Highly-Optimizing and Multi-Target Compiler for Embedded System Models

C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc

Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneideres, and Michael von Wenckstern*
Software Engineering, RWTH Aachen University, Aachen Germany, vonwenckstern@se-rwth.de

1 INTRODUCTION

In embedded and cyber-physical systems software and physical components are deeply intertwined and mostly interact with their environment [45]; examples in automotive industry include ESC1, ABS1, TCS/ASR1, EPS1, LKAS1, ACC1, PW1, and AFL1. Large German automotive manufacturers develop embedded systems using Component and Connector (C&C) models [8, 48], which are later translated to C/C++ code and deployed on embedded devices often exhibiting custom processor architectures. In test-driven modeling [57], which is part of agile development processes [1], it is very important that developers and/or continuous integration systems can test changes quickly and automatically - long compilation and test execution times may hinder the overall development process as slow tests may kill the developer feedback loop [43].

Our first contribution is a compiler toolchain infrastructure, designed with respect to automotive software development needs, for the C&C modeling language EmbeddedMontiArc [27]. This powerful cross-platform compiler serves a series of targets including x86/x64, as well as ARM and WebAssembly; thereby enabling deployment and execution on a variety of devices ranging from micro-controllers to tablets and smartphones. This compiler infrastructure is also highly configurable and combines different frameworks such as Octave [13] or Armadillo [44] as well as various BLAS (Basic Linear Algebra Subprograms) backends such as OpenBLAS [25] or Intel MKL [27] under the hood. The toolchain is completed by a model-based testing framework, a web-based IDE, and middleware integration.

Our second contribution are algebraic and threading optimizations for C&C models that can be derived automatically. These optimizations allow simulations on computationally complex tasks such as local traffic system scenarios for convoys, data mining or image processing tasks on standard personal computers in a small amount of time.

Our third contribution is an evaluation on four applications to measure the runtime performance for the generated code which is produced by compiling C&C models (an image clustering provided by the MathWorks homepage on matrix modifier applications) with the presented infrastructure on an x64 architecture as native code, as well as running it in a web-browser on a normal PC, and a smartphone. Thereby, we compare the runtime performance of the generated code when using the Octave
The mathematics framework with the one when using the Armadillo library. During this case study we compare the runtime performance against the performance of code generated by MATLAB/Simulink, which is the de-facto C&C modeling framework in German automotive industry today, with equivalent models for native applications, and the runtime performance against MathJS code for web-browser applications.

Finally, as an important contribution of our work we made our entire toolchain including its source code and all models needed for our comparison with other frameworks public available from [http://www.se-rwth.de/materials/ema_compiler](http://www.se-rwth.de/materials/ema_compiler). Additionally we produced videos showing our complete setup and experiments, so that all results and experimentation steps are transparent to all readers. We encourage the reader to inspect these materials and use them for their own research.

The outline of this paper is the following: Section 2 presents the image clustering algorithm used as running example in paper. Section 3 shortly introduces the C&C modeling language EmbeddedMontiArc, tools needed for a multi target compiler platform such as Emscripten, LLVM, clang and gcc, as well as the two mathematics frameworks Octave and Armadillo. Section 4 contains our first contribution the complete compiler toolchain infrastructure. Section 5 presents our second contribution, the optimizations to speed up the runtime performance using different mathematics libraries, different BLAS backends, and running it on different targets as well as it compare run-time results when using Simulink or implementing the algorithms directly in JavaScript or in Java. Finally, Section 7 concludes this paper.

2 RUNNING EXAMPLE

We introduce two compelling running examples to thoroughly explore our approach of a highly optimizing multi-target compiler. The first one is an ObjectDetector employing spectral clustering for image recognition. The second one is a C&C model called MatrixModifier which performs different matrix calculations. In practice, these kinds of operations are pervasive in many domains including navigation and routing where maps are represented as large matrices and need to undergo various interpolations and transformations [20].

Object Detector. Unsupervised learning has proven to be an important tool-set for automated data understanding and pre-processing. One prominent application is image segmentation, e.g. enabling self-driving cars to separate objects in a scene captured by a camera. While there is no perfect clustering algorithm and the best fit is highly domain-dependent, spectral clustering methods are known to exhibit an outstanding performance in many complex applications [41, 49]. The basic idea is depicted as a C&C model in Figure 1 and can be summarized as followed. Let \( x_{ij} \in [0, 255]^3 \) be the 3-dimensional pixel value of an image at position \((i, j)\) encoding a point in the HSV (hue, saturation, value) color space. For better handling, an \( N \times M \) image is represented as a vector, mapping a position \((i, j)\) to the vector index \(M \cdot i + j\), where \( N \) and \( M \) are the height and the width of the image, respectively. First, a symmetric similarity matrix \( W \in \mathbb{R}^{N \times M} \) is computed. Consequently, the entry of \( W \) at position \((h, k)\) provides information on the similarity of the two pixels corresponding to the indexes \( h \) and \( k \). Pixel similarity may be defined in terms of distance, color, gradients, etc. Second, the so-called graph Laplacian is computed as \( D = \text{diag}(W1_{NM}) \) with \( 1_{NM} \) being an \( N \cdot M \) dimensional column vector full of ones. Often it is advantageous to use the symmetric Laplacian

\[
L_{sym} = D^{-\frac{1}{2}}LD^{-\frac{1}{2}} = \text{diag}(1_{NM}) - D^{-\frac{1}{2}}WD^{-\frac{1}{2}}
\]

as outlined in [49]. For efficiency reasons, as they do not carry valuable cluster information the identity matrix and the minus depicted in red are often dropped in concrete implementations obtaining the simplified term highlighted in blue. Note that computing \( L_{sym} \) requires a matrix inversion on the diagonal matrix \( D \) as well as two matrix multiplications. Now the eigenvectors corresponding to the smallest eigenvalues of \( L_{sym} \) have to be computed where \( k \) is the number of clusters we want to detect. If this number is unknown an index can be used to estimate it [12]. Furthermore, let \( U \) be an \( NM \times k \) matrix with the \( k \) eigenvectors as its columns. Each row of this matrix represents one pixel in a feature space which should be easier to cluster by the standard \( k \)-means algorithm. Finally, the ObjectDetector can separate the objects shown in Figure 2 (a) as depicted in Figure 2 (b).

For our experiments we use the MATLAB implementation by Asad Ali [3], which is based on the original spectral clustering algorithm [49]. We extend the implementation by an image loader to allow images as input data. The second example depicted in Figure 7 shows a component performing several common matrix modifications. To better grasp the performance benefits gained by using a smart code generator, this example is chosen to be rather abstract but quite computation
intense when the code is written or generated in a naive way. Similar models are often utilized in computationally intense applications where a lot of information is stored and modified in matrices scattered over multiple components. Further details will be provided in the respective sections.

The full source of both examples modeled in EmbeddedMontiArc is available from [14].

3 PRELIMINARIES

C&C Modeling and EmbeddedMontiArc. The main aim of model-driven development is to model the domain knowledge, which in our case is the functionality of embedded systems. In contrast to a general purpose language programmer, the model driven developer should not care about performance issues like multi-threading or optimized algebraic routines. A good modeling language allows the expression of the domain knowledge intuitively and a smart compiler tool chain produces high-performance code to efficiently test and simulate the functional models.

EmbeddedMontiArc [27] is such a textual domain specific language to model logical functions in a C&C based manner. Especially, EmbeddedMontiArc places emphasis on the needs of the embedded, CPS (Cyber-Physical Systems), as well as the automotive domains and is particularly used for controller design [19]. As an example, the elaborate numeric type system allows declarations of variable ranges as well as accuracies. Furthermore, units are an inherent part of signal types and hence tedious and error-prone tasks like checking the physical compatibility of signals (weights cannot be added to lengths) as well as unit or prefix conversion (feet to meters, km to m) are delegated to the EmbeddedMontiArc compiler.

Figure 3 (a) - (c) show how the spectral cluster in Figure 1 is modeled in EmbeddedMontiArc. Figure 3 (b) and (c) represent the subcomponents of (a). As is inherent to C&C languages, the main language elements of EmbeddedMontiArc are component and connect. While the former defines a new component followed by its name, e.g. in line 1 of Figure 3 (a), the latter connects two ports of subcomponents with each other, e.g. in lines 10-15 in Figure 3 (a). The behavior of a component can be either defined by a hierarchical decomposition into subcomponents as in the case of Figure 3 (a) or using an embedded behavior description language.

The principal behavioral language used throughout this work is a matrix based math language, used in lines 6-8 and 5-14 of Figure 3 (b) and (c), respectively. As EmbeddedMontiArc is strongly typed, errors like wrong matrix dimensions are caught at compile-time, in contrast to MATLAB/Simulink where this is a runtime exception. A matrix property system leverages performance optimizations as well as further compatibility checks in the compilation phase. If a matrix is declared to be diagonal, both memory and computational complexity of the generated code can be reduced dramatically. If furthermore, the domain of the matrix is constrained to non-negative entries, it can be inferred that the matrix is positive-semidefinite allowing the inversion function to be used on it and guaranteeing that the result will be positive-semidefinite again [9].

As our spectral clustering example shows each EmbeddedMontiArc component resides in its own text file so that multiple users or teams can work on one large C&C modeling project simultaneously. Compared to other C&C languages where models are stored in a proprietary binary format in one single file, such as Simulink’s sifx format, this facilitates the usage of version control systems, merging, and conflict solving but also textual searching in model repositories.

The EmbeddedMontiArc language family has been developed using MontiCore 5, a language workbench particularly known for its leading language composition technology [22] and hence facilitating the integration of the main C&C architecture description language with behavior description, configuration, testing, and other sub-languages. However, EmbeddedMontiArc is not just a modeling language frontend but rather a holistic model driven software engineering methodology in the sense that all executable code and binary files are generated directly from EmbeddedMontiArc models, i.e., the developer does not have to deal with any target languages, compilation, and linking issues. Therefore, the heart of the proposed tool chain is an extensible highly-optimizing cross-platform compiler presented in the following sections.

Finally, the question arises how components developed in EmbeddedMontiArc can be tested and quality assured in continuous integration environments. Writing unit tests for the generated code using a framework of the target platform not only goes against our holistic model driven engineering principles, forcing the developer to understand the target details and to produce platform-dependent code but is also infeasible in the long run as the interfaces of the generated code might change. Instead we propose a so called stream language allowing a test developer to write black-box tests for EmbeddedMontiArc components by providing sequences of values for the input ports (test data) and the corresponding expected output.

In Figure 4 a stream test example is given for the normalizedLaplacian component of Figure 3 (b) with the generic parameter n=3. In lines 3 and 4 we provide the test inputs for the ports degree and similarity whereas lines 5-8 specify the expected output of the port nLaplacian. The tick keyword separates the values of an input sequence allowing one to test the component behavior for arbitrarily long input and output streams. This is particularly important for stateful components such as PID controllers or automata. By specifying tolerance ranges using the +/− operator as in lines 5-7 we can easily cope with floating point outputs, rounding errors, and numerical inaccuracies.

Multi Target Compiler Platforms. Emscripten [56], developed by Alon Zakai and Mozilla, is a Low Level Virtual Machine (LLVM) to JavaScript or WebAssembly (WASM) compiler.

LLVM is a modular and reusable compiler framework for arbitrary programming languages. The LLVM framework [28] enables transformations at link-, install-, run-, and idle-time, to optimize memory usage, runtime speed, and program size.

Clang is a C, C++, Objective C/C++, OpenCL C to X86-32, X86-64, and ARM compiler based on the LLVM framework. Clang is compatible with GCC, but uses less memory and compiles faster while still delivering programs with a better performance. On the other hand, GCC addresses more targets such as PowerPC or embedded processors. It supports any target architectures where 1rt is 8, 16, 32 or 64-bit wide. A new target platform can easily be added by providing a machine description containing an algebraic formula for each available instruction [47].
Figure 3: EmbeddedMontiArc code of selected components for the spectral clusterer model

Figure 4: Stream test model of the NormalizedLaplacian component.

WebAssembly [51] is a size- and load-time-efficient binary instruction format for a stack-based virtual machine, and it aims to execute at native speed. WebAssembly runs on nearly all smartphone and desktop browsers [39], as well as on the node.js server.

A generator toolchain supporting GCC, Clang, and Emscripten can run its code on multiple targets at the best possible performance. These targets include many embedded platforms, e.g. microcontrollers used in drones, web-browsers on computers, smart phones, or tablets as well as native x86/x64 applications using optimizations such as threading and SIMD (Single Instruction Multiple Data) instruction sets, e.g. SSE, MMX and AVX, but also native GPU (Graphical Processing Unit) support for CUDA and OpenCL.

Octave and Armadillo. As explained above, the behavior of an EmbeddedMontiArc component can be implemented using a math language. For basic mathematical operations the so called Basic Linear Algebra Subprograms (BLAS) specification defines a set of low level routines such as matrix additions and multiplications. Examples of available BLAS implementations include the Intel Math Kernel Library [53] as well as OpenBLAS [55]. To get the last ounce of performance out of the executing processor, these BLAS libraries rely on hardware specific optimizations such as SIMD parallelization and multi-threading.

To generate component behavior code from the math model in the component implementation, e.g. in lines 4-11 of Figure 3 (c) the EmbeddedMontiArc compiler does not use BLAS libraries directly. Instead it lets the user choose between an Octave [13] or an Armadillo [44] backend. These two high level linear algebra and scientific computing libraries cover an even broader range of mathematical functions than the BLAS specification including matrix operations such as eigenvalue decomposition and k-means clustering while offering an easier to use interface. Internally, both Armadillo and Octave use an exchangeable BLAS backend to deliver the best possible performance.

4 TOOLCHAIN

EmbeddedMontiArc Production and Test Compiler. In Figure 5 we show a comprehensive overview of our EmbeddedMontiArc compiler infrastructure featuring its most important parts. This infrastructure consists of two parts: (1) The (production) compiler, depicted on the left side, translates textual EmbeddedMontiArc models to native code for different targets. (2) The test compiler, shown on the right side, translates textual stream test models to native programs executing the compiled native code of (1) and producing test result reports. One contribution of this paper is this toolchain generating native code for different targets as well as testing the generated native code.

The compiler behavior can be controlled via the command line interface (CLI), e.g. to specify the target platform. (1) In a first step the compiler invokes the EmbeddedMontiArc-to-C++ Code generator. The concrete output depends on the mathematics library chosen via CLI, since the C++ code for matrix creation using the Octave library differs significantly from the matrix creation code using the Armadillo library. (2) The generated C++ code is compiled and linked in a next step. Thereby our toolchain chooses the appropriate C++ compiler for the specified target and math library automatically. This hides technical details about compiler options and linker paths for including the required mathematics runtime library from the user. For example, Octave for Windows 64-bit is compiled with GCC, while clang is used for Armadillo on Mac OS X.

To benefit from the highly optimized BLAS libraries working only on primitive Integer and Float types, the compiler throws all units away and converts all variables and constant values during the C++ generation process to the corresponding SI base units.

If the EmbeddedMontiArc compiler needs to compile for the web browser target, the C++ code generator does not generate optimal source code as WebAssembly does not support C++ threads yet [6]. Also the linking process for WebAssembly is different as it cannot link other WebAssembly files at runtime yet [52]. For this reason, no runtime dynamic linking is supported and all libraries are linked...
one alternative is selected based on the CLI options

**Figure 5: Compiler Infrastructure for EmbeddedMontiArc modeling family.**

```plaintext
1. Tags RosTags{
tag nL.degree with RosConnection =
tag value
   {topic= (name=/deg, type=struct_msgs/Mat3)};
   RosConnection with provided middleware meta data
2.ntag nL.W with RosConnection =
tag value
   {topic= (name=/w, type=struct_msgs/Mat3)};
   RosConnection with meta data generated from EMAM model
3.ntag nL.nLaplacian with RosConnection;
```

**Figure 6: Tagging example for the NormalizedLaplacian component.**

at compile time. Otherwise the communication would need to be based on JavaScript which would have a massive negative impact on the runtime performance.

Our compiler also configures emscripten locally, runs clang to compile the generated C++ code to LLVM and then runs emscripten to compile the LLVM code to WebAssembly and JavaScript. The options NO_FILESYSTEM=1 and O3 are used to produce smallest possible JavaScript files.

For the web-browser target we also generate an adapter allowing us to use the C&C model directly in JavaScript; the adapter also accepts matrices as Strings having the same convenient syntax as the matrices defined in EmbeddedMontiArc models. Since JavaScript is typeless, the generated JavaScript adapter also checks whether the input value fits to the type: are the matrix properties and the physical quantities compatible and is the value in the defined range. Additionally, the JavaScript generator produces an HTML file using the JavaScript adapter. Opening such an HTML file in a browser allows the modeler to test the component behavior by specifying input values by hand and receiving the calculated output values; examples of such generated HTML files are available from [14]. For the native code there are plug-ins allowing the developer to generate adapters for different middlewares.

**Middleware Plugins.** Since EmbeddedMontiArc was designed with a particular focus on embedded and cyber-physical systems, quickly the question arose, how models can be integrated into a vast amount of hardware and software platforms without having to vitiate them by technical details or to add hand-crafted glue-code. In domains like automotive, robotics, and cyber-physical systems both hardware and software architectures can be highly distributed. Thus, these systems often rely on middleware protocols such as ROS (Robot Operating System) to ensure loose coupling, exchangeability, and maintainability of the involved components.

Other C&C languages like Simulink [36] or LabView [24] enable the design of middleware-connected systems by providing corresponding component libraries. For instance, instead of outputting a computed actuator value to a standard output port of the system, the signal is sent to a pre-configured ROS [42] component which in turn handles the distribution of the message to a ROS network. Although easy to model and to use, the approach has serious drawbacks:

First of all, the algorithmic part of the system, e.g., the design of our clustering component, gets mingled with technical aspects of the system, thereby violating the separation of concerns principle and making the model platform-dependent.

Second, a variant of the model needs to be developed and stored for each target system or robot configuration. This inflates the required variant management overhead, although the actual model logic remains unaltered.

Third, models contaminated by middleware components exhibiting side-effects or having a behavior not only dependent on their inputs are difficult to test.

To avoid these pitfalls in model-driven engineering of embedded systems it is crucial to separate the conceptual model from the integration aspects. In EmbeddedMontiArc we achieve this aim by employing a so called tagging language to enrich the model by middleware specific information. Tagging allows adding information to the elements of a model such as ports in a separate tag model.
with our running examples, namely the the ROS middleware, we provide full middleware meta data, i.e., a generator for each supported middleware.

This section presents our second contribution: algebraic and thread-optimized code generation. Each generator creates code for the model parts, i.e. symbols, it can deal with. Finally, the orchestrating generator produces glue code melting the actual behavior code with the middleware adapters; thereby it produces an implementation ready to integrate into a simulator or on a real hardware system with zero hand-crafted code.

A tagging example for the NormalizedLaplacian component of Figure 3 (b) is given in Figure 6. We use ROS tags here to integrate the component into a ROS infrastructure. For the two input ports we provide full middleware meta data, i.e., a ROS topic name as well as the corresponding ROS type. It is important to provide this information if the model is integrated into a system with a predefined middleware infrastructure such as a simulator. If, however, the components to integrate are all designed in EmbeddedMontiArc, middleware meta data can be omitted and is generated by the corresponding middleware co-generator. Examples are: (1) EmbeddedMontiArc model controlling a Gazebo robot via ROS using the tagging system [11]; (2) EmbeddedMontiArc model controlling a CarSim car using OpenDaVinci [7] middleware [29]; (3) local traffic system with 10 cars [23] driven by an EmbeddedMontiArc controller in the MontiSim [16] simulator; and (4) an integration of the compiled WebAssembly code into the JavaScript PacMan simulator [18].

Further explanation videos describing the concrete models and the EmbeddedMontiArc development environment using this compiler toolchain are available under Youtube [38, 50].

5 OPTIMIZATIONS

This section presents our second contribution: algebraic and threading optimizations of the C++ code generator. Algebraic optimizations are particularly important for data intensive mathematical matrix-based tasks such as image processing or discretizing ordinary or partial differential equations [4] modeling control systems. Threading optimizations distribute all calculations defined in components into different threads. The effect depends mainly on the target CPU architecture, especially the number of available cores.

Algebraic Optimizations. All algebraic optimizations are explained with our running examples, namely the SpectralClusterer and MatrixModifier model.

Following the definition in (1), the NormalizedLaplacian component of the SpectralClusterer shown in Figure 3 (b) needs to calculate the inverse square root of the degree twice. Note that we use the simplified definition highlighted in blue in order to be comparable to the publicly available MATLAB implementation also using this variant. Since the degree matrix is a diagonal matrix having non-zero entries only on its main diagonal the inverse square root of degree is just the element-wise inverse square root of all elements of the principal diagonal, i.e.,

\[
D \text{ diagonal matrix : } (D^{-0.5})_{i,j} = \begin{cases} 
0 & \text{for } i \neq j \\
\left( D_{i,i} \right)^{-0.5} & \text{else} 
\end{cases} 
\] (2)

The type system of EmbeddedMontiArc is based on the mathematical domain allowing one to declare a matrix to be diagonal, tridiagonal, symmetric, positive-definite, etc. The C++ generator uses these matrix types to select the best suited algorithms [30] for inverting matrices. In most cases (not in the case of diagonal matrices though) when a matrix inversion is combined with a vector or matrix multiplication, e.g. \( A^{-1}x \), no explicit matrix inversion is calculated. Instead linear equation solving algorithms are applied.

Other common compiler optimizations like caching computational expensive results to avoid recomputing the same problem are also implemented; for instance, the two occurrences of degree\(^{-0.5}\) in lines 7-8 of Figure 3 (b) are replaced automatically by a helper variable. At compile-time, the type inference mechanism of EmbeddedMontiArc allows one to derive not only the matrix dimensions but also the algebraic type. Thus the compiler knows that degree\(^{-0.5}\) is a diagonal matrix, as well, and is hence able to choose the best suitable multiplication algorithm only considering the principal diagonal for the left side expression in degree\(^{-0.5}\). The aforementioned statical algebra analyses together with the presented optimizations improve the efficiency of the generated code dramatically.

The next paragraphs show algebraic optimizations based on the matrix dimensions [21] to figure out the best execution order to evaluate matrix expressions. Equations (3) – (6) summarize the formulas used for optimizations, where A, B, and C are matrices, b is a vector, and \( \lambda \) is a scalar:

\[
AC + BC = (A + B)C \quad (3) \quad A(BC) = (AB)C \quad (5) \\
CA + CB = C(A + B) \quad (4) \quad A(B \cdot \lambda) = (AB) \cdot \lambda. \quad (6)
\]

Since the matrix optimizations of equations (3) and (4) are independent of the matrix type and, thus, can be applied always, tools such as Armadillo rewrite expressions following these two rules by using C++ function templates. The MatrixModifier component example, shown in Figure 7, illustrates the optimization process using equation (5); (6) is a special case of (5) with \( \lambda \) being a scalar. MatrixModifier consists of five input ports, one output port, and four Multiplication subcomponents. Based on the matrix dimensions defined in the ports and the mathematical expressions you can estimate the number of needed operations for each atomic component; e.g. multiplying a 5 \( \times \) 7 matrix with a 7 \( \times \) 10 matrix needs \( 2 \cdot 5 \cdot 7 \cdot 10 = 700 \) arithmetic operations.

Equation (7) shows the natively derived calculation of the MatrixModifier component based on its C&C structure and the estimated operations (est. ops) needed for all calculations; for the sake of clarity all matrix dimensions of the input ports and of the
intermediate results are shown, as well. The estimated total number of operations needed to perform all the calculations of the MatrixModifier component is the sum of all estimated (single) operations, in our case $4 \text{ million} + 40 \text{ million} + 20 \text{ million} + 200 \text{ billion} \approx 220 \text{ billion estimated operations}$. 

If the estimated operation count surpasses a specified threshold, the compiler starts to restructure the expressions. To deliver a reasonable trade-off between compilation and runtime performance, the compiler only optimizes computationally expensive expressions. Applying rule (5) transforms equation (7) to equation (8). However evaluating (8) needs only $8 \text{ thousand} + 8 \text{ thousand} + 400 \text{ million} + 40 \text{ million} \approx 440 \text{ million estimated algebraic operations}$. Compared to the originally needed $220 \text{ billion}$ operations, our algebraic optimization reduces the calculation effort by a factor of 500. This factor holds only for algebraic computations, but a program must also load the matrices from the RAM to the processor caches and store them back into the RAM again since these large matrices need several hundreds of megabytes of storage and do not fit into the L3 cache. The case study in the next section shows that the real runtime benefit achieved by rewriting the mathematical formula is only a factor of 8. But real-world image processing systems consist of large hierarchies of filter components offering much more optimization potential then our small MatrixModifier example with only four matrix-processing subcomponents.

**Threading Optimizations.** The threading analysis tries to detect independent calculation paths to schedule them on different CPU cores. All threading optimizations are executed after doing the previously mentioned algebraic optimizations. Since our toolchain presented in Section 4 uses different BLAS backends such as OpenBLAS, some parallelizations on matrix operations can already be done by these libraries.

Thus, the C++ generator has to find a trade-off regarding the number of threads created by itself and the BLAS backend, respectively.

In general it should be avoided to create more threads than the target architecture supports physically (number of cores plus hyper-threading[26]) as this results in scheduling overhead for the operating system and reduces the overall performance.

**CASE STUDY**

Our case study utilizes the following four C&C models:

(i) **ObjectDetector with four SpectralClusterer subcomponents**

(ii) **ObjectDetectorManOpt** is a manually optimized version of (i) to accelerate runtime performance in MATLAB

(iii) **ObjectDetectorLight** is (ii) without the kmeans operation

(iv) **MathUnit** with four MatrixModifier subcomponents.

The object detector C&C models (i)-(iii) consist of four SpectralClusterer subcomponents assuming that a vehicle has a camera
on each side. The incoming images are clustered to detect objects, pedestrians, or other cars.

Model (i) contains the original formula of the spectral clusterer used in [41]. In contrast, model (ii) contains manually optimized code of the NormalizedLaplacian subcomponent to speed up the calculation significantly; this version is the Asad Ali’s optimized MATLAB code [3]. In addition to our real-world examples (i) and (ii), the C&C model (iii) was created to have also a comparison with tools not supporting the kmeans operation. The model (iv) is our syntheticMathUnit C&C example from Section 5.

For each of these four example applications (i)-(iv) this case study executes the following experiments:

(a) measure effect of algebraic optimization on runtime speed
(b) evaluate impact of math backends on runtime performance
(c) compare runtime of code compiled with EmbeddedMontiArc, Simulink, OpenModelica, and Java with time interpreting it in MATLAB, or Octave
(d) measure web-browser performance running WebAssembly code produced by our compiler against handwritten MathJS code
(e) check EMA, Simulink, and Modelica model sizes versus Java, MATLAB, and JavaScript program length.

Figure 10 (a) shows that with a factor between 8 and 10 the algebraic optimizations of our toolchain based on mathematical domain knowledge deliver the largest impact on the runtime performance. Adding threading optimization for the MatrixModifier model has nearly no impact. For the ObjectDetector model the performance gain through threading is as little as 30%.

Figure 10 (b) shows the impact of using different mathematics libraries or frameworks in the compiled code. It is obvious that the Octave backend is always slower than Armadillo. The performance difference when executing the MathUnit model with the Octave backend instead of Armadillo however is quite small as for normal matrix addition and multiplication operations Octave offers native C++ functions, whereas other functionality in Octave is defined via m-code and must be interpreted at runtime. Executing all three ObjectDetector examples with the Octave backend takes significantly more time than executing these models with the Armadillo backend; the main reason is that eigenvalue and element-wise inverse square-root (2) calculations are much slower in Octave than equivalent native C++ implementations used by Armadillo.

The experiment showed that the Blas library performs multi-threading automatically. Using it we were not able to measure any improvement when letting our toolchain create multiple threads for the subcomponents of MatrixModifier and SpectralClusterer. Therefore we aggregated the results for the Blas backend in the column Armadillo (Blas) 1/4 Threads in Figure 10 (b). Another finding of the experiment was Blas being faster than OpenBlas for basic matrix operations such as multiplying and adding matrices. In contrast, the OpenBlas library is faster for more complex matrix operations such as eigenvalue calculation or matrix inversion. For this reason Armadillo (Blas) turns out to be the best library when compiling the MathUnit model whereas Armadillo (OpenBlas) is the best option for translating all object detector models to native Windows applications.

To compare the runtime performance of the compiled EmbeddedMontiArc native Windows 64-bit code with the compiled native code produced by other existing tools, programming languages or libraries we modeled/programmed all four applications in Simulink [36], OpenModelica [15] with OMEdit [5], Java using RelativeGPS [2] library, JavaScript using MathJS [32], and as m-code for MATLAB [33] and Octave [40] again.

The MATLAB code for the ObjectDetectorManOpt model was already given and we did not modify it to have an evaluation with an application not created by us. The EmbeddedMontiArc code contains exactly the same matrix operations (also in the same order) as they are present in the downloaded m-files from the MathWorks website. The only difference is that EmbeddedMontiArc code groups functionality into components instead of functions and interaction takes place via connectors and not via function calls. The Simulink and OpenModelica code is a 1:1 mapping of the (syntax) of the EmbeddedMontiArc code, as all three tools are based on the C&C paradigm. The Java and the JavaScript applications are based on the MATLAB code as all the three are imperative programming languages.

Figure 9 contains selected model and code snippets showing how we remodeled the functionality in different tools/languages. In the left top part (a) the SpectralClusterer subsystem with all its subcomponents is depicted. The two listings below contain the code of the atomic MATLAB Function block types: (b) shows the original code used in the publication to model the behavior of the NormalizedLaplacian block, and (c) presents the manually algebraically optimized code (based on the fact that the degree matrix is a diagonal one, and therefore the power and matrix multiplication operations can be accelerated). The ObjectDetector application uses (d), whereas the ObjectDetectorManOpt and ObjectDetectorLight applications use the optimized one (e). The code in (d) and (e) is the same as used in the m-files for the corresponding applications. Only the additional expression nLaplacian = zeros(2500, 2500) must be added when using the MATLAB code in Simulink as the MathWorks embedded coder needs the exact matrix dimensions for C code generation.

The middle part in Figure 9 represents the OpenModelica implementation of the SpectralClusterer component. The graphical model shown in (f) the structural view in OMEdit must not be consistent with the semantics of the textual Modelica model, since input and output ports are only visible in the graphical model if the textual one is enriched with the correct annotations (see small text snippets in (g)). In our opinion the graphical annotations created by OMEdit pollute the textual Modelica model so that it becomes harder to read; a better solution would be to generate the graphical layout based on the defined in- and output ports or to store the graphical information in an additional file via tagging [17, 31]. The middle bottom listing (g) is the Modelica text code equivalent to the MATLAB/Simulink code shown in (h).

In contrast to the left and middle part in Figure 9 where the spectral clusterer was modeled as a C&C architecture, the code snippets on the right illustrate how to implement the spectral clusterer in an imperative programming language. The MathJS code (i) is the equivalent to the MATLAB code in (g); same holds for the middle right Java code (j). The JavaScript code (k) is very similar to the MATLAB code, as both are untyped languages. On the other hand, the Java code in (i) is strongly typed, cf. line 2 where the matrix is first created with its full-defined dimensions. Compared to equivalent MATLAB code (l) or the equivalent EmbeddedMontiArc
Figure 9: Model Snippets for SpectralClusterer component being modeled in Simulink and Modelica

Code Snippets for SpectralClusterer component using JavaScript/MathJS and Java/RelativeGPS code in Figure 3 (b), the optimized Java code is cumbersome to read having no support for operator overloading - the mathematical infix expressions must be implemented using the cascade pattern (method chaining). In contrast to MATLAB/Simulink [35] EmbeddedMontiArc (Figure 3 (b)) also supports the matrix power operation for non-integer exponents if both the matrix and the exponent are real; therefore in EmbeddedMontiArc the short-form degree^-.5 can be used whereas in Simulink/MATLAB the long and cumbersome syntax sqrtm(inv(degree)) must be used.

As the model and code snippets in Figure 9 suggest that the imperative scripting languages need the least lines of code to develop the functionality, Figure 10 (e) substantiates this fact showing that JavaScript and MATLAB need the least amount of code to implement the four applications. The C&C languages EmbeddedMontiArc and Simulink (where besides the number of MATLAB lines of code also the numbers of in-/output blocks, subsystems and other atomic blocks as well as the number of signal lines to connect two port blocks with each other need to be counted as Simulink is not completely text-based) need nearly the same amount of code/modeling elements as Java does. Due to their verbose syntax and annotations, OpenModelica models are the largest ones being more than twice as large as EmbeddedMontiArc or Simulink models and almost ten times as large as the textual scripting models of MathJS, MATLAB and Octave.

Figure 10 (e) shows the runtime duration to execute the applications with other frameworks. The Octave interpreter executes the same m-file as the MATLAB interpreter. The difference between the duration of the Octave interpreter and the Octave backend is that the Octave backend executed our algebraic optimized code on four parallel threads, and the Octave interpreter interprets the m-file in one thread according to the Windows Task Manager. For the MathUnit application, the EmbeddedMontiArc compiled code gets executed over 80 times faster than the interpreted m-file code by Octave or MATLAB. Since the non-optimized compiled code has a shorter execution time than the interpreted code (1.8s, see Figure 10 (a)), we assume that this time overhead is caused by the fact that the interpreter must parse all m-files and that since version R2015b MATLAB must decide whether it uses its Just-In-Time compiler producing C++ native code [46] or interpreting the statements which in turn results in expensive operations. For the ObjectDetector the compiled code is about six times faster than executing the m-file via the MATLAB command-line. The efficiency of MATLAB compared to Octave, both interpreting the same m-file, is probably due to the fact that MATLAB also recognizes that our degree matrix is sparse and then uses the backslash operator to invert the matrix by solving linear equations [34]. During the execution of our case study we found it remarkable that MATLAB can interpret the MathUnit model via m-file; but if the same model is remodeled in Simulink using the Simulink MatrixMultiply subsystems to multiply matrices we get the following compile error: The Jacobian elements number of 'MatrixModifier' exceeds 2 147 483 647, the maximum Jacobian elements allowed in memory (see screenshot from [14]). This means Simulink cannot be used as compiler for models dealing with large matrices. Also OpenModelica could not execute the MathUnit example as it crashed with an out of memory error message on a machine with 16 GB memory.

Figure 10 (d) shows the execution duration for the applications in the web browser when translating the EmbeddedMontiArc models.
to WebAssembly versus a direct MathJS implementation. While the C&C compiled model of the ObjectDetectorLight finished its execution in about 6.5 minutes in Chrome 66 (64-bit version), the calculation of the same functionality developed in JavaScript using the MathJS library needed more than half an hour where we stopped the evaluation. The original ObjectDetectorLight could not be implemented in MathJS straightforward as in contrast to EmbeddedMontiArc and MATLAB this library offers no k-means clustering. This case study shows that it is possible to develop computationally expensive applications for the browser target with the EmbeddedMontiArc toolchain.

The runtime measurements of Figure 10 (c) and (d) were executed on an Intel Core i7-6700HQ quad core laptop with 2.6GHz clock speed and Hyperthreading support running Windows 10 Professional 64-bit; further software used includes: MATLAB R2018a (64-bit), OpenModelica v1.2.0 (64-bit), Octave 4.2.1 (64-bit), Armadillo version 8.200.2, GCC 7.1, clang 1.37.36 (64-bit), emscripten 1.37.36, OpenBlas version 0.2.20, and CLAPACK [10] 3.2.1 (containing Blas and Lapack).

The presented case study showed that EmbeddedMontiArc enables model driven development on a functional level where the modeler must only care about the domain to be modeled and can leave the implementation and the optimization details over to the proposed toolchain. In comparison to existing C&C tools like Simulink and OpenModelica, EmbeddedMontiArc models are small regarding the code size. The here presented compiler toolchain produces fast executable native code. The EmbeddedMontiArc compiler is the only one in the field producing portable code capable of dealing with matrices of millions of elements such as in our MatrixModifier example. Although MATLAB was able to handle the problem, as well, the code cannot be redistributed without the MATLAB environment itself. Simulink, OpenModelica, and Java failed due to internal restrictions, or deficient memory management. Furthermore, not all languages support complex operations such as k-means clustering. This case study also showed that our compiler produces astonishingly fast code running in web-browsers without the need to adapt the EmbeddedMontiArc models.

7 CONCLUSION

Component and connector (C&C) models, with their corresponding code generators, are widely used in the automotive industry. This paper presented a highly optimizing and multi-target C&C compiler for the component and connector modeling language EmbeddedMontiArc, which enables calculations on ultra large matrices occurring in real-world machine learning applications such as object detection and map transformations.

Due to the domain driven type system of EmbeddedMontiArc focusing on algebraic matrix properties, instead of only reusing the existing byte-based type systems with int, double, or byte, better algebraic optimizations based on mathematical matrix theory are possible.

The case study showed that the toolchain built upon these algebraic optimizations produces not only theoretically faster executable code, but indeed delivers an outstanding performance for existing real-world applications. Some computationally heavy calculations were only possible with the here presented C&C compiler.

The EmbeddedMontiArc modeling methodology focuses primarily on using mathematical algorithms directly from scientific publications. The main aim is to release the modeler from making decisions on how to rewrite the mathematical program in order to ensure runtime efficiency. This paper was a first step further into this direction.

Finally this paper also showed that for executing a modeling language efficiently much more than just a code generator is required; tasks such as including the correct runtime libraries and configuring the used compiler infrastructure with the correct parameters must be supported by the tooling, since all of these decisions may have a large impact on the runtime performance when executing the models of your new modeling language.

Acknowledgements This research was supported by a Grant from the GIF, the German-Israeli Foundation for Scientific Research and Development, and by the Grant SPP1835 from DFG, the German Research Foundation.
REFERENCES

[17] John W. Eaton, David Bateman, Soren Hauberg, and Rik Wehring. 2016. GNU Octave version 4.2.0 manual: a high-level interactive language for numerical computa-
[23] Timo Greifenberg, Markus Luck, Sebastian Roidl, and Bernhard Rumpe. 2015. Engineering Tagging Languages for DSLs. In 18th ACM/IEEE International Confer-
[26] youube.com/GS2dpkFNIPE
[27] Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Mike Lorang, Bernhard Rumpe, Albi Sema, Georg Strobl, and Michael von Wencwester. 2018. Model-
[33] mozilla.org/en-US/docs/WebAssembly
[36] https://doi.org/10.1109/WCSP.2011.6096958
[40] Michalaw Staron. 2017. Detailed Design of Automotive Software. Springer Interna-
[41] tional Publishing, Cham, 117–149. https://doi.org/10.1007/978-3-319-58610-6_5
[42] Ulrike Von Luxburg. 2007. A tutorial on spectral clustering. Statistics and computa-
[52] Details on the automotive software development process using the Device.