Temporal Approximate Function Memoization

Georgios Tziantzioulis, Nikos Hardavellas, and Simone Campanoni
Northwestern University

Improving the performance of applications is a core target of computer systems research and has led to the creation of various techniques. Among them is function memoization, an intuitive technique that, unfortunately, failed to live up to its promised potential. Traditional function memoization falls short mainly because it is input-based, meaning the system needs to pattern-match the current invocation's inputs with previously seen ones to decide which memoized output value to return. This is often computationally intensive and only suitable for a limited set of functions. However, function calls often exhibit temporal value locality on their output. We capitalize on this observation to create the first output-based function memoization technique. We demonstrate that compiler-directed output-based memoization reduces energy and runtime by 50 to 75 percent (2-4x speedup) at the cost of a relatively small degradation of the application's output quality.

Modern applications are written in a modular way, such that repeated computations are encapsulated into functions. Often, these functions are invoked with the same arguments and produce the same output. This has prompted researchers to consider function memoization,[1] the technique of keeping a list of input-output pairs from past executions and, instead of repeating a computation, examining that list upon a function's invocation (in other words, if the input has been seen before, the application will return the previously seen output value instead of executing the function code).

Though promising, traditional function memoization suffers from two significant drawbacks. First, it requires the output to depend only on the function arguments. Unfortunately, many functions compute their output also using global variables and heap elements that are hard to prove remain immutable across calls. Moreover, function memoization requires the pattern-matching of inputs, inducing high overhead.

We observe that consecutive invocations from the same call site tend to produce similar values, even if their inputs are different. This can be intuitively explained by looking at the properties of physical simulation. In physical modeling, the attributes of a single element (such as a particle) typically evolve in time following a smooth progression, which translates to a smooth transition in the output values of the functions that compute these attributes. This observation can extend to

Published by the IEEE Computer Society
0272-1732/18/$33.00 ©2018 IEEE

other domains beyond physical systems. For example, the predicted value of an option or swaption over time produces a relatively smooth line. This is not surprising; the average daily fluctuation of S&P 500 has been 0.8 percent since 1988, and only nine times since 1923 has its daily change been more than 10 percent. This temporal output locality can be exploited to avoid repeating computations that produce similar, but not necessarily identical, results.

We propose temporal approximate function memoization (TAF-Memo), a semantically relaxing compiler transformation that leverages this observation. TAF-Memo replaces the execution of consecutive function invocations by returning the last output actually computed. In essence, TAF-Memo converts the (typically) smooth line of consecutive output values into its quantized equivalent. Because no pattern-matching of the inputs is required, TAF-Memo avoids the computational overheads of conventional function memoization, making it suitable even for small function calls.

TAF-Memo trades computational accuracy for performance and energy efficiency. The application of TAF-Memo on a call site could cause the function's output to differ from the original one, thereby creating distortion to the program's final output. To reduce the output distortion, TAF-Memo is employed only when the relative difference of the output of two consecutively executed function calls is within a predefined range.

Overall, our contributions are:

- We observe that functions often exhibit strong temporal value locality on their outputs.
- We propose the first output-based function memoization technique.
- We demonstrate that compiler-directed output-based memoization can significantly reduce runtime and energy consumption at the cost of a small distortion of the application's output.

FUNCTION MEMOIZATION

Function memoization cannot be applied to any function. Rather, only functions with certain characteristics are amenable to it.

Pure Functions

Function memoization is restricted to pure functions. A function is pure if, when given the same input arguments, it always produces the same output without any side effects (such as mutation of global objects, output to I/O devices, and system calls).

Functions in procedural languages like C/C++ often compute their output by dereferencing pointers to memory or utilizing another global state. These common functions are impure. To measure their coverage, we restrict the criteria by which we identify pure functions in C/C++ programs to a set that modern compilers can tackle. We identify a pure function by validating the following:

- Its input arguments and return type are scalar,
- it uses no state other than its input arguments to compute the output,
- it has no side effects, and
- all functions invoked from that function also adhere to these constraints.

Pure functions account for only a limited fraction of the execution time in most of the considered applications (see Figure 1).[2] We need to relax the strict definition of purity to increase the potential coverage of memoization techniques.

Extended Pure Functions

Implementing an output-based function memoization, rather than an input-based one, allows us to extend the set of functions that it can be applied to beyond the pure ones. Specifically, we can relax two of the pure function criteria, leaving only the following ones:

- The function's return type is scalar,
- it has no side effects, and
- all functions invoked from that function also adhere to these constraints.

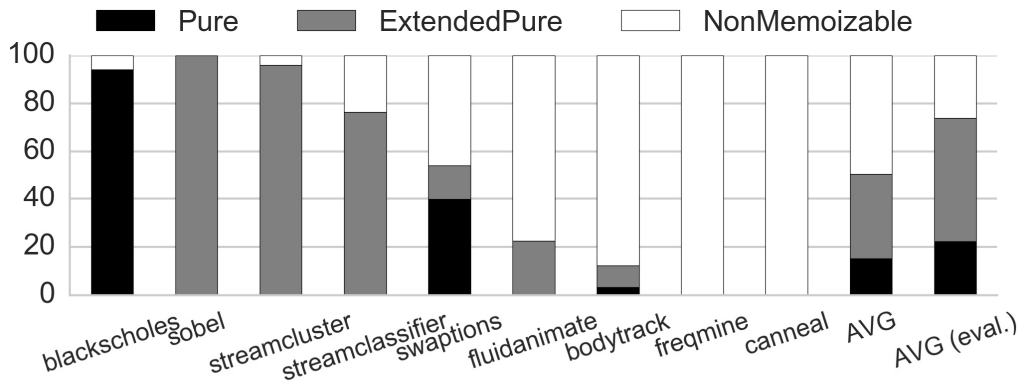

Figure 1. Execution time breakdown of all PARSEC 3.0 benchmarks that LLVM could compile. The AVG column presents the average breakdown across all benchmarks. The AVG (eval.) column presents the average breakdown across the benchmarks we consider in the remainder of this study (which exclude bodytrack, freqmine, and canneal, which have almost no pure or extended pure functions). Pure functions cover a small fraction of the total execution time, while extended pure functions achieve significantly higher coverage.

The extended definition expands the possible sources of data and only excludes functions with side effects or with non-scalar outputs. In particular, unlike pure functions, extended pure functions can have input arguments of any type, and they can use any available state (such as heap memory and global variables) to compute their output. Figure 1 shows that the extended pure function definition captures a significantly higher fraction of function calls. On some occasions, it even leads to 100-percent coverage, while the strict pure function definition has 0 percent (sobel). For output-based memoization to become feasible, however, the outputs of consecutive function invocations from a call site need to be similar.

Temporal Output Locality

We find that a significant portion of extended pure functions exhibit temporal output locality. That is, consecutive function calls from the same call site tend to return similar values. Note that temporal output locality is not necessarily a consequence of input locality. For example, a function that calculates the Euclidian distance between two points will always return the same result when called consecutively on the end points of a fixed-size line that moves in space (high output locality), regardless of its exact location (arbitrary coordinates, no input locality). In our work, we even observed cases of identical output with 31x difference in input values.

To quantify temporal locality, we examine the output of function calls from the same call site and calculate their relative standard deviation (RSD) within a sliding window W:

$$RSD(\vec{W}) = \frac{\sigma_{\vec{W}}}{|\mu_{\vec{W}}|}. \quad (1)$$

RSD is a metric of statistical dispersion used extensively in scientific and engineering fields. It is dimensionless, location-invariant, and scale-independent, and it readily facilitates comparisons

between arbitrary functions; the function with the lowest RSD also has the lowest output variability. Figure 2 presents the RSD for several call sites across applications. Each box shows the inter-quartile range (IQR), and the whiskers below and above the box extend by 1.5x the first and third quartiles, respectively. All points that are greater than the whisker values are considered outliers, and we do not plot them to avoid polluting the figure. The horizontal line on each boxplot signifies the median, and the red dot denotes the mean.

The RSD of output values is small for a significant portion of the investigated functions. This suggests that adjacent function calls from the same site might return similar values. This observation is not surprising, as many applications execute consecutive calls on similar data, which return similar output values. For example, an image processing application may perform transformations in small blocks of pixels, and usually adjacent blocks of pixels exhibit high similarity. A fluid-dynamic simulation may calculate the velocity of particles by calling a function for each particle. Typically, these calls are performed by sweeping the space in some order (not randomly), and nearby particles will likely have similar velocities.

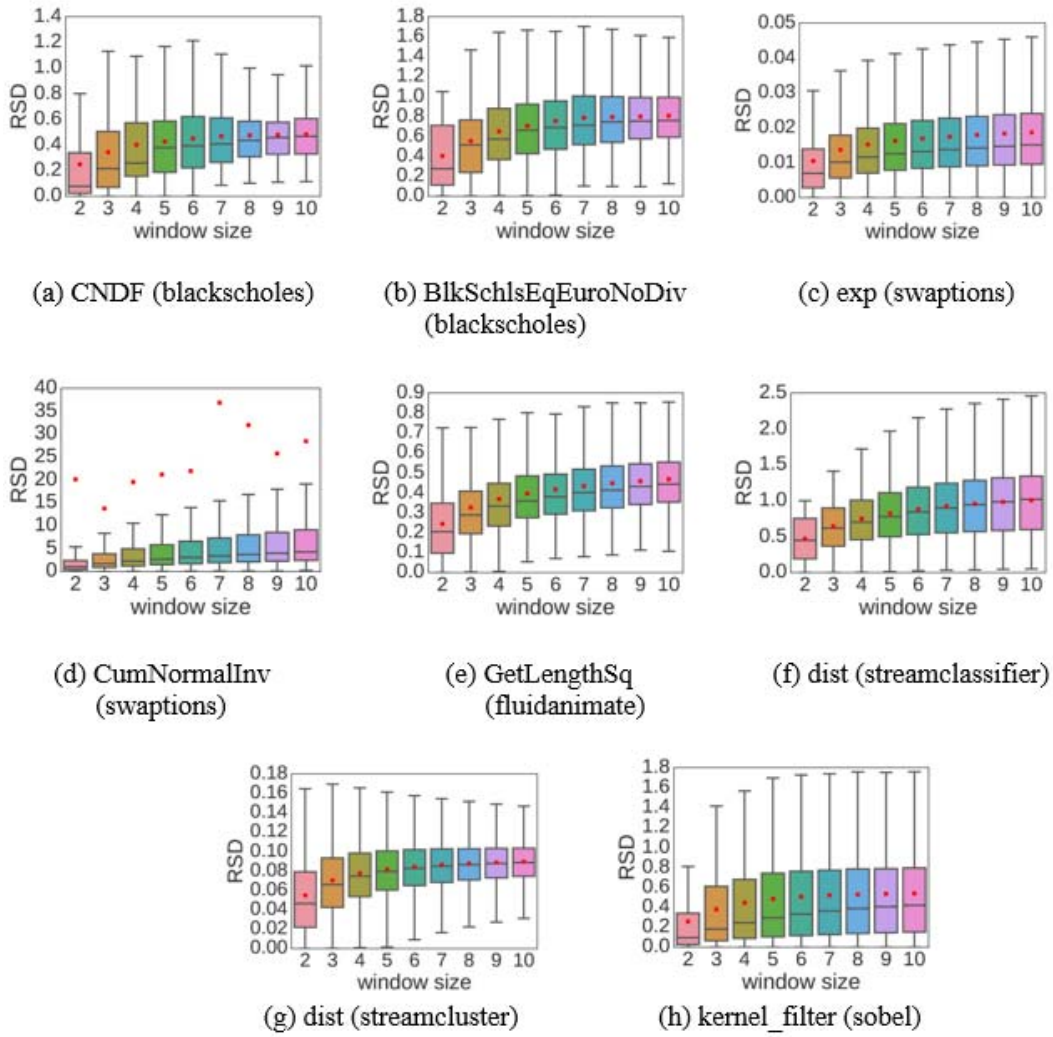

Figure 2. Relative standard deviation (RSD). Small RSD values suggest the function exhibits temporal output locality.

TEMPORAL APPROXIMATE FUNCTION MEMOIZATION

TAF-Memo is the first code transformation that memoizes function invocations based on their output history. The binary generated by TAF-Memo remembers the last output value of a function, and immediately returns it (instead of executing the function) upon identifying that the program has entered an execution segment with temporal output locality. The goal of TAF-Memo is to return a value that is arithmetically close to the one that the actual function execution would have computed.

TAF-Memo identifies temporal locality by checking the relative difference of two consecutive return values from the same call site. If the relative difference is below an acceptable threshold (memoization threshold), the function is considered to exhibit temporal locality. In that case, memoization is triggered for the following w invocations, where w is the memoization window.

To illustrate how TAF-Memo works, Figure 3 presents a snapshot of the output trace of a function. The trace shows that consecutive function calls do not produce wildly different outputs, but rather outputs that are often temporally clustered around similar values. TAF-Memo recognizes this and turns on memoization, which, in effect, quantizes the function output when the function exhibits stable-enough behavior.

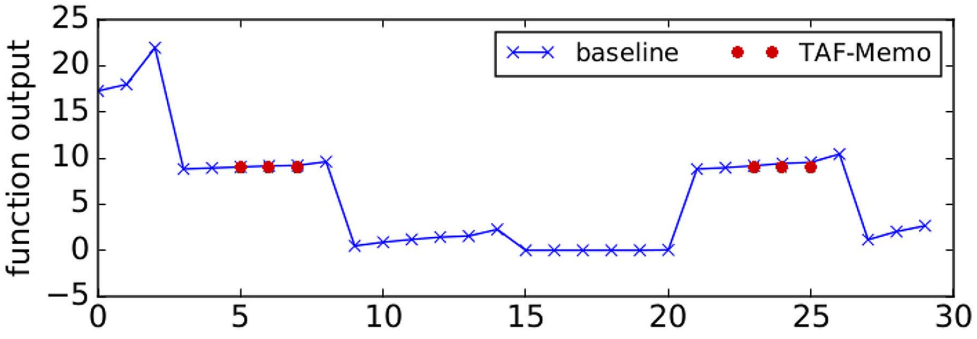

Figure 3. Timeline snapshot of precise and memoized output for BlkSchlsEqEuroNoDiv (blackscholes). TAF-Memo replaces function invocations with memoized values when it detects output stability. A tight threshold in this particular configuration prevents TAF-Memo from memoizing more frequently.

We implement TAF-Memo in two steps. First, the compiler automatically identifies the extended pure functions and their call sites, so they can be instrumented with pragmas denoting their memoization configurations (memoization threshold and window). Then, the compiler wraps the instrumented call sites within code that calculates the relative difference of two consecutive outputs. If the relative difference is below the memoization threshold, it turns on memoization for the next w invocations from that call site.

After identifying a candidate function, memoization can be applied in different granularities: global, per call site, or context-aware (such as by considering the call stack that leads to a call site). We choose to apply memoization at the call-site level, as we found that output variability at each call site can be significantly different. Context-aware memoization that considers the call stack may further improve the observed output value locality. However, tracking the calling context would also impose higher overhead, and thus we do not consider it in our design.

METHODOLOGY

We implement TAF-Memo by extending clang v3.8 and the iACT framework.[3] We evaluate TAF-Memo on an Intel Xeon E5-2695 v3 at 2.3 GHz running Red Hat Enterprise Linux Server r7.4 with Linux kernel v3.10.0.

Measurements

We dynamically analyze a workload by executing the binary iteratively in batches of three, until the collected execution times (R~) show low relative variability and meet the condition in Equation 2:

$$RSD(\vec{R}) = \frac{\sigma_{\vec{R}}}{\mu_{\vec{R}}} \leq 5\%. \quad (2)$$

For this analysis, we use the serial version of each application and profile their regions of interest (ROI) as identified in the original code. The ROI for sobel filter is the execution of function sobel_filtering, which applies the filter to all blocks.

We measure energy using a WattsUp Pro energy monitor, which measures the wall-plug power consumption of our server system at 1-second intervals from the AC side. We collect energy values only for the ROI of each application.

Applications

We evaluate TAF-Memo on a collection of benchmarks from PARSEC 3.0 and a sobel filter application.[2] We run all PARSEC applications with both simlarge and native inputs. We run the public implementation of sobel from the iACT repository with the baboon and Lena reference images as inputs (simlarge and native, respectively).[3] We run streamclassifier on the covtype data using execution arguments similar to streamcluster's execution.[4]

We evaluate distortion *d* using application-specific functions. In all cases, negative distortion values (the TAF-Memo version is more accurate) are clamped to zero (no distortion). If output data are corrupted or floating-point calculations result in *nan*, we set distortion to $+\infty$.

Blackscholes is a financial analysis application that uses the Black-Scholes equation to price a portfolio of stock options. We evaluate distortion as the average absolute relative difference of the baseline and approximate prices, similar to Misailovic *et al*:[5]

$$d = \frac{1}{N} \times \sum_{i=0}^{N} \frac{|P_{base}(i) - P_{approx}(i)|}{P_{base}(i)}. \quad (3)$$

Swaptions is a financial application that uses the Heath-Jarrow-Morton framework to price a portfolio of swaptions. We measure distortion using Equation 3, similar to blackscholes.

Streamcluster is a data-mining application that performs online clustering. We evaluate distortion as the relative difference between the Davies-Bouldin index of the baseline and approximate clustering, averaged across iterations:

$$d = \begin{cases} +\infty & \text{, if corrupted data} \\ \frac{1}{N} \times \sum_{i=0}^{N} (\max(\frac{DB(C_{approx}(i)) - DB(C_{base}(i))}{DB(C_{base}(i))}, 0) & \text{, otherwise} \end{cases}. \quad (4)$$

Streamclassifier shares the same code base as streamcluster, but it performs data classification rather than clustering, thereby exercising different code paths and accepting different inputs. We measure distortion by computing the difference between the b^3 metric of the approximate and baseline classification, averaged across iterations:

$$d = \frac{1}{N} \times \sum_{i=0}^{N} (\max(b^3(C_{approx}(i)) - b^3(C_{base}(i)), 0). \quad (5)$$

Fluidanimate simulates the flow of an incompressible fluid. We evaluate distortion as the Euclidean distance of the final position of a particle between the baseline and approximate executions, averaged across all particles, and normalized by the maximum length between two points of the baseline output:[6]

$$d = \begin{cases} +\infty & \text{, if } \exists i, nan \in \vec{p}_{approx}(i) \\ \dfrac{\sum_{i=0}^{N} (dist(\vec{p}_{base}(i), \vec{p}_{approx}(i))}{N \times \max\_length_{base}} & \text{, otherwise} \end{cases} . \quad (6)$$

Sobel filter identifies regions of an image with high spatial frequency (meaning, an edge). We measure distortion by evaluating the structural similarity index metric (SSIM):

$$d = 1 - SSIM(I_{base}, I_{approx}) . \quad (7)$$

The TAF-Memo Design Space

Each call site of a function memoizable by TAF-Memo is transformed using call-site-specific parameters (memoization window size and memoization threshold). TAF-Memo explores this design space by considering only call sites with runtime coverage of greater than or equal to 5 percent and reasonable memoization configurations (memoization threshold of less than or equal to 10 percent and window of less than or equal to 15). We additionally explore the impact of always memoizing within a window (with no threshold checking).

EVALUATION

Performance

The performance gained by TAF-Memo depends on several factors including:

- the application's tolerance to distortion,
- the fraction of execution time that the memoized function covers, and
- the overhead of TAF-Memo (we measured it at 25 to 30 processor cycles).

The interdependencies among these factors are complex and non-linear. Hence, we evaluate the performance impact of TAF-Memo across a large number of configurations, including a configuration with infinite memoization threshold. A configuration with infinite threshold leads to the removal of the threshold checking code. This "no threshold check" design point provides a bound on the maximum performance improvement that can be achieved by software output memoization, as the entire overhead of the threshold checking code is removed. It also represents an extreme point that relies solely on the temporal output locality of a specific call site and its tolerance to distortion. In effect, it continuously samples the function and replaces a neighborhood of results with a straight line, fully quantizing the function's output. We observe that as output memoization is applied more freely (meaning the memoization window and memoization threshold are increased), the program's execution time is reduced despite TAF-Memo's overhead.

Performance-Output Distortion Tradeoff

The blue circles in Figure 4 show the distortion and runtime (relative to the unmodified application) of different TAF-Memo configurations (varying memoization window size and threshold) on the simlarge input. The solid blue line shows the Pareto frontier on simlarge. Similarly, the purple crosses and line correspond to the native input. The dashed black and green lines signify the normalized baseline runtime (no memoization) and the performance limit of memoization assuming that all extended pure functions are memoized perfectly (using an oracle) with zero overhead.

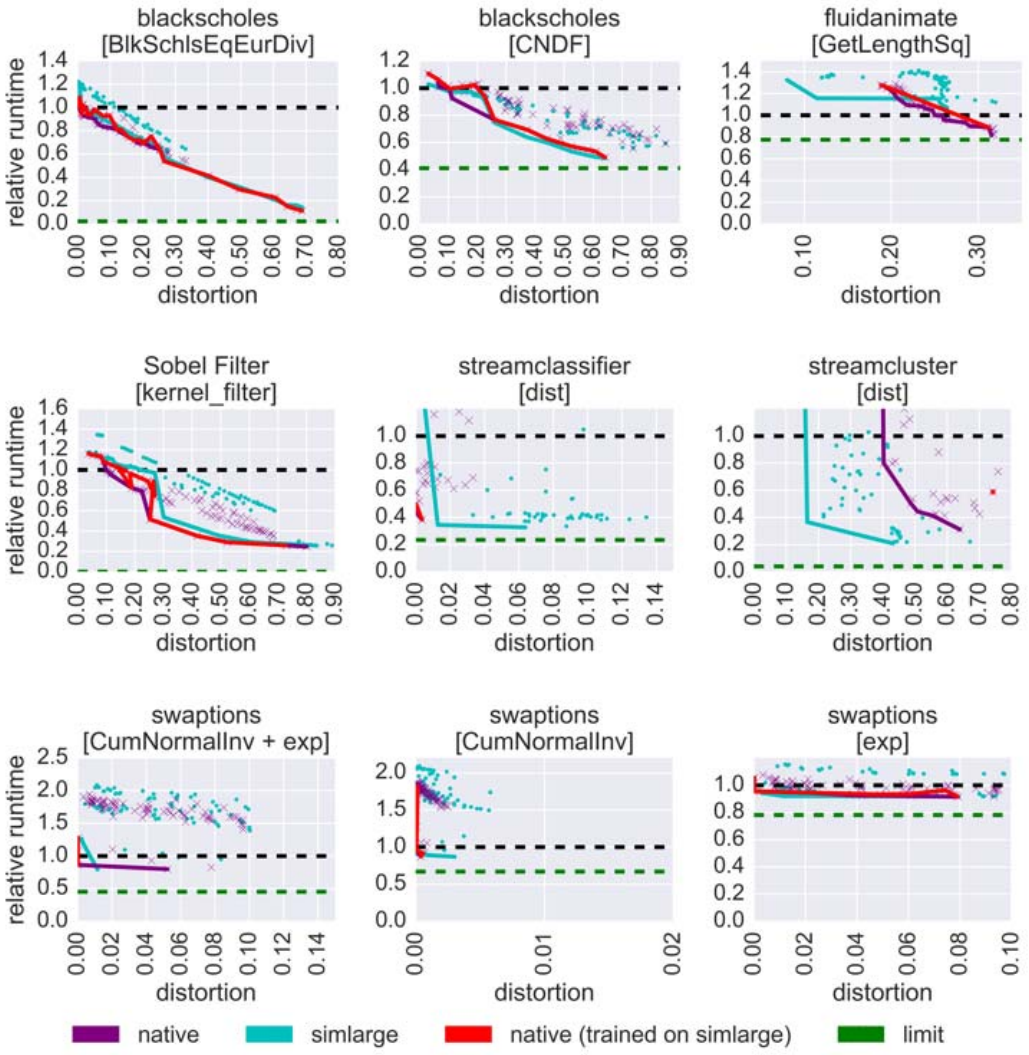

Figure 4: TAF-Memo distortion versus relative runtime. TAF-Memo achieves significant speedups with small distortion for most applications.

The Pareto frontier can be a very short line comprised of few points (sometimes just a couple). Streamclassifier on the native input has one configuration with very small runtime and distortion 0, and another with the lowest runtime and distortion ~0.005. These two configurations strictly subsume all others (which have higher runtimes and distortions). Thus, streamclassifier's native Pareto frontier is a very short purple line close to distortion 0 (and mostly occluded by the red line). Swaptions present a similar case.

Due to the high RSD of swaptions[CumNormalInv] configurations that attempt to memoize the function, speedups are produced only at high values of memoization threshold (and no threshold check) and window size. However, due to swaptions' distortion tolerance when memoizing CumNormalInv, output-based memoization is a viable option despite the high RSD.

All points of fluidanimate's Pareto frontier for simlarge fall above the black line that represents the baseline's runtime. The Pareto frontier for native inputs contains points with lower runtime, but they also exhibit higher distortion.

Finally, while sobel's distortion grows from 0 to 0.61, the output images appear almost identical. The $w = 3$ configuration has acceptable output and is 3.45x faster.

It is important to note that distortion is an application and use-specific metric. Only domain experts can pinpoint the appropriate metric and acceptable distortion level. Thus, 10-percent distortion has different meaning for different applications. Our compiler framework furnishes the user with Pareto frontiers that provide a range of runtimes and distortions, so users can choose a configuration that best fits their particular use-case (we leave auto-tuning to future work). Overall, through careful selection of configurations, TAF-Memo reduces runtime by 50 to 75 percent (2-4x speedup) with acceptable distortion.

Robustness on Training Inputs

To assess the robustness of TAF-Memo, we obtain the configurations that comprise the simlarge Pareto frontier and run them on the native input. The red line in Figure 4 plots the results. Most times, these configurations rank at, or near, the native Pareto frontier in the distortion-runtime design space. Thus, using a smaller (simlarge) input as a guide for selecting configurations for larger (unknown) inputs is sufficiently accurate. Streamcluster presents a special case, showing that TAF-Memo is an inherently unsafe technique with no guarantees. Out of the three points that comprise streamcluster's simlarge Pareto frontier, only one is a valid configuration when running on the native input; the figure shows it as a single red point at (0.75, 0.6).

Energy Savings

We measured the energy savings of workloads memoized with TAF-Memo and observed that they closely track the runtime savings; a workload's relative energy consumption is between -3 percent and +5 percent of its relative runtime (-0.2 percent on average). Thus, Figure 4 could also double as a distortion-versus-energy plot, with reasonable accuracy.

Comparison with Input-Based Memoization

We compare TAF-Memo with iACT, an input-based approximate function memoization technique.[3] iACT compares each incoming function argument against a list of previously seen values to determine whether they are within a predetermined range, and returns the memoized output only if all incoming arguments fuzzy-match in the same historical input vector. In contrast, TAF-Memo compares only the new and old values of the (one) scalar function output. We run iACT using Intel's implementation from GitHub on the same functions that TAF-Memo targets, and we vary its table size and fuzzy-matching error bounds. Figure 5 plots the results.

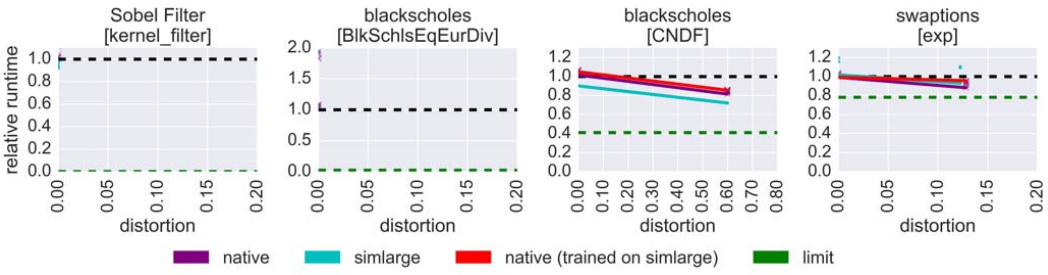

Figure 5. iACT distortion versus relative runtime. iACT achieves limited speedup, mostly due to the cost of input matching in software and the limited coverage of pure functions.

iACT is applicable only to blackscholes and swaptions, as the other applications do not have pure functions. To apply iACT elsewhere requires source-code modifications, such us refraining from using pointers to read data and global state, and passing all data accessed by a function as arguments by value. This, however, constitutes a significant departure from typical coding styles in procedural languages such as C/C++. The modified application is essentially a new one, invalidating any comparison with TAF-Memo on the original, unmodified workload. Thus, in this article, we evaluate memoization on unmodified applications only, and consider source-code

modifications out of scope. For illustration purposes, we modified sobel to implement the filter as a pure function. While iACT achieves 92.6-percent runtime (8-percent speedup) on sobel with minimal distortion, this result pales in comparison to TAF-Memo's 4x speedup. Overall, TAF-Memo's Pareto frontier (see Figure 4) strictly dominates iACT's (see Figure 5) across applications.

RELATED WORK

Memoization was proposed to improve performance.[7] While initially it was restricted to 1-1 matching of inputs to outputs, the approximate computing wave relaxed this constraint to increase memoization's potential benefits. However, memoization techniques maintained their focus on inputs for identifying memoization opportunities. To the best of our knowledge, TAF-Memo is the first output-based memoization technique. TAF-Memo exploits the temporal correlation of consecutive function outputs to trade off quality for runtime and energy reduction.

MCACHE provides hardware support for function memoization in caches.[8] Alvarez *et al*. propose ISA-level approximate memoization of instruction blocks.[9] In contrast, TAF-Memo requires no hardware support.

ATM performs input-based approximate task/function memoization in the runtime system, with pragmas driving task approximations.[10] iACT is a compiler framework for input-based approximate memoization.[3] iACT suffers from reduced coverage and high overhead, requiring hardware extensions to provide speedup even for functions with few inputs.[3] Both works perform input-based approximate memoization, while TAF-Memo is output-based.

Acar *et al*. combine adaptivity and memoization to obtain an incremental computation technique.[11] MemoizeIt is an iterative dynamic analysis that discovers methods amenable to memoization.[12] This work is related to ours, as it also expands the definition of memoizable functions.

CONCLUSION

Code optimizations need to target a large portion of the runtime to be effective. We show that output-based memoization has significantly higher coverage than input-based memoization. We show that function outputs often exhibit temporal value locality, and we harness it to design TAF-Memo—the first output-based approximate function memoization technique. TAF-Memo does not require hardware support and shows significant runtime and energy savings (50- to 75-percent reduction, 2-4x speedup) at the cost of a small quality loss in the application's output.

REFERENCES

1. A. Suresh et al., "Intercepting functions for memoization: a case study using transcendental functions," *ACM Transactions on Architecture and Code Optimization*, 2015.
2. C. Bienia, "Benchmarking modern multiprocessors," dissertation, 2011.
3. A.K. Mishra, R. Barik, and S. Paul, "iACT: a software-hardware framework for understanding the scope of approximate computing," *Workshop on Approximate Computing Across the System Stack*, 2014.
4. "UCI machine learning repository," Irvine University, 2016; https://archive.ics.uci.edu/ml/index.php.
5. S. Misailovic et al., "Quality of service profiling," *ACM/IEEE International Conference on Software Engineering*, 2010.
6. I. Akturk, K. Khatamifard, and U.R. Karpuzcu, *On quantification of accuracy loss in approximate computing*, Workshop on Duplicating, Deconstructing and Debunking, 2015.
7. D. Michie, "Memo functions and machine learning," *Nature*, 1968.
8. G. Zhang and D. Sanchez, "Leveraging hardware caches for memoization," *IEEE Computer Architecture Letters*, 2018.

9. C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Transactions on Computers*, 2005.
10. I. Brumar et al., "ATM: approximate task memoization in the runtime system," *IEEE International Parallel and Distributed Processing Symposium*, 2017.
11. U. Acar, G.E. Blelloch, and R. Harper, "Adaptive memoization," 2004.
12. L. Della Toffola, M. Pradel, and T.R. Gross, "Performance problems you can fix: a dynamic analysis of memoization opportunities," *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

ABOUT THE AUTHORS

Georgios Tziantzioulis has a PhD in computer engineering from Northwestern University. His research interests include energy- and power-efficient computing, computer architecture, and compilers. He is a member of ACM. Contact him at georgios@u.northwestern.edu.

Nikos Hardavellas is an associate professor in the Department of Electrical Engineering and Computer Science at Northwestern University. His research interests include parallel systems, computer architecture, silicon photonics, memory systems, approximate computing, design for dark silicon, and data-oriented software architectures. Hardavellas has a PhD in computer science from Carnegie Mellon University. He is a member of the IEEE and ACM. Contact him at nikos@northwestern.edu.

Simone Campanoni is an assistant professor in the Department of Electrical Engineering and Computer Science at Northwestern University. His main research area is compilers, with special interest in its relation with computer architecture, runtime systems, operating systems, and programming languages. Campanoni has a PhD in computer engineering from Politecnico di Milano, and he was a postdoctoral student at Harvard University working with Professors David Brooks and Gu-Yeon Wei. Contact him at simone.campanoni@northwestern.edu.