The MATISSE MATLAB Compiler*

A MATrix(MATLAB)-aware compiler InfraStructure for embedded computing SystEms

João Bispo[1], Pedro Pinto[1], Ricardo Nobre[1,2], Tiago Carvalho[1,2], João M. P. Cardoso[1,2]
[1] Faculty of Engineering (FEUP), University of Porto, Porto, Portugal
{jbispo, pmsp, ricardo.nobre, tiago.diogo.carvalho, jmpc}@fe.up.pt
[2] INESC-TEC, Porto, Portugal

Pedro C. Diniz[3]
[3] INESC-ID Lisboa, Portugal
pedro@esda.inesc-id.pt

***Abstract*— This paper describes MATISSE, a MATLAB to C compiler targeting embedded systems that is based on Strategic and Aspect-Oriented Programming concepts. MATISSE takes as input: (1) MATLAB code and (2) LARA aspects related to types and shapes, code insertion/removal, and specialization based directives defining default variable values. In this paper we also illustrate the use of MATISSE in leveraging data types and shapes to generate customized C code suitable for high-level hardware synthesis tools. The preliminary experimental results presented here reveal the described approach to yield performance results for the resulting hardware and software references implementations that are comparable in terms of performance with hand-crafted solutions but derived automatically at a fraction of the cost.**

Keywords*— *MATLAB, Source-to-Source Compilers, Aspect-Oriented Programming, LARA, Embedded Systems

I. INTRODUCTION

MATLAB [1] is a *de facto* standard high-level programming language and interactive numerical computing environment in many domains in engineering and science, including embedded computing as it is ubiquitously used by engineers to quickly develop and evaluate their solutions. The flexibility of *MATLAB*, however, relies on runtime interpretation (and JIT compilation) as the language lacks type/shape information, in particular when manipulating arrays. Due to advances in JIT compilation and the use of pre-compiled libraries for the most processor-intensive functions, the *MATLAB* runtime currently exhibits acceptable performance. In many embedded system settings, however, the use of a *MATLAB* runtime is infeasible, due to performance and/or resource constraints. To address this potential shortcoming, a typical solution relies on the development of an auxiliary reference implementation in executable code derived from imperative languages such as C/C++ once the base or original *MATLAB* code has been validated. This reference, C/C++ code, must then in turn be validated against the output of the *MATLAB* code resulting on a lengthy and error prone process that further complicates the overall application development cycle and cost. The existence of two code specifications (the original prototypical *MATLAB* code and the reference C/C++ code) also exacerbates maintenance costs.

An alternative approach relies on a compilation tool to perform advanced analyses and generate a reference C/C++ code directly from *MATLAB* thus greatly reducing the lengthy user intervention. In our approach, we explore the use Aspect-Oriented Programming (AOP) [2][3] concepts, using the LARA language [4][5] as a vehicle to convey information to the compiler regarding types and array variable shapes. The compiler uses the user-provided information complementing and checking its consistency against the information it can derive from its own analyses.

As to the compiler infrastructure, we rely on the concept of *strategic programming* [6] to develop a modular and flexible compiler framework. The infrastructure uses TOM [7], a language extension designed to manipulate tree structures, to attain a built-in rewriting system for analyses and transformations. The end result is a synergy between compiler analysis and the user that allows the compiler to generate very high code quality from *MATLAB* specifications and the possibility to generate different C code versions, a key capability when targeting different embedded systems. In addition, the use of a well-known mature *strategic programming* system such as TOM allows our compiler infrastructure to be easily extended in comparison to other approaches using proprietary data-structures and implementations.

Even when a *MATLAB* compiler integrates sophisticated type and shape inference mechanisms, there is often the need to enforce additional characteristics or features. A common case is the multitude of target embedded architectures that need to be considered, hampering the compiler on generating efficient codes without user's intervention. In addition in some cases developers need to explore the possible solutions and to use specific data types according to accuracy requirements in the desired solution.

This paper mainly focuses on the following aspects of the MATISSE [8] compiler:

- It describes the overall architecture of the MATISSE compiler. This compiler is developed in Java exhibiting an extremely modular software architecture that can be easily augmented with specific transformations and code generation steps, thus facilitating compiler development.
- It describes the use of LARA to complement the compiler analyses for types and shapes of array variables in *MATLAB* programs.
- It presents experimental results of the use of the compiler to generate C code from *MATLAB* examples, guided by LARA specifications with different arithmetic precision contexts and specific array shapes, and targeting both software and hardware implementations (via VHDL code descriptions).

We also describe the aggressive application of array linearization for code generation, an important low-level code generation optimization. The performance results are very promising as the generated C codes are very competitive with the hand-crafted codes. The experiments described here are also clear evidence of the efficiency and effectiveness of our approach when generating different code versions from the same input *MATLAB* code, which is needed when exploring and re-targeting different architectures.

This paper is organized as follows. Section II presents the MATISSE framework and describes its architecture with particular emphasis on its code generation process. Section III presents experimental results. Section IV presents the related work and finally, Section V concludes the paper.

II. MATISSE

MATISSE is a *MATLAB* compiler framework leveraging user knowledge and translation constraints. Currently it supports a subset of *MATLAB* as its input, and generates *MATLAB* and C code (see Fig. 1). It uses LARA aspects to guide the application of source-to-source transformations in its internal high-level code representation, as well as variable type and shape definitions when generating C code from its intermediate representation. MATISSE relies on an automated process called weaving, which combines application code sources and LARA aspects to derive an augmented application at compile-time that satisfies specific concerns described in aspects.

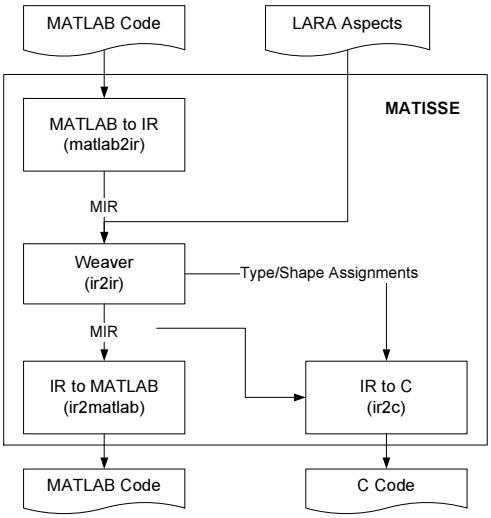

Fig. 1. The MATISSE compiler framework.

MATISSE receives as input: (1) *MATLAB* code and (2) LARA aspects related to types and shapes, code insertion/removal, and specialization directives. It generates C code that can be used by third-party design-flows targeting software/hardware systems. For the purpose of testing, monitoring, and specialization, MATISSE also generates *MATLAB* code.

The knowledge regarding data types and shapes, provided by LARA aspects, allows MATISSE to generate both more efficient C code and stylized C code that conforms to the input requirements of specific tools of a given toolchain. A common example includes the restructuring of source code and the use of statically declared array variables to be compliant with the requirements of high-level synthesis tools (e.g., *Catapult C*).

MATISSE can be used as a source-to-source code transformation and instrumentation tool allowing developers to quickly and reliably generate reference C implementations, a key step in the deployment of embedded system designs. The transformation stage of the compiler performs weaving actions such as insertion of code, definitions of types and shapes, and code specialization based on default values.

A. Compiler Architecture

The input to the MATISSE compiler is a set of *MATLAB* files and LARA aspects. The *MATLAB* files are parsed and represented as a tree-based intermediate representation (MATLAB-IR or MIR). This MIR is then input to a LARA Weaver (*ir2ir* stage). This weaver is based on strategic programming techniques for performing transformations, insertion/removal of code, as well as code specialization based on LARA aspects. Integrated in the *ir2ir* stage are components responsible for the translation of the MIR to a TOM [7] compatible IR (named TIR) enabling the use of TOM and its powerful mechanisms to program tree-transformation strategies. As part of the *ir2ir*, we developed a flexible mechanism to define the semantic rules to be used in type conversions when dealing with arithmetic expressions, of particular importance in the presence of specialized fixed-point representations.

The entire MATISSE infrastructure allows *ir2ir* to deal with a weaving process controlled by LARA. This infrastructure uses an interpreter to process LARA aspects and strategies controlling the weaving at the TIR level. TIR represents the input *MATLAB* code as expression trees directly obtained by the AST produced by the MATISSE front-end. The transformations in *ir2ir* use TOM capabilities to manipulate tree structures. Pattern matching is used to find specific patterns of data-structures in the TIR where transformations are applied. The use of TOM allows us to achieve a flexible compiler infrastructure as transformations rules are described using TOM specifications and thus extensively use strategic programming techniques [6]. By using the advanced capability of TOM regarding rewriting rules and strategies, a compiler infrastructure is able to accommodate easily new transformations as required, maintaining, we believe, a systematic, non-intrusive, approach.

The C code generator (*ir2c*) component of the MATISSE compiler allows us to generate ANSI-C reference codes from MIR representations. According to the complementary information and/or code inserted for monitoring, this C code generator phase may produce different code versions for the same *MATLAB* input program. These code versions reflect different implementations, bearing in mind the target architecture. The current version of *ir2c* supports most *MATLAB* operators over scalars, but includes restrictions when dealing with matrices. At the core of the *ir2c* back-end is a translation of MIR representations to an intermediate representation (named CIR) more suitable for generating C code. Although the CIR strictly represents C, it is augmented to help the *MATLAB* conversion, for example, by providing (several implementations of) matrix as one of its base types. CIR is also used as a bridge for *MATLAB* to C transformations.

Source *MATLAB* code generation (*ir2matlab* back-end) is straightforward from the MIR representation, possibly extended/transformed by compiler analysis and/or by the weaving process, which already represents *MATLAB* input code. The code generated by *ir2matlab* provides the user with the possibility to compare and validate results between the original and transformed variants of the input *MATLAB* code.

B. Type and Shape Inference Analyses

MATLAB is a weakly typed language whose types are implicitly converted. It is also a dynamically typed language, whose variables can be assigned without type declaration, and their types can even change at runtime. This is in stark contrast with C, which has static typing and needs the types of all variables to be declared. Additionally, in C, variables can have only one type during their entire lifetime.

When converting *MATLAB* source code to C, it is necessary to statically determine the types used by each function and by the variables manipulated. This can be a challenge, as the same *MATLAB* function can have very different C implementations, depending on the types of its variables. Fortunately, defining the types of the arguments of a *MATLAB* function is often enough to infer the remaining types of variables the function manipulates. Still, when using statically declared arrays (as opposed to dynamically allocated arrays) it is also necessary to determine the shape of the arrays at compile time.

The data-flow type inference analysis uses a simple data-flow analysis approach [9] where type information is derived by processing each *MATLAB* statement, complemented with information provided by "aspect files", which can define the type and shape of any variable. In most cases, minimal type and shape definitions are needed as a data-flow analysis allows the compiler to propagate types and shapes from the main function to the invoked functions. Types can be set by explicitly indicating the type in an aspect file, or by the use of predefined allocation functions (e.g., *zeros*, *ones*, *eye*) and assignment statements. This information is then propagated along control-flow edges of each of the variable's use and in many cases updated and extended by information determined in other assignments (e.g., information about what value the variable might contain). For instance, an allocation of a variable using the *zeros* function will allow the analysis to determine (should the parameters be evaluated to integer constant values) the shape of the array (i.e., number of dimensions of an array data type along with the respective sizes). In case of unknown compile time values for the shape of declared matrices, the compilation warns the user to define the shape by using an aspect file.

The compiler relies on intra- and inter-procedural propagation of information (e.g., constants) to determine shapes and sizes. For instance, consider a *MATLAB* function which has *N* as one if its inputs, and a call to the function *zeros(N, N)*. If the compiler determine that the function is called with the value *N* equal to 10, it can infer a 10×10 shape, and specialize the function to that input parameter. The compiler also infers shape information under the assumption that the original code would not result in a run-time execution error. For example, an expression with 1×1 shape cannot be divided by an array *Z* with more than one element, as it will result in a runtime error in the *MATLAB* execution environment. Therefore, if *Z* is not reassigned in the code then it has a 1×1 shape and can be safely represented by a single integer or floating-point element.

To address the limitations of a static type/shape inference analyses and the usual cases where the user needs to force and evaluate data types not derived by type analysis, the compiler supports the specification of types and shapes of variables as LARA aspects. The overall goal of this mechanism is to complement the capabilities of the type-inference and allow the compiler to generate code using more accurate type information. However, future work is planned to include a more advanced kind analysis stage as the one presented in [10].

C. Code Generation

The C code generation is performed directly from CIR, making use of a flexible data structure called *tensor* (see Fig. 2). This structure has its dimensions and base type (integer, float and complex) dynamically allocated, thus relying on runtime tests to determine which operation to apply to the elements of multi-dimensional arrays. The *tensor* is used to represent, in C, all variables that have more than one element. The *tensor* structure stores the shape, the size of each dimension, and the type of its elements. The number of elements of the whole matrix represented by a *tensor* is also stored for efficiency. We also developed a C tensor library with all the structures and functions to support some of the most common *MATLAB* matrix built-in functions.

```
typedef struct tensor_struct_d {  /*TENSOR data struct*/
   double* data; //array storing data
   int numel; //total number of elements (e.g., 20×3 matrix ⇒ numel=60)
   int* shape; //array storing number of elements in each dimension
   int dims;     //total number of dimensions
} tensor_d;
```

Fig. 2. Tensor data structure for double precision floating-point data types.

We now illustrate the application of the proposed approach to a *MATLAB* function implementing an FIR (Finite Impulse Response) filter as depicted in Fig. 3. This function takes as input an array named *input* and outputs an array named *output*. In the absence of any type information, our compiler would conservatively generate a C code that uses a tensor variable for the argument as well as for the function's return value as depicted in Fig. 4(a).

```
function output=fir(input, coefficients)
    NTAPS = length(coefficients);
    N = length(input);
    for i = NTAPS:1:N
        sum = 0.0;
        for j = 0:1:NTAPS-1
            sum = sum + input(i-j) * coefficients(j+1);
        end
        output(i) = sum;
    end
end
```

Fig. 3. *MATLAB fir* code example.

With the type and shape specification depicted in Fig. 5, the compiler directly uses this information to resolve the type and shape of the function arguments. This results in a very efficient and compact C code (depicted in Fig. 4(b)), in which the *tensor*

data structure is no longer required. Besides the code of the function, MATISSE also generates main functions for testing purposes, by specifying an .M or .MAT file with the values of the input arguments of the function to test.

```
#include "fir_double.h"
#include "lib/array_creators_alloc.h"
#include "lib/tensor.h"
#include "lib/tensor_struct.h"

tensor_d* fir(
tensor_d* vector_1d, tensor_d*
coefficients, tensor_d** output) {
  int NTAPS; int N; int i;
  double sum; int j;

  NTAPS = length(coefficients);
  N = length(vector_1d);
  zeros_d2(1, N, &*output);
  for(i=NTAPS; i<=N; i=i+1) {
    sum = 0.0;
    for(j=0; j<=(NTAPS-1); j=j+1)  {
      sum = sum+
    (get_tensor_d_1(vector_1d, (i-j))*
         get_tensor_d_1(coefficients,
           (j+1)));
    }
    set_tensor_d_1(*output, i, sum);
  }
  return *output;
}
```

(a)

```
/**
 * 'vector_1d' has shape [1][1024]
 * 'coefficients' has shape [1][32]
 * 'output' has shape [1][1024]
 */
double* fir(double* vector_1d,
double* coefficients, double*
output) {
  int NTAPS; int N; int i;
  double sum; int j;

  NTAPS = 32;
  N = 1024;

for(i = NTAPS; i<=N; i = i+1)  {
    sum = 0.0;
    for(j=0; j<=(NTAPS-1); j=j+1) {
      sum = sum+
            (vector_1d[(i-j)-1]*
            coefficients[(j+1)-1]);

    }
    output[i-1] = sum;
  }
  return output;
}
```

(b)

Fig. 4. *C* codes for the *fir* function generated by MATISSE: (a) for a generic case (no definition of data-types and shapes); (b) for a definition of data-types and shapes. Code in italic highlights the differences between the two codes.

```
aspectdef firSingle
    var typeDef = {
        vector_1d: "single[1][1024]",
        coefficients: "single[1][32]",
        sum: "single"
    };

    select function{name=="fir"}.var end
    apply
        def type = typeDef[$var.name];
    end
    condition $var.name in typeDef end
end
```

Fig. 5. Possible LARA code to specify types and shapes for the *fir* function.

MATISSE uses linearization of multi-dimensional arrays, whereby an element of the multi-dimensional array is accessed through a single pointer variable. Linearization has several benefits. First, simple element-wise operations are compactly executed in a single loop rather than using a loop nested structure. Second, the single allocation of storage and corresponding boundary values also enables one out-of-bound condition check per array access rather than having to perform a verification per array dimension. Lastly, it also provides other advantages regarding the size of code generated. When multi-dimensional arrays are generated without linearization, code may need to consider various pointers (when dimensions are not known at compilation time) in order to allocate the space needed for all dimensions.

Array linearization is an important transformation to reduce execution time when dealing with dynamic memory allocated data structures. Our previous experiences showed that focusing on the improvement of element-wise operations can have a significant performance impact of the generated C code, because applying the same operation (e.g., element-wise sum, subtraction, and multiplication) to each element of equally sized arrays is very common in *MATLAB* programs.

III. EXPERIMENTS

We carried out a series of experiments to evaluate the impact of the type and shape information on the performance of the generated C code in combination with the array linearization transformation, as this transformation proved to be very effective using shape and type knowledge.

A. Methodology

We use MATISSE to automatically derive a series of C codes corresponding to kernels written in *MATLAB*. We then compare the execution time after compiling the resulting code to the following three target architectures:

- Xilinx MicroBlaze (MB) processor using a single precision floating-point unit, data and instruction cache, integer divider and optimized for speed. We use the *gcc* and the *reflectc* compiler [5] and a MB cycle accurate simulator;
- PowerPC 604 (PPC) processor using the *gcc* compiler and PSIM [11], a cycle-based simulator for this processor.
- Application-specific architectures generated with the DWARV2.0 hardware compiler [12] integrated in the *reflectc* hardware/software compiler [5].

The experiments rely on the use of LARA-based data-type and shape specifications to generate C code with statically declared arrays. In the absence of array shape information, the compiler uses dynamically allocated arrays (relying on a flexible data-structure named *tensor*). In addition, we have also carried out experiments where we use different precision requirements (e.g., double and float).

B. Benchmarks

We conducted these experiments using the *MATLAB* codes briefly described in Table I. We include *fsubband* and *gridlt*, two critical functions from the *3D Path Planning* and the *MPEG audio encoder* applications [4]. In addition, we include an application to perform correlation using FFTs and with 3D matrices as input. This application named *cfd*, uses forward and inverse 2D FFTs provided by a *MATLAB* function able to perform N-dimensional FFTs (identified as *fft2D*), and a dot product between 3D matrices (identified as *cpx*).

C. Results

The experimental validation of the C code generated by MATISSE for all benchmarks in Table I showed results within a relative error of $1e10^{-4}$ when compared to the *MATLAB* results (with both codes using double precision data types).

Fig. 6 shows a performance comparison of C code automatically generated by MATISSE from *MATLAB* using single precision types, when compared with C manual versions developed by experienced programmers. The code was compiled with GCC compilers, with optimization level O2, and was evaluated using PPC and MB processor simulators. For these experiments, we consider that the shapes of matrices are known at compile time and thus matrices are implemented as statically declared arrays. We do not include comparisons for *cpx* and *cfd* examples as we do not have manual C code for those functions. Values greater than one for the ratios between execution times presented in Fig. 6 indicate performance improvements of MATISSE generated C code over manual C code. For the examples used, the performance of the C codes generated by MATISSE is very close to the performance of the C code developed manually, both for the PPC and for the MB processors (between 95-99% of the original performance). In some cases, the MATISSE-generated codes for the MB processor outperform the manually-derived C codes (see *gridIt* and *fft2d*).

TABLE I. DESCRIPTION SUMMARY OF THE MATLAB CODES USED. (1) NESTED LOOPS PERFORMING OPERATIONS WITH ARRAYS OF DOUBLES.

Benchmark (name used)	Description	Base Types/ Shapes Used	Code Structure - MATLAB #LoCs
FIR (fir)	32-tap Finite Impulse Response filter	1D array of doubles	(1) - 27 lines
filter_ subband (fsubband)	Filter bank, key component in MPEG audio compression	1D arrays of doubles	(1) - 16 lines
gridIterate (gridIt)	Laplacian's iteration function for a 3D Path Planning app.	3D arrays of doubles (32×64×16)	(1) - 38 lines
fftRealImag (fft2D)	Forward and inv. N-D FFT using complex values	2D arrays of doubles complex	(1) - 114 lines
cpxdotprod (cpx)	Dot product between two 3D matrices as input	3D arrays of doubles complex	(1) - 16 lines
cfd	Correlation in the Freq. Domain using 3D matrices as input	3D arrays of doubles complex	2 calls to fft2D, 1 call to cpx, and sum of 2 L×2D matrices - 34 lines

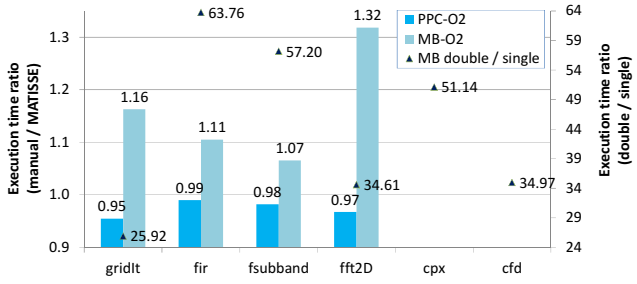

Fig. 6. Performance between code versions of the benchmarks when executed in PowerPC (PPC) and MicroBlaze (MB) processors.

Fig. 6 also compares the execution times for the MB when using single and double precision floating point (MB single vs. double). As the MB only has hardware support for single precision floating point, double precision operations are emulated in software and have a significant impact when used (i.e., a slowdown between 30× and 61×, in the tested cases). In these cases, it is very important to avoid unwanted uses of double precision data types when single precision code is the target. The following are 3 situations avoided by MATISSE when targeting single precision data types: (1) intermediate variables being inferred as double precision data types; (2) use of double precision for constants instead of single precision (e.g., `2.0` vs. `2.0f`); (3) implicit casts of double to float due to the use/generation of functions with parameters of type double instead of float. When these issues are not addressed, we have measured slowdowns of up to an order of magnitude for single precision code. These issues are cumbersome and error-prone to correct when manually adapting double precision to single precision C code, and are well-suited for an automatic approach, such as the one proposed by the MATISSE compiler.

Additional experiments using dynamically allocated arrays showed increases in performance between 1.04× (for *fir*) and 2.04× (*gridIt*). Furthermore, dynamically allocated arrays are not supported by hardware compilers (including the most representative academic/industrial high-level synthesis tools). This clearly shows the importance of the specialization provided by shape inference and user information.

Fig. 7 shows the results obtained for the function *fsubband* considering the use of manual C code, and the C code obtained from the *MATLAB* code using MATISSE. The C codes were then compiled to the MB processor and to custom FPGA hardware using the *reflectc* compiler [4], which uses DWARV2.0 [12] as its hardware generator. We used three compiler strategies (identified as 0, 1, and 2) for each target. The results reveal the efficiency of the C code produced by MATISSE when considering a complete toolchain from *MATLAB* to microprocessor binaries and to hardware. The results even show important improvements over the use of manual C code. Noticeable are the latencies' improvements of 36.1%, 29.5%, and 32.8% for SW 0 and HW 2 with single precision data types, and for HW 2 with double precision, respectively.

IV. RELATED WORK

Given the importance of *MATLAB* there have been research efforts to improve the execution of JIT *MATLAB* compilers. A recent example is the compiler presented in [13] which performs function specialization based on the runtime knowledge of the types of the arguments of the functions. Yet, a lack of a *MATLAB* runtime for most embedded systems has led to the development and research on how to best translate *MATLAB* programs into equivalent C code.

DeRose and Padua developed the FALCON environment [14] that translates *MATLAB* to FORTRAN90 code. They leverage an aggressive use of static and type inference for base types (doubles and complex) as well as shape (or rank) of the matrices. Other researchers have explored the reuse of storage for array variables across a *MATLAB* code thus reducing the memory footprint of the corresponding C reference code [15]. Joisha et al [16][17][18] focused on type and shape inference

techniques. Researchers have also relied on a mix of type inference approaches and user's provided information. For instance, [19][20] use annotations to specify data types and shapes and simple type inference analysis and target VHDL code specification for hardware synthesis onto FPGAs. We specifically note that our approach is focused on embedded implementations of the *MATLAB* programs. In this context, an efficient translation to an implementation language (mainly C) is needed. One of the possibilities is to consider a subset of *MATLAB* allowing feasible and efficient static compilation. One example using such a subset is the embedded *MATLAB* (a subset of *MATLAB*) to C code translation existent in the MathWorks Real-Time Workshop [21], which allows the user to embed annotations with *MATLAB* code to achieve C code implementations for embedded systems [22].

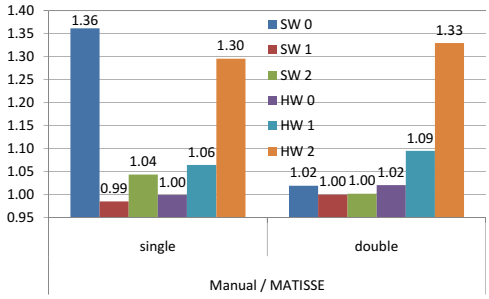

Fig. 7. Results for function *fsubband* considering SW (MicroBlaze: MB) and HW implementations. The results show execution time improvements between manual C code and C code generated by MATISSE, and the use of 3 compiler sequences (0, 1, and 2).

The popularity of the *MATLAB* language is also reflected in the similar languages that have been proposed. Examples of those languages are *Scilab* [23] and *Octave* [24]. Recently, a *Scilab* to C translator [25], named *Sci2C*, has been developed. The *Sci2C* translator focus entirely on embedded systems, and is completely dependent on annotations embedded in the *Scilab* code to specify data sizes and precisions. Our compiler distinguishes from *Sci2C* as it is able to generate C code even without annotations and specialization of the generated C code can be achieved without polluting the original code (*MATLAB*, in our case). Furthermore, *Sci2C* requires that the size of arrays is fixed and statically known while our compiler also produces C code (including calls to *realloc*) when those sizes are unknown.

The use of user-specified rules and strategies for code transformations has been focused by various authors. As with the compiler to optimize Octave programs presented in [26], our compiler relies on a strategic programming approach. They present an approach, based on Stratego/XT [27], in the context of code specialization for Octave [24] programs. They consider loop vectorization and partial evaluation of types and values. In our approach we extend the strategic programming approach with an AOP approach provided by LARA.

In this work we described a mechanism for conveying information about types and shape/rank similar in spirit with the notion of Aspects [4]. Previous work has proposed aspect-oriented extensions to *MATLAB* [28] and an aspect-oriented code transformation language for *MATLAB* [29]. Other authors have explored aspect-oriented approaches for *MATLAB* [30], but do not use aspects to specify complementary information that can be used by compilers to produce more efficient implementations. For our compilation infrastructure we used a strategic programming approach, based on TOM, and we focused on C code generation, especially for embedded computing systems, along with the aggressive use of the linearization and tensor internal representation to deal with unknown shape/rank dimensions.

Some of the benefits of using aspect-oriented extensions to *MATLAB* have been already exposed by a number of software metrics [31]. However, the inclusion of the versatility of LARA [4][5] on providing strategies for *MATLAB* tools adds further suitable dimensions to the use of AOP approaches in the context of the *MATLAB* programming language.

V. CONCLUSION

This paper presented MATISSE, a compiler infrastructure for *MATLAB*. MATISSE relies on LARA aspects for specifying data types, shapes, and code instrumentation and specialization. The MATISSE approach not only assists in the early code development phases but the implementation phases as well, by providing data type and shape information for the subsequent code generation steps. When coupled to the *reflectc* [5] compiler, MATISSE provides a complete toolchain able to generate code for both software and hardware components. The experimental results presented here reveal the described approach to yield performance results for the resulting software references implementations that are comparable in terms of performance with hand-crafted solutions.

ACKNOWLEDGMENT

This work was partially funded by the European Comission's Framework Programme 7 (FP7) under contract No. 248976 (REFLECT project) and by FEDER/ON2 and FCT project NORTE-07-124-FEDER-000062. The authors are also grateful to the FCT support provided through grant PTDC/EIA/70271/2006.

REFERENCES

[1] MATLAB – the Language of Technical Computing, http://www.mathworks.com/products/matlab

[2] G. Kiczales, "Aspect-oriented programming," ACM Computing Surveys (CSUR). Volume 28, Issue 4es (Dec. 1996), 154.

[3] J.D. Gradecki, and N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*, John Wiley & Sons, Inc., NY, USA, 2003.

[4] J.M.P. Cardoso, et al., "LARA: An Aspect-Oriented Programming Language for Embedded Systems," in *Proc. Int. Conf. on Aspect-Oriented Software Development (AOSD'12)*, Potsdam, Germany, March 25-30, 2012, pp. 179-190.

[5] J.M.P. Cardoso, Pedro Diniz, José Gabriel Coutinho, and Zlatko Petrov (eds.), *Compilation and Synthesis for Embedded Reconfigurable Systems*, Springer, May 2013.

[6] R. Lämmel, E. Visser, and J. Visser, "Strategic programming meets adaptive program-ming," in *Proc. 2nd Int. Conf. on Aspect-Oriented Software Development (AOSD'03)*, Boston, Mass., March 17-21, 2003. ACM, New York, NY, USA, pp. 168-177.

[7] J.-C. Bach et al., *TOM Manual*, Version 2.7, May, 2009 (http://tom.loria.fr)

[8] MATISSE, http://specs.fe.up.pt/tools/matisse/

[9] A. Aho, J. Ullman, M. Lam and R. Sethi, *Compilers: Principles, Techniques and Tools*, Addison Wesley, 2006.

[10] J. Doherty, L. Hendren, and S. Radpour, “Kind analysis for MATLAB,” in *Proc. ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA'11)*. New York, pp. 99-118.

[11] A. Cagney, “PSIM - Model of the PowerPC Architecture,” March 2012. http://sources.redhat.com/psim/

[12] R. Nane, et al., “DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler,” in *Proc. 22nd Int. Conference on Field Programmable Logic and Applications (FPL'12)*, Oslo, Norway, 29-31 Aug. 2012, pp. 619-622.

[13] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge, “Optimizing MATLAB through Just-In-Time Specialization,” in *Int. Conf. on Compiler Construction (CC'10)*, March 2010, pp. 46–65.

[14] L. De Rose, and D. Padua, “Techniques for the Translation of MATLAB programs into Fortran 90,” in *ACM Trans. Program. Lang. Syst.*, 21, 2 (Mar. 1999), pp. 286–23.

[15] P. Joisha, and P. Banerjee, “Static array storage optimization in MATLAB”, in *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'03)*, June 09-11, 2003, San Diego, CA, USA.

[16] P. Joisha, and P. Banerjee “Correctly detecting intrinsic type errors in typeless languages such as MATLAB,” in *Proc. ACM Conf. on Array Processing Languages (APL'01)*, June 2001, ACM, New York, NY, USA, pp. 7–21.

[17] P. Joisha, and P. Banerjee, “The MAGICA type inference engine for MATLAB,” *Proc. 12th Int. Conf. on Compiler Construction*, April 2003 (LNCS, vol. 2622). Springer, Berlin, 2003, pp. 121–125.

[18] P. Joisha, P. Banerjee, “An algebraic array shape inference system for MATLAB,” in *ACM Transactions on Programming Languages and Systems*, 2006; 28(5), pp. 848–907.

[19] A. Navak, M. Haldar, A. Choudhary, and P. Banerjee, “Parallelization of MATLAB Applications for a Multi-FPGA System”, in *Proc. 9th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'01)*, Rohnert Park, CA, USA, May, 2001, pp. 1-9.

[20] P. Banerjee, at al., “Automatic Conversion of Floating Point MATLAB Programs”, in *Proc. 11th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'03)*, Napa, CA, USA, 2003.

[21] Real-Time Workshop: Generate C code from Simulink models and MATLAB code, http://www.mathworks.com/products/rtw/

[22] H. Zarrinkoub, and G. Martin, “Best Practices for a MATLAB to C Workflow Using Real-Time Workshop,” MATLAB Digest - Nov. 2009.

[23] Scilab, http://www.scilab.org/

[24] The Octave Home Page. http://www.gnu.org/software/octave/

[25] Scilab 2 C - Translate Scilab code into C code, http://forge.scilab.org/index.php/p/scilab2c/

[26] K. Olmos, and E. Visser, “Turning dynamic typing into static typing by program specialization in a compiler front-end for Octave,” in *Proc. 3rd IEEE Int. Workshop on Source Code Analysis and Manipulation (SCAM'03)*, 2003, 26-27 Sept. 2003, pp. 141-150.

[27] Stratego/XT, http://www.stratego-language.org.

[28] J.M.P. Cardoso, J.M. Fernandes, and M. Monteiro, “Adding Aspect-Oriented Features to MATLAB,” in *SPLAT! 2006, Software Engineering Properties of Languages and Aspect Technologies*, workshop affiliated with AOSD 2006, March 21, 2006. Bonn, Germany.

[29] J.M.P. Cardoso, et al., “A Domain-Specific Aspect Language for Transforming MATLAB Programs,” in *Domain-Specific Aspect Language Workshop (DSAL'2010)*, part of AOSD'2010, March 15-19, 2010, Rennes & Saint Malo, France.

[30] T. Aslam, J. Doherty, A. Dubrau, and L. Hendren, “AspectMatlab: An Aspect-Oriented Scientific Programming Language”, *in Proc. Aspect Oriented Software Development Conference (AOSD)*, March 2010, ACM, NY, USA, pp. 181-192.

[31] P. Martins, et al., “Program and Aspect Metrics for MATLAB,” in *Proc. 12th Int. Conf. on Computational Science and Applications (ICCSA'12)*, June 18-21, 2012, Salvador de Bahia, Brazil. LNCS 7336, Springer-Verlag, pp. 217–233.