A SOURCE-LEVEL MATLAB TRANSFORMER FOR DSP APPLICATIONS

Arun Chauhan
achauhan@cs.rice.edu

Ken Kennedy
ken@cs.rice.edu

Department of Computer Science, Rice University, Houston, TX 70005, USA.

ABSTRACT

We present an automatic source-level transformer for DSP applications written in MATLAB®. Our transformer is based on a novel approach to specify the transformations in an XML-based language resulting in a simple implementation of the transformer as a rewriting engine. In an earlier study we have identified transformations that pay off handsomely for DSP applications. This paper describes the design of the transformer and demonstrates the practical utility of the system on real DSP applications.

KEYWORDS

MATLAB, compiler, performance, XML, rewriting, source-level

1 Introduction

Our survey of several researchers in the Electrical and Computer Engineering department at Rice University revealed that MATLAB is a very popular programming language to write DSP and image processing applications. MATLAB affords very high-level operations that enable application writers to encode their algorithms easily. In addition the MATLAB package comes with domain-specific libraries, or toolboxes, which contribute to its huge popularity. Unfortunately, the *performance* delivered by MATLAB falls short of end users' requirements. The performance metric could be either the running time or the memory footprint of the application or a combination of the two, depending on the application requirements. For this paper we focus on the application running times. All the applications studied in this work are real simulation applications that have been, or are being, used by the researchers in the ECE department.

DSP applications are often coded modularly. In other words the application is often divided into functions that are called from the application. These functions themselves call other lower-level functions, and so on. Our past studies of DSP applications have shown that many frequently used functions are abstracted in a general way to be widely applicable. These general functions then form a user-level library of DSP functions. However, the functions tend to be used in certain predictable ways that are amenable to source-level program transformations resulting in significant improvements in running times.

Additionally, our past studies have also shown that several well-known program transformation techniques in compilers turn out to be high-payoff for DSP applications written in MATLAB.

®MATLAB is a registered trademark of MathWorks Inc.

The main contribution of this paper is the development of an automatic source-level transformer to carry out the transformations that we have previously discovered (or identified) as high-payoff for DSP applications written in MATLAB. The transformer works as a MATLAB rewriting system that carries out a sequence of source-level rewritings specified externally as an XML-based language that we have designed for this purpose.

2 DSP Applications

We studied a set of DSP applications and functions, written in MATLAB, which were obtained primarily from the Center for Multimedia Communications at Rice. A few were downloaded from the contributed section of the web-page of MathWorks Inc. In this paper we present results for a few selected DSP applications and functions. The functions that were studied are briefly described below.

`jakes_mp1` This function computes fast fading signals using the Jakes model. It is used in an application called `ctss` that simulates a complete system with convolutional coding and overlapping codes. The function consists of a single loop that performs trigonometric computations on matrices.

`codesdhd` This is a Viterbi decoder that uses other lower level functions. It is the most computationally intensive component of the `ctss` application. Most of its computation is evenly distributed among the lower level functions.

`newcodesig` Used to simulate the transmitter and the channel of a system within the `ctss` application, it is the second most computationally intensive component. Most of its computation is performed inside a single `for` loop.

`ser_test_fad` This procedure implements a value iteration algorithm for finite horizon and variable power to minimize outage under delay constraints and average power constraints. It is used inside an application that simulates outage minimization for a fading channel. It is invoked inside a doubly-nested loop and itself consists of a five-level deep loop-nest where it spends most of its time.

`sML_chan_est` This is a piece of MATLAB code that implements a block in a SimuLink® system that consists of several interconnected blocks. It primarily consists of a single loop that is inside a conditional statement. The procedure is the most time-consuming part of the entire simulation.

®SimuLink is a registered trademark of MathWorks Inc.

`acf` This procedure to compute auto-correlation of a signal is a part of a collection for time-frequency analysis. The computation is performed inside a simple `for` loop.

`artificial_queue` Almost all the computation in this small function is inside a loop that contains a vector statement that resizes an array. The characteristic feature of this procedure is that it typically operates on huge arrays.

`ffth` This function computes an FFT on a `real` vector in half the space and time needed for a general FFT. Essentially, this is a version of FFT specialized for `real` input.

`fourier_by_jump` This function implements Fourier analysis by the method of jumps. The implementation has been motivated by the lack of accurate results by the intrinsic MATLAB `fft` function in certain cases. It consists of two loops, only one of which is invoked depending on the value of an input argument.

`huffcode` This function computes Huffman codewords based on their lengths. The primary computation inside the procedure occurs in a doubly nested loop. The outer `for` loop encloses a `while` loop that is guarded by an `if` condition.

3 Source-level Transformations

Several source-level transformations have a high-impact on the performance of DSP applications. We described these transformations in an earlier paper, but summarize them below for easier reference.

3.1 Beating and Dragging Along

MATLAB code for DSP makes liberal use of the `reshape` primitive to facilitate ease of indexing into arrays. Unfortunately, this can often result in copying of the array at runtime. Beating and dragging along is a compiler technique to “beat” the reshaped array into the original shape by rewriting the index expressions of the reshaped array so that they refer to the original shape of the array. Further, this shape is “dragged along” as long as possible.

3.2 Loop Vectorization

Users often find it easier to think in terms of loops. However, even when running on a scalar machine, we have discovered that vectorizing loops is a big win. In a DSP function that simulates fast fading signals with the Jakes model, replacing a doubly nested loop by equivalent vector statements resulted in a speedup by a factor of 33! The reason behind this remarkable improvement is the large overheads of library calls in MATLAB that get amortized in the vectorized equivalent form.

3.3 Type-based Specialization

It is often possible to disambiguate the variable types in a MATLAB program using a process called *type inference*. Type inference is necessary because MATLAB variables are not typed. This opens up the possibility of replacing calls to generic library functions by variants that are specialized for specific argument types. We call this *type-based specialization*.

3.4 Procedure Strength Reduction

Inspired by the well-known loop optimization technique called operator strength reduction, procedure strength reduction attempts to split a function call inside the loop into two components—the slice of the function that performs loop-invariant computation (f_μ) is moved outside the loop and the remaining loop-varying slice ($f_\triangle$) remains inside the loop. The transformation is applicable when several of the arguments to a function call inside the loop are loop-invariant, which is a situation that occurs frequently in the DSP code that we studied. It is particularly useful when the function has been sliced in advance, speculatively. In that case a simple rewriting at the source level can replace the call inside the loop as shown below.

```
for i = 1:N
    ....
    x = f (c1, c2, i, c3);
    ....
end
```

$\Rightarrow$

```
fμ(c1, c2, c3);
for i = 1:N
    ....
    x = f△(i);
    ....
end
```

3.5 Procedure Vectorization

Procedure vectorization is applicable in a context that resembles that of procedure strength reduction in that there is a function call inside a loop. However, in this case the loop-varying arguments to the function may be more complicated, including an array section defined using the loop index. the following simple example illustrates the transformation.

```
for i = 1:N
    ....
    y = f (c1, c2, i, x(i));
    ....
end
```

$\Rightarrow$

```
for i = 1:N
    ....
end
y(1:N) =
    fν(c1, c2, [1:N], x(1:N));
for i = 1:N
    ....
end
```

3.6 Constant Propagation

Constant propagation is the process of propagating statically known constant values as far as possible in the code, replacing variable references by their constant values wherever possible. Combined with constant expression folding, this can lead to significant runtime savings.

These source-level transformations can have significant performance impact. For example, figure 1 shows the impact of procedure strength reduction applied to the application `ctss`, reprinted from [3].

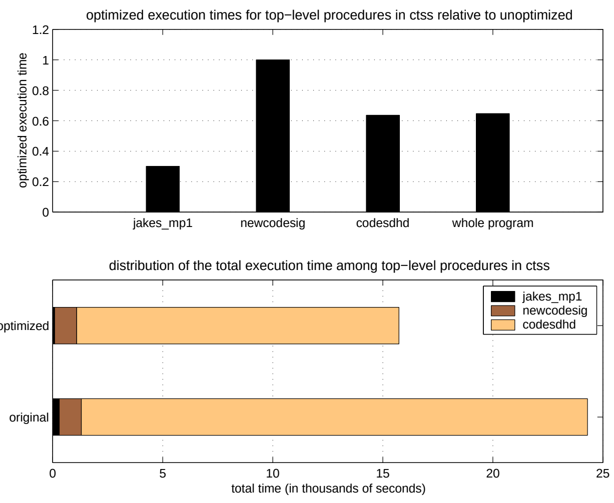

Figure 1. Effect of procedure strength reduction (reprinted from [3])

If an array's size can be computed before the array is ever used then a `zeroes` call could be used to pre-allocate it, eliminating expensive runtime array-resizing. While array sizes can sometimes be computed using static techniques, often source-level transformations must supplement the static size computation. We developed a technique called slice-hoisting to accomplish this in [4]. Figure 2, reprinted from [4], shows how this source-level transformation plays an important role in array-size computation.

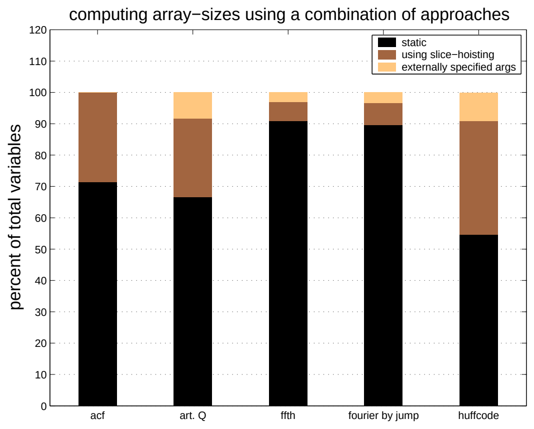

Figure 2. Importance of source-level slice-hoisting for array-size computation (reprinted from [4])

4 XML-based Language

Figure 3 shows the high-level outline of the schema for the XML specification language. The entire set of transformations are written as a sequence of *rewriting rules*. Each rule consists of a *context*, a *match* and a *substitute*. A match or a substitute consists of a sequence of statements each of which can be any of the four recursively defined types—simple statement, loop, two-way branch, and multi-way branch. The four statements that the schema allows are sufficient to express any MATLAB program structure. The "variable names" allowed under the schema are SSA renamed so that the names refer to values rather than memory locations. The *application* of a rule involves searching for the pattern specified by the match of the rule under the specified context and replacing it with the substitute as long as no dependencies are violated.

In general, an XML description following the given schema can exactly describe a control-flow graph. Indeed, the class hierarchy that is used internally in the front-end of the compiler exactly mirrors the statement structures permissible under the schema. Thus, there is a simple statement class, a loop class, and so on. Effectively, this class hierarchy defines a grammar that can generate a class of control-flow graphs and there is an isomorphism between the grammar and the schema. There are three important consequences of this mirroring:

1. Since the internal class-hierarchy is capable of representing the control-flow graph of any arbitrary MATLAB program, it follows that the XML schema has the same power of expression.

2. Control-flow graph is a widely used intermediate representation in compilers. Therefore, describing structures directly in the form of control-flow graph provides a language-independent way of specifying specializing transformations so that it can be easily used (or extended) for languages other than MATLAB.

3. The isomorphism between the specification language and the grammar of the control-flow graph (defined in terms of the hierarchy of CFG classes) greatly simplifies the process of recognition of the specified structures in a given program.

Even though the power of the language is illustrated by arguing in terms of the control-flow graph, the language is more easily understood in terms of abstract syntax tree. Since there is a one-to-one correspondence between an abstract syntax tree of a MATLAB program and its control-flow graph in our MATLAB compiler, the rewriting engine has a choice of working on either representations.

In addition to the ability to specify a template to match it is often useful to be able to refer to all occurrences of a variable or expression. In lieu of the `<substitute>` element the schema allows writing a `<replaceAllOccurs>` element that specifies replacing all occurrences of a given variable or an array section by another.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:annotation>
    <xs:documentation xml:lang="en">
      Optimizations as Specializations.
      Rice University, 2004.
    </xs:documentation>
  </xs:annotation>

  <xs:element name="specializations">
    <!-- sequence of transformation -->
  </xs:element>

  <xs:complexType name="transformation">
    <xs:complexType>
      <xs:choice>
        <xs:sequence maxOccurs="unbounded">
          <xs:group ref="stmtGroup">
          <xs:element name="context" type="preCondition"/>
          <xs:element name="match" type="stmtList"/>
          <xs:element name="substitute" type="stmtList"/>
        </xs:sequence>
        <xs:sequence maxOccurs="unbounded">
          <xs:group ref="stmtGroup">
          <xs:element name="context" type="preCondition"/>
          <xs:element name="replaceAllOccurs"
            type="replacementSpec"/>
        </xs:sequence>
      </xs:choice>
    </xs:complexType>
  </xs:complexType>

  <xs:group name="stmtList">
    <xs:sequence maxOccurs="unbounded">
      <xs:choice>
        <xs:element name="simpleStmt" type="simpleSpec"/>
        <xs:element name="twowayBranchStmt"
          type="twowayBranchSpec"/>
        <xs:element name="multiwayBranchStmt"
          type="multiwayBranchSpec"/>
        <xs:element name="loopStmt" type="loopSpec"/>
        <xs:element name="wildcardStmt"
          type="wildcardSpec"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="context">
    <xs:choice>
      <xs:element name="type" type="typeAssertion"/>
      <xs:element name="value" type="valueAssertion"/>
    </xs:choice>
  </xs:complexType>

  <xs:complexType name="replacementSpec">
    <xs:choice>
      <xs:element name="occurrence" type="stmtList"/>
      <xs:element name="replacement" type="stmtList"/>
    </xs:choice>
  </xs:complexType>

</xs:schema>
```

Figure 3. Outline of the XML schema for describing rewriting rules

As the name suggests the match element specifies a *pattern* that is matched by the rewriting engine against the procedure being compiled. Before describing how the pattern is matched, one more syntactic entity must be added to the language to make it sufficiently powerful to handle the relevant transformations—wild card. Often the match of a rule can tolerate arbitrary intervening statements, which would normally also appear in the substitute. In fact, this is the more common and the more powerful form of the match element. A wild card matches any simple or compound statement and can can match multiple occurrences according to the attributes `minCount` and `maxCount`. The default value for both is one. Each wild card element must also be supplied with a unique integer label that can be used within the substitute element to identify the corresponding match[1].

4.1 Applying the Rewriting Rules

```
input:
    rewriting rule, $\mathcal{R}$ =<$C$, $P$, $S$>
    abstract syntax tree, $T$
output:
    transformed syntax tree, $T'$
uses:
    search_pattern
    replace_pattern
    replace_occurrences

procedure rewrite
  return if the context $C$ not verified
  $L$ = list of the top-level statements in $T$
  pattern_handle = search_pattern($P$, $T$)
  if found
    if $S$ is a substitute then
      if replacing $P$ by $S$ does not violate any dependencies
        $T'$ = replace_pattern($T$, pattern_handle, $S$)
      else
        $T'$ = $T$;
      endif
    else
      $T'$ = replace_occurrences($T$, pattern_handle, $S$)
    endif
  endif
  // now repeat the process for each statement recursively
  for each compound statement, $M$, in $L$
    $H$ = abstract syntax tree for $M$
    $H'$ = rewrite($R$, $H$)
    $T'$ = $T$ with $H$ replaced by $H'$
    $T$ = $T'$
  endfor
  return $T'$
```

Figure 4. The algorithm for the rewriting engine

In order to apply a rule the specialization engine first converts the match part of the given specification into a control-flow graph. However, this is not a regular program control-flow graph but a *pattern* that must be matched against the control-flow graph of the function being compiled.

A rewriting rule, $\mathcal{R}$, can be denoted by a three tuple $<C, P, S>$ where,

C is the context that must be satisfied for the rule to be applicable

P is the pattern that must be matched for the rule to be applicable, and

S is the equivalent code that must be substituted for the matched part if the rule is applicable, or a replacement rule that replaces all occurrences of a variable or array section.

[1]Those familiar with Perl might recognize this as a generalization of tagged regular expressions.

Given the abstract syntax tree, T, of the function being compiled and a sequence of rewriting rules, $\mathcal{R}_1$, $\mathcal{R}_2$, $\ldots$, $\mathcal{R}_n$, the rewriting engine follows the algorithm in figure 4 by calling the `rewrite` procedure for each rule.

An important part of this algorithm is the subroutine `search_pattern`. The ground rules for matching include:

- A "variable" in the specification can match any expression tree, including simple variables and constants. Multiple occurrences of a variable must match exactly identical expression trees. All program variables are assumed SSA-renamed to avoid spurious matchings. However, a "variable" may be marked as "LVAL" in which case it matches only an lvalue.
- A "constant" in the specification matches a program constant of the same value.
- A "statement wild card" (`<anyStmt>`) can match any simple or compound statement.

It is important to recognize that the matching process that takes place in `search_pattern` is very similar to matching regular expressions over the entities of the abstract syntax tree.

The next section uses the XML schema described above to write the relevant transformations as rewriting rules.

5 MATLAB Rewriting System

This section illustrates how source-level transformations can be written as rewriting rules using the XML schema of figure 3. It should be noted that the rewriting engine carries out a transformation only if the transformation will not violate any dependencies. Thus, the rewriting rules induce more than simple pattern replacement—dependence information is a critical component in applying these rules.

5.1 Beating and Dragging Along

Beating and dragging along in the context of MATLAB is the elimination of the `reshape` primitive by replacing all occurrences of an array section in terms of its new indices induced by the `reshape` call. For example, if a `reshape` call changes the indices of a linear array `A` into a two-dimensional array then all subsequent uses of a section of `A` are rewritten in terms of the original linearized indices. Figure 5 shows the XML specification for this rewriting rule. It uses the second major construct provided by the XML schema—the `<replaceAllOccurs>` element.

The rule will change if the original array is reshaped into a different number of dimensions. Writing a rule for each possible number of dimensions is not so bad since in real code array dimensions rarely exceed a small number.

Another primitive function that is a candidate for beating and dragging along is array transpose. Once again, the rules will vary depending on the number of dimensions involved.

```
<specialization>
  <context>
    <type var="x" dims="2" sizes="m n"/>
  </context>
  <match>
    <!-- a = reshape(b, m, n) -->
  </match>
  <replaceAllOccurs>
    <occurrence>
      <!-- match section a(i, j) -->
    </occurrence>
    <replacement>
      <!-- replace by
        b(scalarADD(scalarMULT(scalarSUB(j,1),m)),i)
      -->
    </replacement>
  </replaceAllOccurs>
<specialization>
```

Figure 5. Rewriting rule for beating and dragging along

5.2 Loop Vectorization

Loop vectorization involves converting a scalar loop into a vector statement. Suppose that a function `f` has a vectorized version called `f_vect`. If `f` takes one input and returns one value, then a rewriting rule for vectorizing a single-loop could be written as in figure 6.

In a more readable format, the specification performs the following transformation:

```
for i = L:S:U                 for i = L:S:U
  S1*                             S1*
  b(i) = f(a(i))      ⇒      end
  S2*                         b(L:S:U) = f_vect(a(L:S:U))
end                           for i = L:S:U
                                  S2*
                              end
```

Notice that there may be other statements in the loop that are preserved. The dependence check in the rewriting engine ensures that the transformation is legal in a specific case, i.e., no dependencies are violated with respect to the statement groups S_1 and S_2.

Even though the XML specification appears complicated, it is, in fact, very simply structured. Moreover, it is an intermediate representation that the library developer or the rule writer never sees—a front-end editor presents the specification in a graphical or easy-to-edit format.

Other relevant transformations, i.e., procedure strength reduction, procedure vectorization, and constant propagation, can also be described using the XML-based language, the details of which have been omitted. Further, we have found that several other code transformations can be expressed using the specification language. However, those are beyond the scope of this paper.

```
<specialization>
<match>
  <forLoopStmt index="i">
    <lower>L</lower> <upper>U</upper> <step>S</step>
    <body>
      <anyStmt label="1" minCount="0" maxCount="unlimited"/>
      <simpleStmt>
        <function>f</function>
        <input><asection var="a"><dim><lower><var>i</var></lower>
            <upper><var>i</var></upper></dim></asection></input>
        <output><asection var="b"><dim><lower><var>i</var></lower>
            <upper><var>i</var></upper></dim></asection></output>
      </simpleStmt>
      <anyStmt label="2" minCount="0" maxCount="unlimited"/>
    </body>
  </forLoopStmt>
</match>
<substitute>
  <forLoopStmt index="i">
    <lower>L</lower> <upper>U</upper> <step>S</step>
    <body><putStmt label="1"/></body>
  </forLoopStmt>
  <simpleStmt>
    <function>f_vect</function>
    <input><asection var="a"><dim>
      <lower><var>L</var></lower> <upper><var>U</var></upper>
      <step><var>S</var></step></dim>
    </asection></input>
    <output><asection var="b">
      <dim><lower><var>L</var></lower> <upper><var>U</var></upper>
      <step><var>S</var></step></dim>
    </asection></output>
  </simpleStmt>
  <forLoopStmt index="i">
    <lower>L</lower> <upper>U</upper> <step>S</step>
    <body><putStmt label="2"/></body>
  </forLoopStmt>
</substitute>
</specialization>
```

Figure 6. Rewriting rule for Loop-vectorization

6 Related Work

Source-level transformations were pioneered by David Loveman in his classic paper [6]. Indeed, there are several similarities between the approach outlined in this paper and Loveman's approach of source-to-source transformation. He envisioned performing most optimizations at the source level and then having a relatively straightforward code generator. Our design is particularly targeted at MATLAB—or MATLAB-like languages—that allow certain simplifications due to their characteristics, such as absence of aliasing. Finally, libraries play a key role in MATLAB, therefore, most of our transformations are designed to operate at the library-level and make use of the already optimized lower-level libraries.

Procedure strength reduction and procedure vectorization have been described in detail in [3]. Beating and dragging along was described in the context of compiling APL by Abrams [1]. Source-level transformations of MATLAB have been found to be useful by other studies before us, including that by Menon and Pingali [8].

MATLAB has been attracting a lot of attention from the compilers community recently. One of the earliest attempts at compiling MATLAB was made in the FALCON compiler, and later in the MaJIC just-in-time compiler, both from the University of Illinois [5, 2].

A complete reference to MATLAB is available at the MathWorks web-site [7] and to XML is available at the web-site of the World Wide Web Consortium (W3C) [9]. In our implementation we have used the Xerces C++ libraries to parse XML [10].

7 Conclusion

While many end-users vastly prefer programming in MATLAB to lower-level languages such as C or Fortran, the benefits of MATLAB are mitigated by its poor performance. The performance could be measured in terms of the running time or memory usage or a combination of the two, depending on the application requirements. In this paper we have focused on the application running time.

We have built an automatic source-level transformer for MATLAB that can carry out the transformations that our earlier studies have found to be very high-payoff for DSP applications. These transformations are expressed using an XML-based language that we developed for this purpose. The purpose of the language is to serve as an intermediate representation for the transformations. Together with dependence analysis, our transformer is capable of performing source-level transformations of MATLAB programs, resulting in significant performance improvements of the DSP applications that we studied.

References

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

[2] George Almási and David Padua. MaJIC: Compiling MATLAB for speed and responsiveness. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 294–303, June 2002.

[3] Arun Chauhan and Ken Kennedy. Procedure strength reduction and procedure vectorization: Optimization strategies for telescoping languages. In *Proceedings of ACM-SIGARCH International Conference on Supercomputing*, June 2001.

[4] Arun Chauhan and Ken Kennedy. Slice-hoisting for array-size inference in MATLAB. In *16th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[5] Luiz DeRose and David Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Transactions on Programming Languages and Systems*, 21(2):286–323, March 1999.

[6] David B. Loveman. Program improvement by source-to-source transformation. *Journal of the Association of Computing Machinery*, 24(1):121–145, January 1977.

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

[8] Vijay Menon and Keshav Pingali. A case for source level transformations in MATLAB. In *Proceedings of the ACM SIGPLAN / USENIX Workshop on Domain Specific Languages*, 1999.

[9] `http://www.w3.org/`. The World Wide Web Consortium (W3C).

[10] `http://xml.apache.org/xerces-c/index.html`. Xerces-C: The Apache Xerces XML parser for C++.