Compiling Scilab to high performance embedded multicore systems

Timo Stripf, Oliver Oey, Thomas Bruckschloege, Juergen Becker, Gerard Rauwerda, Kim Sunesen, George Goulas, Panayiotis Alefragis, Nikolaos S. Voros, Steven Derrien, Olivier Senteys, Dimitrios Kritharidis, Nikolaos Mitas, Thomas Perschke

1. Introduction

Efficient, flexible, and high performance chips are needed. Many performance-critical applications (e.g. digital video processing, telecoms, and security applications) that need to process huge amounts of data in a short time would benefit from these attributes. Research projects such as MORPHEUS [1] and CRISP [2] have demonstrated the feasibility of such an approach and presented the benefit of parallel processing on real hardware prototypes. Providing a set of programming tools for respective cores is however not enough. A company must be able to take such a chip and program it, based on high-level tools and automatic parallelization/mapping strategies without detailed knowledge of the underlying hardware architecture. Only then, when combining the advantages of an Application-Specific Integrated Circuit (ASIC) in terms of processing density, with the flexibility of a Field-Programmable Gate Array (FPGA), in addition to it being affordable since it could be manufactured in larger numbers (like general purpose processors or FPGAs), it will profit from benefits of programmability and system level programming.

The Architecture oriented parallelization for high performance embedded Multicore systems using scilab (ALMA) European project aims to bridge these hurdles through the introduction and exploitation of a Scilab-based toolchain which enables the efficient mapping of applications on multiprocessor platforms from a high level of abstraction. The holistic solution of the ALMA toolchain allows the complexity of both the application and the architecture to be hidden, which leads to better acceptance, reduced development cost, and shorter time-to-market. Driven by the technology restrictions in chip design, the end of exponential growth of clock speeds and an unavoidable increasing request of computing performance, ALMA is a fundamental step forward in the necessary introduction of novel computing paradigms and methodologies.
ALMA-specific extensions, enables a simplified parallelism extraction. A novel Architecture Description Language (ADL), the ALMA ADL, is integrated into the whole toolflow for gaining platform-independence from the target architecture. The ALMA parallel software optimization environment will be combined with a SystemC simulation framework for Multiprocessor System-on-Chip (MPSoC). The overall framework is evaluated by targeting two architectures as well as two application test cases.

In this paper, we present our concept of the ALMA toolset enabling compilation of Scilab source code to multicore architectures. The rest of this paper is organized as follows: First, Section 2 discusses the Scilab input language. Section 3 gives an overview of the ALMA toolset followed by in-depth descriptions of the individual components. The toolset is based on an ADL that is explained in Section 4. Section 5 introduces the ALMA front-end tools for parsing, optimizing, and early performance evaluation of the Scilab input language. The coarse-grain parallelism extraction (Section 6) partitions, maps, and schedules the tasks to the target processor cores while the fine-grain parallelism extraction (Section 7) exploits data-level parallelism on instruction level. Parallel platform code generation (Section 8) compiles the optimized ALMA IR to machine code that could be simulated by the multicore architecture simulator (Section 9). In Section 10, the ALMA target architecture and application test cases are introduced and Section 11 concludes the paper.

2. Scilab input language

With the end of exponential growth of clock frequencies caused by the power wall, Multi-processor System-on-Chip (MPSoC) architectures approaches arise as one of the most popular ways to gain high performance on embedded systems. From the architecture perspective, efficient usage of MPSoCs requires the exploitation of parallelism on different granularities. On system level, coarse-grain parallelism must be exploited by parallelizing and mapping algorithms to different processing cores. Fine-grain parallelism is exploited on instruction level by targeting Single Instruction, Multiple Data (SIMD) instructions that require the usage of small integer data types and the vectorization of the source code. Additionally, the usage of efficient supported data types (integer or fixed-point data types over floating-point data types) offers a performance improvement and energy reduction but is coming along with accuracy reduction. In general, the efficient programming of MPSoCs requires significant experience and knowledge of target-specific optimizations. Thus, the programmability is one of the major problems of these systems.

On the other side, the end user does not want to care about parallelism and data types. In general, a typical end user does not have – or does not want to have – a deep knowledge of the underlying hardware. The end user wants to develop and explore algorithms on a high level using a simple and comfortable language within a numerical computing environment such as MATLAB [4]. For mapping his algorithm to the target architecture, the end user wants a one-button solution that provides a high performance and energy efficient result. In our approach, we try to bridge the gap between the end user and architecture perspective by providing an integrated toolchain for semi-automatic mapping of Scilab code to MPSoC architectures. Scilab is a platform-independent, numerically-oriented, high-level programming language. MATLAB code, which is similar in syntax, can be converted to Scilab. Scilab is one of several open source alternatives to MATLAB.

While using the Scilab language for targeting MPSoC architectures offers a lot of advantages to the end user, a compiler architect would not select Scilab as a first choice. The language utilizes matrix-based computation, dynamic typing, automatic memory management and lacks the ability for expressing concurrency, thus making it hard to produce efficient code for MPSoC architectures. Besides that, the Scilab language is very beneficial for automatic parallelization since it does not use pointers. In the following, we explain the advantage as well as our approach for addressing the language difficulties of the Scilab input language.

Dynamic typing

The Scilab language uses dynamically typed variables. Each variable can contain any Scilab data type (e.g. strings, boolean, integers, floating-point scalars and especially n-dimensional matrices of these types) and the variable’s data type is specified by value assignment. Within the Scilab environment, the type checking is performed at run time – as opposed to at compile time. The run-time type checking is computational intensive and implies the usage of automatic memory management, thus hindering efficient code generation for MPSoC architectures. To solve this issue, we extended the Scilab language with annotations of static type information. This approach allows the end user to soft migrate Scilab applications for supporting the ALMA compilation process.

Matrix-based computation

Scilab uses matrices as the main data type. A variable can contain arrays of 1 (vectors), 2 (matrices), or more dimensions. The language provides simple matrix operations on the data type such as multiplication. At run time, the size of matrices or vectors is not fixed and can be changed by matrix operations. Therefore, the user must provide the maximum size and dimension of array data types within our ALMA annotations in order to avoid unpredictable memory consumption as well as run-time overhead of dynamic memory allocation. In that way, changing the size of matrices is still possible (and is commonly used for constructing matrices) but only the maximum size is limited.

Data type usage

Scilab supports integer data types of various bit widths but they are not used in common practice. End users typically rely on floating-point data types. In general, floating-point operations are slower and less energy efficient than fixed-point data operations. Additionally, the corresponding floating-point unit within a processor consumes a significant amount of die area making the processor more expensive. The usage of integer or fixed-point operations can speed up computation and avoids using expensive hardware but is coming along with a loss of accuracy or a limited variable range. Therefore, we provide Scilab annotations to the end user to specify the dynamic range of variables and the maximum quantization error caused by the reduced accuracy. With this additional information, the ALMA toolchain is able to automatically select appropriate integer data types for floating-point variables. The integer operations can then be further optimized by using SIMD instructions.

Pointer free

A pointer (also called reference) is a programming language data type whose value refers directly to (or “points to”) another value stored elsewhere in the computer memory using its address. Scilab is a pointer-free language in common practice, i.e. a typical end user does not use pointers for expression algorithms. That is in contrast to the C programming language that requires pointers e.g. for strings, efficiently passing of values to functions or returning more than one variable from a function. In contrast to many common programming languages, Scilab allows to specify more than one output parameter per function, thus enabling the pointer-free programming model. All function input parameters are call-by-value in Scilab since there exist no pointers for realizing call-by-reference. The abdication of pointers within the Scilab language is very beneficial for compiler optimization since it avoids
the pointer aliasing problem. Pointers can be dynamically changed at run time and thus making it – in general – impossible to determine where a pointer points at compile time. Aliasing refers to the situation where the same memory location can be accessed using different names. Since a pointer can point to any variable or to the same location as another pointer, a compiler does not know the side effects of a pointer access. It is thus not allowed to reorder pointer accesses and that finally limits the exploitation of instruction- and thread-level-parallelism within the compiler.

2.1. ALMA-specific Scilab extension

The ALMA toolchain recognizes an extended version of the Scilab language in order to assist the automated mapping of Scilab specification to a multicore system. The standard Scilab development environment works as an interpreter with dynamic type checking meaning that type context of expressions is validated during execution. Matrices, which are the dominant data type, may change their type and size at run time by simple assignment statements. ALMA allows in the same way matrices dynamic resizing but restricts the matrices’ elements type to be determined statically at compile time. For this reason, ALMA input specification composes of a declarative section accepting type and variable declarations using the CDecl language and a Scilab language section accepting Scilab programs. The two regions lie in a single file (.sce) and are separated by the //%% delimiter with the declarative region being first in sequence.

The Scilab compiler engine of ALMA translates Scilab source code to annotated C code. The parser supports every specified feature of Scilab 5.3.3. However, idsyntactic elements of Scilab such as embedded C code blocks are not supported. More specifically, the language features as can be seen in Table 1 are supported.

The CDecl language is an extension of a specific subset of the C99 declarative syntax to adapt to Scilab compilation requirements. The subset of the C language declarations includes declarative statements for arrays, character strings, and functions. Scalars are considered as single element arrays while one-dimensional arrays are modeled as row or column vectors. Using the widely known C language declarative syntax has the advantage of requiring minimal effort for Scilab designer to start developing programs for ALMA. Every variable or function should be declared before appearing in the Scilab section. The user should declare the type and an upper bound (static size) for the size of matrix variables. The size is dynamically allocated during program initialization and refers to a steady data pool where the matrix data reside in memory. The dynamic size of the array cannot exceed the static size declared in the declarative region. Moreover, matrix variables may reside in either global or function scope. For every global variable in Scilab, a declaration of the following type is made in CDecl where for matrix A a size of 10*10 integers is allocated upon program initialization:

\[ \text{int } A[10][10]; \]

The following declaration depicts the declaration of matrix B inside the scope of function foo. In this case, a size of 10*100 integers is reserved during program initialization. The scope operator :: has been adopted in CDecl language to state the scope of the declared variable.

\[ \text{int } \text{foo} :: B[10][100]; \]

Scilab function declaration required the adaptation of C declarative syntax to handle the case of multiple output parameters. CDecl language has two special specifiers, in and out, for declaring whether a function parameter is an input or an output parameter. Their usage is shown on the example below:

\[ \text{int } \text{foo}(\text{in int } * * gfa, \text{in int } * * fb, \text{out int } * * k); \]

where this declaration stands for Scilab function

\[ \text{function } k = \text{foo}(gfa, fb) \]

It is important to notice that formal parameter variables inherit the size of the actual function parameter variables during a function call. For this reason, formal function parameters have their size unspecified (declared as double pointers) while the type declaration of the elements is mandatory.

Moreover, an important point for end users is the policy regarding the support of Scilab intrinsic functions. Scilab uses two forms of intrinsic functions: (a) “fundamental” ones written in C language and (b) “derived” ones written in Scilab and accessible from the single input specification file.

Finally, the Scilab Front-End tools (SAFE) assists the subsequent automatic coarse- and fine-grain parallelism exploitation engines by transferring user information regarding task identification in the form of SCILAB comments.

3. ALMA toolset overview

The ALMA toolset provides an end-to-end toolchain from Scilab [5] code to executable code on embedded MPSoC platforms. The typical use case involves an end user to develop and provide an application in an ALMA-specific Scilab dialect, as well as an abstract description of the target description using the ALMA Architecture Description Language (ADL), described in Section 4. The ALMA-specific Scilab dialect is a subset of the Scilab language, enhanced with comment-type annotations, and is outlined in this paper in Section 2. In the above use case, the ALMA toolset will produce parallelized executable code ready to run on the designated multicore embedded platform.

The ALMA toolset workflow is presented in Fig. 2. The ALMA-specific Scilab dialect source code is consumed by the Scilab Front-End (SAFE), which produces a C representation of the original code. Next, the C code is loaded into the GeCoS open source compiler framework [6] and is converted to the ALMA-specific Intermediate Representation (IR). The ALMA-specific IR is a GeCoS IR, extended to meet the needs of the ALMA project. Several transformations are applied to the ALMA-specific IR, implemented as GeCoS passes, before platform-independent MPSoC code is produced. The fine-grain parallelism extraction step, described in Section 7, targets the exploitation of the Single Instruction, Multiple Data (SIMD) instruction set of the underlying MPSoC architectures, addressing the data type selection and memory access aware vectorization problems. The coarse-grain parallelism extraction and optimization step, described in Section 6, analyzes and modifies the Control and Data Flow Graph (CDFG) in order to cluster, partition and schedule subgraphs to the available cores taking into account temporal and spatial constraints imposed by the architecture, the computational load, and the memory transactions of the various tasks. The parallelism extraction steps rely on the
platform ADL description, which is available to them through the ADL Compiler. In addition, the ALMA Multi-Core Simulator, which is an abstraction of the platform specific simulators, assists the code optimization steps by providing more accurate performance estimations.

The diagram in Fig. 1 shows the ALMA approach from the target MPSoC perspective. The figure distinguishes between hardware and software. On the bottom, embedded MPSoC architectures are implemented, such as platforms based on Recore’s reconfigurable DSP cores or Kahrisma [7–9] cores. The ALMA toolset from Fig. 1 shows how the ALMA approach from Fig. 2 is integrated with the multicore hardware/simulator. Fig. 1 depicts that the output of the ALMA tools (e.g. Fig. 2) is C-based code with parallel descriptions. This C-based code is taken as input for the target multicore platform hardware-specific compilers (e.g. Recore Xentium compiler, Kahrisma compiler, etc.). The executable binaries that are created by the hardware specific compilers can be run in the multicore simulators or can be directly executed on the multicore hardware. An abstract ADL description of the multicore hardware architecture will be used as an input for the ALMA approach. The ADL provides two goals:

1. ADL defines an abstract hardware description of the multicore hardware target. This abstract information is used to build a multicore simulation environment for the multicore hardware target.
2. Additional characteristics about the multicore hardware are defined which will be used during the optimization steps of the ALMA tools.

4. ALMA architecture description language

The ALMA Architecture Description Language (ADL) is a fundamental component of the ALMA toolset and is used by all other components as a central database to gather information about the current target architecture. The ADL is a key component to enable the target independence of the overall ALMA tool flow. Within the project, the architecture independence is shown by targeting two different architectures. Beyond that, the ADL-based approach enables the extensibility of the ALMA toolset to other target architectures as well as parametric design-space exploration of the target templates. The ADL serves as the hardware description input for the ALMA approach and therefore it provides the following features:

- Abstract hierarchical structural description for simulation of multi-core architectures.
- Behavioral annotations to the structural description for compiler-oriented application mapping to multi-core target architectures.
- Microarchitecture, resource and instruction set description for performance estimation, SIMD instruction selection and platform-specific C code generation.
- Configuration description for supporting reconfigurable architectures.
- Extensibility by using a special markup language.
- Compact description of regular structures using loop and conditional constructs.
- Parameterizable description by using variables.

While there exists several ADLs for MPSoCs [10–12], none is suitable to fulfill the special requirements of the ALMA toolset including structural specification for simulation and behavioral information for compilation. Therefore, we developed a novel ADL that is tailored for the special needs of the ALMA project and the ALMA tools described within the following sections.

4.1. ADL data description

The ADL is based on a special markup language for coding hierarchical structured data in a text document. It is comparable to XML [13] and JSON [14] and creates a tree representation of the described data. The language uses scalar data types as leaf nodes and
vector or object containers as inner nodes. While elements in a vector container are referenced with numbers, the elements inside an object can be referenced with a string key. Furthermore, the data description language offers the flexibility to use variables as well as constant mathematical expressions, for-loops and conditional constructs. This enables the flexibility to describe regular MPSoC structures very abstract. After variable propagation, mathematical expression calculation, and “for”/“if” statement interpretation, the format can be converted to an XML or JSON representation and is thus further reusable.

4.2. ADL architecture description

Based on the markup language, the structure of the ADL description is specified. The ADL is structured in various major sections that allow the specification of the ALMA target architectures from a structural perspective annotated with behavioral information. Thereby, we rely on the concept of modules, instances, and connections as widely used by hardware description languages such as VHDL or SystemVerilog but without describing the individual modules and connections on bit- or Register Transfer Level (RTL) granularity. Instead, the modules and connections are only specified in an abstract fashion in order to enable the analyzability that would be nearly impossible for a lower level of abstraction.

In detail, the ADL comprises the following top sections:

**Global** is used for global architecture definitions like base frequency. In addition to that, a boot configuration can be defined for reconfigurable architectures.

**Interfaces** is a library of usable connection types. An interface connects two or more modules with predefined ports and can provide behavioral information about connection type, transmission constraints, throughput, and other connection details.

**Modules** is a library of available system parts, describing their behavior and functionality. Modules can be instantiated and can be connected by ports using interfaces. A single module consists of a port definition, simulation information and one or more behavioral annotations that define different module properties. Additionally, a module can hierarchically instantiate other modules that are implemented as submodules.

**TopLevel** is a special base module of each system description. In this part of the system description the top-level modules are instantiated and connected via interfaces.

**Configurations** allows expressing reconfigurable architectures that can change the functionality of one or a group of modules. A configuration consists of required modules and their connections as well as the functionality of the grouped modules.

**Microarchitectures** specifies information about one or more processor architectures. A microarchitecture is referenced within the **Core** behavioral type annotated to modules or configurations.

4.3. Behavioral annotation

The structural specification of the target architecture within the ADL is annotated with behavioral information. A behavioral annotation can be applied to a **Module**, **Configuration** or **Interface** (see Table 2 column three). A behavioral annotation can consist of one or more behavioral types. A behavioral type categorizes e.g. a module as memory, cache, network router or core (processing element). Each Type is described by a set of different properties, e.g. a **Memory** type would include the size and delay properties. An overview of the possible behavioral types and their supported properties is given in Table 2.

The behavioral annotations do not represent an exact specification of the system’s behavior. They rather provide an approximate description for optimizing application mapping to as well as performance estimation of the target architecture. The behavioral annotations are all well-defined in order to enable their analyzability within the ADL Compiler. The structural description as well as non-structural but behavioral information is extracted by the ADL Compiler. To enable accurate simulations additional simulation parameters are available.

4.4. Microarchitecture description

A microarchitecture description is a special behavioral annotation, as specified in Section 4.3, to describe a processor core. This description contains the available data types (i.e. the register formats), the resources within the processor pipeline, the instruction set and some compiler-specific information.
The instruction set within the microarchitecture section is thereby described for three reasons: (1) for early performance estimation within the compilation flow, (2) for a list of SIMD instructions to be used by our auto-vectorization approach within fine-grain parallelism extraction and (3) for translating the ALMA Intermediate Representation (IR) to C code during parallel code generation. As we do not target compiler generation out of the instruction set description, we are able to describe the instruction set from the ALMA IR perspective instead of using the architecture perspective, as it is common practice in state-of-the-art ADLs. For each platform-independent operation and data type within the ALMA IR, the resource consumption and C code generation rules (i.e., for using intrinsics) are described. Instructions not natively supported can be added using function calls.

In Listing 1 an example for specifying an ADD and SIMD ADD operation within the ADL is given. Beyond the plain specification of the instruction set, this perspective additionally allows including instruction-set-specific compiler decisions and thus enables efficient performance estimation.

### 4.5. Hierarchy

The ADL supports hierarchical modeling of the architecture structure including behavioral details in the Modules section of the ADL. A module is defined in the Modules section of the ADL and can instantiate other modules as submodules. The ADL uses a strict hierarchical model. Therefore, connections between submodules of different parent modules are not allowed, all submodules of a module belong to the same level but submodules can themselves have submodules and the ADL Compiler checks for the existence of hierarchical loops.

The hierarchical modeling allows a compact description of the target architecture. Beyond that, hierarchy enables architecture modeling across different abstraction levels, namely the algorithmic and architectural level as known from the Gajski-Kuhn-Y-Chart. Therefore, in a single ADL description it is possible to model e.g. a Network-on-Chip (NoC) on architectural abstraction level as a Module and additionally provide structural information (network topology) on the algorithmic level using submodules.

For simulation, either a fast behavioral or a cycle-accurate simulation can be independently setup for subsystems within the overall simulation environment by ignoring simulation related information. E.g., the memory/cache subsystem could be simulated cycle-accurate while the NoC is only simulated behaviorally. This finally enables a flexible trade-off between performance and accuracy within the simulator.

Behavioral annotations can be combined from different abstraction levels. E.g., the routing protocol on architecture level is combined with topology details, like delays, on algorithmic level to extract all communication paths between pairwise communication partners. The communication paths feed as input for the mapping algorithm within the coarse-grain parallelism extraction.

### 4.6. ADL compiler

The interface between the ADL and the other tools of the toolchain is realized by the ADL Compiler. It is a tool that parses the given ADL file, analyzes the structural ADL, extracts compiler-specific information, and stores the information into JSON files. A well-defined API is available within the GeCoS framework for accessing the extracted information from the ADL Compiler. The individual commands enable the extraction of the following information:

- **getCoreCount()**
  The maximum number of parallel processing cores.

- **getCommunicationInformation(Core1, Core2)**
  Communication information, delay and throughput, for pairwise data transfer between Core1 and Core2 if it is possible.

- **getCoreMemoryInformation(Core)**
  A list of all available memories of this core including access delays, unaligned memory access overhead and access width.

- **getDataTypes(Core)**
  A list of supported data types by the target architecture.

- **getInstructions(Core)**
  List of supported assembly instructions by the target processor core including a list of supported SIMD instructions.

- **getCoreArchitecture(Core)**
  High-level information about the architecture of one core.

```plaintext
1 ['ADD'] = {
2   ['Datatype'] = [ {'*'} = ('Int<8>', 'Int<16>', 'Int<32>'); ];
3   ['Semantic'] = ['set', 'ADD', ('add', 'DSrc1', 'DSrc2')];
4   ['CodeGeneration'] = '(DSrc1 + DSrc2)';
5   ['Resources'] = [ [] = { ['ALU'] = (1, 1); }; ];
6};
7
8 // This is a SIMD Operation for 4 x 8-bit ADD in parallel
9 ['SIMD_8_ADD'] = {
10   ['Datatype'] = [ {'*'} = ('4xInt<8>'); ];
11   ['Semantic'] = ['set', 'ADD', ('add', 'DSrc1', 'DSrc2')];
12   ['CodeGeneration'] = '__builtin_kahrima_sind8_add(DSrc1, DSrc2)';
13   ['Resources'] = [ [] = { ['ALU'] = (1, 1); }; ];
14};
```

Listing 1. Example specification of an ADD and SIMD ADD instruction.
Thereby, the ADL Compiler is a key component for enabling the description of multi-purpose information with different requirements within a single ADL description. On the one side, the ADL description is simulation oriented and requires structural information about the components within the target architecture. On the other side, the compilation phases, namely the fine- and coarse-grain parallelism extraction and parallel code generation, require information on a higher abstraction level. Therefore, the ADL Compiler acts as a tool to connect the different abstraction layers. The ADL Compiler will extract the information of the structural domain and combine it with the behavioral annotations as can be seen in the representation within the Gajski-Kuhn-Y-Chart in Fig. 3.

5. Scilab frontend

The Scilab ALMA frontend is the first stage within the framework workflow and comprises of three tools:

- **The Scilab Frontend Environment (SAFE)**, a source-to-source compiler that parses a Scilab program and generates three output representations of this input: (1) a parse tree based on the C/89 BNF grammar and produced by a LALR parser, (2) a tree-like High-Level intermediate representation generated by analyzing the parse tree, and (3) C code that will later be used as the input specification for GeCoS framework.

- **The ALMA profiler (aprof)** offers block- and instruction-level profiling at an early stage. Its purpose is used for providing basic block and Scilab variable statistics for enabling an initial partitioning in the context of coarse-grain optimization. A secondary use of aprof is to assist end users in identifying application hot-spots.

- **The ALMA High-level Optimizer (ALMA HLO)** attempts to apply generally beneficial optimizations to the HLIR generated by SAFE.

5.1. SAFE (Scilab Frontend Environment)

**Scilab Front-End (SAFE)** is a source-to-source compiler that translates Scilab 5.3.3 source code to equivalent C code. The compiler front-end consists of two modules: (1) a parser recognizing the syntax of Scilab, and (2) a parser recognizing C declarative statements that bridge the gap between the untyped Scilab syntax and C language type system. The Scilab parser produces a tree like representation, called *Scilab High Level Intermediate Representation* (SHLIR), of the Scilab program while the CDecl parser produces a tree like intermediate representation, called *CDecl HLIR* (DHLIR), representing the declarative statements of the declarative section. DHLIR passes through a decoding process (Decipher Types) as shown in Fig. 4 to convert the DHLIR tree into appropriate types of the Scilab program symbols. In the end, the overall program information is carried by the SHLIR and Symbol Table. SHLIR and Symbol Table information are combined into the C Generator module to produce the output C code. It was agreed that this output C code will be the representation that will later be used as the input specification for UR1’s GeCoS framework. Fig. 4 gives an overview of the SAFE compiler architecture.

SAFE’s front-end requires a single file (.sce file) organized into two successive regions separated by the special //%% delimiter. The first region involves declarations written in a C-like language, assigning type to Scilab user variables (CDecl), while the second region is for purely Scilab input. The output of the front end is a tree-like representation (SHLIR) that corresponds to a refined version of the Scilab parse tree along with type information for each Scilab variable.

SAFE’s compiler back-end can be configured to produce different forms of output. For this reason, the user may provide specific directives in the form of pragmas to configure specific aspects of code generation. In addition, the compiler may pass information concerning parallelization to the back-end in a transparent way. This information is expressed by the user, in the input Scilab code, in the form of comments. Being inside comments, the annotations do not affect in any way the rest of the Scilab or C code.

The SAFE back-end performs the SHLIR linearization transforming the SHLIR to equivalent C code. The generated C code is organized into multiple header (.h) and implementation (.c) C files. These files are logically separated into three categories: the first one contains the code derived from the direct translation of the Scilab input, the second contains declarations of data structures corresponding to Scilab data types and prototypes of the implementations of internal Scilab functions in C. The implementation of these internal functions are provided in the third file.

---

**Fig. 3.** The ADL compiler in the context of the Gajski-Kuhn-Y-Chart.

**Fig. 4.** SAFE compiler architecture.
5.2. aprof (ALMA profiler)

The ALMA Scilab frontend includes an intermediate representation (IR) profiler, named aprof (ALMA profiler), that offers block- and instruction-level profiling at an early stage. Its purpose is to compute application performance bounds given generic architectural assumptions. It uses an internal IR called NAC (N-Address Code) [15]. The aprof high-level performance estimator is used for providing basic block and Scilab variable statistics for enabling an initial partitioning in the context of coarse-grain optimization. A secondary use of aprof is to assist end-users in identifying application hot-spots.

The basic steps involved in aprof are shown in Fig. 5. As input to aprof, SHLIR is supplied, corresponding to Abstract Syntax Trees (ASTs) produced by SAFE. The SHLIR-to-NAC generator produces code for the internal NAC IR. NAC operations specify a mapping from a set of \( n \) ordered inputs to a set of \( m \) ordered outputs as follows: \( o_1, \ldots, o_m \leftarrow \text{op}_1, \ldots, \text{op}_n \); where \( \text{op} \) specifies the IR-level instruction. The memory access model defines dedicated address spaces per array, so that both loads and stores require an explicit array identifier. An indexed load in C \( \text{b} = \text{a}[i] \); is translated as \( \text{b} \leftarrow \text{load} \text{a}, i \);, while an indexed store \( \text{a}[i] = \text{b} \); is translated as \( \text{a} \leftarrow \text{store} \text{b}, i \);. Procedure calls are non-atomic operations; for instance \( y \leftarrow \sqrt{x} \); computes the square root of \( x \).

At this stage, annotations can be placed as NAC operations for tracking events such as entering basic blocks. Then, static and dynamic analyses are ready to proceed: static analyzers are used to extract statistics such as the static instruction mix and data type coverage for the application. aprof extracts ODGs (Operation Dependence Graphs) in non-SSA, SSA (Static Single Assignment) or pseudo-SSA form, which are partial CDFGs (Control-Data Dependence Graphs) in non-SSA, SSA (Static Single Assignment) or pseudo-SSA form, which are partial CDFGs (Control-Data Dependence Graphs) in non-SSA, SSA (Static Single Assignment) for each task/function of the application for estimating performance. Pseudo-SSA form applies variable renaming and out-of-SSA conversion locally, at the basic block level. Scheduling engines account for generic architectural parameters to model a range of machines: (a) a sequential machine, (b) maximum intra-block parallelism (ASAP scheduling), (c) ideal block processor (inter-block parallelism) and (d) ideal thread processor (task parallelism).

The latter two options are used to estimate the amount of inter-block parallelism in the application at compile time. Scheduling for the ideal block or thread processor explores the potential of executing certain basic blocks or tasks in parallel. This is performed by identifying mutually independent basic blocks or functions in a given CFG (Control Flow Graph) or call graph, respectively. Reduced acyclic forms of these graphs must be examined incorporating profiling information, e.g., their spanning trees. Scheduling trees can then be scheduled in order to detect mutually independent tasks/procedures. Thus, task-level scheduling will involve mutually independent tasks/procedures that correspond to different threads/cores, assuming a homogeneous configuration. Each version of an ideal machine should be considered as a superset of the previous ones, i.e., the ideal thread processor offers all levels of parallelism. Realistic constraints such as communication and synchronization effects are not modeled at this point, thus only an upper bound on achievable performance is estimated.

Dynamic performance analysis will use a compiled simulator accepting NAC as its input. For the fast compiled simulation approach, NAC is back-converted to its corresponding unstructured C code. Host tools can be used to compile and execute the customized simulator for the specific application to collect basic block execution frequencies and extract the dynamic instruction mix. Then, these results are combined with the ILP profile produced by the scheduler to obtain an estimate on abstract machine cycles.

5.2.1. Estimating hardware resources utilization

The task of hardware resources utilization estimation by aprof is to provide a resource utilization report for reference by the end user application developers. The estimation will be based on static analysis of the NAC code executed on the NAC virtual machine using compiled simulation. This metric establishes the proposed generic architecture of a NACVM hypothetical processor that would compare to the resource utilization of a real-world architecture. Thus, resource utilization estimation at the NACVM level should be indicative of good resource utilization by real-world architectures such as the embedded multi-core targets that are of interest to the ALMA project. Other techniques proposed in the past involve CDFG state analysis [18], and execution trace analysis [19].

The NACVM corresponds to an abstract machine, for which the following hold:
• An interblock parallel machine can initiate an unconstrained number of basic blocks in parallel, if allowed by the static dependencies in the program.
• Similarly, a task parallel machine can initiate an unconstrained number of tasks.
• Homogeneous register file organization with unlimited number of read/write ports.
• Unlimited storage (actually limited by the capabilities of the host machine) and memory accesses.
• Operation costs are assigned based on a data-independent table with estimated cycles; the effect of mechanisms such as branch prediction, cache misses, pipeline hazards are not considered.

5.3. High-level Optimizer (ALMA HLO)

The ALMA High-level Optimizer (ALMA HLO) attempts to apply generally beneficial optimizations to the SHLIR as expressed in C form. Each transformation is implemented as a separate executable compiled from the corresponding rule-based transformation source file, written in the TXL [20] programming language.

In context of the HLO, basic loop transformations can be applied based on syntactical transformations written in TXL. The GeCoS fine-grain optimizer provides a comprehensive framework for optimizing static control parts (SCoPs) of programs. This applies for certain transformations such as strip mining, partial/full loop unrolling, and loop reversal, which are then enabled in a much more controlled manner. Transformations that do not require data flow analyzes can be applied by the HLO as well; this includes loop bump, extension, reduction, loop coalescing, and loop normalization.

The frontend optimizations provided by the HLO will focus on the following aspects:

• Generic code canonicalization including simplifications and algebraic identities.
• Syntactical code restructurings among iteration and if-conversion schemes.
• Advanced arithmetic/algebraic optimizations such as
  – constant and variable modulo optimization,
  – single and multiple constant multiplication/division,
  – constant multiplication by a vector or matrix,
  – optimizing linear structures,
  – polynomial expression rewriting.
• Source-level matrix flattening.
• Aggressive/selective procedure inlining.
• Loop invariant code motion.

Arithmetic optimizations include generally beneficial optimizations such as strength reduction of constant multiplications and divisions. Multiplication by an integer constant (kmul) [21,22] allows for significant speed improvement and area reduction since a full variable multiplier needs not be used. Contemporary FPGAs offering high-speed embedded multipliers might seem to render this optimization unnecessary, however it is always relevant in microprocessors without full-range hardware multipliers. Division by an integer constant uses a multiplication with the multiplicative inverse of the constant followed by a number of adjustment steps. Efficient algorithms for calculating the multiplicative inverse can be found in [23].

TXL is unsuitable for numerical computation, thus the constant multiplication/division code generators are written in C. Fig. 6 illustrates the combined TXL and ANSI C approach.

The first two steps apply preprocessing for splitting local scope variable declarations and removing those that are redundant or unused. Then, they are localized and subsequently function calls to specialized constant multiplication routines are introduced. These routines are then generated by the C tool and concatenated to the application ALMA IR/C. The optimized routines can be inlined at their call site by the platform compiler to eliminate argument-passing overheads.

6. Coarse-grain parallelism extraction and optimization

The coarse-grain parallelism extraction and optimization step of the ALMA toolset is responsible to provide parallel implementations for the input source code, with respect to a global view of the program, while selectively ignoring specific code blocks to reduce the search space. Inputs to the coarse-grain parallelism extraction step include the program in the form of the ALMA Intermediate Representation (ALMA IR) which exposes task-level parallelism possibilities along with the platform description in ADL through the ADL Compiler.

For initial coarse-grain parallelism extraction, profiling information (basic block execution frequencies, Scilab variable size statistics, and early performance estimation) from the ALMA profiler as part of the ALMA frontend is used. This is updated by accurate run-time profiling for the generated programs provided by the specific architecture simulator modules through the ALMA Multicore Simulator. The profiling is used as feedback in order to fine-tune the optimization process and is valuable since no architecture representation can fully model the MPSoC platform performance. Sample program input can be provided by the user to increase the simulator feedback accuracy.

The coarse-grain parallelism extraction and optimization subsystem follows an iterative process that follows the steps presented in Fig. 7, which are analyzed in the following paragraphs.

6.1. Coarse-grain parallelism extraction

The coarse-grain parallelism extraction step consists in analyzing and transforming the intermediate representation used in GeCoS to obtain a hierarchical task graph exposing task level dependencies. Because the GeCoS initial intermediate representation is unlikely to offer a sufficient amount of such parallelism, this task graph extraction is preceded by several program transformations. The goal of these transformations is to expose additional task parallelism, and consists of the following stages:

• A loop coarse-grain parallelization stage based on a combination of loop fusion and tiling. Scilab programs already expose a significant amount of data parallelism thanks to its use of matrix operation. However, using this parallelism as is is generally not a good solution as it may lead to parallel tasks suffering from a very poor (and hence harmful) communication/computation ratio, which in turn will prevent from obtaining an efficient parallel allocation and schedule.

• A domain-specific parallelization stage, which will rely on the semantics of some of the most widely used Scilab built-in functions (e.g. FFT, correlation, convolution, etc.) to propose specific parallelization transformations as proposed by Milder et al. [24]. For example, an FFT kernel can be easily split into several tasks (with additional parallelism) thanks to recursive decomposition into several smaller FFT kernels.

These transformations can be parameterized to perform a more or less aggressive parallelization of the program in order to increase the number of independent tasks in the resulting ALMA IR. This feature is exposed to the following stages enabling an iterative optimization flow.
6.2. Task clustering and partitioning

As strongly connected elementary tasks would incur high communication and synchronization penalties if mapped to different cores, we use clustering heuristics as a first step to combine elementary tasks into composite structures, which in the ALMA context are called hyperblocks. This process is performed iteratively, in order to achieve the desired level of granularity, balancing the number of the resulting hyperblocks between search space reduction and optimization opportunities exposed from smaller hyperblocks.

During the graph partition step, independent partitions of the modified CDFG, are generated. The graph-partitioning step produces larger clusters of hyperblocks that exhibit minimal dependencies between them. Each graph partition defines a potential task that is going to be executed by a single core and inherits in and out control and data dependencies of the original program. The communication nodes may include node duplication or data transfer nodes, with node duplication used when it is cheaper to re-compute a result instead of transferring it. Data transfer nodes are used besides transferring data as a synchronization mechanism.

After tasks are mapped to cores, the task-scheduling step reorders hyperblocks for each core in order to improve performance. The scheduling problem is modeled as a project-scheduling problem with multiple workers. For each task, earliest and latest deadlines for beginning and ending are defined and each task is scheduled to exactly one core. A heuristic algorithm generates an initial solution based on the result of the graph partitioning algorithms, although the task mapping may be changed if during the scheduling algorithm a better solution can be found. Several optimization goals can be defined. In the ALMA context, the smallest critical execution path or smallest average workload imbalance between worker cores is used. To address different behavior depending on input parameters, different schedules are generated with an injection of a conditional statement to decide the active schedule during run-time.

6.3. Task mapping and scheduling

After graph partitioning, task mapping assigns tasks to different cores for execution on the target architecture. In the context of the ALMA project, only homogenous multicore architectures are addressed, which simplifies the mapping process. Assigning tasks to cores imposes the generation of communication and synchronization nodes in the CDFG in order to maintain control and data dependencies of the original program. The communication nodes may include node duplication or data transfer nodes, with node duplication used when it is cheaper to re-compute a result instead of transferring it. Data transfer nodes are used besides transferring data as a synchronization mechanism.

After tasks are mapped to cores, the task-scheduling step reorders hyperblocks for each core in order to improve performance. The scheduling problem is modeled as a project-scheduling problem with multiple workers. For each task, earliest and latest deadlines for beginning and ending are defined and each task is scheduled to exactly one core. A heuristic algorithm generates an initial solution based on the result of the graph partitioning algorithms, although the task mapping may be changed if during the scheduling algorithm a better solution can be found. Several optimization goals can be defined. In the ALMA context, the smallest critical execution path or smallest average workload imbalance between worker cores is used. To address different behavior depending on input parameters, different schedules are generated with an injection of a conditional statement to decide the active schedule during run-time.

6.4. Modeling the optimization problem

For the above steps, we implemented various exact and approximate solution methods. A Mixed Integer Programming (MIP) model is used to solve small instances providing optimal solutions. A modified model is used to solve sub-problems of larger instances to optimality, although the process does not guarantee globally optimal solutions. MIP has been used before for automatic parallelization of embedded software [25]. Our approach combines MIP with single path metaheuristic methods like simulated annealing [26], tabu search [27], great deluge [28], and late acceptance hill climbing [29] to provide high quality solutions in a timely manner. The use of single-path and population-based metaheuristic methods for the task-planning problem for heterogeneous multiprocessors is presented in [30]. Beg presents in [31] a heuristic for graph partitioning the data dependency graph in order to assign computational workload on cores of a multicore system, with the main feature to identify the critical path of the code. Ferrandi et al. propose an ant colony optimization to optimize the hierarchical task graphs for MPSoC parallel applications [32]. Tournavitis
et al. identify that traditional target-specific mapping heuristics are inflexible and propose a machine-learning based prediction mechanism for the mapping decisions [33].

7. Fine-grain parallelism extraction and accuracy exploration

This component is responsible for fine-grain parallelism extraction targeting at the Single Instruction, Multiple Data (SIMD) instruction set of the RECORE and KARISHMA target architectures. Two interrelated problems are addressed in this task: SIMD vectorization and floating-point to fixed-point conversion.

Many embedded processors and DSPs [34], including the RECORE and KARISHMA cores, offer SIMD or Sub-Word Parallelism (SWP) instructions to exploit data-level parallelism on short integers data types. These instruction sets are designed to take advantage of sub-word parallelism available in many embedded applications (multimedia, wireless communications, image processing). The principle behind these extensions is simple: an operator (multiplier, adder, shift) of word-length \( W \) having SWP capabilities can be decomposed into \( P \) operations in parallel on sub-words of \( W/P \) length (e.g., a 64 bit SWP adder can execute \( 2 \times 32.4 \times 16 \) and \( 8 \times 8 \) bits vector additions). Efficient compilation for such instruction set is known to be difficult as their performance is very sensitive to memory layout decisions, loop-level data dependencies, and of course integer data encoding choices.

This last difficulty is worsened by the fact that the vast majority of Scilab programs operate on matrix and vector of floating-point data. This is an important issue when targeting heterogeneous embedded multi-core architectures, as only a few architectures offer native hardware support for floating-point arithmetic and even fewer provide floating-point SIMD capabilities. Obtaining an efficient implementation hence involves a costly floating-point to fixed-point conversion step, which is generally done manually and requires exhaustive simulations. To address this issue, we instead propose to rely on an analytical modeling of the program accuracy in terms of Signal to Quantization Noise Ratio (SQNR) as a function of the fixed-point encodings used by the program [35]. This modeling can then be used to automatically construct the set of fixed-point encodings satisfying application requirements.

As a consequence, when moving from the initial floating-point Scilab implementation to a fixed-point integer version, the designer has the opportunity to choose short bit width data encoding to be able to benefit from the SIMD extensions (provided the program exposes enough parallelism). This then comes at the cost of a loss of numerical accuracy in the results due to quantization noise [35], and exposes complex performance/accuracy trade-off optimization problems, that we want to address in our flow [36].

In ALMA, the extraction of fine-grain parallelism hence comprises two stages: data type binding and data parallelization. This parallelization will only be applied to the subset of programs that is amenable to polyhedral analysis, this subset being known as Static Control Part (SCoP). This module then generates an expanded ALMA IR where all Scilab vector- or matrix-based operations are expanded into scalar-level operations in nested loop constructs, in which the target processor SIMD instructions are exposed using intrinsic functions.

7.1. Data type selection

The aim of this stage is to select the data types that will enable the use of highly parallel vector operations, while enforcing the accuracy constraints provided by the user in the Scilab source code through annotations. This problem can be formulated as a constrained optimization problem in which performance/accuracy trade-offs are explored. Such optimization requires on one hand the definition of a realistic performance model of the SIMD instruction set, in which the penalties caused by unaligned memory accesses and packing/unpacking instruction must be taken into consideration. This information will be obtained from the input program and from the ADL description of the target architecture. On the other hand, the constraint function corresponding to the numerical accuracy will be obtained by considering the quantization noise power as a numerical accuracy metric. Approaches based on analytical models will be favored over simulations to obtain reasonable optimization times. An important aspect of the problem is that, to the difference of previous approaches that explore accuracy/performance trade-off, the number of data encoding types remains very limited, as it must correspond to a machine supported type (byte, word, double word), and therefore reduces the optimization search space.

7.2. Memory-aware vectorization

The aim of this stage is to perform the parallelization process, in other words, to expose a vectorized code, in which the target SIMD machine instructions are used through intrinsic function calls. Although the initial Scilab specification already exposes vector parallelism through vector/matrix level operations, this parallelism may lead to very suboptimal performance, as it rarely exhibits good spatial and/or temporal memory access locality. Our approach consists in restructuring the program control flow through complex loop transformations (loop interchange, fusion, and tiling) that will jointly address parallelization and vectorization [37]. In addition to the control flow, we will also explore complex array layout transformations, to reduce the program memory footprint and also to limit as much as possible the need for unaligned memory access and vector packing/unpacking instructions [38]. This loop and array layout transformation framework will be based on the well-known polyhedral model, which enables the exploration of a large space of program transformations, and provides efficient code generation features.

8. Parallel platform code generation

The parallel platform code generation compiles the ALMA IR from fine- and coarse-grain parallelization into executable binary for the target architectures. The binary could then be either simulated by the multicore architecture simulator or executed by the target architecture. It uses two inputs, (1) the CDFG of the ALMA IR including coarse-grain mapping and scheduling to processor cores as well as fine-grain SIMD instructions and (2) high-level information of the target cores provided by the ADL description. The parallel code generation relies on the code generation for single processors. It does not directly compile the ALMA IR into executable binaries. Instead, the ALMA IR is first converted into C source code within the GeCoS framework and afterwards compiled to target binaries. The advantage of this approach is that it allows to reuse existing, state-of-the-art C compilers generating optimized assembler code.

8.1. Parallel C code generation

The parallel platform code generation will be part of the ALMA extensions for the GeCoS framework and compiles the ALMA IR into target-specific C source code. The tasks of the CDFG are directly translated into C statements and functions including dedicated communication primitives for transferring data between tasks mapped to different cores. The layout of the available memories is calculated and the variables are placed into the available memory locations. For each core, an individual task schedule is available and
control code for implementing the scheduling is generated. The generated C code is passed to the target-specific code generation tools that compile the C source code for each processor individually.

The C code generation for Scilab variables and operations is dependent on the variable type. For scalar variables, only the arithmetic and logical C operators (e.g., +, *) are generated. Whereas for n-dimensional array variables, operator functions are called. The operator functions are available as a library and perform the vector or matrix operations using for loops. The C compiler later inlines this functions and can unroll the for loops for small arrays with a fixed number of elements. The SIMD instructions introduced by fine-grain parallelism extraction are converted into target-specific intrinsic function calls. These functions are replaced by single assembler instructions during C compilation. Therefore, the ADL description contains a list of SIMD instructions including their intrinsic function name.

The communication on multiprocessor systems can present a huge impact on the whole system performance. In a first step, the communication primitives use an architecture-independent API. As API, we rely on a subset of the Message Passing Interface (MPI) [39] that can be efficiently supported by both target architectures. Using a standard API also enables the evaluation of the generated parallel code on general-purpose processors. Both target architectures support a streaming-oriented data communication allowing direct transfer of registers between two processor cores. In a second step, we optimize the communication code and reduce the start-up costs by using coarse-grained target-specific API to transfer Scilab variables between processors. The target-specific API must then be specified within the ADL.

8.2. Target-specific code generation

The target-specific code generation compiles the C source code from parallel platform code generation into an executable binary for the target architectures. The binary could then be either simulated by the multi-core architecture simulator or executed on the target architecture. Both ALMA target architectures are coming along with a software toolchain including a C compiler, an assembler, and linker [40]. Depending on the specified target architecture within the ADL description, the appropriate software toolchain is selected and the C source code is compiled into executable binaries. The compiler translates C source code into assembly language source code. The assembler accepts assembly language source code and generates machine language object files. The linker combines a list of object files into a single executable binary. The linker accepts object files and object file libraries.

Both C compilers of the target architecture are based on the LLVM compiler infrastructure [41]. The back-end of the LLVM compiler generates the target-specific assembler code. Thereby, amongst other things, the instruction selection performs a pattern matching to transform target-independent instructions used by the compiler to target-dependent instructions available within the architecture. Thereby, the intrinsic SIMD functions inserted by fine-grain parallel extraction are converted into target instructions. For RISC processors, the scheduling assigns an order to the instructions and places each instruction into one time slot. Within the ALMA project, both architectures use a VLIW ISA. Scheduling for VLIW processors must additionally perform a fine-grain parallelism extraction and utilize the available Instruction Level Parallelism (ILP) in order to decide which instructions are executed in parallel.

9. Multicore architecture simulation

The multi-core architecture simulator plays an important role for application testing and performance estimation in the ALMA toolflow. In contrast to a common architecture simulator, the ALMA simulator must be flexible to enable the simulation of all ALMA architectures and configurations that could be expressed within the ADL. Therefore, it relies on the SystemC/TLM [42] simulation language and provides a library of SystemC Modules that can be referenced inside the ADL.

The ADL simulator uses two input files. The first input will be the application binary generated by the parallel code generation. As output, it generates profiling information that is used by coarse- and fine-grain parallelism extraction for profile-based optimizations.

As the second input, the simulator will use an ADL file to specify the simulated target architecture and its configuration. Each ADL Module has a reference to a SystemC simulation class in the simulators module library and supports additional simulation-specific parameters.

In order to support different abstraction levels that make use of the hierarchical description of modules, each module can define a deactivate variable. This variable controls the simulation of a module as Black Box Model on a higher abstraction level or as a detailed model using its submodules within a lower level of abstraction.

Fig. 8 shows a Gajski-Kuhn-Chart that demonstrates the use of different abstraction levels. The simulator is able to extract different structural abstraction levels per module as well as the according behavioral annotations. This allows a mixed simulation using different abstraction levels per module.

For initialization of the simulation environment, the simulator first parses the ADL description into internal data structures. Therefore, the ADL Compiler is embedded as a library into the simulator. Afterwards, the initialization uses the structural information from the internal data structure to instantiate and connect existent SystemC modules according to the target architecture/configuration described within the ADL file. After initialization of the SystemC modules, the application binary is loaded into the available memory. The simulation is started by calling the SystemC simulation kernel.

In the ALMA toolflow two processor cores, Recore’s X2014 and KIT’s Kahrisma architecture, will be made available in the library that could be realized by either an Instruction Set Simulator (ISS) or a Cycle-Accurate Simulator (CAS).

During simulation, the simulator will collect statistics, profiling, and tracing information of the individual processor cores, network, and memory components. The information will be made available to the coarse- and fine-grain parallelism extraction modules to enable profile-guided optimizations. Furthermore, the information could be used by the end user for application or architecture optimizations.

10. Evaluation of the ALMA approach

The ALMA approach and toolchain is evaluated by targeting two state-of-the-art MPSoC architectures from industry and academia using two application test cases from telecommunication and image processing domain.

10.1. Embedded multicore target platforms

Some of the most exciting recent developments in embedded architectures are the advances made in multicore and reconfigurable architectures [43]. The ALMA toolchain targets two such novel architectures.

10.1.1. Kahrisma

The Kahrisma architecture [7–9], as shown in Fig. 9, is a hyper-morph reconfigurable processor architecture. It features dynamic
reconfiguration of the instruction set as well as instruction format, e.g., switching between 2-issue and 8-issue Very Long Instruction Word (VLIW), to execute a configurable number of statically-sched operations in parallel. Multiple processor instances (each instance may be configured to execute a different instruction format) can co-exist in parallel. Each instruction format requires a different amount of resources and provides different peak performance characteristics.

Each processor instance is a clustered dynamic VLIW processor supporting precise interrupts. It comprises – dependent on the configuration – between 1 and 4 Encapsulated Datapath Elements (EDPE), between 1 and 4 Instruction Rename Tiles, 1 Control Flow Tile, and 1 or 2 Instruction Cache Tiles. Dependent on the number of EDPE elements, the VLIW processor can execute 2, 4, 6, or 8 operations in parallel and has 32, 64, 96, or 128 registers. The 32-bit instruction set features 8-bit and 16-bit vector operations.

The Kahrisma architecture supports a hardware centric programming model. The memory model defines three layers: (1) Local processor registers, which are only accessible locally, (2) local scratchpad memory, which is only accessible locally, and (3) main memory, which is accessible globally through local L1 data caches. There exists no cache coherency between the data caches. Additionally, a communication network for direct data transfer between processor core instances is used. The communication network can be accessed by dedicated communication assembly instructions. They are available in C through inline assembler. The communication network is self-synchronizing causing a processor instance to automatically stall until a communication assembler instruction is being completed. On top of the communication network, an MPI 1.3 library implementation is available.

The Kahrisma architecture comes along with a software toolchain for the C programming language including an LLVM-based C compiler, an assembler, a linker, and a cycle-approximate single-core simulator. The software toolchain supports code generation for and simulation of all configurable Kahrisma processor instances.

10.1.2. Tile-based multicore DSP architecture

Fig. 9b illustrates Recore Systems’ tile-based multicore architecture template. In the tiled architecture, a number of tiles are interconnected with I/O blocks and a bus-based General-Purpose Processors (GPP) subsystem using a 2D mesh Network-on-Chip (NoC). The main tiles are Xentium and Montium DSP cores and memory blocks. For the GPP subsystem, an embedded microprocessor is used such as an ARM or a LEON. The Xentium processor is a 32/40-bit fixed-point DSP core designed for high-performance embedded signal processing. Its VLIW architecture features ten parallel execution slots with support for Single Instruction Multiple Data (SIMD) operations and zero-overhead loops. The Montium is a coarse-grained reconfigurable 16-bit fixed-point DSP processor designed for low-power, compute-intensive signal processing. With its five execution units and ten parallel memory banks, the Montium resembles a VLIW architecture. However, it does not have a fixed instruction set as a conventional VLIW core. The Montium execution units do thus not fetch instructions but are instead explicitly (re) configured to perform the required functionality. Once configured the Montium resembles more ASIC than a DSP processor. This coarse-grained reconfigurability of execution units is fast and done at run time. Instances of the tile-based multicore architecture template are demonstrated in Integrated Circuit (IC) and Field-Programmable Gate Array (FPGA).

Programming this kind of heterogeneous multi-core architectures entails some additional complexities as each type of core has its own optimized toolchain. These point tools must be used to generate efficient target code for each specific core. Even though...
DSP and GPP cores can often be programmed using the C program-
ing language, the toolchains can vary substantially; providing differ-
ent C run-time libraries, different levels of C language support
(C89, C99, language extensions, etc.), different built-in or intrinsic
functions, and different levels of Operating System (OS), simulation,
debugging, and profiling support. Devising a single multi-target
toolchain that natively supports all (combinations of) target cores
and that can compete with vendor-specific point tools does not
seem viable. Like the ALMA toolchain, such a toolchain needs in-
stead to be built on top of existing toolchains.

In [52], a multicore platform with forty-five Xentium DSP cores
one ARM™ microprocessor core is programmed using a run-time re-
source manager. At run time, the resource manager computes and
executes the mapping of applications to the currently available re-
sources. When mapping an application, the resource manager relies
on an annotated task graph defining inter-task dependencies and
specifying processing and communication requirements. The con-
struction of annotated task graphs and compilation of application
code for the processing tasks are done before deployment. Code
compilation is done using single-core toolchains. The task graph
construction is typically mostly manually. The construction includes
task dependency analysis and splitting, merging, and parallelizing
tasks as well as calculation of bounds on processing and communi-
cation requirements. A (quasi) automatic toolchain for this con-
struction would extend further the applicability of the approach.

10.2. Application test cases

To ascertain that programmers can directly apply the program-
ing toolchains that the ALMA consortium develops in the course
of the project, ALMA targets two cases in market domains with dif-
f erent, complementary requirements.

The first case is from the field of telecom, and intends to support
the fast development of next generation of Point-to-Point/Point-to-
Multipoint wireless communication systems. Multi-core architec-
tures are a very good fit, since many performance-critical functions
must be executed in parallel in order to meet real-time constraints.
The call for short time-to-market at a minimum design effort re-
quires high-quality models that produce optimized code, and re-
lease the designer from the code optimization burden through a
set of tools that allow efficient optimization and parallelization of
system-level models with minimum designer intervention.

The second case is from the image-processing domain. It uses
object features to simultaneously track a number of different ob-
jects. The use of standard scale-invariant features like SIFT features
[53] has a high computational load and a real-time implementa-
tion on embedded systems with strong restrictions on power con-
sumption is very hard to achieve. The computational load can also
vary heavily from image to image. For embedded systems offering
low computational power, the algorithm must be adapted and par-
allelized over multiple processors. Partitioning the algorithm for
such systems by hand is difficult, error-prone and in nearly all
cases not optimal. The final goal is to demonstrate the tracking
of objects in a video live stream in real-time. This could be the basis
for the implementation of such algorithms in smart cameras.

11. Conclusion

In this paper, we presented the ALMA toolset that aims to deli-
ver an end-to-end solution for semi-automatic parallelization of
Scilab code to embedded multicore architectures. Two distinct
phases are identified, the parallel code production and the parallel
platform code generation. Two important tools integrate all the
parts of the toolset. (1) the ALMA Intermediate Representation
(ALMA IR) representing the Scilab code during the parallelization
steps and (2) the Architecture Description Language (ADL) enabling
the architecture independence of the toolset by providing an ab-
tract specification of the target architectures. The parallel code
production includes the frontend Scilab code parsing, a set of
transformations for the code IR followed by coarse- and fine-grain
parallelism extraction. The frontend generates the ALMA IR out of
the annotated Scilab input language. Coarse-grain parallelism
extraction partitions, maps, and schedules the tasks to the target
processors. Fine-grain parallelism extraction exploits the available
data-level parallelism and selects appropriate data types. The par-
allel platform code generation compiles the ALMA IR to executable
machine code. The ALMA parallel software optimization environ-
ment is combined with a fully functional SystemC simulation
framework for multicore architectures, which will be defined
through generic SystemC interfaces/protocols to connect existent
simulation modules targeting multiple architectures. The ALMA
toolset, although extensible at the hardware platform level, will
use the Recore’s multicore architecture as well as KIT’s Kahrisma
multicore embedded architecture as hardware targets. On the soft-
ware side, the ALMA toolset is evaluated by two application test
cases from telecommunication and image processing domain with
different, complementary requirements.

Acknowledgment

This work is co-funded by the European Union under the 7th
Framework Programme under Grant Agreement ICT-287733.

References

Guyeant, E. Schuler, K. Muller-Glaser, J. Becker, MORPHIEUS: heterogeneous
reconfigurable computing, in: International Conference on Field
et al., CRISP: cutting edge reconfigurable ICs for stream processing, in: J.M.P.
Cardoso, M. Hübner (Eds.), Reconfigurable Computing: From FPGAs to
www.mathworks.de/products/matlab/).
KAHRIEMA: a novel hypermorphic reconfigurable-instruction-set multi-
grained-array architecture, in: Design, Automation Test in Europe Conference
that enables dynamic code execution for variable-issue clustered processors,
in: IEEE International Symposium on Parallel and Distributed Processing,
exploration of run-time scalable issue-width processors, in: International
modeling of MPSoCs using ArchC, in: Proceedings of the Forum on
Specification & Design Languages FDL’05, 2005,
[12] L. Thiele, I. Bacivarov, W. Hard, K. Huang, Mapping applications to tiled
multiprocessor embedded systems, in: Application of Concurrency to System
ASIP compilers, in: Proc. Int. Conf. on Engin. of Recof. Sys. and Applications
[17] C. Ayache, N. Horspool, Simple generation of static single assignment form, in:
Proc. 9th Int. Conf. in Compiler Construction, 2000, pp. 110–125.
synthesis, in: Proceedings of the 9th Great Lakes Symposium on VLSI, Ann
Gerard Rauwerda obtained his MSc degree in Electrical Engineering at the University of Twente (The Netherlands) in 2002. He received his PhD degree from the University of Twente in January 2008 for his thesis titled “Multi-Standard Adaptive Wireless Communication Receivers – Adaptive Applications Mapped on Heterogeneous Dynamically Reconfigurable Hardware”. He worked for Ericsson and later at the University of Twente (The Netherlands), and he was a visiting researcher at Atmel Germany. He has over 10 years’ experience in wireless communications systems and reconfigurable embedded systems. He is co-founder of Recore Systems and currently engaged as Chief Technology Officer.

Kim Sunesen has studied at Aarhus University, Denmark, Université Louis Pasteur Strasbourg, France, and the Technical University of Munich, Germany. He holds a PhD. in computer science from Aarhus University. He manages research and development projects for Recore Systems’ digital signal processing platform chips. His responsibilities include both hardware and software developments projects. He has more than ten years of experience working in the semiconductor and EDA/Case tooling industry in The Netherlands, France and Denmark. He is R&D project manager at Recore Systems.

George Goulas holds a diploma and PhD from the Electrical and Computer Engineering Department, University of Patras. Currently, he is a fellow at Telecommunications Systems and Network Department, Technological Educational Institute of Mesolonghi and a postdoctoral researcher at the Electrical and Computer Engineering Department, University of Patras. His research interests include multicore and manycore systems, grid and cloud computing, web services, metaheuristics and timetabling. He with Dr. Valouxis, Dr. Alefragis and Prof. Housos submitted the software solution that won the first place on all three tracks of the 1st International Nurse Rostering Competition 2010.

Panayiotis Alefragis holds a Diploma and PhD in Electrical & Computer Engineering from University of Patras. His research interests include software engineering, parallel/distributed computing, programming languages and compilers, resource scheduling algorithms, integer and combinatorial optimization and embedded design. He has coauthored over 30 articles in refereed scientific journals & conference proceedings and has extensive research and industrial experience in the above areas. He is currently faculty of the Dept. of Telecommunication Systems of the TEI of Mesolonghi. He is a member of the IEEE and the Engineering Chamber of Greece.

Nikolaos S. Voros received his Diploma in Computer and Informatics Engineering, in 1996, and his PhD degree, in 2001, from University of Patras, Greece. His research interests fall in the area of embedded system design and hardware-software co-design. Currently, he is Assistant Professor at Technological Institute of Mesolonghi, Department of Telecommunication Systems and Networks where he is leading the Embedded Systems and Applications Lab. During the last years, he has participated in more than 15 research projects funded by European Commission while he has also published significant number of refereed articles in well-known international journals and conferences.

Oliver Sentieux joined University of Rennes (ENSSAT) and IRISA Laboratory, France, as a full Professor of Electronics Engineering, in 2002. He is leading the CARN Research Team common to INRIA Institute (national institute for research in computer science and control) and IRISA Lab. (research institute in computer science and random systems). His research activities are in the two complementary fields of embedded systems and signal processing with interests firstly on the definition of new system-on-chip architectures, especially the paradigm of reconfigurable systems, and their associated CAD tools, and secondly on some aspects of signal processing like finite arithmetic effects and cooperation in mobile systems. He is the author or co-author of more than 150 journal publications or peer-reviewed conference papers and holds 5 patents.

Nikolaos Kavvadias received the B.Sc. degree in physics and M.Sc. in electronic physics from Aristotle University of Thessaloniki (AUTH), Greece in 1999 and 2002, respectively. In 2008, he received his PhD. on custom processor design methodologies from the same department. From 2008 to 2012 he was a lecturer at the Dept. of Computer Science and Technology of the University of Peloponnese. He works as a scientific researcher and is the cofounding CTO of Ajax Compilers. His research interests include high-level synthesis of algorithms to hardware, application-specific/embedded processors, FPGAs, and compilers. He is a Member of the ACM and the IEEE.

Grigoris Dimitroulakos was born in March 26, 1976 in Athens, Greece. He received his Physics Degree in 1999 and the MSc in Electronics in 2001 from the Physics Department. He has received his PhD degree from Patras University in 2007. Currently, he is a Research Associate at the University of Peloponnese. His research interests are in the area of compilers, reconfigurable computing, computer architecture and the optimization of embedded systems on chip. He has authored or co-authored more than 40 scientific articles in conferences and Journals.

Konstantinos Masselos received a first degree in Electrical Engineering from University of Patras, Greece and a MSc degree from University of Manchester and a PhD degree from University of Patras, Greece. From 1997 until 1999 he was a visiting researcher at IMEC in Belgium. From 2001 to 2004 he was with INTRACOM SA, Greece. In 2005 and 2006 he was a lecturer in the Department of Electrical Engineering and Electronics of Imperial College London. Currently he is an Associate Professor in the Department of Computer Science and Technology of University of Peloponnese, Greece and an honorary lecturer of Imperial College. His main interests include compiler optimizations and high level synthesis, high level power optimization and FPGAS.

Steven Derrien obtained his PhD from University of Rennes 1 in 2003, and is now professor at University of Rennes 1. He is also a member of the Cairn research group at IRISA. His research interests include high level synthesis, loop parallelization, and reconfigurable systems design.
Dimitrios Kritharidis, MSc received his Diploma in Electrical Engineering from National Technical University of Athens (NTUA), Greece in 1988 and his MSc in Microprocessor Engineering and Digital Electronics from University of Manchester Institute of Science and Technology (UMIST), UK in 1989. After spending a year in Siemens Medical Division, in 1991 he joined INTRACOM, Athens, Greece, as an ASIC Designer. Since 1993 he has led the IC Design activity in INTRACOM, being the Head of the respective Group. He has designed numerous ASICs and FPGAs for Telecom Systems, and he has introduced new technologies and methodologies in the company's design flow. Currently he is responsible for the company's research in Microelectronics and Embedded Systems.

Nikolaos Mitas, MSc received his Diploma in Physics from University of Ioannina, Greece in 2003 and his MSc in Computer Engineering from Delft University of Technology in 2007. He has worked in IMEC (Leuven, Belgium) in 2006 and after spending a year in Intel (Braunschweig, Germany) working on EDA tool development in 2007, he joined INTRACOM in the beginning of 2008. Since then he has been at the Microelectronics lab working on research and development for various products as well as microelectronics and embedded research projects.

Thomas Perschke holds a PhD in physics. He has done his diploma thesis at the Max-Planck Institute for nuclear physics in Heidelberg and his PhD thesis at the Max-Planck Institute for physics in Munich. He worked three years as HW/SW developer in the biosensor development group of a biotech company based in Heidelberg. Currently he is working at the Fraunhofer Institute of Optronics, System Technologies and Image Exploitation IOSB. His research interests are the application of heterogeneous computing systems and their design methodologies in the image processing domain.