A Framework for Optimizing Function Call Sequences in MATLAB or Inter-procedural Optimization without Inter-procedural Analysis

Arun Chauhan
achauhan@cs.indiana.edu

Chun-Yu Shei
cshei@cs.indiana.edu

School of Informatics and Computing, Indiana University, Bloomington, IN 47405, USA

Tech Report TR705, February 2013

Abstract

Modern processors are getting harder to program. At the same time, wider availability of high-level dynamic languages is enabling relatively novice users to write sophisticated applications. In this paper, we argue that memory bandwidth-related problems on modern multi-core processors are exacerbated in the context of high-level languages. Compilers can help alleviate these problems, but lack a robust framework to perform the kind of inter-procedural analysis that is required to solve these problems for high-level dynamic languages.

We identify some specific issues related to memory accesses that arise in optimizing applications in high-level programming systems, such as MATLAB*. We demonstrate that the severity of the memory bottleneck in such languages forces us to reconsider several traditional compiler optimizations and, in some cases, to perform transformations that are the exact opposite of what "conventional wisdom" dictates.*

We propose a theoretical framework to solve several of these related problems. The framework relies on simplifying sequences of function calls based on separately measured "savings" functions. It focuses on the specific problem of rewriting function call sequences, rather than attempting to be a completely general rewriting system that many past systems for describing compiler optimizations have tried to be. As a result, we are able to devise efficient algorithms to implement the framework. This seemingly simple framework turns out to be powerful enough to be applicable to a variety of inter-procedural problems without paying the price of live inter-procedural analysis. These problems include, library function selection, procedure specialization, "de-vectorization" (converting vector statements to parallelizable loops), computation partitioning for heterogeneous platforms, and grouping operations based on data formats and distributions.

Keywords: High-level programming systems, compilers, multicore, memory bandwidth, library functions

1. Motivation

As major hardware manufacturers have turned to multi-core architectures to continue the exponential growth in processing power, the onus of translating the raw processing power into real gains in software performance has fallen squarely on software developers. Unfortunately, this comes at a time when software development expertise is in continued short supply with no sign of relief in the near future. Therefore, development of automatic or semi-automatic methods becomes vitally important in order to extract good performance from multi-core processors. Moreover, such methods must work while allowing relatively novice end-users to write their software directly, without relying on the increasingly precious community of expert programmers.

Motivated by this observation we conducted a small experiment to find where the most notable performance bottlenecks could be in high-level programming systems, such as MATLAB. Figure 1 shows the results for an array expression involving extensive subscript expressions. The expression is taken from a MATLAB version of the NAS MG benchmark [3] and is shown below in an abbreviated form.

```
m =
 f(1).*(n(c,c,c)) +
 f(2).*(n(c,c,u)+n(c,c,d)+...+n(d,c,c)) +
 f(3).*(n(c,u,u)+n(c,u,d)+...+n(d,d,c)) +
 f(4).*(n(u,u,u)+n(u,u,d)+...+n(d,d,d));
```

In the above code `f` is a vector of length 4 and `n` is a three-dimensional cubic array. Variables `c`, `d`, and `u` are used to index into `n` in different patterns. Thus, the additions of the variously indexed subsections of `n` are all vectorized. This means that several intermediate

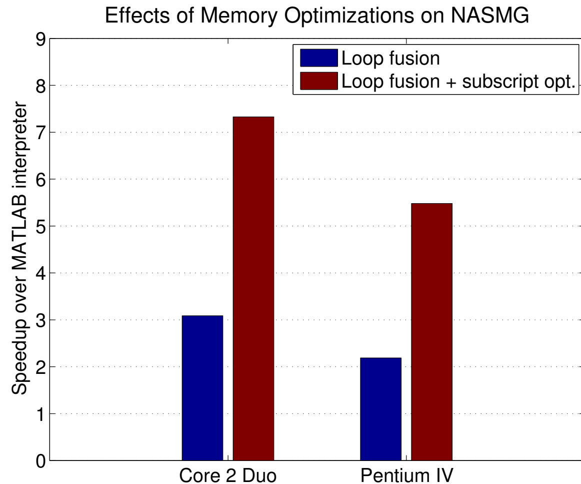

Figure 1. “De-vectorizing” a loop-nest.

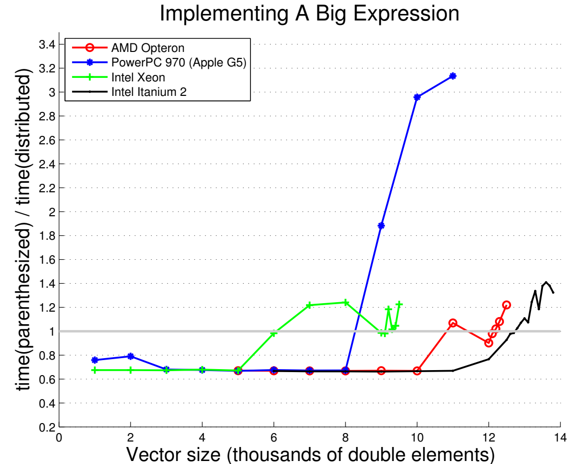

Figure 2. Memory bandwidth bottleneck.

array values must be computed and stored to evaluate this large expression. It turns out that this code segment is the core computation in the benchmark, consuming more than 80% of the total running time. Fortunately, it is easy to convert the vector operations in this case into a loop to perform the same computation. The resulting loop eliminates all temporary arrays, replacing them by temporary scalar values. This conversion alone results in a factor of 2.25 to 3 speedup of the whole application (left bars in Figure 1). Further optimization can be performed by realizing that the indirection into `n` is simply a permutation of the elements of `n`. Once subscript computation is optimized the application, with the above code segment translated into C, runs 5.5 to 7 times faster (right bars in Figure 1).

The transformation described above can be thought of as inter-procedural loop fusion with array contraction [12]. An important consequence of starting with vector statements is that the loops have no loop-carried dependences and can be parallelized, e.g., to run in multi-threaded mode to leverage the modern multi-core processors. This finding is exactly opposite of the “conventional wisdom” that operations should be vectorized whenever possible [**?**, 11, 4] in MATLAB-like array languages. The most major performance improvement comes from a dramatic reduction in memory traffic in the transformed code.

Another example demonstrating the crucial role that memory traffic plays is in Figure 2, reported earlier by McFarlin and Chauhan [10]. The plot shows the speedup obtained when a complex matrix expression is implemented by minimizing the number of temporaries, even at the cost of increased floating point operations. The graph clearly demonstrates that as the matrix sizes increase the memory bandwidth becomes the bottleneck and reducing the memory traffic becomes the most critical optimization, even if that means increasing the number of arithmetic operations.

A third motivating example in Figure 3 is a code

```
function [s,r,j_hist] = ...
   min_sr1(xt,h,m,alpha)
....
while ~ok
 ....
 invsr=change_form_inv(sr0,h,m,low_rp);
 big_f=change_form(xt-invsr,h,m);
 ....
 while iter_s < 3*m
  ....
  invdr0=change_form_inv(sr0,h,m,low_rp);
  sssdr=change_form(invdr0,h,m);
  ....
 end
 ....
 invsr=change_form_inv(sr0,h,m,low_rp);
 big_f=change_form(xt-invsr,h,m);
 ....
 while iter_r < n1*n2
  ....
  invdr0=change_form_inv(sr0,h,m,low_rp);
  sssdr=change_form(invdr0,h,m);
  ....
  end
 ....
end
```

Figure 3. Repeating function call patterns.

fragment from an image processing application [?]. The functions `change_form_inv` and `change_form` are repeatedly called in tandem. As an array output of the former goes into the latter, it presents an excellent opportunity to combine the two functions and eliminate the array temporary.

In the rest of the paper we present an abstract framework and its solution in Sections 2 and 3 that is able to model and solve all the three optimization problems illustrated above. Section 4 describes how the framework is integrated in our MATLAB / Octave compiler to solve these problems. In addition, the framework can also be applied in other scenarios, as described in Section 5. Section 7 discusses related work.

2. Problem Abstraction

We define an abstract problem as follows. Suppose that we are given a sequence of calls to functions, f_i, $1 \leq i \leq n$, where the function f_i takes the arguments α_i and returns the values β_i, each of which is a list of values. Thus, α_i is the list of actual inputs to the call to f_i and β_i is the list of actual outputs.

$$\begin{aligned} \beta_1 &= f_1(\alpha_1) \\ \beta_2 &= f_2(\alpha_2) \\ &\cdots \\ \beta_n &= f_n(\alpha_n) \end{aligned}$$

We are given a translation table where the jth entry is of the form:

$$\begin{aligned} f_{i_1} : \iota_{i_1}(n_{i_1}) &\rightarrow \rho_{i_1}(m_{i_1}) \\ &\cdots \\ f_{i_{k_j}} : \iota_{i_{k_j}}(n_{i_{k_j}}) &\rightarrow \rho_{i_{k_j}}(m_{i_{k_j}} \\ f_{i_1} f_{i_2} \cdots f_{i_{k_j}} &\rightarrow \tilde{\mathbf{G}}_{\mathbf{j}}(f_1, \iota_1, \rho_i, \ \ldots \ , f_n, \iota_n, \rho_n) \end{aligned}$$

Further, each entry is subject to a predicate, P:

$$P(f_1, \iota_1, \rho_1, \ldots \ , f_n, \iota_n, \rho_n)$$

that can be computed efficiently. ρ_i and ι_i are the formal output and input parameters to the function f_i. The numbers following them within the parentheses are the number of formal output and input parameters. The predicate P has access to simple properties of its arguments, such as queries about the type or value of a certain function parameter, as well as dependence information such as whether there is a dependence between certain pairs of actual input or output parameters and whether there are dependences connecting a specific actual parameter to pieces of code outside the matched sequence. Finally, there is a savings function, S_j, associated with the jth entry:

$$S_j(\iota_{i_1}, \ldots, \iota_{i_{k_j}}, \rho_{i_1}, \ldots, \rho_{i_{k_j}})$$

S_j might be a symbolic expression that may depend on the parameters and their types. The target function $\tilde{\mathbf{G}}_{\mathbf{j}}$ is a *meta-function* that, in general, is a specification for generating the target function from the matched function sequence. For dynamic and interpreted languages, which this work primarily targets, it is most convenient to use the source language itself as the meta-language for specifying $\tilde{\mathbf{G}}_{\mathbf{j}}$. The need and use of this will become clear when we discuss applications of the framework in Section 5. Similarly, the predicate P, and even the savings function S_j, may also be expressed within the source language.

In general, the correctness of a specification cannot be verified by the compiler. However, in an important special case the compiler can verify the correctness related to dependences. If the effects of the function $\tilde{\mathbf{G}}$ on the dependence graph related to the matched function sequence can be evaluated then the compiler may be able to verify automatically if any dependences are violated.

3. Solution Framework

3.1. Preliminaries

A *reference point* in a program is a static lvalue or an rvalue that accesses a memory location. A reference point refers to a syntactic location in a program and not to a dynamic or run-time instance of a reference in a loop or a function call.

We define a *dependence graph* to be a data-dependence multi-digraph with three types of edges:

1. δ: true dependence
2. δ^{-1}: anti-dependence
3. δ^{o}: output dependence

Unlike the traditional dependence graphs where nodes in the graphs are program statements [2], we assume that nodes in the graphs are reference points. This allows the dependence graph to more precisely point to the reference that causes the dependence. Notice that even though we use the term "graph" it is, in fact, a multi-digraph since there may be multiple edges between a pair of reference points if the reference points happen to be inside a loop-nest. There may also be self-loops corresponding to loop-carried dependences.

We assume the statements in the given program to be in completely *flattened form*. Thus, large expressions are broken down into a series of expressions involving only the most basic forms of operations by introducing suitable temporaries. This applies to expressions occurring on the right-hand sides of assignment statements, as also to complex expressions occurring in function arguments, array subscripts, and loop and branch conditionals. Such a form allows a more accurate representation of the computations that must be carried out eventually and exposes the implicit temporaries. This is especially important for array languages, such as MATLAB that this work targets, since failing to account for array temporaries can dramatically alter the computation and memory traffic estimates, which can lead to incorrect decisions by a compiler trying to optimize the code. "Flattening" is among the very first operations performed by the MATLAB parallelizing compiler that we are currently developing, in order to normalize the input.

Using a completely flattened form also enables a cleaner theoretical model by using a convenient method to represent all operations, including the built-in primitive operations, by abstract function calls and assignments. For example, a statement `c = a op b` may be represented by a function call `f_op` as `[c] = f_op(a,b)`, using MATLAB-like syntax for (potentially multiple) return values. Such representation allows handling of all operations in a uniform way and also provides a seamless mechanism to handle the overloaded operators, some of which, in fact, may map directly to function calls. In short, each simple statement is a function call. The function name may refer to either a "real" function or an operator. To make a distinction between standard function calls and the "pseudo-functions" that represent statements we will call the latter **statement functions** or **s-functions**. Thus, an **s-function** may either be a standard function or an operator represented as a function.

3.2. Basic Blocks

Our experiments with MATLAB applications have shown that basic blocks are the most important common case for this problem. Most sequences of s-functions that profit from coalescing occur within single basic blocks.

As indicated before, if the compiler can compute the dependences in the code that replaces a matched sequence then it may be able to automatically detect and avoid dependence violations. In such cases, it is tempting to frame the problem of searching for the given function sequence patterns in a basic block as sub-

$$
\begin{array}{ll}
f_1 : \iota_1(2) \rightarrow \rho_1(3) & f_1 f_3 \rightarrow g_1 \\
f_3 : \iota_3(3) \rightarrow \rho_3(2) & P : \text{depend}(\rho_1[1] \rightarrow \iota_3[2]) \\
g_1 : \delta_1(4) \rightarrow \theta_1(2) & \quad \wedge \ \text{depend}(\rho_1[2] \rightarrow \iota_3[3])
\end{array}
$$

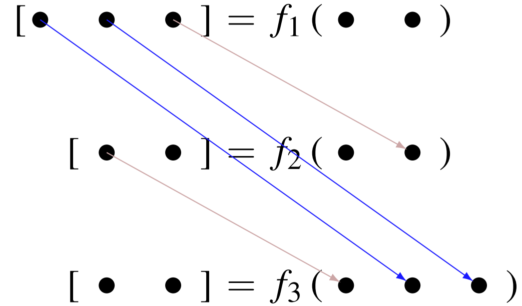

Figure 4. Intermediate dependences interfere.

graph isomorphism over the dependence graph of the basic block. Even though the sub-graph isomorphism problem is $\mathcal{NP}$-complete, in general, we are dealing with only a subset of graph types because the dependence graph within a basic block is a Directed Acyclic Graph (DAG). However, sub-graph isomorphism is not enough. Consider the example in Figure 4.

The top part of the figure is a specification for replacing a sequence of $f_1 f_3$ by g_1 provided the first return value from f_1 is the second input value to f_3 and the second return value from f_1 is the third input value to f_3. The bottom part of the figure shows a sequence where this relation is satisfied among the actual parameters. The dependence graph is superimposed on the function sequence. The darker edges satisfy the relation among the arguments of f_1 and f_3 that is sought. However, if we were to actually perform the fusion of f_1 and f_2 into g_1 the intermediate dependences, represented by lighter lines, will get violated. Thus, it is not enough to match a sub-graph of dependences. We also need to ensure that actual replacement of the sequence will not violate any other dependences.

To keep the exposition concise we assume that the predicate P has been suitably enhanced or verified by the compiler, if it is able to do that. We define a *candidate sequence* as follows.

> **Candidate Sequence:** A sequence of actual functions in a linear code segment is a candidate sequence if, (a) the sequence is part of some entry in the translation table; and (b) the

Algorithm **Find_Best_Candidate_Sequence**

Input: basic block B, dependence graph D restricted to B, table T of sequence replacement rules
Output: the set of candidate sequences, C, which results in the biggest overall saving
begin
1 Construct a finite automaton from the pattern specs in T
2 Run the finite automaton to find all the candidate sequences in B, using D
3 Create a new graph, $G = (V, E)$, such that:
   $V =$ candidate sequences found on line 2
   $E = \{(u, v) \,|\, \text{if } u \text{ does not overlap with } v\}$
4 Return the clique of G with largest saving
end

Figure 5. Sequence matching for basic blocks.

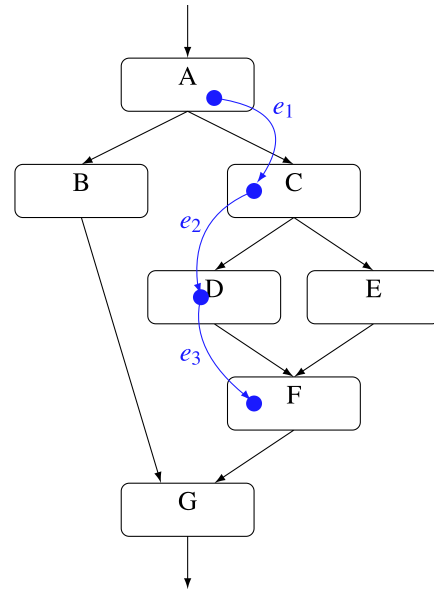

Figure 6. Forward control flow

predicate in the translation table entry evaluates to true for the sequence.

Observe that if there are dependence edges lying wholly within the sequence that were not part of the specification then the predicate should reject the sequence as not a valid candidate. Similarly, if dependence edges entering or leaving the sequence get deleted because the reference points where the edges terminated or started disappeared in the replacement then also the sequence is not a valid candidate. This might happen, for example, if an intermediate result is needed later, but replacing the sequence by a single function will make the intermediate result unavailable.

The algorithm in Figure 5 makes use of the above definition to look for candidate sequences. Matching sequences in B can be efficiently found using a finite automaton encoding the function sequences and running the finite automaton over the dependence graph D. The predicate for the sequence has to be evaluated only if the automaton finds a syntactic match. In the common case, the predicate takes no more time than the size of the matched sequence. The graph G can be created in time proportional to the square of the total number of matching sequences. A clique in G represents a set of sequences that do not interfere with each other and hence can all be replaced simultaneously. We are interested in the clique that results in the maximum amount of saving. Clique finding is a hard problem, however, the size of G is likely to be very different from B. Specifically, we expect the number of candidate sequences, and hence the size of graph G, to be much smaller than the size of B. This is borne out by our experience with MATLAB applications in the domain of linear algebra and image and digital signal processing.

Finally, if the savings functions cannot be completely evaluated at compile time then one possibility is to resolve them with additional input from the user and / or profiling data. Alternatively, the evaluation can be deferred until run time when, combined with generation of specialized variants for different expected resolutions, a "dynamic compilation" phase can choose the right specialized variant to load dynamically.

3.3. Forward Control Flow

The solution for basic blocks can be extended to a program consisting of forward control flow (i.e., no back edges). In that case the control flow graphs forms a DAG. If a candidate sequence spans multiple blocks the savings are tempered by the additional computation that the fused function must (presumably) perform even when the actual control flow path does not require computing the downstream functions in the sequence. An example will clarify this point.

In Figure 6 the edges e_1, e_2, and e_3 connect four s-functions that constitute a candidate sequence. The resulting combined function will need to be placed in the block A. However, this means that the combined function might perform unnecessary computations when the control does not pass through the blocks C, D, and F. This represents a trade-off that is impossible to judge without additional information. If the control flow edges are marked with probabilities of their execution

then the compiler can weigh the savings based on the execution probability of the path on which the candidate sequence lies. Alternatively, an expert user (or a profiler) can provide feedback to the compiler with hints regarding the most probable paths. Other than this consideration, the algorithm used for basic blocks can be extended in a straightforward manner to work with basic blocks in a control flow DAG.

3.4. Side-Effects and Aliases

So far we have assumed that functions have no side effects, which works for a vast majority of MATLAB programs. However, if arguments may be passed by reference (in another language), or if the program uses global variables, then we need additional analysis. Alias analysis can be used to refine the dependence graph when function arguments can be passed by reference. To be safe, alias analysis must always report an alias whenever it can actually occur. Side-effects can be similarly handled by adding dependence edges for global variables associated with an s-function.

3.5. Loops

Loops can potentially add substantial complexity to the analysis of function call sequences with unclear benefits. In fact, one of the motivating examples mentioned earlier was to convert vector statements to loops. We believe that the ability to handle function sequences across loop boundaries does not present a compelling immediate need. Therefore, we leave the investigation of function call sequences across loop boundaries to future work.

3.6. Combining On Demand

Sometimes, it may be possible to coalesce a variable number of operations together. The framework described so far has no way to represent such a scenario. We enhance the framework with a regular-expression-like syntax to allow a variable number of operations to be combined using positive closure. For example:

$$(f_1 \,|\, f_2 \,|\, \ldots \,|\, f_n)^+ \quad \rightarrow \quad \tilde{\mathbf{G}}$$

The above rule says that any combination of functions f_1 through f_n may be combined together into code obtained by evaluating the meta-function $\tilde{\mathbf{G}}$. The arguments to $\tilde{\mathbf{G}}$ and the savings function depend on the actual sequence of functions matched and are implicitly constructed. Notice that this extension of the framework does not change the efficiency of the finite-automaton-based algorithm to search for matching sequences.

This extension is intended to capture the case of "de-vectorizing" operations that the compiler can then recombine automatically into a loop-based code that, presumably, $\tilde{\mathbf{G}}$ constructs.

3.7. Savings Function

Computing the actual savings functions is outside the scope of this paper. These could be computed in several ways. If a function can be modeled analytically, then the savings could be computed analytically. If the functions involved are too complex, empirical modeling, or a combination of analytical and empirical modeling could be employed [16].

Often, as our experiments have shown, the most important consideration in high-level programming systems is the amount of memory traffic. This is especially true on modern multi-core processors. Thus, a practical savings function could be simply the savings in memory traffic.

4. Implementation

We are implementing the function sequence optimization framework presented in this paper in a MATLAB and Octave compiler that we have been developing. The overall architecture of the compiler is shown in Figure 7. The figure shows only the main components. The gray boxes represent the components that use the framework presented here.

The compiler works by compiling portions of the given code into C++, while keeping the remainder of the code that will not benefit from such compilation, intact. In this way those parts of the code for which the interpreter does a good job can continue to use the interpreter. Additionally, the compiler can be deployed incrementally—whatever language features or optimizations the compiler does not support still work as before within the interpreter. Finally, the compiler can be used either as a Just-In-Time or an offline compiler. We have been using it as both by providing MATLAB and Octave compilation scripts as well as a standalone executable that can be invoked at the command line.

Integrating the compiler for MATLAB—or for any dynamic language, for that matter—with the interpreter provides another compelling benefit. Any optimizations based on partial-evaluation can make use of the language interpreter, eliminating the potential hazard of differences in operational semantics between the compiled language and the language used to implement the compiler. In the case of MATLAB it also makes it possible to partially evaluate complex functions, e.g., those

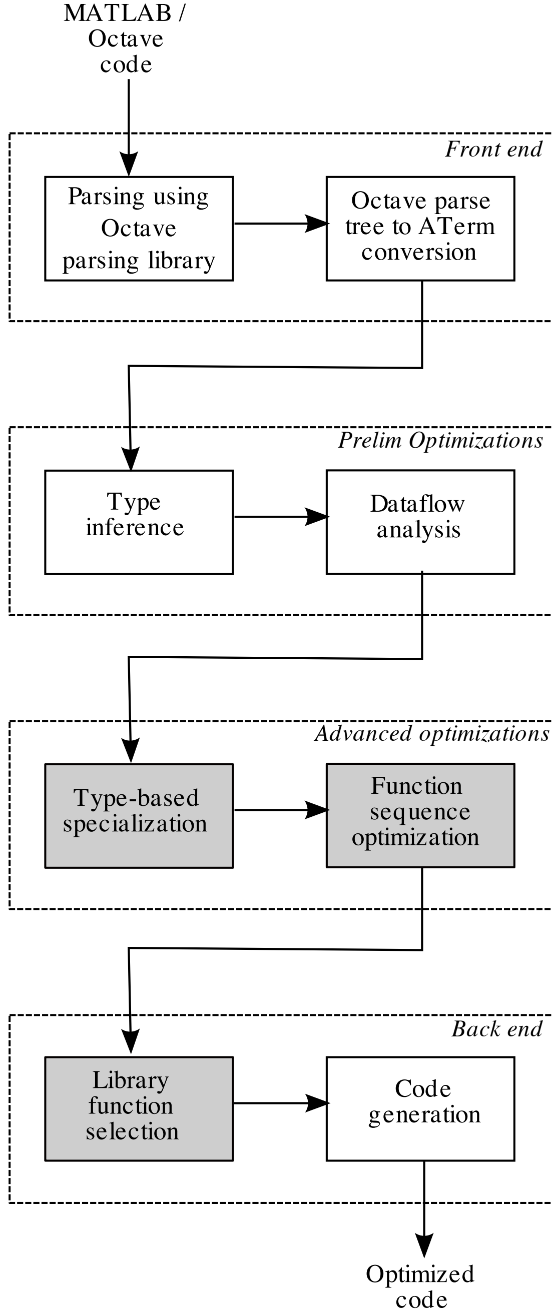

Figure 7. The overall compiler architecture

related to matrices, which would otherwise be impractically hard to implement and involve large duplication of effort. The type inference component makes use of this opportunity, however the detailed discussion of that component is beyond the scope of this paper.

Function sequence optimization and library-function selection are the two major components that are most relevant to the work presented here. The availability of the interpreter is also leveraged in these components, especially, for function sequence optimization, by using the MATLAB language itself to specify the templates for dynamically-generated functions. The next section illustrates this with an example. It also illustrates some other advanced optimizations that can make use of the framework, some of which will be integrated into the compiler.

We also note that function sequence optimization closely depends on the prior phases of type inference and dependence analysis (not shown in the figure) as it uses their results.

5. Applications

The abstract formulation described in the last two sections can be used in several optimization scenarios. This section describes some of them.

"De-vectorization" Consider a specification using the regular-expression-like extension of the basic framework. The individual functions are element-wise addition and multiplication and the coalescing function, **loop_addmul**, is a template to generate code to compute the matched functions inside a loop.

$$\begin{aligned} \text{vec_add} : I_a(2) &\rightarrow O_a(1) \\ \text{vec_mul} : I_m(2) &\rightarrow O_m(1) \\ (f_1|f_2)^* &\rightarrow \textbf{loop_addmul} \end{aligned}$$

Notice that the actual replacement rule uses placeholders for functions. This is useful to be able to refer to functions in a sequence uniquely when they have the same name. This specification matches the scenario in the first motivating example in Section 1 and is able to perform the "de-vectorization" that results in dramatic performance improvement in NASMG.

This optimization may also be easily combined with loop-parallelization by modifying the template **loop_addmul** to be a parallel loop. In effect, this allows inter-procedural loop fusion without paying the price of inter-procedural analysis. If the resulting combined function is parallelized, e.g., using threads, then the transformation can be highly profitable on multi-core processors by reducing memory traffic and exploiting concurrency at the same time.

Library Function Selection Our second motivating problem concerned mapping matrix operations in MATLAB to an underlying matrix library, such as the BLAS. This is easily done in the framework presented here. For example `DGEMM` can be a target function (g) of a

sequence of operations involving matrix scaling, multiplication and addition. The savings could be modeled based on the savings in memory traffic due to the elimination of array temporaries.

$$\begin{aligned} \text{mx_scale} : I_{s1}(2) &\rightarrow O_{s1}(1) \\ \text{mx_scale} : I_{s2}(2) &\rightarrow O_{s2}(1) \\ \text{mx_mul} : I_m(2) &\rightarrow O_m(1) \\ \text{mx_add} : I_a(2) &\rightarrow O_a(1) \\ f_1 f_2 f_3 f_4 &\rightarrow \mathbf{gen_dgemm} \end{aligned}$$

For the sake of brevity we do not show the accompanying predicate. The predicate enforces the dependences between scaling, multiplying, and additions that together map to a single call to `DGEMM`. This example also brings out the importance of type information. It exactly models, and supersedes, our earlier heuristic-based approach [10].

Recurring Function Sequences The third motivating example is the canonical case that is simplest to handle within the presented framework. All it requires is a simple mapping of the sequences of calls to `change_form_inv` and `change_form` to a combined function that is able to eliminate at least one of the array temporaries.

Grouped Operations The combined operation can also be viewed as a grouping operation. This would be relevant for working on sparse data where there might be different, competing, representations for the sparse data. A single grouped operation identifies those sequences of operations that all benefit from using a specific data representation. In such cases, the grouped operation may still be implemented as a sequence of operations. Another scenario could be operations that work on distributed data and grouping those operations together that work with the same distribution is profitable.

Computation Partitioning When targeting heterogeneous platforms the framework can aid in partitioning the computation. For example, it may be more efficient to implement certain operations on an FPGA-based co-processor, or an accompanying Graphical Processing Unit (GPU). Identifying maximal sequences of operations that can be offloaded onto a co-processor can reduce the overheads compared to repeatedly transferring data back-and-forth between the main processor and the co-processors.

Traditional Function Specialization Predicates on arguments allow the traditional function specialization to be expressed easily within the framework. The specialization could be based on values (e.g., when specific argument values can create more optimized code) or types (e.g., when functions are overloaded).

In all of the scenarios, except the last, a savings function that accounts for memory traffic is likely to work well.

6. Contributions and Future Directions

The main contribution of this paper is in recognizing that a simple function-sequence replacement operation allows expressing the most critical optimizations in high-level programming systems, especially those based on high-level dynamic languages such as MATLAB. Unlike earlier efforts aimed at developing complete systems that could express all, or most, compiler optimizations, we started out by identifying the optimizations that were most important in compiling MATLAB. We then created the simplest framework that was sufficient to express those optimizations. By keeping the framework simple we have been able to devise a simple and efficient algorithm to implement it.

Even with the simplicity of the framework, its programmable predicates, savings, and meta-functions lend it a great amount of power. We have found that these mechanisms work naturally with a dynamic and interpreted programming language because the framework implementation can leverage the interpreter to evaluate the predicates, savings, and meta-functions.

In addition to implementing and testing the framework on a large number of applications, our future plans include evaluating the effectiveness of the framework to express other inter-procedural transformations that are relevant in the context of high-level programming systems. Some of these advanced transformations, in addition to the motivating optimizations, were described in Section 5, but evaluating them in the context of real applications is a part of future work. Even though memory traffic is often the primary bottleneck, we will also be exploring the cost-saving metrics more inclusive than memory traffic. Finally, handling the savings functions that evaluate to symbolic values that cannot be compared at compile time and generating them automatically or semi-automatically is also a part of future work.

7. Related Work

Several general frameworks for expressing compiler optimizations have been proposed. One such recent framework, called Pavilion developed by Willcock, uses regular-expressions enhanced with existential and universal path quantification and trace transduc-

tion [14]. Willcock also provides a detailed overview of numerous other similar efforts in the past. All of these attempt to build general frameworks, usually Turing complete, to be able to capture the myriad code optimizations. Our goal in this paper has been to develop a specific and more narrowly focused mechanism that lends itself to more efficient implementation and simple specification.

There have been several recent research efforts at handling library-level optimizations. The Broadway compiler makes use of a domain-independent annotation language to capture expert knowledge about libraries [6]. Our work has overlapping goals with this project. The annotation language in Broadway is extensive and sufficiently expressive to capture aliases and pointer information to be handle their primary target language, C. However, their documentation suggests that their compiler does not handle library call sequences, instead focusing only on specializations of single library calls. Replacing "code patterns" of library call sequences has been mentioned as their future work.

The telescoping languages approach has been proposed as a way to achieve efficient interprocedural optimizations without paying the price for it at "script-compilation" time [7]. Even though annotations and recognition of library identities have been suggested as desirable techniques, there is no documented research on those techniques within the telescoping languages approach.

Tools, such as ROSE, let users describe tree-transformations for automated program rewriting [15]. Our own compiler is implemented using a domain-specific tree-rewriting language, called Stratego [13]. However, tree-rewriting is in the worst case inadequate, and in the best case tedious, for transforming function sequences that might be related through dependences, but may be widely separated in the parse tree of the program.

The classic technique of instruction selection based on tree matching relies on closeness between the source and target instruction sets [1]. Subsequent work on targeting more complex instruction sets, such as those of Digital Signal Processing (DSP) processors using tree coverage also assumes that the target instructions have known and predictable costs [8]. More recent work on DSP targets have focused on making the selection process more efficient and accurate, but the underlying assumptions about the cost model remain unchanged [5]. To select most appropriate library functions the solver needs to be able to account for costs that are given as expressions, possibly evaluating to symbolic values.

The Falcon compiler used a heuristic-based approach to select the most appropriate BLAS calls for MATLAB operations [9]. This approach was later extended by McFarlin and Chauhan to regions bigger than basic blocks [10]. This paper presents an abstract formalization of the problem that can apply to a variety of problems in addition to library function selection. The abstraction also allows handling of arbitrary libraries through appropriate parameterization.

8. Conclusion

This paper has presented a framework to optimize sequences of function calls or operations by reducing them to equivalent integrated functions. The need for such optimizations arises in several situations, especially when optimizing for memory traffic. The framework provides a simple model that can be efficiently implemented and is applicable in a range of scenarios, especially those involving inter-procedural optimizations, which are the most important optimizations for high-level programming systems.

9. Acknowledgments

The MATLAB implementation of the NAS MG benchmark was developed at the Ohio Supercomputer Center. The image processing code was obtained from the Center of Multimedia Communication at Rice University. The work was supported in part by a grant from the Faculty Research Support Program at Indiana University.

References

[1] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):491–516, Oct. 1989. DOI: 10.1145/69558.75700.

[2] J. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 2001.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991. DOI: 10.1177/109434209100500306.

[4] N. Birkbeck, J. Lévesque, and J. N. Amaral. A dimension abstraction approach to vectorization in Matlab. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 115–130, 2007. DOI: 10.1109/CGO.2007.1.

[5] E. Eckstein, O. König, and B. Scholz. Code instruction selection based on SSA-graphs. In *Software and Compilers for Embedded Systems*, volume 2826/2003 of *Lecture Notes in Computer Science*, pages 49–65. Springer Berlin / Heidelberg, 2003. DOI: 10.1007/b13482.

[6] S. G. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2):342–357, Feb. 2005. DOI: 10.1109/JPROC.2004.840489.

[7] K. Kennedy, B. Broom, A. Chauhan, R. J. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping Languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(2):387–408, Feb. 2005. DOI: 10.1109/JPROC.2004.840447.

[8] R. Leupers and P. Marwedel. Instruction selection for embedded DSPs with complex instructions. In *Proceedings of the Conference on European Design Automation*, pages 200–205, 1996. DOI: 10.1109/EURDAC.1996.558205.

[9] B. A. Marsolf. *Techniques for the Interactive Development of Numerical Linear Algebra Libraries for Scientific Computation*. Doctoral dissertation, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1997.

[10] D. McFarlin and A. Chauhan. Library function selection in compiling Octave. In *Proceedings of the Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL), held in conjunction with the 21st IEEE Parallel and Distributed Processing Symposium (IPDPS)*, Mar. 2007. DOI: 10.1109/IPDPS.2007.370645.

[11] V. Menon and K. Pingali. A case for source-level transformations in MATLAB. *ACM SIGPLAN Notices*, 35(1):53–65, Jan. 2000. DOI: 10.1145/331963.331972.

[12] J. Ng, D. Kulkarni, W. Li, R. Cox, and S. Bobholz. Inter-procedural loop fusion, array contraction and rotation. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*, 2003. DOI: 10.1109/PACT.2003.1238008.

[13] K. Olmos and E. Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In R. Bodik, editor, *Proceedings of the 14th International Conference on Compiler Construction (CC '05)*, volume 3443 of *Lecture Notes in Computer Science*, pages 204–220. Springer-Verlag, Apr. 2005. DOI: 10.1007/b107108.

[14] J. J. Willcock. *A Language for Specifying Compiler Optimizations for Generic Software*. Doctoral dissertation, Indiana University, Bloomington, Indiana, USA, Dec. 2008.

[15] Q. Yi and D. Quinlan. Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In *Proceedings of the 17th International Workshop on Languages and Compilers for High Performance Computing (LCPC)*, volume 3602/2005 of *Lecture Notes in Computer Science*, pages 253–267, Berlin / Heidelberg, 2004. Springer. DOI: 10.1007/11532378_19.

[16] K. Yotov, K. Pingali, and P. Stodghill. Think globally, search locally. In *Proceedings of the 19th Annual International Conference on Supercomputing*, pages 141–150, 2005. DOI: 10.1145/1088149.1088168.