A Modular MATLAB Compilation Infrastructure Targeting Embedded Systems

Ricardo Nobre

Faculdade de Engenharia da Universidade do Porto (FEUP)
Departamento de Engenharia Informática, Rua Dr. Roberto Frias s/n,
4200-465 Porto, Portugal
Email: ricardo.nobre@fe.up.pt

Abstract. Dynamic programming languages make extremely difficult for static compilation to achieve efficient results. MATLAB is one of the programming languages offering dynamic features and abstractions to easily perform matrix operations. These features, the vast portfolio of domain-specific libraries, and the tools to simulate MATLAB system specifications, make MATLAB a widespread programming language, ranging from embedded to scientific computing domains. However, MATLAB programs are usually seen as models and require aggressive specializations in order to achieve efficient embedded computing implementations. For example, the same MATLAB model may be implemented using double, single, and/or fixed-point precision data types. This kind of specialization needs analysis steps exploring different alternatives, measuring the impact on accuracy and precision errors, before deciding to a specific implementation. Furthermore, a selection of a particular implementation may have to be done according to the target embedded computing architecture. This paper presents a MATLAB compiler infrastructure able to generate different implementations from the same MATLAB input code. Specifically, our MATLAB compiler is modular and can be easily extended by using a strategic programming approach, and specific implementations can be derived by aspect-oriented information specified by the user. We also present the use of the current version of the compiler for translating MATLAB codes to C code implementations for embedded computing systems using different input type specifications, thus simulating different target scenarios.

Keywords: modular, MATLAB, C, compilation, specialization

1 Introduction

MATLAB [2] is a *de facto* standard high-level language and an interactive numerical computing environment for many areas of application, including embedded computing. It is widely used by engineers to quickly develop and evaluate their solutions [3]. By including extensive domain-specific and visualization libraries, MATLAB substantially enhances engineers productivity [3].

The flexibility of MATLAB, however, comes at the cost of interpretation (and JIT[1] compilation) and lack of type/shape information. In MATLAB the same identifier can be used to hold various data types and array shapes (i.e. the number of elements of each dimension of a multi-dimensional array) throughout the code execution. This makes it very handy for quick program prototyping but it is not ideal for static analysis, and in many cases precludes the application of advanced program transformations. The flexibility of MATLAB programs results in lower execution performance. This lack of performance is typically addressed by the development of an auxiliary reference implementation once the MATLAB code has been validated. This amplifies the maintenance costs as the programmer must now deal with multiple language versions of the application.

An alternative approach relies on a compilation tool to perform advanced analyses and to generate a reference C code directly from MATLAB, thus avoiding the lengthy user intervention. This automatic approach, however, is also fraught with obstacles as there are inherent limits to what information static analyses can extract. Rather than pushing the limits of static analysis - sometimes too conservative due to the lack of complementary information about the problem - we have adopted the pragmatic approach of allowing the user to control some of the aspects of the translation process at a very high-level. In our approach the compiler is aware of separate type and shape specifications provided by the user as a vehicle to convey information to the compiler regarding the types and array shapes of the used variables. The compiler uses the user-provided information and complements/checks 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[2] to accomplish a modular and flexible compiler framework. The infrastructure uses Tom[3] to achieve a built-in rewriting system for analysis, transformations, and code generation. The end result is a synergy between compiler analysis and the user that allows the compiler to generate very high quality code from MATLAB specifications and the possibility to generate different C code versions, important when targeting different embedded systems. In addition, the use of a well-known and mature strategic programming system as Tom allows our compiler infrastructure to be easier to extend in comparison to an approach using proprietary and specific data-structures and implementations.

This paper is organized as follows. Section 2 describes the compilation infrastructure. Section 3 presents some experimental results. Section 4 describes related work. Finally, Section 5 draws some conclusions and presents future work.

[1] Just-in-time compilation is a method that compiles code at run-time in order to improve the runtime performance of computer programs.

[2] Strategic programming is "a generic programming idiom for processing compound data such as terms or object structures", relying on the "separation of two concerns: basic data-processing computations vs. traversal schemes". Traversals are composed by passing the former as arguments to the latter [4].

[3] Tom is a language extension that provides pattern matching facilities for manipulating tree structures and XML documents [5,6].

2 Compilation Infrastructure

Our compiler infrastructure is organized according to the component diagram depicted in Fig. 1: a single front-end (*mat2tir*) and two back-ends (*tir2mat* and *tir2c*). The compiler is modular enough to allow back-ends for other output programming languages.

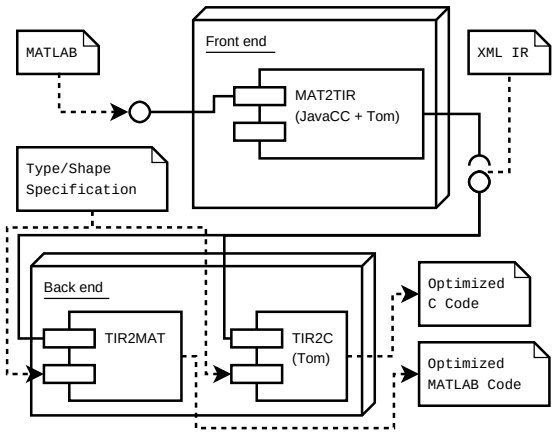

Fig. 1. Component diagram of MATLAB Compilation infrastructure.

The mat2tir component uses a parser generated from an LL MATLAB grammar [4] with JavaCC[5] to parse MATLAB M-files. The abstract syntax tree (AST) generated by the parser is then converted to a Tom [5,6] intermediate representation (IR) exported in XML format. This *mat2tir* component is internally structured as a traditional front-end parser and a Tom IR creator step.

The IR AST grammar is declared by Gom [5,6]. Gom provides a syntax to define ASTs. The Tom IR structure is built from the JavaCC AST by recursively visiting each node, respecting the specification declared using Gom syntax.

The Tom IR represents the input MATLAB code as expression trees directly obtained by the AST produced by the parsing front-end. The IR also represents the line numbers of MATLAB statements, code annotations, and information about types and shapes of the MATLAB expressions. Information about types and shapes is unknown until an inference engine is executed. Annotations are embedded in the MATLAB comments (they start with *%@*) and are used in our approach to identify specific MATLAB code locations (join points), or are inside a separated file loaded by the backend.

[4] LL grammars are a subset of contex-free grammars parsable from left to right, producing the leftmost derivation of the input.

[5] JavaCC is a parser generator for use with Java applications [8]

Table 1 depicts MATLAB code for a recursive factorial, part of the correspondent Tom IR and the generated C code specifying the input variable n as scalar and integer.

Table 1. (a) MATLAB code, (b) XML IR for the factorial function and C code generated from MATLAB with the following type and shape information input to the compiler: *n:int32:1x1*.

a)

```
1 function x = factorial(n)
2     if n == 1
3       x = 1;
4     else
5       x = factorial(n−1);
6       x = n∗x;
7   end
```

b)

```
1  <Start>
2  <FunctionMFile>
3    <ConcIdentifier>
4     <Identifier>
5      "x" "1"
6     </Identifier>
       ...
7     "1"
8  </FunctionMFile> "1"
9  </Start>
```

c)

```
1 #include "tensor.c"
2 void factorial(int n, int∗ x) {
3   if(n==1) {
4     (∗x) = 1;
5   } else {
6     factorial(e_minus(n, 1),
            &(∗x));
7     (∗x) = e_times(n,(∗x);
8   }
9 }
```

A *tir2x* form component then takes the Tom IR as input and, by combining it with the type/shape specifications written by the user, applies specific IR-based transformations or code generation steps. We have developed two backend applications, namely *tir2mat* and *tir2C*.

The MATLAB Generator backend (*tir2mat*) is used for validation purposes and generates MATLAB code from the IR. This MATLAB code is helpful as it gives the user the possibility to compare results obtained by using transformed and/or specialized code with the original code. This can be a way to evaluate and analyze accuracies in the case of type and word-length transformations, and to trace and debug in the case of inserting code for monitoring.

The C Generator backend (*tir2C*) allows us to generate C code from the MATLAB input code with the possibility to use the complementary information added by the user.

The transformations in the Tom IR use Tom [5,6] capabilities to manipulate tree structures. Pattern matching is used to find specific patterns of data-structures in the Tom IR where transformations are applied. The use of Tom allows us to achieve a flexible compiler infrastructure as transformations rules are described using Tom specifications.

Regarding the back-end, and when translating typeless/shapeless languages such as MATLAB to C, a key step is the definition of specific types and in particular array shapes. To this extent we have implemented a limited form of data-flow analysis [9] as well as an external user-provided mechanism for static determination of types and shapes of variables as described next. The idea is that just declaring information about the argument variables of a function,

the external knowledge gained by the inference engine is sufficient to statically determine the shapes and types of all, or at least almost all variables.

2.1 Pipeline architecture and Strategies

The compiler is structured as a set of distinct engines attached to a single pipeline. An engine has both accesses to the compiler internal variables, like symbol tables, and the IR representing the input code at each step.

Fig. 2 depicts a diagram representing the main classes backend.

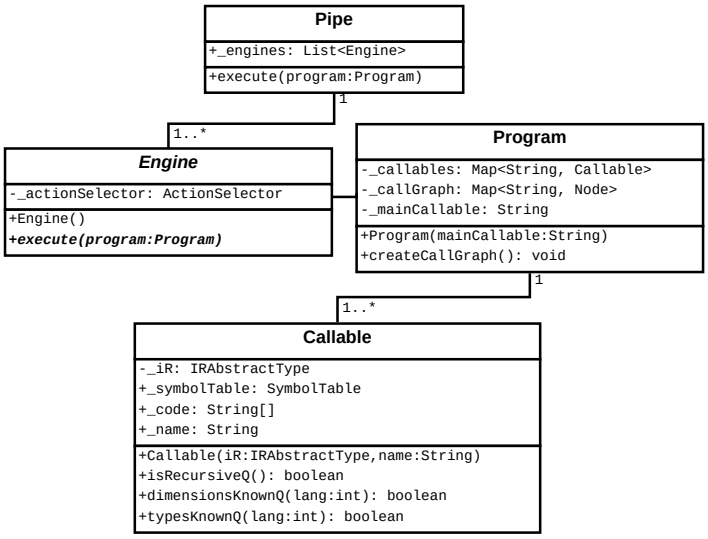

Fig. 2. Main classes composing a backend instance.

The main class of the *tir2c* backend, *Tir2C*, instantiates a single pipeline (*Pipe* class) and attaches engines (extend *Engine*) in the order they should execute. Engines can manipulate the IRs representing the input MATLAB files, can change the internal state of the compiler (e.g. symbol tables, generated code), or both. For instance, the C generator engine, *CGeneratorEngine* class, is responsible for generating the C code from the imported IR, using the type and shape information about variables inferred by the inference engine (*InferenceEngineFilter* class) with information directly specified in external specification files, in case this files exist and contain type or shape declarations. The script inliner engine (*ScriptInlinerEngine*) is an example of an engine that inlines IRs for multiple MATLAB script files into a single IR.

The pipeline is executed after the call graph creation method, which returns an instance of the *Program* class holding information about all M-Files, each represented by an instance of the *Callable* class. A Callable object holds all

compiler internal variables, symbol tables and IR structure representing an M-File. When executing (*execute()* method) the pipeline the *Program* object is passed as argument, and then each engine attached to the pipeline is executed (*execute()* method) with it passed as argument. Engines have access to all the internal variables (e.g. IRs and symbol tables) of the compiler.

2.2 Engine creation methodology

The execution of an engine is performed by passing each Callable from the *Program* object to a new instance of a class extending an abstract class called *ActionSelector*. The class extending the *ActionSelector* is always described using primitives from the Tom language extension in a *NAMEActionSelector.t* file, where *NAME* represents the name of the engine.

The logic to traverse the IR is encapsulated in *NAMEActionSelector.t* by a Tom strategy that for each Tom IR structure found in the IR calls a method from a class extending the *ActionSelection* class, which encapsulates the only methods that are allowed to change the compiler internal state for a given *Callable* passed as argument.

The *ActionSelector* of an engine acts as a selector for choosing the appropriate method to call from the *ActionSelection* component. Having the logic for the traversal of the IR completely separated from the actions to perform during the traversal of each IR node in separated files allows interesting software development techniques as the *ActionSelector* and the *ActionSelection* parts of each engine require different levels of understanding about the compiler architecture.

Extending the ActionSelector to create a new engine only requires knowledge about the IR and the Tom primitives used to implement strategies for traversing the IR. Extending the *ActionSelection* neither requires knowledge about how to specify Tom strategies nor how the IR is structured, as the *ActionSelection* part only deals with updating the symbol table and the IR and all the methods needed are automatically created when extending the abstract class. Transformations are only specified, using a strategic programming approach, in the class extending the *ActionSelector* abstract class. These classes are represented in 3.

2.3 C Generation

In terms of the implementation, the generated C code relies on a very flexible data structure called tensor. The tensor is used to represent, in C, all array variables. This structure has its dimensions and base type (integer, real and complex) dynamically allocated, thus relying on run-time tests to determine which operation to apply to the elements of the multi-dimensional array, only in case the precise dimensions are not known at compilation time (e.g., when such information is not determined by the static shape analysis stage). The number of elements of the whole structure is also stored for efficiency. We also developed a C tensor library with all the structures and functions to support the most common MATLAB matrix built-in functions.

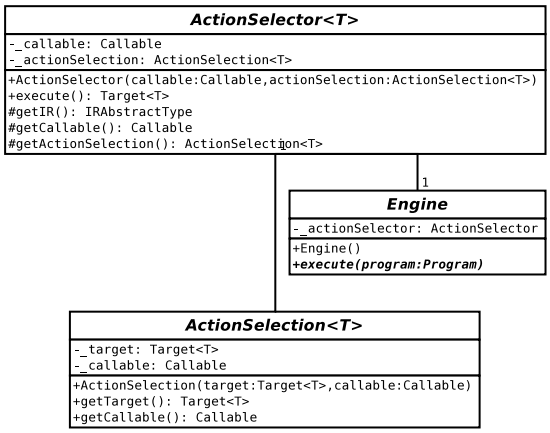

Fig. 3. Action Selection Framework abstract classes.

2.4 User-provided Type and Shape Information

To address the limitations of a static type/shape inference analyses and/or to force implementations according to user knowledge about the target implementations, our MATLAB compiler supports the specification of types and shapes of variables as a form of annotations in a separate file. This mechanism complements the capabilities of the type/shape-inference and allows the compiler to generate code using more accurate type information and/or specialized implementations.

The code generated by our compiler when considering the specification of the type and shape of the arguments of the top function results in a specialized code without type checks which gives an efficient C implementations, both in terms of performance and memory, whenever such specialization can be used in the final implementation. The generated code is clean when compared to human made code.

3 Experimental Results

We carried out a series of experiments to evaluate our MATLAB compiler and the impact of the type and shape information on the performance of the generated C code.

In our experimental evaluation we used a set of 11 code kernels originally written in MATLAB. We used our compiler to automatically derive C codes. These kernels are drawn from simple arithmetic operations in some cases using linear algebra operations such as dense matrix-vector multiplications, from digital signal processing applications, and from an industrial application.

One of the set of kernels used is an industrial code delivered by Honeywell for the REFLECT project [7,22]. The code is related to a Three-dimensional (3D)

Path Planning algorithm. This kernel plans a 3D path between the current position of an autonomous vehicle and a predetermined goal position. In this paper we consider the most computationally intensive task of the 3D path planning algorithm: the gridIterate function. This function consists of 4 nested for type loops and uses 3D arrays. In the experiments we consider a partial map of the environment of size $32 * 64 * 16$.

We then compared the execution time of each sample code using different type and shape declaration files. To compare the performance of the base types the programmer can specify, we carried out experiments where the different base types corresponding to distinct precision requirements were used. These include the specification of double (default), float, and fixed-point precision.

Preliminary tests comparing the number of cycles needed to run executables generated with Mathworks MCC tool[6] and the C code generated by our compilation infrastructure shown that, as expected, the compiled C code is much faster than Mathworks JIT approach (between 2 and 4 times faster). These tests were performed on an x86 microprocessor (in our case the Intel Core 2 Duo), not at all a processor for embedded systems, as it is one of the few platforms compatible with executables generated by MCC (execution requires run-time compiled libraries). Therefore MCC can not be considered as an alternative to our compiler for use with embedded systems.

3.1 Methodology

Experiments were conducted using the SimpleScalar/ARM simulator [10,11] to execute the object output by gcc[7] with the *-O3* flag from the C code generated by our compiler from MATLAB input files. The simulator models Intel StrongARM SA-11xx microprocessors, has been validated against a large collection of workloads, and is believed to be within 4% of real hardware for performance estimation [11]. The StrongARM family of microprocessors implements the ARM V4 instruction set architecture. This simulator reports the cycle count with high accuracy.

Note that we only declared the type and shape of the parameters of each function in the M-files, thus asserting that for all call-sites the same information holds. The information about all other variables is inferred at compilation time by evaluating the expressions where the input parameters are present. Furthermore, for these examples, we relied on type inference to determine the type and shape of all other local variables. This is clearly not the most flexible situation and we will continue to explore scenarios where type/shape specialization makes sense. In addition to the impact of shape information we also tested the performance improvement gained by using single precision floating point rather than the standard double precision.

[6] The MATLAB compiler "prepares MATLAB file(s) for deployment outside of the MATLAB environment, generates wrapper files in C or C++, optionally builds standalone binary files, and writes any resulting files into the current folder, by default." [23]

[7] The GNU Compiler Collection [24]

3.2 Results and Discussion

We show in Fig. 4 the results obtained when using the SimpleScalar/ARM simulator.

The results consider the execution time improvements between float vs. double, fixed vs. double, and fixed vs. float implementations. The last group of columns (labeled "average") corresponds to the average speedup for our set of code kernels.

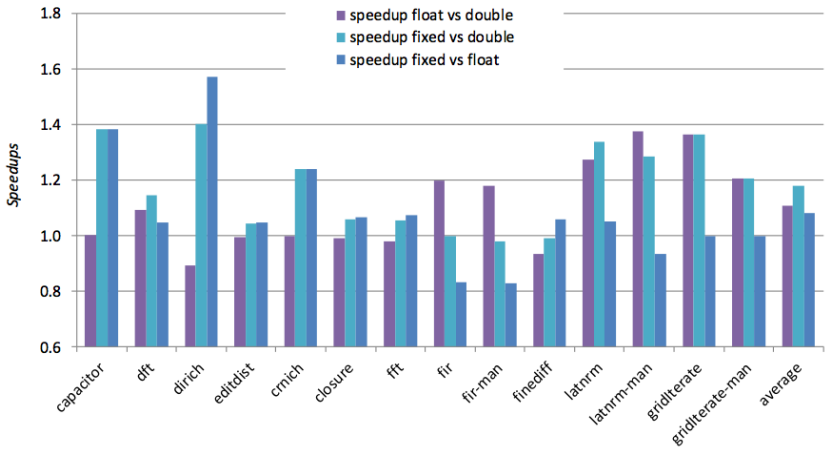

Fig. 4. Speedups for different implementations of the benchmarks.

As the performance results reveal there is on average an increase in performance of 1.16 times and 1.09 times when using the fixed-point arithmetic representation versus double and float precision, respectively. This is mainly due to the use of integer operations, that are faster than instructions operating on float and double data types for the specific ARM architecture used.

The fact that the performance relation between double and float is not consistent between benchmarks (mainly because of float-to-double and double-to-float conversions) is a proof that the specification of types and shapes can be valuable, even when the target architecture does not change. For instance, when considering fixed vs. float implementations, a speedup of 1.57 times is achieved for the dirich code and a slowdown is achieved for the fir code.

4 Related Work

Given the significance of MATLAB in early algorithm design and evaluation, many other research efforts focused on translating MATLAB code to languages that provide a more efficient execution environment (see, e.g., [12,13,14]). Due to

the importance to execute directly MATLAB programs there has been research efforts to improve the execution of JIT MATLAB compilers. A recent example is the compiler presented in [15] that performs function specialization based on the runtime knowledge of the types of the arguments of the functions. DeRose and Padua developed the FALCON research project [12] that translates MATLAB to FORTRAN90 code with performance improvements that resemble the ones we report here for codes of similar complexity. They leverage an aggressive use of static and type inference for base types (double and complex) as well as shape (or rank) of the matrices.

Other researchers have also relied on simple type inference approaches such as ours. Banerjee et al [16,17] have used annotations to specify data types and shapes, simple type inference analysis and target VHDL code specifications for hardware synthesis onto Field-Programmable Gate Arrays (FPGAs). Later on, this work resulted on the commercial AccelFPGA compiler which supports, among others, annotations to define the shape of MATLAB variables.

As with the compiler to optimize Octave programs presented in [18] our compiler relies on a strategic programming approach. However, in our approach we extend the strategic programming approach (using Tom in our case instead of the Stratego approach in [18]) with user information regarding types and shapes. One of the closest related work is the embedded MATLAB (a subset of MATLAB) to C code translation existent in the MathWorks Real-Time Workshop [19] which allows the user to embed annotations with MATLAB code to achieve C code implementations for embedded systems [20].

The popularity of the MATLAB language is also reflected in the similar languages proposed. Examples of those languages are Scilab and Octave. Recently, a Scilab to C translator [21], named Sci2C, has been proposed. The Sci2C translator focuses entirely on embedded systems and it is completely dependent on annotations, to specify data sizes and precisions, embedded in the Scilab code. Our compiler distinguishes from Sci2C as it is capable 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 not statically known.

5 Conclusion

This paper presented a compiler infrastructure for MATLAB. The infrastructure relies on the concept of strategic programming, and especially on the Tom framework to analyse, transform and generate code from a Tom intermediate representation of the input MATLAB program.

The architecture of the compiler infrastructure is sufficiently flexible and modular to allow an easy integration of compilation stages by taking advantage of Tom rewriting capabilities, in the form of what we called the "Action Selection" framework.

One important stage of the compiler is the generation of C code. In order to achieve more efficient and/or specialized C code our approach allows users to add information about types and array shapes. This additional information is of paramount importance in most cases, especially in cases where advanced type/shape inference analysis needs to be conservative, and in the cases where one needs specialized implementations dependent on the target system and/or specific concerns. Additionally, the compiler infrastructure is sufficiently modular to allow the integration of other output languages generators, other than MATLAB and C.

Our ongoing work focuses on the use of an aspect-oriented language to specify more powerful type and shape information (possibly parametrically) as well as high-level code transformations as recently proposed. These code transformations will allow users to easily explore specific code optimizations at the MATLAB level.

The comparison of code generated by *tir2c* with code generated by Sci2C (performance, memory, and readability-wise) is work in progress.

Future plans include a static analysis (possibly further extended with profiling information) to identify automatically the variables in a MATLAB program whose types/shapes are more important for the type/shape inference engine.

Acknowledgment

This research has been partially funded by the Portuguese Science Foundation Fundao para a Ciência e Tecnologia (FCT) under research grant PTDC/EIA/70271/2006.

The author also acknowledges the support of the FP7 EU-funded project REFLECT for the access to MATLAB codes related to industrial applications.

The author acknowledges support from João M. P. Cardoso, affiliated with Faculdade de Engenharia da Universidade do Porto (FEUP) and Pedro Diniz, affiliated with Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento em Lisboa (INESC-ID.

References

1. Ricardo Nobre, Joo M. P. Cardoso, Pedro C. Diniz 'Leveraging Type Knowledge for Efficient MATLAB to C Translation', paper presented at CPC2010, Vienna University of Technology, Vienna, Austria, July 9, 2010.
2. MATLAB the Language of Technical Computing, `http://www.mathworks.com/products/matlab`.
3. Richard Goering, "Matlab edges closer to electronic design automation world" EE Times, 10/04/2004 `http://eetimes.com/electronics-news/4050334/Matlab-edges-closer-to-electronic-design-automation-world`
4. R. Lämmel, E. Visser, and J. Visser, Strategic programming meets adaptive programming, In Proc. 2nd Intl Conf. on Aspect-Oriented Software Development (AOSD'03), Boston, Mass., March 17 - 21, 2003. ACM, New York, NY, USA, pp. 168-177.

5. Jean-Christophe Bach et al.: Tom Manual - Version 2.7, http://tom.loria.fr
6. Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles, Tom: Piggybacking rewriting on java, In 18th International Conference on Term Rewriting and Applications (RTA07), Paris, France, June 26-28, 2007, Springer LNCS 4533, pp. 36-47.
7. João M. P. Cardoso et al., REFLECT: Rendering FPGAs to Multi-Core Embedded Computing, Book Chapter, Reconfigurable Computing, Springer.
8. The JavaCC Home, https://javacc.dev.java.net/
9. A. Aho, J. Ullman, M. Lam and R. Sethi, Compilers: Principles, Techniques and Tools, Addison Wesley, 2006.
10. T. Austin, E. Larson, and D. Ernst, SimpleScalar: An infrastructure for computer system modeling, Computer, 35(2), 2002, pp. 5967.
11. SimpleScalar Version 4.0, Test Releases, http://www.simplescalar.com/v4test.html
12. De Rose, L. and Padua, D. Techniques for the Translation of MATLAB programs into Fortran 90. ACM Trans. Program. Lang. Syst. 21, 2 (Mar. 1999), pp. 28623.
13. Joisha, P. G. and Banerjee, P. 2007. A translator system for the MATLAB language: Research Articles. Softw. Pract. Exper. 37, 5 (Apr. 2007), pp. 535-578.
14. G. Almsi , D. Padua, MaJIC: Compiling MATLAB for Speed and Responsiveness, In Proc. of the ACM 2002 Conf. on Programming Language Design and Implementation (PLDI'02), June 17-19, 2002, Berlin, Germany.
15. Amina Aslam, and Laurie Hendren, McFLAT: A Profile-based Framework for Loop Analysis and Transformations, in the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC2010), October 7 - 9, 2010, Rice University, Houston, Texas, USA.
16. A. Navak, M. Haldar, A. Choudhary and P. Banerjee, Parallelization of MATLAB Applications for a Multi-FPGA System, in Proc. of the 9th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'01), Rohnert Park, Calif., May, 2001.
17. P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, R. Uribe, Automatic Conversion of Floating Point MATLAB Programs, in proc. of the 11th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM03), Napa, Calif., 2003.
18. K. Olmos, E: Visser, Turning dynamic typing into static typing by program specialization in a compiler front-end for Octave, in Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM03), 26-27 Sept. 2003, pp. 141- 150.
19. Real-Time Workshop: Generate C code from Simulink models and MATLAB code, http://www.mathworks.com/products/rtw/
20. Houman Zarrinkoub, and Grant Martin, Best Practices for a MATLAB to C Workflow Using Real-Time Workshop, MATLAB Digest - November 2009, http://www.mathworks.com/company/newsletters/digest/2009/nov/matlab-to-c.html
21. Scilab 2 C - Translate Scilab code into C code, http://forge.scilab.org/index.php/p/scilab2c/
22. REFLECT, FP7 EU Project: http://www.reflect-project.eu.
23. MATLAB R2011b Documentation, MATLAB Compiler. http://www.mathworks.com/help/toolbox/compiler/mcc.html
24. GCC, the GNU Compiler Collection. http://gcc.gnu.org/