Appeared in Proc. 12th IEEE Int. Working Conference on Source Code Analysis and Manipulation (SCAM), 2012.

Optimizing Expression Selection for Lookup Table Program Transformation

Chris Wilcox, Michelle Mills Strout, James M. Bieman
Computer Science Department
Colorado State University,
Fort Collins, Colorado, USA 80523–1873
Email: wilcox,strout,bieman@cs.colostate.edu

***Abstract*—Scientific programmers can speed up function evaluation by precomputing and storing function results in lookup table (LUTs), thereby replacing costly evaluation code with an inexpensive memory access. A code transform that replaces computation with LUT code can improve performance, however, accuracy is reduced because of error inherent in reconstructing values from LUT data. LUT transforms are commonly used to approximate expensive elementary functions. The current practice is for software developers to (1) manually identify expressions that can benefit from a LUT transform, (2) modify the code by hand to implement the LUT transform, and (3) run experiments to determine if the resulting error is within application requirements. This approach reduces productivity, obfuscates code, and limits programmer control over accuracy and performance. We propose source code analysis and program transformation to substantially automate the application of LUT transforms. Our approach uses a novel optimization algorithm that selects Pareto optimal sets of expressions that benefit most from LUT transformation, based on error and performance estimates. We demonstrate our methodology with the Mesa tool, which achieves speedups of 1.4-6.9× on scientific codes while managing introduced error. Our tool makes the programmer more productive and improves the chances of finding an effective solution.**

***Keywords*-lookup table; performance optimization; error analysis; code generation; scientific computing; memoization; fuzzy reuse; Mesa tool**

I. Introduction

Math-intensive scientific codes are often performance limited by elementary functions such as *sin*, *exp*, and *log* that consume many CPU cycles. For example, calling *cos* in the math library is 35-45× slower than floating-point addition on current architectures. A lookup table (LUT) improves the performance of function evaluation by precomputing and storing function results, thereby allowing replacement of subsequent evaluations with less expensive memory lookups [1], [2]. Practical concerns limit the size and accuracy of LUT results, so LUT methods introduce error to gain performance.

Hardware designers have long used LUTs to improve elementary function performance [3], [4]. Dedicated memory is expensive in hardware, but linear interpolation and polynomial reconstruction can provide high accuracy with small LUTs [5]. Scientific programmers use similar LUT techniques to expedite function evaluation. For example, the Fastest Fourier Transforms in the West (FFTW) libraries incorporate cosine and sine tables to achieve "significant reductions in computation time" [6]. As another example, the Rapid Radiative Transfer Model (RRTM) software uses LUT transforms for the exponential and tau functions, yielding a 1.75× improvement on code that consumes 25% of the execution time of a global climate model [7].

Lacking a methodology and tools, scientific programmers typically transform source code by manually inserting LUT code. The *ad hoc* nature of such transforms makes it difficult to predict and control application accuracy and performance. Developing performance code by hand is also a substantial effort that impacts programmer productivity [8] and can obfuscate application code [9]. We propose a methodology and algorithms that make LUT development more efficient and effective. We demonstrate our approach with Mesa [10], a tool that uses program analysis and transformation to substantially automate the application of LUT transforms.

Our interest in LUT methods started with application code that we wrote for the Small Angle X-ray Scattering (SAXS) project [11]. We reduced the execution time of our original code by manually incorporating a LUT transform for the dominant calculation. Through lengthy experimentation we achieved an application speedup of 6-7× while maintaining reasonable accuracy. However, the manual process was inefficient and provided limited control over accuracy and performance.

Table I shows the results achieved using Mesa to performance-tune the SAXS code and five additional scientific applications. The performance speedup, calculated as the ratio of the original time divided by the optimized time, varies from 1.4-6.9×. The maximum error shown is the difference between the output of the original and optimized applications. The evaluation of these applications is described in Section VI.

In prior papers, we described versions 1.0 [12] and 1.1 [13] of Mesa. Mesa 1.0 lacked support for domain profiling and linear interpolation and required a separate specification of candidate expressions outside the source code. A programmer using this version had to (1) man-

Table I
RESULTS OF LUT OPTIMIZATION WITH MESA.

(Intel Core 2 Duo, E8300, family 6, model 23, 2.83GHz, single core)

Application Name	Original Time	Optimized Time	Performance Speedup	Maximum Error	Memory Usage
Saxs Scattering (discrete)	196.2s	29.0s	6.8X	4.06 x 10^{-3}%	4MB
Saxs Scattering (continuous)	10.1s	2.5s	4.0X	1.48 x 10^{-4}%	4MB
Stillinger-Weber (simulation)	14.6s	10.4s	1.4X	2.91 x 10^{-2}%	1MB
Neural Network (logistics)	8.0s	3.6s	2.2X	8.70 x 10^{-2}%	4MB
Neural Network (hypertan)	10.9s	3.9s	2.8x	6.30 x 10^{-1}%	4MB
PRMS (slope aspect)	242ns	56ns	4.3X	8.21 x 10^{-6}%	4MB
PRMS (solar radiation)	13.7s	6.1s	2.2X	2.97 x 10^{-4}%	4MB

ually identify candidate expressions and their domains, (2) specify the expressions and constituent variables in a file, (3) run Mesa to generate code, and (4) manually integrate the resulting code back into the application. With Mesa 1.1 a user could apply LUT transformation to expressions in C and C++ source code by identifying the relevant statements with pragmas, thereby operating directly on application source code. The dissertation by Wilcox [14] includes a comprehensive description of our research on automated LUT program transformation and the development of Mesa.

This paper introduces our approach to optimizing the selection of expressions for LUT transformation to maximize the benefits and minimize costs. The current version of Mesa is a source-to-source translation tool that automatically finds and evaluates candidate expressions through source code analysis and program transformation. Mesa finds the most effective set of LUT transforms by building and solving a multi-objective LUT optimization problem. Our goal is to identify sets of expressions that provide the most performance benefit with the least impact on accuracy. Error analysis and a performance model provide the criteria for choosing between potential LUT transforms.

The contributions of this research are as follows:

- A comprehensive methodology and tool support to apply LUT transforms to source code,
- error estimation and a performance model to characterize LUT transforms,
- a novel LUT optimization technique that maximizes performance and minimizes error, and
- case studies that demonstrate the effectiveness of our methodology and tool.

II. BACKGROUND

We present LUT terminology using example LUT data for $exp(x)$, as shown in Figure 1. We restrict the input values to $0 \leq x \leq 1$ so the LUT *domain* is limited to $[0, 1]$. The exponential function is $f(x)$ and its approximation via the LUT is $l(x)$. Tables are created by partitioning the domain into uniform intervals, and assigning a LUT *entry* for each interval. The number of intervals is the LUT *size*, which is 16 for our example. The interval width represents the LUT *granularity*. Equation 1 shows the relationship between domain, granularity, and size, from which we compute a granularity of 0.0625 for both tables.

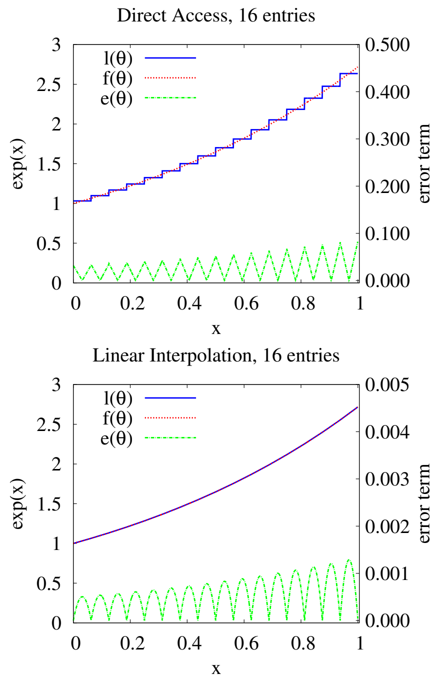

Figure 1. Table Data for Exponential Function

$$\text{Granularity} = \text{Domain}/\text{Size} \tag{1}$$

The performance and accuracy of a LUT transform depend on how LUT data is sampled. Common sampling methods are *direct access* and *linear interpolation*. Direct access simply finds the interval that contains the input value and returns its LUT entry. Linear interpolation selects the two closest LUT entries and combines them according to their respective distances from the exact input. Figure 1 shows direct access and linear interpolation sampling for a table with 16 entries.

LUT *error* is the absolute difference between a function and its LUT approximation. In Figure 1 the error is the distance between $f(x)$ and $l(x)$, plotted separately as $e(x)$. Each step in $l(x)$ represents a single LUT entry. For direct access the maximum absolute error is 0.0836; for linear interpolation the maximum absolute error is 0.0013. The equations for computing the maximum error are given in Section IV. The graphs illustrate that linear interpolation yields much smaller error terms (we reduced the error scale as shown on the right side of the graph by two orders of magnitude for linear interpolation to make the smaller error visible).

Regardless of the sampling method, the purpose of a LUT transform is a reduction in execution time that we refer to as LUT *benefit*. We compute the benefit as the difference between the cost of expression evaluation and the cost of LUT access. Linear interpolation has less benefit because of the extra LUT access and computation, and is therefore 1.5-1.8× slower than direct access on current architectures.

III. Defining the LUT Optimization Problem

Following current practice a programmer must explicitly identify candidate expressions and manually apply LUT transforms. The primary contribution of this paper is the automatic selection of expressions for which LUT transforms are most effective, meaning they provide the highest performance benefit with the least error. Our algorithm (1) enumerates expressions within the subroutines specified by the programmer, (2) estimates the error and performance impact of applying a LUT transform to each of these expressions, and (3) constructs and solves a numerical optimization problem that selects Pareto optimal sets of the most beneficial expressions from the enumeration. Pareto optimal is a term used in game theory to describe solutions with the most favorable tradeoff between multiple objectives.

To use the Mesa tool, the programmer identifies the optimization scope by inserting pragmas above subroutine definitions (method definitions in C++). Mesa considers LUT transforms for expressions in the bodies of these subroutines. Mesa can optimize any number of subroutines in a single pass, however, the cost of analysis increases quickly as subroutines are added. Within the specified subroutines, Mesa currently enumerates only expressions that contain elementary function calls, and the optimization solution is based on a model of the error and benefit for each component expression.

In optimization terms, a solution is a set of expressions (X_i) for which we wish to minimize the sum of errors (E_i) and maximize the sum of benefits (B_i). The error is computed based on an equation that takes into account the LUT size (S_i), domain (D_i), and maximum slope (M_i) of the function within the domain. The optimization problem also constrains LUT data to fit within the cache size specified by the programmer. Figure 2 shows the mathematical definition of the LUT optimization problem.

The complexity of the LUT optimization problem is exponential, with up to 2^N solutions for N expressions. To reduce the computational complexity, we have developed an algorithm for culling the solution space when N is large. Both the error equation and performance benefit are non-linear functions, and the problem mixes integer and floating-point numbers. The LUT optimization problem has competing objectives, thus it produces multiple optimal solutions. Our algorithm discards suboptimal solutions, leaving the programmer to select from the much smaller set of Pareto optimal solutions, which are ranked by accuracy and performance.

INPUT VALUES
D_i: real - input domain for expression
M_i: real - maximum slope for expression
B_i: real - potential benefit for expression
CS: integer - cache size

VARIABLE DEFINITIONS
X_i: boolean - expression selector
S_i: integer - table size ($0 < S_i < CS$)
E_i: real - computed as $(D_i/S_i) * (M_i/2.0)$

PROBLEM OBJECTIVES
maximize TotalBenefit = $\sum((X_i?1{:}0)*B_i)$ - maximize benefit
minimize TotalError = $\sum((X_i?1{:}0)*E_i)$ - minimize error

CACHE CONSTRAINT
$\sum S_i = CS$ - use entire cache

INTERSECTION CONSTRAINTS
$X_i + X_j \leq 1$, for intersecting X_i and X_j

Figure 2. Mathematical definition of optimization problem.

IV. Solving the LUT Optimization Problem

As previously stated, the LUT optimization problem is mixed-integer and non-linear. Optimization frameworks exist that handle these attributes, however, our problem is unusual because the solver must simultaneously select expressions and allocate cache for them. As a result we have not yet been able to use existing solvers including Couenne, Bonmin, and MINLP [15]. Our solution to the LUT optimization problem is divided into two parts. The first part allocates cache memory for the LUT data associated with each solution. We call this *local optimization*. The second part selects the set of LUT transforms that are Pareto optimal. We call this *global optimization*. We now present several algorithms that we use to construct and solve the LUT optimization problem. First we describe estimation of the error introduced by a LUT transform. Next we describe the performance model that estimates the potential benefit of a LUT transform. Finally we address our methodology for solving the local and global optimization problems.

A. Error Analysis

Error analysis provides an estimate of the maximum error for individual LUT transforms, which we use as a proxy for application error. A better solution would be to characterize the propagation of the introduced error over the entire application, but this is a complex numerical analysis problem. Techniques such as interval analysis can bound introduced error through a sequence of operations [16], but this is beyond the scope of our work. Our methodology instead provides support for empirically measuring application error for representative inputs.

Equation (2) shows the error equation for direct access. The maximum error depends on LUT size, input domain, and the function slope over the domain. The domain is known so we compute only the maximum slope, deferring the LUT size computation until local optimization. The error is inversely proportional to LUT size, so a $2\times$ increase in size decreases error by $2\times$.

$$\text{MaxError} = (\text{Domain}/\text{Size}) * (\text{MaxSlope}/2) \qquad (2)$$

We find the maximum slope analytically or numerically. For elementary functions we analyze the function derivative to determine the maximum slope over the domain. Alternatively we can use numerical traversal of the domain to find the maximum slope for arbitrary functions. We have experimented with exhaustive traversal, stochastic sampling, and boundary sampling, and found that the latter has the best performance.

Equation (3) shows the error computation for linear interpolation [17]. The maximum delta is the maximum change in the function slope over the domain because the error for linear interpolation is related to the curvature of the function over the interval. Because the error term contains the square of the granularity, a $2\times$ increase in size decreases the error by $4\times$.

$$\text{MaxError} = (\text{Domain}/\text{Size})^2 * (\text{MaxDelta}/8) \qquad (3)$$

B. Performance Modeling

Performance modeling estimates the benefit associated with each LUT transform based on the execution time of arithmetic operators, elementary functions, and memory access. Mesa incorporates a benchmark that measures the average performance of these operations for direct access and linear interpolation.

Equation 4 shows the performance model, which uses a count of the arithmetic operators and elementary function calls per expression. These counts are multiplied by the cost of each operation, and the LUT access time is subtracted. For linear interpolation, the result is divided by the relative performance compared to direct access. This result is an estimate of the benefit of replacing the expression with a LUT access.

$$\begin{aligned}\text{Benefit} = ((&\text{Cost(Op)} * \text{Count(Op)}) \\ &-\text{Cost(Access)}) * \text{Frequency}\end{aligned} \qquad (4)$$

For example, consider the optimization of an expression with a cosine call. On our test system the sine call takes 45ns. Subtracting 7.4ns for the LUT access gives a savings of 37.6ns per execution. We multiply this by the call frequency to get the total expression benefit. For example, a call frequency of 10^8 for the same expression would yield a benefit 37.6ns$\times 10^8$ or 3.76s. When the expression is optimized with linear interpolation, the benefit would be reduced by the relative factor, which is $\sim 1.8\times$ on our test system.

We do not expect the performance model to predict exact timing. Instead, it establishes the relative performance to allow comparison of solutions. Even so, we are often within 5-10% when estimating performance speedup for individual elementary functions. When optimizing code with multiple elementary function calls, the model tends to overestimate benefit, mainly because it does not model compiler optimizations. However, our case studies show that our estimates are usually within $\sim$1-2$\times$ of the actual performance benefit.

C. Local Optimization

The purpose of local optimization is to find the optimal allocation of cache resources for the set of expressions within each solution. Equation (5) shows the closed-form formula that computes LUT sizes for each LUT transform to minimize solution error [14]. A variant of the formula directly computes the maximum error for a single LUT access, as shown in Equation (6). However, our algorithm only has to compute one or the other, since size and error are related through Equations (2) and (3).

$$S_i = CS / \left(\sum_{j=1..n} \sqrt{(M_j D_j / M_i D_i)} \right) \qquad (5)$$

$$E_i = M_i D_i \left(\sum_{j=1..n} \sqrt{(M_j D_j / M_i D_i)} \right) / CS \qquad (6)$$

D. Global Optimization

We find Pareto optimal solutions by sorting the solutions by estimated error and performance, and selecting solutions that lie on the convex hull. To find the convex hull we use a modified Graham Scan algorithm [18]. Figure 3 shows the Pareto chart for the example code presented in Section V. Solution error is on the x-axis and solution benefit is on y-axis. To make the chart we plotted both the Pareto optimal (circles) and suboptimal (triangle) solutions. The Pareto solutions are joined by a line called the Pareto curve. The optimal solutions C0 through C4 lie above and the left of suboptimal solutions, because they provide more performance with the same or less error.

As demonstrated in Figure 3, the Pareto curve gives insight into the effectiveness of the LUT transforms that are combined into Pareto solutions. LUT transforms that contribute lots of benefit but little error appear on the steep left side of the curve. LUT transforms that introduce lots of error but little performance appear on the flatter right side of the curve. The programmer can examine the Pareto curve and decide how much benefit is possible for the amount of error they can tolerate.

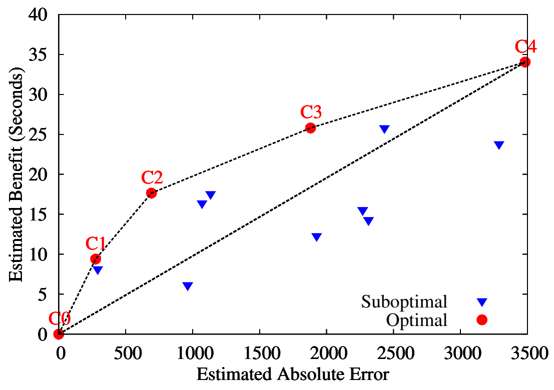

Figure 3. Pareto chart for example program.

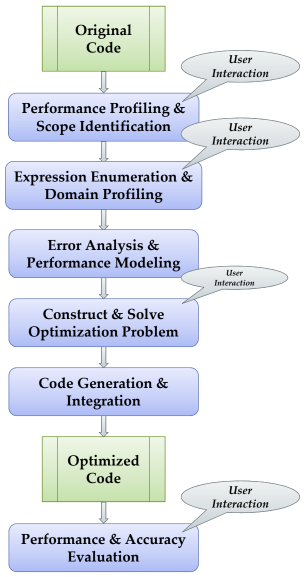
Original Code
User Interaction
Performance Profiling & Scope Identification
User Interaction
Expression Enumeration & Domain Profiling
Error Analysis & Performance Modeling
User Interaction
Construct & Solve Optimization Problem
Code Generation & Integration
Optimized Code
User Interaction
Performance & Accuracy Evaluation

Figure 4. Methodology for automated optimization.

V. MESA TOOL

We have implemented error and performance models and optimization algorithms from Section IV in a standalone tool called Mesa. Mesa incorporates ROSE compiler infrastructure [19], [20] to support static and dynamic source analysis and program transformation on C/C++ application code. Figure 4 shows the automated optimization methodology implemented by Mesa.

Figure 5 shows the example program that we use to explain our methodology and tool. The performance of this code is limited by the computation of elementary functions, and the domain of input values for the program enables us to satisfy accuracy requirements while fitting in cache memory. We now describe the six stages of the methodology displayed in Figure 4 in terms of the example code.

A. Stage 1: Profile Performance and Identify Scope

In the first stage the user *profiles performance* by manually running a tool such as **gprof** to find the most costly subroutines. For the example code, the subroutine ScatterSample consumes >90% of the execution time. The user inserts the pragma shown in Figure 5 to identify the *optimization scope* as the body of this subroutine. The pragma causes Mesa to evaluate expressions in statements S35 through S48 for potential LUT transforms.

```
#pragma LUTOPTIMIZE
double ScatterSample(Sample sample, ...
{
S35     double dProduct;
S36     double dSum0 = 1.0;
S37     double dSum1 = 1.0;
S38
S39     // Iterate geometry
S40     for (int j = 0; j < vGeometry.size(); j++)
S41     {
S42         dProduct = (sample.x * vGeometry[j].x) ...
S43         dSum0 += exp(dProduct) + sin(dProduct);
S44         dSum1 += exp(dProduct) + cos(dProduct);
S45     }
S46
S47     // Return answer
S48     return dSum0 * dSum0 + dSum1 * dSum1;
}
```

Figure 5. Source listing for example program.

Table II
ENUMERATED EXPRESSIONS FOR EXAMPLE CODE.

Expression Identifier	Expression Description	Statement Identifiers	Input Variables
X0	exp(vProduct)	S43	dProduct
X1	sin(dProduct)	S43	dProduct
X2	exp(vProduct) * sin(dProduct)	S43	dProduct
X3	exp(vProduct)	S44	dProduct
X4	cos(dProduct)	S44	dProduct
X5	exp(vProduct) * cos(dProduct)	S44	dProduct

B. Stage 2: Enumerate Expressions and Profile Domain

In the next stage Mesa *enumerates expressions* that are candidates for a LUT transform. Our enumeration algorithm extracts expressions from each statement in the scope, and rejects those that do not match our criteria. For example, expressions must contain one or more elementary function calls, since these functions are the focus of our methodology. Enumeration includes individual elementary function calls and more complex expressions that combine elementary function calls with arithmetic operators. Complex expressions are considered because additional performance gains can occur when an expression with multiple elementary functions is handled by a single LUT.

Table II lists the enumerated expressions for statements S43 and S44 in Figure 5, labeled as X0 to X5. For each expression we show the identifier, syntax, statements, and input variables. Extracted expressions can overlap other expressions in the same statement. For example, expressions X0 and X1 are contained in X2 so they cannot be optimized simultaneously without

```
Original:    X0 ∩ X2, X1 ∩ X2, X3 ∩ X5, X4 ∩ X5
Introduced: X0 ∩ X6, X3 ∩ X6
Inherited:   X2 ∩ X6, X5 ∩ X6
```

Figure 6. Intersection constraints for example program.

Table III
INPUT DATA FOR OPTIMIZATION PROBLEM.

Expression Identifier	Expression Description	Statement Identifiers	D_i	M_i	B_i
X0	exp(dProduct)	S43	**2.44**	**3.31**	**6.15s**
X1	sin(dProduct)	S43	**2.44**	**1.00**	**8.15s**
X2	exp(dProduct) + sin(dProduct)	S43	**2.44**	**3.67**	**16.40s**
X3	exp(dProduct)	S44	**2.44**	**3.31**	**6.15s**
X4	cos(dProduct)	S44	**2.44**	**0.95**	**9.40s**
X5	exp(dProduct) + cos(dProduct)	S44	**2.44**	**2.38**	**17.65s**
X6	exp(dProduct)	S43, S44	**2.44**	**3.31**	**12.30s**

causing redundant computation. Mesa maintains *intersection constraints* to prevent overlapping expressions from being combined into a solution. Figure 6 lists the original intersection constraints for the example code.

During this stage Mesa combines similar expressions via *expression coalescing*. Coalescing saves resources by sharing LUT code and data between two or more identical expressions. For the example code, coalescing creates a new expression X6 that combines the identical exponential calls in X0 and X3. The coalesced expression adds new intersection constraints and inherits old ones, as shown in Figure 6

After expression enumeration Mesa initiates *domain profiling*, where it instruments the source code to capture runtime data such as the domain of input variables and the execution frequency of statements. The user must compile and run the instrumented code with representative data sets to perform the dynamic analysis. At program completion the instrumented code stores input domain and call frequency information for the subsequent stages.

C. Stage 3: Error Analysis and Performance Modeling

Mesa applies the algorithms for error analysis and the performance model from Section IV to compute the maximum slope and performance benefit. Table III shows the candidate expressions and their estimated error and benefit parameters, which is the input data for the optimization problem. D_i lists the extent of the input domain, M_i lists the maximum slope of the function, and B_i lists the potential benefit in seconds.

D. Stage 4: Optimization Problem

Mesa builds and solves the LUT optimization problem. Figure 7 is a partial listing from Mesa that shows the optimization result, ending with the presentation of Pareto optimal solutions to the programmer. Mesa displays the number of possible solutions as 2^7 or 128 for the 7 expressions extracted from the example code, and the number of actual solutions as 29 after intersection culling. The $4\times$ decrease in solutions is not unusual when expression coalescing is enabled because of introduced intersection constraints. From these, Mesa finds five Pareto optimal solutions numbered from C0 to C4. Mesa shows these to the programmer along with the corresponding error and benefit estimates. The listing gives absolute error, benefit in nanoseconds, and lists the expressions that comprise each solution.

```
128 solutions (possible)
29 solutions (actual)
5 solutions (pareto optimal)
Solution    Error        Benefit      Optimizations
C0          0.000e+00    0.000e+00
C1          2.759e+02    9.400e+09    X4
C2          6.921e+02    1.765e+10    X5
C3          1.881e+03    2.580e+10    X5,X1
C4          3.483e+03    3.405e+10    X5,X2
Select solution: 4
X5 Di=2.44 Mi=2.38 Ei=1.553e+03 Bi=1.765e+10
   Si = 1869833 (1826KB)
X2 Di=2.44 Mi=3.67 Ei=1.930e+03 Bi=1.640e+10
   Si = 2324471 (2270KB)
Mesa 2.0: Generating optimized code
Realizing X5 in statement S44
Realizing X2 in statement S43
Mesa 2.0: Optimization completed.
```

Figure 7. Mesa optimization result for example code.

Figure 7 shows that the solutions range from C0, which has zero benefit and error, to C4, which has the maximum benefit. The latter combines expressions X2 ($exp(dProduct) * sin(dProduct)$) and X5 ($exp(dProduct) * cos(dProduct)$), each of which contains two elementary function calls. The coalesced exponential call X6 ($exp(dProduct)$) does not appear in the solution so expression coalescing has not helped.

The listing further shows user selection of solution C4, which causes Mesa to replace expressions X2 and X5 with LUT accesses. The X2 expression receives a cache allocation of 2270KB. This is larger than the 1826KB allocation of X5 because X2 has a larger maximum slope. The global optimization has identified the most effective LUT transforms based on the error and performance model, for which the local optimization has allocated the ideal amount of cache.

E. Stage 5: Code Generation and Integration

After the programmer selects a solution, Mesa realizes the code for the specified set of LUT transforms, including the LUT data, constructor, destructor, initialization code, table access, and original function. Mesa then integrates the generated code into the application and replaces original expressions with LUT access calls. An optimized version of the application code is written to the file specified on the command line. The programmer rebuilds the application using the normal build process, and the resulting executable should behave in an identical manner to the original program, except for differences in performance and accuracy.

```
// Start of code generated by Mesa, version 2.0
// LUT constants
const double X5_lower = -1.2461340000e+00;
const double X5_upper =  1.1962890000e+00;
const double X5_gran  =  5.2249019600e-06;
class CLut {
public:
  // LUT Constructor
  CLut() {
    double dIn, dOut;
    for (dIn=X5_lower; dIn<=X5_upper; dIn+=X5_gran) {
      dOut = X5_orig(dInput+(X5_gran/2.0));
      X5_data.push_back(dOut);
    }
  }
  // LUT Destructor
  ~CLut() {
    X5_data.clear();
  }
  // LUT function for expression
  float X5_lut(float X5_param) {
    X5_param -= X5_lower;
    int uIndex = (int) (X5_param*(1.0/X5_gran));
    return(X5_data[uIndex]);
  }
  // Original expression
  double X5_orig(double dProduct) {
    return (exp(dProduct)+cos(dProduct));
  }
private:
  // LUT data structures
  std::vector<float> X2_data;
};
// Object instantiation
CLut clut;
// End of code generated by Mesa, version 2.0

// Expressions replaced by Mesa
S43 dSum0 += clut.X2_lut(dProduct);
S44 dSum1 += clut.X5_lut(dProduct);
```

Figure 8. Optimized code generated by Mesa.

Figure 8 shows a partial listing of the code generated by Mesa. To save space, we show only the code for the X5 expression $(exp(dProduct) * cos(dProduct))$ and the modifications to the original subroutine.

F. Stage 6: Accuracy and Performance Evaluation

In the last stage, the programmer evaluates the benefit and accuracy of the optimized program version against the original version. We compute performance speedup as the original execution time divided by the optimized execution time. Accuracy is evaluated as the absolute or relative deviation from original program output.

The optimized version of the program shows a $8.1\times$ speedup over the original version. We compute a relative error of $1.13{\times}10^{-4}\%$, based on the accumulation of return values from the ScatterSample subroutine.

The evaluation stage completes when the programmer has the accuracy and performance information needed to evaluate whether the optimization is worthwhile. The programmer can accept the optimization and use the code generated by Mesa, run Mesa again and select another solution, or revert to the original code. A Mesa parameter specifies the selection of a solution so that the iterative process just described can be scripted.

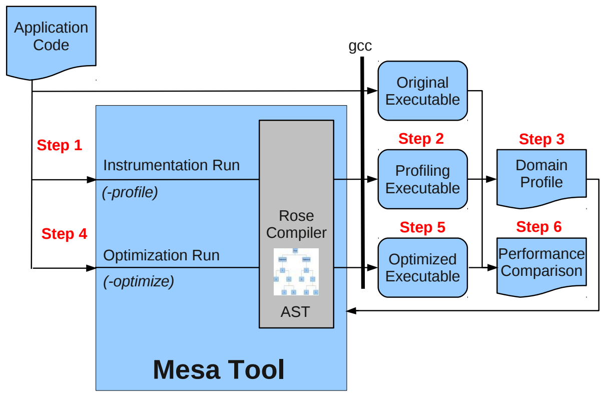

Figure 9. Diagram of Mesa workflow.

G. Mesa Workflow

Figure 9 shows the Mesa workflow. The programmer inserts pragma statements in the source code to identify the optimization scope, and runs Mesa to instrument the source code for domain profiling (Step 1). Next the programmer compiles and runs the instrumented code (Step 2) to capture domain profiles (Step 3). The programmer runs Mesa again to request program optimization (Step 4). Based on user selection of a Pareto optimal solution, Mesa generates optimized code (Step 5) that the programmer compiles, runs, and compares with the original program to evaluate performance and accuracy differences (Step 6).

The code instrumentation and generation are implemented by calling the Rose libraries to read the original code and parse it into an abstract syntax tree (AST). Mesa analyzes the portion of the AST identified by the pragma to find expressions that may benefit from LUT transforms, then instruments or optimizes the code by modifying the AST and calling Rose to unparse it back into C/C++ application code.

H. Tool Limitations

The following limitations apply to the tool, not the methodology. Mesa parses only C and C++ code and generates only C++. Mesa also only handles a single source module. Additionally there are syntactic elements that are not handled, including type casts and structure and pointer access. Mesa handles only the following functions: sin, asin, sinh, cos, acos, cosh, tan, atan, tanh, exp, log, and sqrt. In addition, Mesa optimizes assignment expressions and initializers, but will not find computations in other constructs.

VI. Evaluation

We evaluate our methodology in terms of ease of use, accuracy, and performance by using Mesa to optimize six scientific applications. Two case studies evaluate the *slope aspect* and *solar radiation* computations from the

Table IV
APPLICATION RESULTS FROM MESA OPTIMIZATION.

(Intel Core 2 Duo, E8300, family 6, model 23, 2.83GHz, single core)

Application Name	Lines of Code	Number of Expressions	Possible Solutions	Actual Solutions	Pareto Solutions	Processing Time
PRMS Slope Aspect	35	9	512	384	**9**	**13.7s**
PRMS Slope Aspect	35	11	2048	425	**9**	**15.5s**
PRMS Solar Radiation	7	6	64	64	**8**	**14.1s**
SAXS Discrete	60	3	8	4	**3**	**11.2s**
SAXS Discrete	60	3	8	4	**3**	**16.5s**
SAXS Continuous	30	5	32	20	**4**	**10.8s**
Stillinger-Weber	44	6	64	36	**3**	**9.3s**
Neural Network (logistics)	5	2	4	3	**2**	**4.9s**
Neural Network (hypertan)	5	1	2	2	**2**	**2.8s**

Precipitation-Runoff Modeling System (PRMS), developed by the United States Geologic Survey [21]. The third and fourth case studies come from two applications written for the previously cited SAXS project [11]. The fifth application is Stillinger-Weber, a molecular dynamics program developed and used for research at Cornell University [22]. The sixth application is neural network code [23] developed by a faculty member in our department.

A. Evaluation Methodology

To evaluate our methodology we apply the process that a programmer would follow to use Mesa. We run gprof to find bottleneck subroutines that we identify with a pragma. We run Mesa to instrument the application for domain profiling, and to optimize the application. Part of our evaluation is to examine the number of possible and actual solutions for each application, and we measure tool processing time. We finish by compiling and running the original and optimized code and comparing the benefit and error predicted by Mesa to the actual application performance and accuracy.

B. Evaluation Results

Table IV summarizes the results of our case studies. The table shows program and tool statistics: lines of code analyzed, number of expressions, number of possible, actual, and Pareto optimal solutions, followed by the tool processing time. Only the lines of code in the function optimized by Mesa are listed. For example, the Neural Network code has a simple transfer function with only five lines of code. Refer to Table I for performance speedup and error relative to the original output. For example, Mesa extracts 9 expressions from the PRMS slope aspect code, from which it analyzes 384 solutions, and finds 9 to be Pareto optimal. The processing time is 13.7 seconds.

C. Model Evaluation

We now evaluate our error and performance model when applied to the solar radiation application. Figure 10 shows that the performance model predicts the trend of the actual performance, but overestimates the benefit. Some solutions are clearly below the estimated performance, which may be due to instruction-level parallelism and other compiler optimizations.

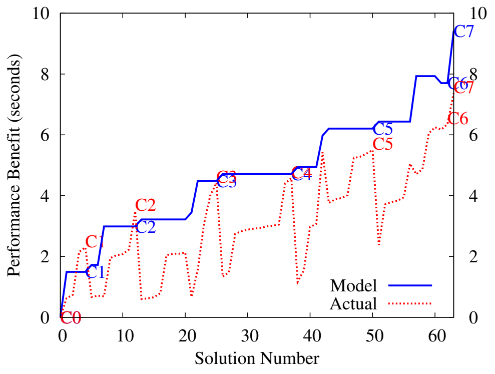

Figure 10. Performance model versus application accuracy.

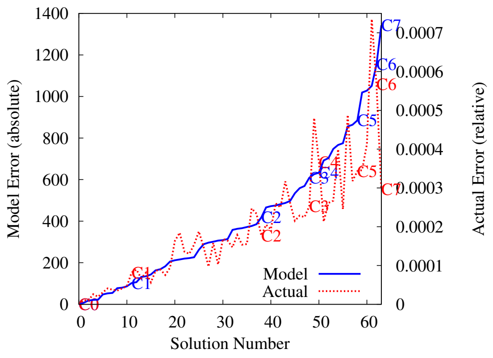

Figure 11. Error model versus application accuracy.

Figure 11 compares the estimated maximum error from the error analysis against the actual maximum error of the optimized application. The model again correctly predicts the trend but has some local variation. We attribute this to the difficulty of modeling application error. Despite being able to quantify many aspects of the error introduced by a LUT transform, a general method to compute the effect on application accuracy remains an open problem. The propagation of error through arbitrary sections of application code poses a complex numerical analysis problem that is unique to each application. This implies that some level of experimentation will still be necessary to evaluate the accuracy of a LUT transform.

D. Summary and Evaluation

Our case studies demonstrate that LUT optimizations are effective on the applications shown, because they show significant performance improvements while meeting the accuracy requirements of the application. Mesa also reduces the programming effort in several ways. First, our tool automates source code analysis, which frees the programmer from having to search for candidate expressions. Second, Mesa gives the programmer error and performance estimates without them having to manually instrument code. Third, our tool generates and integrates LUT code, freeing the programmer from coding. Fourth, our tool computes optimal cache allocations for LUT data using a method that would be very cumbersome by hand. Finally, the tool simplifies experimentation with different solutions.

Mesa has improved our own process for tuning applications whose performance is bound by elementary functions. Our original *ad hoc* LUT implementation for the SAXS discrete code required several weeks of development time and experimentation, even after the base algorithm was implemented and tested. Characterization of error and performance was especially time-consuming, because it required multiple runs of the entire SAXS application. In addition, we simply had no way to estimate the performance or error impact of our LUT transforms. In contrast, we can now evaluate applications with Mesa and quickly receive feedback on whether LUT methods will help the application. For example, we were able to optimize the SAXS continuous code in a matter of minutes, achieving the results shown in Section 6.

E. Threats to Validity

The primary threat to validity for our research is external validity. Empirical research is always limited with respect to the number and scope of the applications that can be evaluated. Our empirical evaluation consists of case studies of six applications in four scientific areas, of which two were partially written by the authors. Further research is required to demonstrate applicability to other domains. However, we expect that our results will generalize to applications that have the same limitations on performance caused by elementary function calls, assuming that other environmental factors (compilers, languages, hardware) are consistent.

Other threats to validity are as follows. First, LUT methods depend on the relative performance of function evaluation versus memory access, which can change as processor architectures evolve. Second, our error model considers only introduced error, not the error that propagates through the application. Third, our performance model sometimes overestimates benefit because it does not account for compiler optimizations.

VII. Related Work

There is considerable precedence for methods that reduce accuracy to gain performance. Linderman et al. [24] show the reducing precision can be beneficial, and Buttari et al. [25] argue that single-precision math can sometimes replace double-precision. These papers place the burden of analyzing numerical stability of the application on the programmer, and this is equally true with our methodology.

Some of the history of hardware LUTs is presented in Section I. Software LUTs have few academic references, but some books [2], [26] discuss the topic. We found one tool that supports software LUT transforms [27]; it is a standalone compiler that analyzes mathematical expressions written in a language that is similar to MATLAB, and transforms these expressions either into an FPGA design or C/C++ code.

Memoization is a related technique that also reuses previous evaluation results to avoid future computation. Memoization employs a mapping function to map input values to cached results. The mapping function can require precise reuse or the comparison can be imprecise or fuzzy [28]. Memoization differs from LUT methods in that results are cached results only as needed, instead of computing the entire table in advance [29].

VIII. Conclusion

This paper demonstrates a novel approach to LUT optimization that is supported by error analysis and performance estimation. Our methodology and Mesa tool substantially automates the application of LUT optimization to scientific programs. Our approach is effective at speeding up code that is performance limited by elementary function calls. Case studies demonstrate speedups from 1.4-6.8$\times$ with reasonable accuracy. Automation improves programmer productivity by reducing the effort required to identify and implement LUT transforms, and by providing information that helps the programmer make the critical tradeoff between error and performance. The Mesa tool provides an alternative to current *ad hoc* practices that require significant programmer effort.

Acknowledgments

We acknowledge support for our research from Award Number 1R01GM096192 from the National Institute Of General Medical Sciences. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institute Of General Medical Sciences or the National Institutes of Health. Our research is also supported by grant number DESC0003956 from the Department of Energy, with additional seed funding from the Vice President of Research and the Office of the Dean of the College of Natural Sciences at Colorado State University (CSU) and from a Department of Energy Early Career grant. This research utilized the CSU ISTeC Cray HPC System supported by NSF Grant CNS-0923386.

References

[1] D.-U. Lee, A. Abdul Gaffar, O. Mencer, and W. Luk, "Optimizing Hardware Function Evaluation," *IEEE Trans. Comput.*, vol. 54, pp. 1520–1531, Dec. 2005.

[2] M. Pharr and R. Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Boston, MA, USA: Addison-Wesley Professional, 2005.

[3] S. Gal, "Computing Elementary Functions: A New Approach for Achieving High Accuracy and Good Performance," in *Proceedings of the Symposium on Accurate Scientific Computations*. London, UK: Springer-Verlag, 1986, pp. 1–16.

[4] P.-T. P. Tang, "Table-lookup Algorithms for Elementary Functions and their Error Analysis," in *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, 1991.

[5] J. A. Piñeiro, J. D. Bruguera, and J. M. Muller, "Faithful Powering Computation Using Table Look-Up and a Fused Accumulation Tree," in *ARITH '01: Proceedings of the 15th IEEE Symposium on Computer Arithmetic*. Washington, DC, USA: IEEE Computer Society, 2001.

[6] D. L. Jones, "Efficient FFT Algorithm and Programming Tricks," *Connexions*, 2007, http://cnx.org/content/m12021/1.6/.

[7] Rapid Radiative Transfer Model, 2010, http://rtweb.aer.com/rrtm_frame.html.

[8] S. Faulk, A. Porter, J. Gustafson, W. Tichy, P. Johnson, and L. Votta, "Measuring HPC productivity," *International Journal of High Performance Computing Applications*, vol. 2004, pp. 459–473, 2004.

[9] E. Loh, M. L. Van De Vanter, and L. G. Votta, "Can Software Engineering Solve the HPCS Problem?" in *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*. New York, NY, USA: ACM, 2005, pp. 27–31.

[10] MESA Project, 2012, http://www.cs.colostate.edu/hpc/MESA.

[11] SAXS Project, 2010, http://www.cs.colostate.edu/hpc/SAXS.

[12] C. Wilcox, M. Strout, and J. Bieman, "Mesa: Automatic Generation of Lookup Table Optimizations," in *Proceedings of the 4th International Workshop on Multicore Software Engineering*, ser. IWMSE '11. New York, NY, USA: ACM, 2011.

[13] ——, "Tool support for software lookup table optimization," *Scientific Programming*, vol. 19, no. 4, pp. 213–229, Dec. 2011.

[14] C. Wilcox, "A methodology for automated lookup table optimization of scientific applications," Ph.D. dissertation, 2012.

[15] NEOS Solverss, 2012, http://www.neos-server.org/neos/solvers/index.html.

[16] R. E. Moore and F. Bierbaum, *Methods and Applications of Interval Analysis)*. Philadelphia, PA, USA: Society for Industrial and Applied Math (SIAM), 1979.

[17] J. Epperson, *An Introduction to Numerical Methods and Analysis*. New York, NY, USA: John Wiley & Sons, 2007.

[18] T. H.Cormen, C. E. Leiserson, R. L.Rivest, and C. Stein, *Introduction to Algorithms (Second Edition)*. Cambridge, MA, USA: The MIT Press, 2001.

[19] ROSE Project, 2011, http://www.rosecompiler.org/.

[20] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski, "A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries," in *IWOMP*, 2010, pp. 15–28.

[21] PRMS Project, 2010, http://water.usgs.gov/software/PRMS.

[22] M. Haran, J. A. Catherwood, and P. Clancy, "Diffusion of Group V Dopants in Silicon-Germanium Alloys," *Applied Physics Letters*, vol. 88, no. 17, p. 173502, Apr. 2006.

[23] Neural Network Software, 2011, http://www.cs.colostate.edu/~anderson/meOther.html.

[24] M. D. Linderman, M. Ho, D. L. Dill, T. H. Meng, and G. P. Nolan, "Towards program optimization through automated analysis of numerical precision," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 230–237.

[25] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov, "Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy," *ACM Transations on Mathematical Software*, vol. 34, pp. 1–22, 2008.

[26] R. Hyde, *The Art of Assembly Language*. San Francisco, CA, USA: No Starch Press, 2003.

[27] Y. Zhang, L. Deng, P. Yedlapalli, S. Muralidhara, H. Zhao, M. Kandemir, C. Chakrabarti, N. Pitsianis, and X. Sun, "A Special-Purpose Compiler for Look-up Table and Code Generation for Function Evaluation," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, Mar. 2010, pp. 1130 –1135.

[28] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy Memoization for Floating-Point Multimedia Applications," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 922–927, 2005.

[29] M. Hall and M. Paul, "Improving Software Performance with Automated Memoization," *The Johns Hopkins APL Technical Digest*, vol. 18, pp. 254–260, 1997.