown here is already unrolled and pipelined. All LUTs are 8-way set-associative (4-byte LUT data).

	Area (mm^2)	Energy (pJ)	Latency (ns)
CRC32 Unit	0.0146	2.9143	0.4133
Hash Register	0.0018	0.2634	0.1121
LUT (4KB)	0.0217	3.2556	0.1768
LUT (8KB)	0.0364	4.4221	0.2175
LUT (16KB)	0.0666	7.2340	0.2658

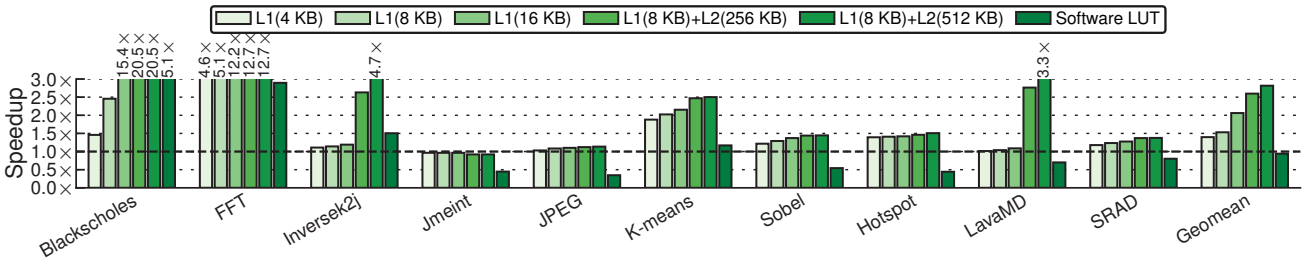
(including both tag and data) range between `4 KB`, `8 KB`, and `16 KB`. In this case, we use small-sized LUTs to limit the areal overhead of dedicated SRAM arrays necessary for larger L1 LUTs (synthesis results listed in table 5). The memoization unit can have an *optional* L2 LUT. The L2 LUT is partitioned from last-level cache (L2 cache in this case) and does not require dedicated SRAM. The size of L2 LUTs is either `256 KB` or `512 KB`. When we evaluate a configuration with L2 LUT, we fix the L1 LUT size to 8KB. This design decision is aiming to strike a balance between the performance and cost of hardware resource.

Energy modeling. We use CACTI 6.5 [17] to estimate access energy and latency of the LUTs and the 1 KB SRAM in the CRC unit. The total dynamic energy of the processor is estimated using McPAT version 1.3 [10]. We configure McPAT according to the gem5 configuration for HPI processor and use statistics from gem5 to calculate the number of accesses to each component. Finally, we feed number of accesses to McPAT to calculate the application's energy consumption.

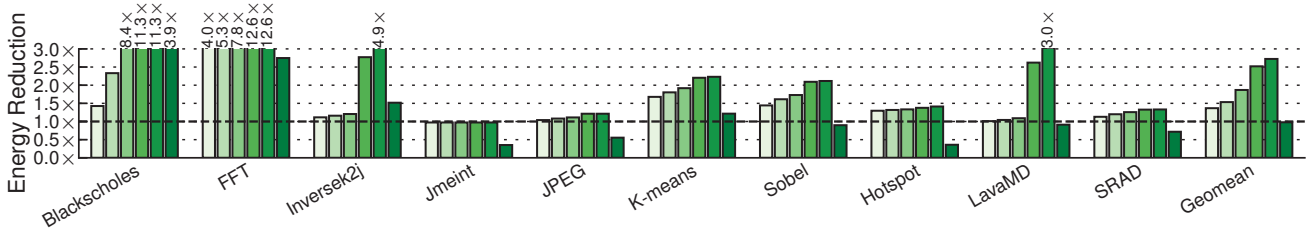
6.2 Experimental Results

We evaluate the benefits of AxMemo across a diverse set of benchmarks, including financial analysis, robotics, molecular dynamics, and machine learning. In all evaluations, the baseline is a regular ARM HPI processor with the same configuration but not equipped with memoization hardware. All results from hereon are normalized to this baseline. For each evaluation, we also perform a sensitivity study of the results for various AxMemo configurations: `L1(4 KB)`, `L1(8 KB)`, `L1(16 KB)`, `L1(8 KB)+L2(256 KB)`, `L1(8 KB)+L2(512 KB)`,

and bit truncation is applied identically for memoization inputs. In addition to the baseline, we implement a software LUT for memoization as another contender. The software implementation of CRC uses an 8-bit parallel algorithm (Fig. 3, Section 3). To calculate the CRC value of a 4-byte input, the software implementation requires at least $4 \times 3 = 12$ instructions (1 AND, 1 LOAD and 1 XOR for each byte). We then use the CRC value to index entries of the LUT array using $CRC\%2^N$, where 2^N is the total number of entries in the LUT array. To reduce the cost of the software implementation, we use a simple array as the LUT. In contrast to the hardware implementation, the cost of increasing the number of LUT entries in software is significantly lower. As such, we simply increase the size of LUT arrays to the point where



(a) Speedup



(b) Energy Savings

Figure 7: (a) Speedup and (b) energy saving (defined as $E_{baseline}/E_{AxMemo}$) using different configurations of LUTs. L1 LUT is fixed at 8KB and the L2 cache size is reduced accordingly when L2 LUT is enabled. Software LUT is the software memoization implementation.

speedup plateaus. Based on these trials, we fix the number of LUT entries for the software implementation to 2^{28}, equivalent to `1 GB` for `4 B` data types. Recall that we use the remainder operation to obtain the array index, which means only the last 28 bits of CRC value are used to index the array.

Performance and energy benefits with AxMemo. Fig. 7a shows full application speedup with AxMemo for different LUT configurations normalized to the non-memoized ARM HPI baseline, where the application executes normally on the CPU. Of all the benchmarks, Blackscholes enjoys the highest speedup of 20.5× for the `L1(8 KB)+L2(512 KB)` configuration. The main reason for such a high speedup in Blackscholes is that almost the entire computation kernel of the benchmark, consisting of no less than 40 instructions, is replaced with a single LUT access. The repetitive input patterns needed for quantitative financial analysis [16] also make the overall hit rate very high. Out of ten benchmarks, only Jmeint exhibits virtually zero speedup for all tested AxMemo configurations. This is due to the low lookup hit rate for Jmeint (less than 0.1%), an indicator of low computation reuse available for AxMemo to exploit. We study the source of these benefits in the following paragraphs. On average, AxMemo delivers 1.40× and 2.82× speedup for the `L1(4 KB)` and `L1(8 KB)+L2(512 KB)` configurations respectively. Some benchmarks, such as Blackscholes, do not benefit from a large LUT. These benchmarks only need small LUT to capture most of their computation reuse. This is similar to the case when data cache becomes much larger than the working set of an application. In contrast to AxMemo, the software implementation, on average, suffers a slow down by 0.94×. The underwhelming results of the software LUT implementation is largely due to the significant overhead of CRC calculation in software. Out of ten benchmarks, only Blackscholes, FFT, Inversek2j and K-means enjoy speedup from the software implementation. For these four benchmarks, the sheer amount of computation replaced by lookup hits outweighs software overhead from performing memoization. Fig. 7b shows the energy reduction for each benchmark compared to the baseline with no memoization. Similar to the trend of speedup results, the highest energy reduction is achieved for Blackscholes (11.3×), FFT (12.6×), and Inversek2j (4.94×). The `L1(4 KB)` and `L1(8 KB)+L2(512 KB)` configurations respectively deliver a 1.37× and 2.72× energy reduction on average for these benchmarks. Once again, the software LUT implementation provides no overall energy reduction, which we still attribute to the large overhead of CRC calculation.

We further study the sensitivity to the total L2 cache size using a 256kB L2 LUT. When the total L2 cache size is decreased from 1MB to 512kB (768kB to 256kB in terms of capacity available for caching), the average performance degradation is 0.44%, with hotspot showing the highest degradation of 1.55%.

Dynamic instruction count. Fig. 8 shows the total dynamic instruction count of each evaluated benchmark normalized to the baseline without memoization. We show the breakdown of the dynamic instruction count between memoization instructions and normal instructions. We consider **`ldr_crc`** instructions not as a memoization instructions, but as a normal instruction because they simply substitute the original load. On average, AxMemo effectively reduces the number of dynamic instructions by 20.0% and 50.1% for `L1(4 KB)` and `L1(8 KB)+L2(512 KB)`, respectively. We observe the largest reduction in the number of dynamic instructions for FFT, where an overwhelming section of the application is replaced by memoization and hit rate is higher than

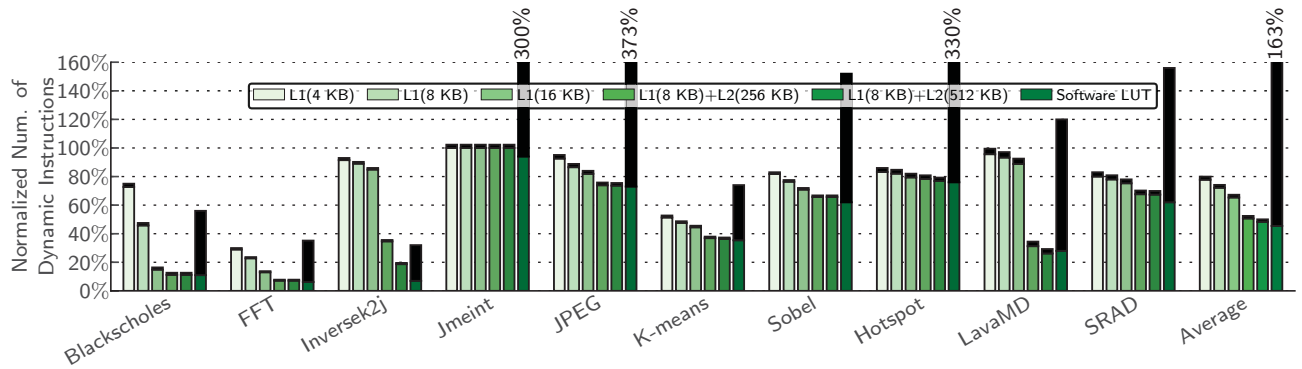


Figure 8: Dynamic instruction count normalized to the baseline with no memoization. The top bar in the stack (black-colored bars) represents memoization instructions, which include the AxMemo instructions (Section 4) and the additional branch instructions.

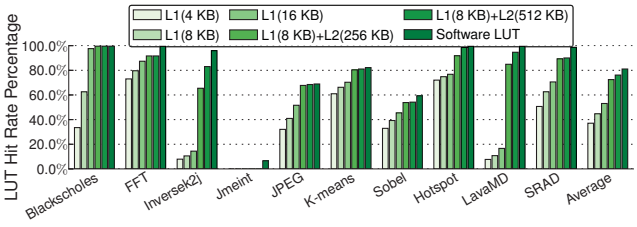
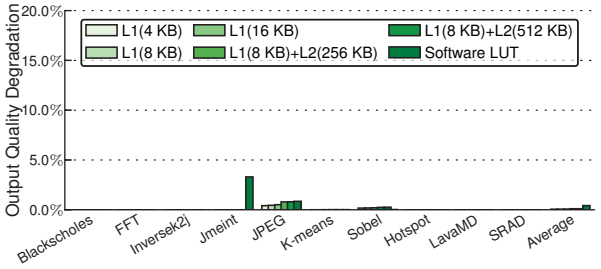


Figure 9: The LUT hit rate of various LUT configurations, including AxMemo and the software LUT implementation.

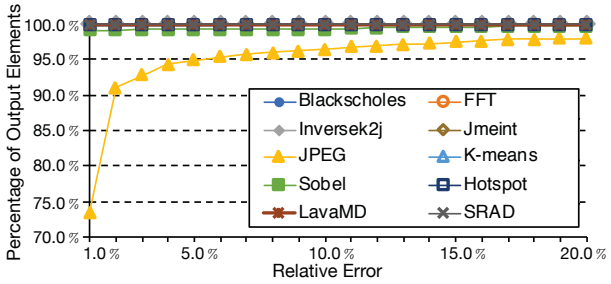
90%. This coincides with earlier compiler analysis showing a high memoization coverage for FFT in Section 5. The software memoization implementation, on the other hand, increases dynamic instruction count by ≈2.0×, causing most applications to see a slowdown and virtually zero energy reduction using the software implementation (See Fig. 7). The total number of dynamic instructions is a proper indicator for the amount of benefits that AxMemo can deliver. However, different instructions have different latencies, so dynamic instruction count alone does not determine exact speedup or energy savings.

Lookup-Table (LUT) hit rate. Fig. 9 shows the LUT hit rate of the evaluated benchmarks across different AxMemo configurations. For AxMemo configurations with multiple levels of LUT[1], we calculate the total lookup hit rate across both levels. The last bar for each benchmark shows the software implementation of our proposed memoization approach. On average, across all benchmarks, `L1(4 KB)` (the smallest LUT size) and `L1(8 KB)+L2(512 KB)` (the largest LUT size) provide a 37.1% and 76.1% total hit rate respectively. Increasing from the smallest LUT size to the largest LUT size yields a 39.1% improvement in the lookup hit rate on average. This result shows the effectiveness of AxMemo's multi-level LUT design in improving hit rate. Note that the L2 LUT is inclusive, therefore adding the L2 LUT is effective in increasing the total hit rate but has minimal impact on the L1 LUT hit rate. The software LUT implementation, delivers an average hit rate of 81.1% across all benchmarks, slightly better than the 76.1% of the `L1(8 KB)+L2(512 KB)` configuration. As previously noted,

[1] `L1(8 KB)+L2(256)` and `L1(8 KB)+L2(512)`



(a) Output Error (E_r)

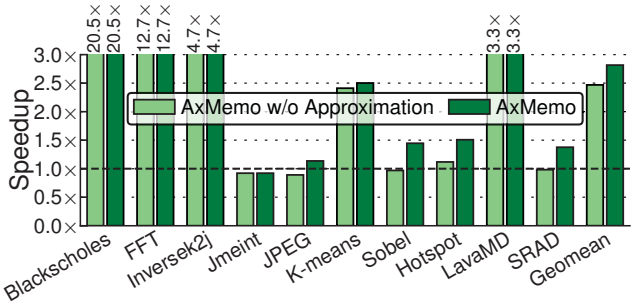


(b) Cumulative Distribution Plot of Element-wise Relative Error

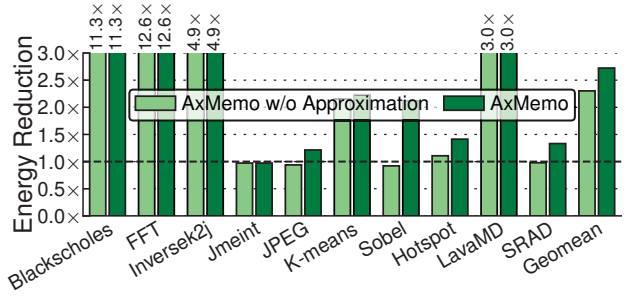
Figure 10: (a) Whole application quality loss with AxMemo for all the configurations compared to a baseline with no accuracy loss and (b) cumulative distribution function (CDF) plot of the applications' output quality loss for AxMemo when L1(8KB)+L2(512KB) LUT configuration is used. Software LUT has higher error rate due to its higher collision rate.

the small overhead of increasing memory used in software allowed us to increase the LUT array's size to 1 GB, after which further increases no longer improve speedup. Fig. 9 indicates that all the benchmarks except Jmeint exhibit significant computation reuse. The reason for Jmeint's failure is its lack of repetitive or similar input patterns to the memoized computation block.

Output quality degradation. To assess the output quality degradation, we use the quality metric defined in Equation 2. Fig. 10 shows the final output quality degradation of the evaluated applications across all the AxMemo configurations. To provide more detailed information about the quality, we also show the cumulative distribution function



(a) Speedup



(b) Energy Saving

Figure 11: Speedup and energy savings of AxMemo, without approximation and with approximation. In both cases, AxMemo uses 8 KB L1 LUT and 512KB L2 LUT.

of the element-wise relative error of the output in Fig. 10. On average, the output error E_r across all configurations of AxMemo falls below 1%, and none of the execution has the memoization disabled by the quality monitoring unit. The main reason for this low output error with AxMemo is its virtually zero hashing collision rate. Furthermore, since we use the compiler analysis to determine the number of truncated bits in Section 5, the effects of truncation on the output quality will be minimal. Finally, errors from the LUT outputs do *not* always affect final values of the application. The last bar in Fig. 10 shows output quality degradation for the software implementation, which has a *non-zero* collision rate (1% on average and up to 6.6%). As we previously detailed, the reason is that the 4 most significant bits of the hash value are discarded when indexing the LUT array.

Effectiveness of approximation. We use bit truncation as an approximation technique. To show the effectiveness of approximation, we assess the speedup and energy penalties of AxMemo when no truncation is performed on the memoization inputs. Fig. 11a and Fig. 11b show these speedup and energy reductions compared to AxMemo with truncation. On average, approximation improves the speedup and reduces energy by 14.1% (max. 49.1%) and 17.4% (max. 130%) respectively. Three benchmarks: JPEG, Sobel, and SRAD suffer from slowdowns and no longer enjoy energy savings without approximation. Without approximation, average LUT hit rate drops from 76.1% to 47.2% across all benchmarks. Therefore, we conclude that input truncation is an effective approximation technique for AxMemo.

Comparison with prior work. The closest work to AxMemo is Approximate Task Memoization (ATM) [3]. To generate the hash key, ATM first concatenates the inputs into a 1D vector, and create a vector of indices, each of which points to one byte in the concatenated input vector. Then indices are shuffled, and input bytes pointed by the first n indices are used to generate a hash key for the lookup table accesses. Nonetheless, we note that the only publicly available benchmark and datasets used both by ATM and AxMemo is Blackscholes from PARSEC benchmark suite [1]. As such, we have to implement ATM based on the description in [3] and apply the technique on our benchmarks. Our implementation of ATM only shows speedup for four benchmarks: Blackscholes (5.8×), FFT (2.6×), Inversek2j (1.3×) and K-means (1.3×). Other benchmarks show various levels of slowdown: Jmeint (0.31×), JPEG (0.4×), Sobel (0.3×), Hotspot (0.3×), LavaMD (0.7×) and SRAD (0.4×). On average, ATM shows a slowdown of 0.8× (geometric mean). The results again show that software-only approaches generally do not work well with applications with relatively small memoization code blocks.

7 RELATED WORK

AxMemo lies at the intersection of memoization and approximate computing. However, this work is fundamentally different from the prior work in the following two major ways: (1) we proposed a novel use of CRC for fixed-size LUT tag, which enables us to efficiently apply memoization to distinct regions of code with flexible number and data types of inputs using same hardware structures, and (2) we further exploit the fault-tolerant nature of the applications and the correlation between input similarity and output similarity to combine approximation with memoization. Below, we discuss the most related work in these two disjoint domains.

Memoization. Traditionally, memoization is used in very limited scope, such as avoiding unnecessary expensive arithmetic operations [20]. More recently, there have been proposals using both software-based and hardware-based techniques for more general-purpose memoization. Razlighi *et al.* [19] proposed a memoization-based neural network that does not need multiplier. It implements LUTs with content addressable memories (CAMs) to replace the multipliers in neural networks. However, this implementation is only for neural network and is not suitable for other applications. Brumar *et al.* [3] proposed a software implementation for task-level memoization. This approach is limited to task-based programs while our approach does not have this constraint. Performance-wise, the pure software approach only benefit a few benchmarks due to the large overhead. Tziantzioulis *et al.* [30] also proposed a software-only temporal approximate function memoization, which suffers high output error and does not actually check input values. Tuck *et al.* [29] proposed a memoization method using their hardware-based memory access disambiguation approach. However the memoization itself is still software-based and

brings little speedup. Connors *et al*. [5] proposed a compiler-directed instruction-level computation reuse, and Tsumura *et al*. [28] proposed an auto-memoization processor. However, these two proposals need either significant modification to the processor pipeline or complex hardware to track the input trees, which do not justify the relatively small performance improvement. Imani *et al.* [8] proposed memoization scheme on GPU using resistive content addressable memory (CAM). The CAM-based design is expensive and requires significant modification to the baseline GPU architecture. Sinha et al. [27] proposed a memoization-based approximate computing design on FPGA. However, this proposal only works for reconfigurable logic, because it cannot handle different number of inputs without using reconfigurable logic. Zhang *et al*. [34] proposed to leverage caches for memoization. This proposal uses the concatenated inputs as LUT tag and it can only support up to 128-bit of total input, and the authors mostly showed the effectiveness of the scheme with one-input or two-input functions. In contrast to these proposals, we used simple hardware that are loosely integrated with the CPU core, which does not require significant modification to the CPU pipelines. We also avoided the problematic large LUT tag by using the CRC hashing scheme.

Approximate computing. Several software and hardware techniques have been proposed for approximate computing, such as (1) loop perforation and loop early termination [26], (2) neural acceleration [7, 32], (3) approximate storage [12, 23], (4) computation substitution [21, 22], and (5) value prediction [11, 14, 15, 33]. Our work takes inspiration from neural acceleration, computation substitution, and value prediction techniques. In contrast to neural acceleration and computation substitution techniques in which code blocks are substituted with simpler computations, AxMemo substitutes large code blocks with only simple LUT accesses; hence significantly reduces the computation cost of the substituted code. Furthermore, similar to value prediction techniques, our work also predicts the value of outputs based on the similarity among inputs to code blocks. For example, Bunker-Cache [14] leverages the fact that approximately similar data exhibit spatial regularity in memory. In contrast, our work leverages a fundamentally different data similarity. That is, AxMemo is built on this insight that invocations of a code block with similar inputs produces similar outputs.

8 CONCLUSION

This paper proposes AxMemo, an approximate memoization scheme. AxMemo focuses on replacing a long sequence of instructions with a few lookup table accesses, therefore reducing the total number of instructions executed. To enable memoization for large computation blocks with different number of inputs and data types, AxMemo uses CRC to generate hash values and uses them as tags to perform lookups. Furthermore, we apply approximation by truncating inputs to improve LUT hit rate and observe output errors of only 0.2% on average. AxMemo is implemented with simple hardware with an area overhead of 2.1% and requires minimal software changes backed by our proposed compiler support. Our experiment shows that an AxMemo provides speedups up to 2.64× and energy savings up to 2.58×, yielding a 6.68× energy efficiency improvement compared to the baseline processor.

ACKNOWLEDGMENTS

This work was in part supported by NSF awards CNS #1703812, ECCS #1609823, CCF #1553192.

REFERENCES

[1] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques*.

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

[3] Iulian Brumar, Marc Casas, Miquel Moreto, Mateo Valero, and Gurindar S. Sohi. 2017. ATM: Approximate Task Memoization in the Runtime System. In *IEEE International Parallel and Distributed Processing Symposium*.

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization*.

[5] Daniel A. Connors and Wen-Mei W. Hwu. 1999. Compiler-directed Dynamic Computation Reuse: Rationale and Initial Results. In *IEEE/ACM International Symposium on Microarchitecture*.

[6] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *IEEE/ACM International Symposium on Computer Architecture*.

[7] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *IEEE/ACM International Symposium on Microarchitecture*.

[8] Mohsen Imani, Daniel Peroni, and Tajana Rosing. 2018. Nvalt: Nonvolatile Approximate Lookup Table for GPU Acceleration. *IEEE Embedded Systems Letters* 10, 1 (2018).

[9] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro* 31, 5 (Sept. 2011), 7–17.

[10] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *IEEE/ACM International Symposium on Microarchitecture*. ACM, 469–480.

[11] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value Locality and Load Value Prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[12] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[13] Zhenhong Liu, Daniel Wong, and Nam Sung Kim. 2018. Load-Triggered Warp Approximation on GPU. In *The International Symposium on Low Power Electronics and Design*.

[14] Joshua San Miguel, Jorge Albericio, Natalie Enright Jerger, and Aamer Jaleel. 2016. The Bunker Cache for Spatio-value Approximation. In *IEEE/ACM International Symposium on Microarchitecture*.

[15] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load Value Approximation. In *IEEE/ACM International Symposium on Microarchitecture*.

[16] Alexander Moreno and Tucker Balch. 2014. Speeding up Large-Scale Financial Recomputation with Memoization. *Seventh Workshop on High Performance Computational Finance* (2014).

[17] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. [n. d.]. CACTI 6.0: A Tool to Model Large Caches. In *HP Technical Report HPL-2009-85*.

[18] W. W. Peterson and D. T. Brown. 1961. Cyclic Codes for Error Detection. In *Proceedings of the IRE*, Vol. 49.

[19] Mohammad Samragh Razlighi, Mohsen Imani, Farinaz Koushanfar, and Tajana Rosing. 2017. LookNN: Neural Network with No Multiplication. In *Design, Automation & Test in Europe Conference & Exhibition*.

[20] Stephen E. Richardson. 1992. *Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation*. Technical Report.

[21] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based Approximation for Data Parallel Applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[22] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. SAGE: Self-tuning Approximation for Graphics Engines. In *IEEE/ACM International Symposium on Microarchitecture*.

[23] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. 2013. Approximate Storage in Solid-state Memories. In *IEEE/ACM International Symposium on Microarchitecture*.

[24] Yakun Sophia Shao and David Brooks. 2013. ISA-Independent Workload Characterization and its Implications for Specialized Architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software*.

[25] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. In *IEEE/ACM International Symposium on Computer Architecture*.

[26] Stelios Sidiroglou, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing Performance vs. Accuracy Trade-offs With Loop Perforation. In *ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering*.

[27] Sharad Sinha and Wei Zhang. 2016. Low-Power FPGA Design Using Memoization-Based Approximate Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 8 (2016).

[28] Tomoaki Tsumura, Ikuma Suzuki, Yasuki Ikeuchi, Hiroshi Matsuo, Hiroshi Nakashima, and Yasuhiko Nakashima. 2007. Design and Evaluation of an Auto-memoization Processor. In *The IASTED International Multi-Conference: Parallel and Distributed Computing and Networks*.

[29] James Tuck, Wonsun Ahn, Luis Ceze, and Josep Torrellas. 2008. SoftSig: Software-exposed Hardware Signatures for Code Analysis and Optimization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[30] Georgios Tziantzioulis, Nikos Hardavellas, and Simone Campanoni. 2018. Temporal Approximate Function Memoization. *IEEE Micro* 38, 4 (2018), 60–70.

[31] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. 2017. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design & Test* 34, 2 (2017), 60–68.

[32] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. 2015. Neural Acceleration for GPU Throughput Processors. In *IEEE/ACM International Symposium on Microarchitecture*.

[33] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C. Mowry. 2016. RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads. *ACM Transactions on Architecture and Code Optimization* 12, 4 (2016).

[34] Guowei Zhang and Daniel Sanchez. 2018. Leveraging Hardware Caches for Memoization. *IEEE Computer Architecture Letters* 17, 1 (2018), 59–63.