Optimizing Strategies for Telescoping Languages: Procedure Strength Reduction and Procedure Vectorization

Arun Chauhan
achauhan@rice.edu

Ken Kennedy
ken@rice.edu

Department of Computer Science
Rice University
Houston, TX 77005, USA

ABSTRACT

At Rice University, we have undertaken a project to construct a framework for generating high-level problem solving languages that can achieve high performance on a variety of platforms. The underlying strategy, called *telescoping languages*, builds problem-solving systems from domain-specific libraries and scripting languages. To accomplish this it extensively preanalyzes and transforms the library to produce a scripting language precompiler that optimizes library calls within the scripts as if they were primitives in the underlying language.

A major technical issue is how to preoptimize a library for use in applications that are yet to be seen. To address this issue, we have conducted a study of applications written in Matlab by the signal processing group at Rice University. This has identified a collection of old and new optimizations that show promise for this particular domain. Two promising new optimizations are *procedure vectorization* and *procedure strength reduction*. The latter of these is particularly useful in this problem domain and we expect it to be just as applicable in other contexts as well.

We report on the results of an exploration of the effectiveness of procedure strength reduction on three real Digital Signal Processing (DSP) applications. By transforming these programs according to the strategies described in this paper, we were able to achieve speedups ranging from 10 to 40 percent over the entire application – speedups for individual functions were even more dramatic.

Keywords

Telescoping Languages, Matlab, Specialization, Vectorization, Reduction in strength

General Terms

High-performance computing, High-level languages, Scripts, Partial evaluation, Procedure specialization, Automatic differentiation

Appears in the Proceedings of the 15th ACM International Conference on Supercomputing, June 16-21, 2001, Sorrento, Naples, Italy.

1. INTRODUCTION

As high-performance computers have become more complex, application development in traditional programming languages has become more difficult because of the need to structure programs to achieve high performance on specific architectures. As a result, scientific application development in such languages is becoming the exclusive domain of experts. This has the unfortunate side effect of limiting the rate of scientific progress. One strategy for overcoming this problem is to make it possible for end users to develop applications in very high level domain-specific programming systems. The popularity of Matlab™ is evidence of the promise of this approach. However, such systems typically do not achieve performance levels that are sufficient to support many application domains, so many programs written in Matlab and other such languages are rewritten in traditional languages by professional programmers for better performance.

At Rice University, we have undertaken a project to construct a framework for generating high-level problem solving languages that can achieve high performance on a variety of platforms. The underlying strategy in this framework is to use advanced compiler technology to construct such systems from domain-specific libraries that are specifically programmed to be invoked from flexible scripting languages. This is similar to the way such systems are constructed today. However, because existing scripting languages typically treat libraries as black boxes, they fail to achieve acceptable performance levels for compute-intensive applications. Previous research has shown that this problem can be ameliorated by translating scripts to a conventional programming language and using whole-program analysis and optimization to improve performance.

This approach suffers from the disadvantage that script compilation times can be long and there is no provision to take advantage of the specific knowledge of the library developer on how to improve code involving multiple calls to the library. To overcome these disadvantages, we propose a new approach called *telescoping languages* [20, 7], in which the libraries that provide component operations accessible from scripts are extensively analyzed and optimized in advance for use in a smart script compiler. By using the knowledge gained from this analysis and preliminary compilation and from annotations provided by the library designer, the compiler will be able to process scripts containing references to the library components as rapidly and effectively as if they were primitive op-

[1] Matlab is a registered trademark of MathWorks Inc.

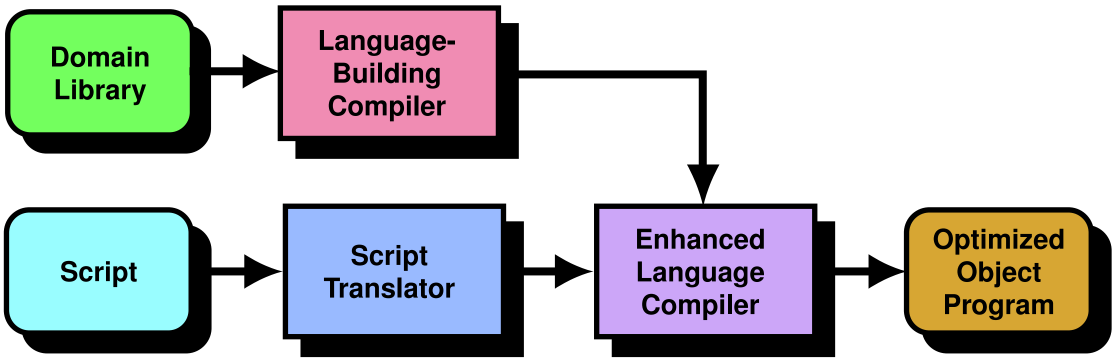

Figure 1: Telescoping Languages

erations in the base language. The telescoping languages approach is diagrammed in Figure 1.

In this strategy, the translator generator (language building compiler), which could run for many hours, would analyze and optimize the domain library under a variety of assumptions about the calling program, many of which are generated from advice and sample calling sequences provided by the library developer. The result would be an "enhanced language compiler" which would optimize and specialize calls to the domain library to produce code that would feed into the vendor optimizing compiler.

The basic idea of this approach is to invest substantive time in library analysis and optimization, on the assumption that the domain libraries will be recompiled far less often than scripts that invoke them. By exhaustively exploring the implementation space for library modules in advance, it will be possible to automatically build a powerful framework for transforming and optimizing uses of library primitives. The preliminary analysis and optimization pass may take hours, but it would be worth the cost if the libraries were reused in many different scripts.

A critical technical challenge is how to use computation power to support speculative optimizations of library routines that will yield high enough performance benefits to make the scripting language efficient enough for developing real applications. To address this challenge we will need to develop new optimizations that can be systematically applied to the procedures of a domain library. With this in mind, we have entered into a collaboration with signal-processing researchers at Rice who are engaged in the development of a number of different applications in Matlab. These users are unwilling to give up Matlab for a lower-level programming language, but they need much higher performance and efficiency levels in the final applications, particularly if these applications are to run efficently on embedded computers in hand-held wireless devices.

Our study of their applications has led to the development of two new optimizations: *procedure vectorization* and *procedure strength reduction*. Our preliminary investigations indicate that these optimizations can dramatically improve the performance of Matlab applications. Specifically, our experiments show that procedure strength reduction alone yields performance improvements of 10 to 40 percent on whole applications, with much higher local gains.

The remainder of this paper is organized as follows. In the next section, we survey some of the conventional optimizations that can have high benefit on library routines written in a language like Matlab and describe how they might be useful in the telescoping langauges framework. We, then, introduce procedure strength reduction and procedure vectorization. In Section 3 and Section 4, we discuss these two optimizations in details. Section 5 presents the results of our preliminary experiments with these optimizations. Related work is surveyed in Section 6. Finally, we discuss the on-going development and future research in Section 7.

2. STRATEGIES FOR ANALYSIS AND OPTIMIZATION

The telescoping languages framework requires that decisions about which optimizations are to be applied to libraries must be made far in advance of compilation of a program that invokes these libraries. Fortunately, because it can expend enormous computing power on the library processing phase, the langauge generation system can overcome this problem by generating many different specialized variants of the same routine, each optimized for different potential contexts. Then, at application compile time, the right specialized version of a library component can be rapidly selected based on compile-time approximations of the types and properties of the parameters passed to that libarary procedure.

For this strategy to work, we need to incorporate three types of compilation technologies:

- *Library analysis strategies* are used to construct *jump functions* that can quickly summarize the side-effects to parameters passed from the script to the library, making it possible to instantly propagate parameter properties (e.g., type and value) through procedure calls at script compilation time [16]. In addition, the analysis phase identifies promising optimization points in the library and reasons backward from those points to the preconditions at the call that are needed to make the optimizations feasible.
- *Recognition and exploitation of identities* is a critical component that makes it possible to translate sequences of calls to library routines into equivalent sequences that are more efficient. In addition, library annotations can be used to help the language generation system to identify opportunities for specialized optimizations, such as reduction in strength. derived from annotations provided by the library designer.
- *Procedure specialization* is used to produce variants of the library routines optimized in advance to different potential

```
function z = jakes_mp1 (blength, speed, bnumber, N_Paths)
....
for k = 1:N_Paths
   ....
   xc = sqrt(2)*cos(omega*t_step*j') ...
      + 2*sum(cos(pi*np/Num_osc).*cos(omega*cos(2*pi*np/N)*t_step.*jp));
   xs = 2*sum(sin(pi*np/Num_osc).*cos(omega*cos(2*pi*np/N)*t_step.*jp));
   ....
end
```

Figure 2: Code fragment from a DSP application showing opportunity for replacing an expensive library call by a simpler expression.

calling contexts. At script compilation time, the compiler first uses a property-propagation procedure, which invokes jump functions at library invocation points, and produces as precise an estimate as possible of the run-time properties of each parameter at each library call in the program being compiled. Then the compiler selects the best variant for the called procedure using a fast matching procedure similar to unification (from theorem proving) [27].

In our exploration of Matlab programs for signal processing, we have discovered several important opportunities for using each of these. Because Jump functions have been dealt with extensively in the literature, we will concentrate on exploitation of identities and code specialization here.

2.1 Recognition and Exploitation of Identities

In the telescoping languages framework, library designers will be able to provide annotations that help the language generation system produce fast, effective optimizing compilers for the target language. These annotations can be used for two purposes – to help the language generation system focus on producing the right specialized variants and as a basis for a high-level optimization system that improves performance by replacing sequences of library calls with equivalent, but more efficient sequences. To support the first purpose, the annotations will need to provide hints as to how a given library routine is likely to be used. These hints might indicate which parameters are likely to be constant from call to call, and which procedures might be called in a loop with the loop index passed as one parameter (this will become important later). Many of these annotations can be derived from a well-designed collection of sample calling sequences.

To illustrate how the compiler can make use of library annotations, consider figure 2 that shows a code fragment from a real Digital Signal Processing (DSP) application. The expression in bold face is a common argument to `sin` and `cos` and does not change between the calls. Computing trigonometric functions can be expensive. If the compiler knows that the `sin` or `cos` are both used on the same argument, it can replace the two calls with a single call to a routine that computes both sin and cos of a single argument in time only slightly greater than the time to compute the `sin` alone. Such a relationship among trigonometric functions may be common knowledge, but a similar relationship between library routines can only be known through user annotations.

Another case where user annotations can be valuable is in identifying sequences of library routines that tend to be called in certain ways and, possibly, an equivalent less expensive call that can replace such a sequence in certain situations. Figure 3 shows an example of such a case that occurs in another real DSP code. This example shows that the function calls `change_form_inv` and `change_form` always occur in pairs. Moreover, the input to the latter is always the direct output of `change_form_inv`, or a simple modification of it. This can lead to opportunities for optimization by combining the two functions. In the very least, it will eliminate a copy operation and a function call overhead which can be significant in Matlab.

```
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: Code fragment from a DSP application showing pattern of library call sequences.

2.2 Procedure Specialization

One way to think of specialization is as a kind of procedure cloning, which is a well known compiler technique [10]. In Matlab programs for DSP there are many specializations that will yield significant benefits. Before coming to those, however, we will survey some of the high-payoff optimizations for these programs.

2.2.1 Useful Optimizations for Matlab

This section describes some source level transformation techniques that are highly relevant from the perspective of compiling high-level languages like Matlab.

Vectorization is a technique that has traditionally been used in compiling for vector machines [3, 4]. It turns out that it is also a very effective source transformation technique to improve Matlab programs. The reason for this is that loops in Matlab have very high overheads and often library procedure calls inside vectorizable loops can be replaced by their vector counterparts. Consider

```
function z = jakes_mp1 (blength, speed, bnumber, N_Paths)
....
for k = 1:N_Paths
   ....
   xc = sqrt(2)*cos(omega*t_step*j') ...
      + 2*sum(cos(pi*np/Num_osc).*cos(omega*cos(2*pi*np/N)*t_step.*jp));
   xs = 2*sum(sin(pi*np/Num_osc).*cos(omega*cos(2*pi*np/N)*t_step.*jp));

   % for j = 1 :  Num
   %  xc(j) = sqrt(2) * cos (omega * t_step * j);
   %  xs(j) = 0;
   %  for n = 1 :  Num_osc
   %     cosine = cos(omega * cos(2 * pi * n / N) * t_step * j);
   %     xc(j) = xc(j) + 2 * cos(pi * n / Num_osc) * cosine;
   %     xs(j) = xs(j) + 2 * sin(pi * n / Num_osc) * cosine;
   %  end
   % end
   ....
end
```

Figure 4: Code fragment from a DSP application showing opportunity for vectorization.

figure 4 that shows an example. This is the procedure `jakes_mp1` that was also shown in figure 2. The complicated looking expressions that compute `xc` and `xs` are, in fact, equivalent to the commented loop nest that occurs just below the statements. This translation from a nested loop to vector statements was done by hand by the end users, but it is something that a compiler can easily handle with current vectorization technology. In this case, the vectorization reduces the total running time of the procedure from about 50 seconds to about 1.5 seconds on a 336 MHz SPARC processor – a speedup of more than 33!

The same example also demonstrates the opportunities for *common subexpression elimination* [6]. The common subexpressions that occur in Matlab programs are often vector expressions, therefore, eliminating them can lead to significant performance gains.

Dynamic array re-shaping can be expected to occur frequently in programs written in scripting languages like Matlab. This is confirmed by our study of several DSP simulation applications. Array re-shaping can occur either implicitly when a column or row is added to a matrix, or it can be explicit through the `reshape` library call. A classic technique to optimize array re-shaping is *beating and dragging along* [1]. This technique defers data movement as much as possible and re-computes array indexes, whenever possible, in terms of the original shape. Another, source level, technique for Matlab is to compute the largest possible size for a dynamic array and allocate all of it in the beginning through the `zeros` call. This approach has been reported to result in significant performance improvement in some cases [23].

2.2.2 New Specializing Transformations

Based on our experience we have identified two types of specialization transformations that offer high promise for improvements in Matlab programs for DSP. Our study revealed that frequently a procedure is called within a loop in which the only thing that varies from call to call is the loop index (or a simple function thereof) which is passed as a parameter. This presents two opportunities for optimization.

First, if the loop containing the call to the library can be distributed around that call so that the call is the only thing in the resulting loop, it may be feasible to interchange the loop and the call and then vectorize statements in the procedure body with respect to the loop [18]. To use this strategy, which we call *procedure vectorization*, in the telescoping language framework, we would need to generate a variant of the procedure in which the loop is already interchanged to the inside and vectorized. (This variant would have extra parameters for the loop bounds.) In addition, the procedure would need to have jump functions that can be used to determine whether loop distribution around the call is possible. However, given the value of vectorization in Matlab, we expect that this would be a high-payoff optimization on many applications. Section 4 discusses this strategy in more details.

Second, we could break the procedure into two component procedures, one that computes and saves the values of variables that do not change when only the single parameter (a function of the loop index) is changing from call to call. The other computes the next value on each call from the previous one. Once these two procedures are available, the first can be called outside the loop and the second, which is presumably much more efficient, inside the loop. This optimization, which we call *procedure strength reduction* is discussed in section 3. For this transformation to be applied in telescoping languages, we need some mechanism for determining which parameters might be the only ones to vary during a loop. Otherwise the number of variants needed will be large. Programmer annotations or sample calling sequences can be very helpful here.

3. PROCEDURE STRENGTH REDUCTION

Our approach of procedure strength reduction is reminiscent of operator strength reduction [2], hence the name. In our case, however, the *strength reduction* is applied to library procedures that are the *operations*.

It is often the case that library routines are called in a loop with a number of arguments remaining loop invariant. The computations that depend only on these loop invariant arguments can be abstracted away into an *initialization* part that can be moved outside the loop. The part that is called inside the loop depends on the loop index and performs the *incremental* computation. Figure 5 illustrates this in a simple abstract case. Function f is called inside a `for` loop in which the arguments c_1, c_2, and c_3 remain invariant. Thus, f can be split into an initialization function f_μ that computes

```
for i = 1:N
        x = f ($c_1$, $c_2$, i, $c_3$);
end
                ⇓
$f_\mu$($c_1$, $c_2$, $c_3$);
for i = 1:N
        x = $f_\triangle$(i);
end
```

Figure 5: Example of Procedure Strength Reduction. Arguments c_i are invariant in the `for` loop.

the invariant part and $f_\triangle$ that computes the iteration dependent part.

Figure 6 illustrates this idea with concrete example of a real DSP application. The upper part of the figure shows the original code and the lower part shows the code after applying procedure strength reduction. In this case, procedure strength reduction can be applied to *all* the three procedures that are called inside the loops. The arguments in bold are the only ones that vary across loop iterations. As a result, computations performed on all the remaining arguments can be moved out of the procedures.

```
% Initialization
....
for ii = 1:200
  chan = jakes_mp1(16500,160,ii,num_paths);
  ....
  for snr = 2:2:4
    ....
    [s,x,ci,h,L,a,y,n0] = ...
    newcodesig(NO,l,num_paths,M,snr,chan,sig_pow_paths);
    ....
    [o1,d1,d2,d3,mf,m] =
    codesdhd(y,a,h,NO,Tm,Bd,M,B,n0);
    ....
  end
end
....
```

⇓

```
% Initialization
....
jakes_mp1_init(16500,160,num_paths);
....
[h, L] =
  newcodesig_init_1(NO,l,num_paths,M,sig_pow_paths);
m = codesdhd_init(a,h,NO,Tm,Bd,M);
for ii = 1:200
  chan = jakes_mp1_iter(ii);
  ....
  a = newcodesig_init_2(chan);
  ....
  for snr = 2:2:4
    ....
    [s,x,ci,y,n0] = newcodesig_iter(snr);
    [o1,d1,d2,d3,mf] = codesdhd_iter(y);
    ....
  end
....
end
....
```

Figure 6: Applying Procedure Strength Reduction to procedures called in `ctss`.

Notice that in the case of the procedure `newcodesig` not all the arguments are invariant inside the second level of the enclosing loop nest. Therefore, the *initialization* part for `newcodesig` cannot be moved completely out of the loop nest, resulting in two initialization components – `newcodesig_init_1` and `newcodesig_init_2`.

In principle, for maximal benefit, a procedure should be split into multiple components so that all the invariant computation is moved outside of the loops. Thus, in telescoping languages model, `newcodesig` would be a library routine that would have been specialized for a context in which only its fifth argument varies across invocations. Such a specialization of procedures is clearly context dependent. As mentioned before, example calling sequences and annotations by the library writer would be used to guide the specialization.

```
for $i_1$ = 1:$N_1$
        for $i_2$ = 1:$N_2$
        ....
                for $i_k$ = 1:$N_k$
                        x = f ($\alpha_0$ $\alpha_1$ $\alpha_2$ ... $\alpha_k$);
                end
        ....
        end
end
                ⇓
$f_{\mu_0}(\alpha_0)$;
for $i_1$ = 1:$N_1$
        $f_{\mu_1}(\alpha_1)$;
        for $i_2$ = 1:$N_2$
        ....
                $f_{\mu_{k-1}}(\alpha_{k-1})$;
                for $i_k$ = 1:$N_k$
                        x = $f_{\mu_k}$ ($\alpha_k$);
                end
        ....
        end
end
```

Figure 7: Example of a general case of Procedure Strength Reduction. Argument sub-sequence α_i represents the arguments that are invariant at level i but not at $i - 1$.

Figure 7 shows a general case for multi-level splitting of procedures. In this case α_0 ... α_k are argument sub-sequences where the sub-sequence α_i is invariant at loop level i but not at $i - 1$. Indiscriminate specialization can lead to a combinatorial explosion of clones. The extent of reduction in strength must be weighed against the extra overheads of calling the initialization (μ) procedures, the extra space needed to store library clones, and the script compilation time. Appendix A discusses some of the trade-offs involved.

Depending on the arguments that are invariant, one or more of the return values of a procedure may also be invariant. This knowledge is needed to be able to propagate the invariance property. The decision about whether to reduce a procedure in strength may affect the decision for other procedures whose arguments depend on the first procedure.

4. PROCEDURE VECTORIZATION

Vectorization of statements inside loops turns out to be a big win in Matlab programs. We can extend this idea to procedure calls, where a call inside a loop (or a loop nest) can be replaced by a single call to the vectorized version of the procedure. In the context of telescoping languages this can be done by generating an appropriate variant of the procedure.

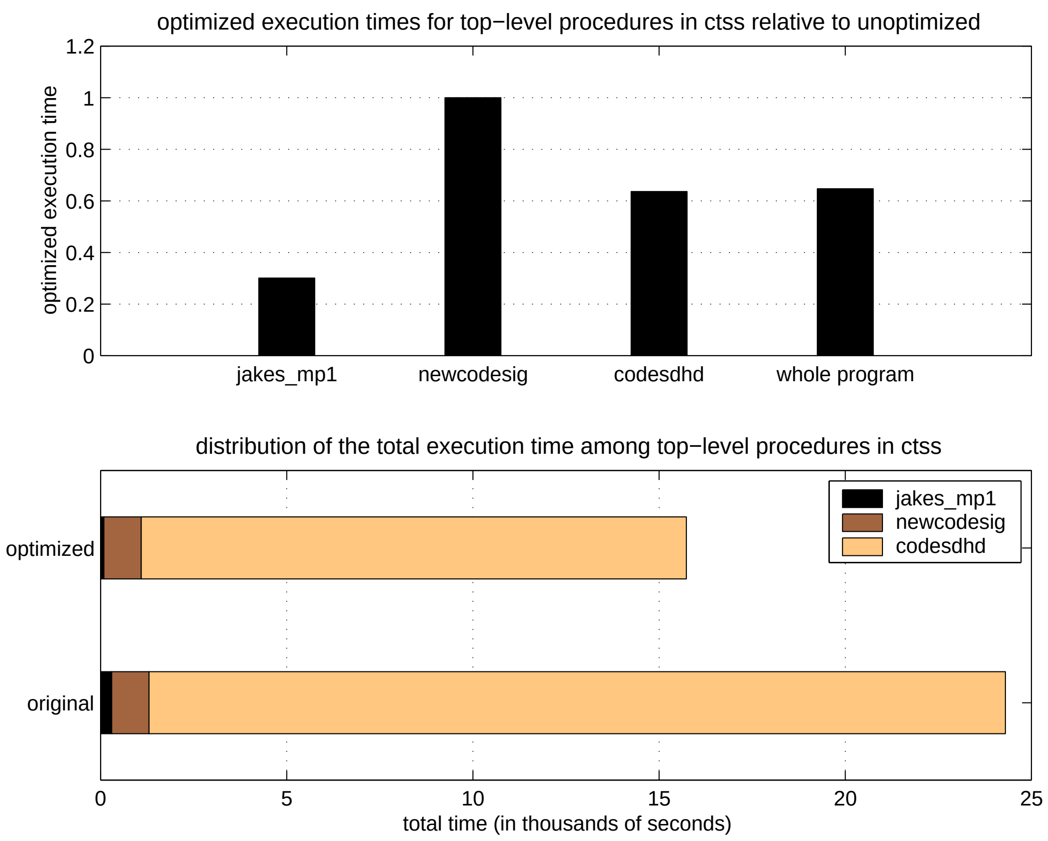

Figure 9: Performance improvements in `ctss` after applying Procedure Strength Reduction. The graphs show only the top level procedures. Almost all performance gains for the program come out of `codesdhd`.

```
% Initialization
....
chan =
jakes_mp1_vectorized(16500,160,[1:200],num_paths);
for ii = 1:200
  ....
  for snr = 2:2:4
    ....
    [s,x,ci,h,L,a,y,n0] = ...
    newcodesig(NO,l,num_paths,M,snr,chan(ii,:,:), ...
               sig_pow_paths);
    ....
    [o1,d1,d2,d3,mf,m] =
               codesdhd(y,a,h,NO,Tm,Bd,M,B,n0);
    ....
  end
end
....
```

Figure 8: Applying Procedure Vectorization to `jakes_mp1`.

Consider again the DSP program `ctss` in figure 6. It turns out that the loop enclosing the call to `jakes_mp1` can be distributed around it, thus giving rise to an opportunity to vectorize the procedure. If `jakes_mp1` were to be vectorized, the call to it inside the loop could be moved out as shown in figure 8. This involves adding one more dimension to the return value `chan`.

To effectively apply this optimization in the telescoping languages setting, it must be possible to distribute loops around the call to the candidate for procedure vectorization. This requires an accurate representation of the load-store side effects to array paramters of the procedure, which would be encapsulated in specialized jump functions that produce an approximation to the patterns accessed. An example of such a representation is a *regular section descriptor (RSD)* [8, 19]. Methods for computing these summaries are well-known.

In practice, the benefit of vectorization will need to be balanced against the cost of a larger memory footprint as well as the costs of specialization as indicated in the previous section.

5. EXPERIMENTAL EVALUATION

We studied three different DSP applications to evaluate the idea of procedure strength reduction. Procedure vectorization was applicable in one of the cases. All these applications are part of real simulation experiments being done by the wireless group in the Electrical and Computer Engineering Department at Rice University.

In order to evaluate our idea we transformed the applications by hand carefully applying only the transformations that a practical compiler can be expected to perform and those that are relevant to procedure strength reduction. The transformations included common subexpression elimination, constant propagation, loop distribution, and procedure strength reduction. The transformations were carried out at the source-to-source level and both the original as

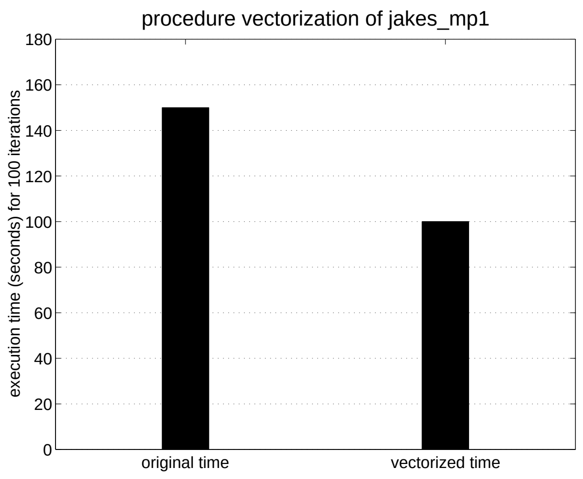

Figure 10: Performance improvement in `jakes_mp1` after applying Procedure Vectorization.

well as the transformed programs were run under the standard Matlab interpreter, unless noted otherwise.

All timing results are from experiments conducted on a 336 MHz SPARC based machine.

5.1 `ctss`

`ctss` is a program that simulates a digital communication system where the data is encoded by a convolutional code and is then transmitted over a wireless channel using a new modulation scheme that exploits the characteristics of the channel to improve performance. The top level program consists of a nested loop and procedure calls inside those loops. While these procedures are written in Matlab and are part of the program, some of these are actually being used as domain-specific library routines since they are used by multiple programs.

Figure 9 shows the improvements resulting from our transformations applied by hand. The first part of the figure shows the performance improvements achieved in various top-level procedures relative to the original running time. Notice that the procedure `jakes_mp1` achieved more than 3 fold speedup. These results do not include the dramatic performance improvement that results from vectorizing the loop inside the procedure `jakes_mp1` since that vectorization had already been performed by the user.

The whole program achieved only a little less than 40% speed improvement since most of the time in the application was spent in the procedure `codesdhd` as shown in the second part of the figure 9. After applying reduction in strength to this procedure its execution time falls from 23000 seconds to 14640 seconds, and accounts for almost the entire performance gain for the whole application.

In all cases the initialization parts of the procedures were called much less frequently than the iterative parts, effectively making the time spent in the initialization parts comparatively insignificant.

Figure 10 shows the timing results for `jakes_mp1` upon applying procedure vectorization. This procedure is fully vectorizable with respect to the loop index of the loop inside which it is called in the main program. The chart shows a performance improvement of

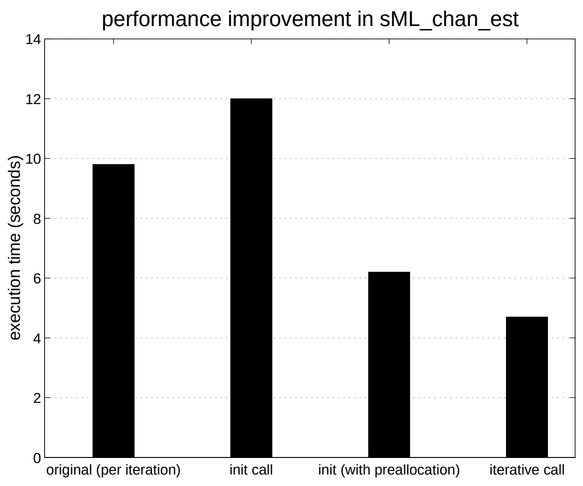

Figure 11: Performance improvement in `sML_chan_est` code after applying Procedure Strength Reduction. This procedure is called inside a complicated SimuLink application and is the most time consuming component.

33% over the original code for a 100 iteration loop. This performance gain was almost unchanged down to one iteration loop.

Vectorization resulted in a smaller improvement in performance compared to strength reduction because of two possible reasons. First, vectorization of the procedure necessitated an extra copying stage inside the procedure. Second, vectorization resulted in a much higher memory usage giving rise to potential performance bottlenecks due to memory hierarchy. For very large number of loop iterations the effects of memory hierarchy can cause the vectorized loop to perform even poorer than the original code. However, these effects can be mitigated by strip-mining the loop.

5.2 `sML_chan_est`

This application computes the delay, phase, and amplitude of all the echoes produced by the transmissions from a group of cell phones in real world scenario. The application is written under the Simulink™ environment provided with Matlab. We studied a particular procedure in the application called `sML_chan_est` that is the one where the application spends most of its time.

Figure 11 shows the result of applying strength reduction to this procedure. The chart shows the time taken by the initialization call and the iterative call. The initialization call has a loop that resizes an array in each iteration. If the entire array is preallocated (using `zeros`) then the time spent in initialization drops from 12 seconds (init call) to 6.2 seconds (init with preallocation). This illustrates the value of the techniques needed to handle array reshaping and resizing.

5.3 `outage_lb_fad`

This application computes a lower bound on the probability of outage for a queue transmitting in a time varying wireless channel under average power and delay constraints. It is a relatively small application that spends almost all of its time in a single procedure call. However, complex and deeply nested loops make the application run time very long (several hours).

[2]Simulink is a registered trademark of Mathworks Inc.

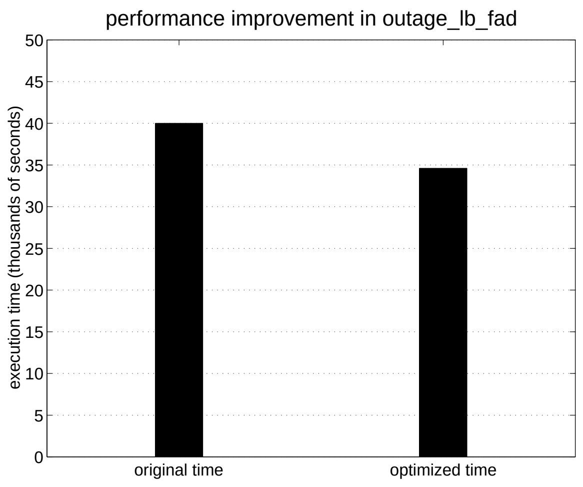

Figure 12: Performance improvement in `outage_lb_fad` after applying Procedures Strength Reduction. This is a stand-alone simulation application that spends almost the entire time in one procedure. This improvement is for the entire application all of which comes from optimizing the single procedure.

A straight forward application of procedure strength reduction reduces the run-time of this application by 13.5%. This indicates the power of our approach in benefiting even relatively tightly written code.

5.4 Effect of Compilation

Matlab has a compiler called `mcc`. The benefit of the compiler can be expected to be maximum when interpretive overheads are high. This would be the case when the application spends considerable time in deeply nested loops.

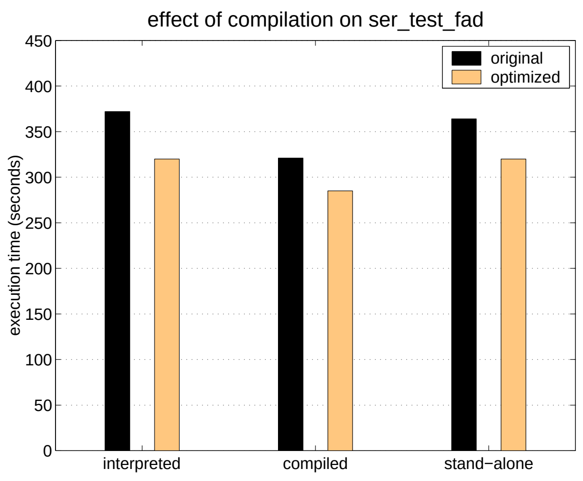

Figure 13: Effect of compilation on `ser_test_fad` that is called from the application `outage_lb_fad` and where the application spends almost all of its running time.

We studied the effects of compilation on procedure strength reduction for `outage_lb_fad`. The procedure `ser_test_fad`, that accounts for almost the entire running time for the application, has a 5-level deep loop. Figure 13 shows the results. While the performance improvement due to procedure strength reduction falls a little for the compiled code, it is still significant at more than 11%. Combining procedure strength reduction and compilation results in a performance improvement of 23.4%.

It should be observed here that this application has very high interpretive overheads due to deep loop nesting and, therefore, represents a best-case scenario for compilation. Other applications showed no significant performance gains due to compilation.

The "stand-alone" column indicates run-times for the code that was compiled to run as a stand-alone application. It is not clear why the stand-alone version of the code is slower than the compiled version that runs under the Matlab environment.

6. RELATED WORK

As mentioned before, Matlab package comes with a Matlab compiler, called `mcc`, by The Mathworks, Inc [22]. The `mcc` compiler works by translating Matlab into C or C++. Since the source of the compiler is not publicly available, the analysis it performs is not known. However, it does not seem to perform the type of advanced inter-procedural analysis proposed for Telescoping Languages.

Type inferencing for Matlab has been addressed in DeRose's PhD thesis [12] in details. Their approach is to translate Matlab programs into Fortran 90. The biggest drawback of their approach is that they handle all function calls through inlining. This, clearly, leads to a potential blowup in compilation time if library routines have to be optimized. Moreover, recursive functions can not be handled with this approach.

Building on the FALCON system based on DeRose's work, University of Illinois and Cornell University are developing a Just In Time (JIT) compiler for Matlab under their joint project called MAJIC [21]. In this context, some recent work by Menon and Pingali has explored source level transformations for Matlab [23, 24]. Their approach is based on using formal specifications for relationships between various operations and trying to use an axiom based approach to determine an optimal evaluation order. They evaluated their approach for matrix multiplication.

Currying in functional programming languages is a concept related to our idea of procedure strength reduction [11]. In the functional programming world, partial evaluation has been well studied [26]. However, many ideas from functional languages do not carry over well into languages like Matlab where functions (procedures) have side-effects.

A technique called "automatic differentiation" of programs has been successfully used in replacing high-cost procedure calls inside loops by incremental computation [15]. This technique is mainly employed in numerical codes for replacing the original computation by a less expensive, but often approximate, computation. The code to compute the increments is generated automatically from the original code through automatic differentiation [5]. While our focus is on moving strictly invariant code outside procedures, user directed automatic differentiation could complement our approach in specific contexts.

APL programming language is the *original* matrix manipulation language. Several techniques developed for APL can be useful in Matlab compilation, for example, the techniques to handle array re-shaping [1].

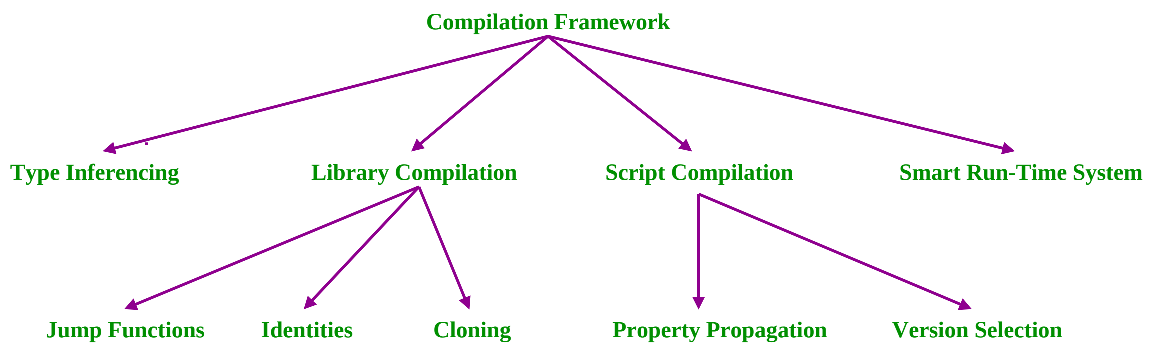

Figure 14: Compilation Framework.

Designing annotation languages for optimizing libraries is a subject of ongoing research [17].

There are many projects underway to translate Matlab into lower level languages like C, C++, or Fortran. Some of these target parallel machines using standard message passing libraries. These include the Otter system at Oregon State University, the CONLAB compiler from University of Umea in Sweden, and Menhir from Irisa in France [28, 13, 9]. The MATCH project at Northwestern University attempts to compile Matlab directly to special purpose hardware [25].

7. CONCLUSION AND FUTURE WORK

We described a compilation framework for Scripting Languages that we call Telescoping Languages. Figure 14 depicts this framework graphically. Our work focuses only on the library and script compilation components. Type inferencing is needed since Matlab programs do not have variable declarations and a smart run-time system that includes dynamic or Just In Time (JIT) compilation can be useful for Grid-computing scenarios [14].

Many existing compilation techniques are useful for compiling Telescoping Languages, especially, vectorization, CSE (common subexpression elimination), and invariant code motion. We introduced two new techniques, procedure strength reduction and procedure vectorization. Both these techniques work within the telescoping languages model by providing the benefits of inter-procedural analysis without incurring extra costs at script compilation time or requiring library sources. We evaluated procedure strength reduction by studying three real DSP applications. Application of procedure strength reduction leads to up to 40% overall application performance improvement and up to 300% procedure level performance improvement through source level transformations. For a procedure in one of the applications, applying procedure vectorization leads to a 33% performance improvement.

The idea of procedure strength reduction can be extended to include operator strength reduction. Thus, not only can the expressions that depend only on the invariant arguments be extracted out of a procedure, the remaining expressions can also be reduced further. In many cases there may not be an easy way of reducing these remaining expressions. In these cases, automatic differentiation can be helpful.

We are in the process of implementing a compiler to do these source level transformations automatically. Currently we have a front end for Matlab and plan to leverage the existing compiler infrastructure at Rice, developed for dHPF and Java.

8. ACKNOWLEDGMENTS

We thank Srikrishna Bhashyam, Dr Vidya Venkataraman, Dr Hyeokho Choi, Shriram Sarvotham, Dinesh Rajan, and Bryan Jones for making DSP codes available and for helpful discussions.

9. REFERENCES

[1] P. S. Abrams. *An APL Machine*. PhD thesis, Stanford Linear Accelerator Center, Stanford University, 1970.

[2] F. E. Allen, J. Cocke, and K. Kennedy. Reduction of operator strength. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 79–101. Prentice-Hall, 1981.

[3] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[4] R. Allen and K. Kennedy. *Advanced Compilation for Vector and Parallel Computers*. Morgan Kaufmann Publishers, San Mateo, CA, 2001. To be published.

[5] `http://www.cs.rice.edu/~adifor/`. ADIFOR Project: Automaric Differentiation of Fortran.

[6] P. Briggs and K. D. Cooper. Effective partial redundancy elimination. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–170, June 1994.

[7] B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnson, K. Kennedy, J. Mellor-Crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*. Accepted for publication.

[8] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, June 1987.

[9] S. Chauveau and F. Bodin. Menhir: An environment for high performance MATLAB. In *Proceedings of the Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Pittsburgh, PA, May 1998.

[10] K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of IEEE International Conference on Computer Languages*, Apr. 1992.

[11] H. B. Curry and R. Fayes. *Combinatory Logic*, volume 65 of *Studies in Logic and the Foundation of Mathematics*. North Holland Pub. Co., Amsterdam, 1958.

[12] L. A. DeRose. Compiler techniques for Matlab programs. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[13] P. Drakenberg, P. Jacobson, and B. Kagstrom. A CONLAB compiler for a distributed memory multicomputer. In *SIAM Conference on Parallel Processing for Scientific Computing*, volume 2, pages 814–821, 1993.

[14] http://www.hipersoft.rice.edu/grads/. GrADS: Grid Application Development Software Project.

[15] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Philadelphia, PA, 2000.

[16] D. Grove and L. Torczon. Interprocedural constant progapagation: A study of jump function implementations. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 90–99, Albuquerque, NM, June 1993.

[17] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Proceedings of the ACM SIGPLAN / USENIX Conference on Domain Specific Languages*, 1999.

[18] M. Hall, K. Kennedy, and K. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing*, pages 424–434, Albuquerque, NM, Nov. 1991.

[19] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[20] K. Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *Proceedings of International Parallel and Distributed Processing Symposium*, Cancun, Mexico, May 2000.

[21] http://polaris.cs.uiuc.edu/majic/. MAJIC Project.

[22] http://www.mathworks.com/. Mathworks, Inc.

[23] V. Menon and K. Pingali. A case for source level transformations in Matlab. In *Proceedings of the ACM SIGPLAN / USENIX Conference on Domain Specific Languages*, 1999.

[24] V. Menon and K. Pingali. High-level semantic optimization of numerical code. In *Proceedings of ACM-SIGARCH International Conference on Supercomputing*, 1999.

[25] A. Nayak, M. Haldar, A. Kanhere, P. Joisha, N. Shenoy, A. Choudhary, and P. Banerjee. A library-based compiler to execute MATLAB programs on a heterogeneous platform. In *Proceedings of the Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, Aug. 2000.

[26] http://www.acm.org/pubs/contents/proceedings/series/pepm/. ACM SIGPLAN Workshop on Partial Evaluation and Semantics Based Program Manipulation.

[27] M. S. Patterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.

[28] M. J. Quinn, A. Malishevsky, and N. Seelam. Otter: Bridging the gap between MATLAB and ScaLAPACK. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing*, Aug. 1998.

APPENDIX

A. PROCEDURE STRENGTH REDUCTION FOR NESTED LOOPS AND ITS RELATIVE GAIN

Suppose that we apply reduction in strength to a procedure that is called inside a loop nest k levels deep. We associate execution times with each reduction as shown in figure 15.

```
for i_1 = 1:N_1
    for i_2 = 1:N_2
    ....
        for i_k = 1:N_k
            x = f ($\alpha_0$ $\alpha_1$ $\alpha_2$ ... $\alpha_k$);
        end
    ....
    end
end
```

$$\Downarrow$$

```
f_{$\mu_0$}($\alpha_0$);                // $T_{\mu_0}$
for i_1 = 1:N_1
    f_{$\mu_1$}($\alpha_1$);            // $T_{\mu_1}$
    for i_2 = 1:N_2
    ....
        f_{$\mu_{k-1}$}($\alpha_{k-1}$);    // $T_{\mu_{k-1}}$
        for i_k = 1:N_k
            x = f_{$\mu_k$} ($\alpha_k$);   // $T_{\mu_k}$
        end
    ....
    end
end
```

Figure 15: Applying reduction in strength in a general case along with the execution times of various components.

The original running time, T_θ is given by:

$$T_\theta = \left(\prod_{i=1}^{k} N_i\right) \times T$$

where T is the original running time of one call to `f`. The new execution time, T_ν, for the translated code is given by

$$T_\nu = T_{\mu_0} + (N_1) \times T_{\mu_1} + (N_1.N_2) \times T_{\mu_2} + ...+$$
$$\left(\prod_{i=1}^{k-1} N_i\right) \times T_{\mu_{k-1}} + \left(\prod_{i=1}^{k} N_i\right) \times T_{\mu_k}$$

Thus, the difference in running time, $T_\triangle$, is

$$T_\triangle = T_\theta - T_\nu$$

and relative improvement in speed, $T_\triangle/T_\theta$, is given by

$$\frac{T_\triangle}{T_\theta} = 1 - \frac{T_\nu}{T_\theta}$$

or

$$\frac{T_\triangle}{T_\theta} = 1 - \frac{1}{T} \times \left[T_{\mu_k} + \frac{T_{\mu_{k-1}}}{N_k} + \frac{T_{\mu_{k-2}}}{N_{k-1}.N_k} + ...+ \right.$$
$$\left. \frac{T_{\mu_1}}{\prod_{i=2}^{k} N_i} + \frac{T_{\mu_0}}{\prod_{i=1}^{k} N_i}\right]$$

This clearly provides an upper bound on the amount of performance improvement that can be achieved with this method, which is $1 - T_{\mu_k}/T$.

In addition, this equation also provides another useful insight. It is usually the case that the sum of the running times of all f_μs is equal to the original running time T of f, i.e., $T = \sum_{i=0}^{k} T_{\mu_i}$. Clearly, to obtain maximum performance improvement the summing series in the brackets must be minimized. It is minimized if we can make all $T_{\mu_1} ... T_{\mu_{k-1}}$ values zero while maximizing T_{μ_0} given the constraint that all T_{μ_i} sum to T. This corresponds to the intuition that computation should be moved out of the entire loop nest, if possible. However, notice that except the first term, all other terms in the brackets have iteration range in the *denominator*. Thus, for any reasonably large loop the contribution from all those terms is insignificant. For example if N_k is 100 the effect of all terms after the first is of the order of only 1%. This leads to the conclusion that except for the case when the innermost loop is very short (in which case the compiler should consider loop unrolling) splitting the procedure `f` more than once may not provide significant benefits. From compiler's perspective, it need not spend much time attempting to reduce procedures multiple times for a multi-level loop since the marginal benefits after the first split are minimal.