A Flexible Approach for Compiling Scilab to Reconfigurable Multi-Core Embedded Systems

(Invited Paper)

Timo Stripf, Oliver Oey, Thomas Bruckschloegl, Ralf Koenig, Michael Huebner*, Juergen Becker
Karlsruhe Institute of Technology, Germany
{stripf, oey, bruckschloegl, ralf.koenig, huebner, becker}@kit.edu
*since April 2012 at Ruhr-University of Bochum, Germany

George Goulas, Panayiotis Alefragis, Nikolaos S. Voros
Technological Educational Institute of Mesolonghi, Greece
{ggoulas, alefrag, voros}@teimes.gr

Gerard Rauwerda, Kim Sunesen
Recore Systems, The Netherlands
{gerard.rauwerda, kim.sunesen}@recoresystems.com

Steven Derrien, Daniel Menard, Olivier Sentieys
Université de Rennes I, INRIA Research Institute, France
{steven.derrien, daniel.menard, olivier.sentieys}@irisa.fr

Nikolaos Kavvadias, Grigoris Dimitroulakos, Kostas Masselos
University of Peloponnese, Greece
{nkavv, dhmhgre, kmas}@uop.gr

Diana Goehringer†, Thomas Perschke
Fraunhofer-Institute of Optronics, System Technologies and Image Exploitation, Germany
{diana.goehringer, thomas.perschke}@iosb.fraunhofer.de
†since April 2012 at Karlsruhe Institute of Technologie, Germany

Dimitrios Kritharidis, Nikolaos Mitas
Intracom S.A. Telecom Solutions, Greece
{dkri, nmitas}@intracom.gr

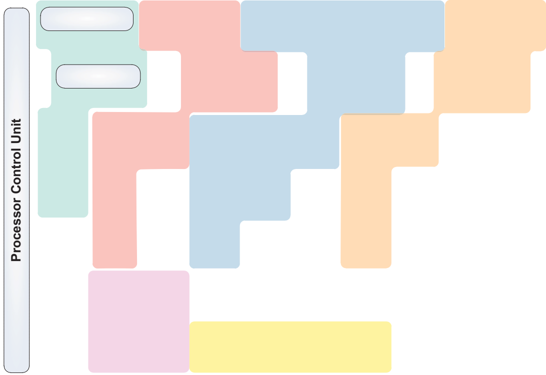
***Abstract*—The mapping process of high performance embedded applications to today's reconfigurable multiprocessor System-on-Chip devices suffers from a complex toolchain and programming process. Thus, the efficient programming of such architectures in terms of achievable performance and power consumption is limited to experts only. Enabling them to non-experts requires a simplified programming process that hides the complexity of the underlying hardware – introduced by software parallelism of multiple cores and the flexibility of reconfigurable architectures – to the end user. The Architecture oriented paraLlelization for high performance embedded Multi-core systems using scilAb (ALMA) European project aims to bridge these hurdles through the introduction and exploitation of a Scilab- and architecture-description-language-based toolchain which enables the efficient mapping of applications on multiprocessor platforms from high level of abstraction. This holistic solution of the toolchain allows the complexity of both the application and the architecture to be hidden, which leads to a better acceptance, reduced development costs, and shorter time-to-market.**

I. Introduction

Chips are needed that are efficient, high-performance, and programmable. Many performance-critical applications (e.g. digital video processing, telecommunications, 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 *reconfigurable* and *parallel* processing on real hardware prototypes. Providing a set of architecture-specific programming tools for respective cores is however not enough. A company must be able to take such a chip and have non-experts program it using high-level languages. The complexity of the underlying hardware – introduced by software parallelism of multiple cores and the flexibility of reconfigurable architectures – must be hidden to the end-user. Only then, when combining the advantages of an *Application-Specific Integrated Circuit* (ASIC) in terms of processing density, with the flexibility of an *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.

However, hiding the architecture parallelism and reconfigurability from the end-user necessitates a complex software toolchain featuring an automatic mapping process of high-performance embedded applications to today's reconfigurable multiprocessor *System-on-Chip* (SoC) devices. The realization of such a toolchain involves two challenges: (1) hiding the parallelism requires an automatic coarse- and fine-grained parallelism extraction. The problem of conventional approaches is the expression of application with a pure imperative programming language which is commonly C. Thereby, the natural usage of pointers and the lack of parallelism directives limits the mapping, partitioning and the generation of optimized parallel code, and consequently the achievable performance and power consumption of applications from different domains. (2) supporting architecture reconfigurability. Traditional approaches are typically tailored to static, non-reconfigurable processor architectures. In contrast, reconfigurable architectures require the selection of configurations e.g. by a configuration-space exploitation. Thus, a flexible software framework is required that is independent of any configurable parameter during the parallelism extraction, mapping, and code generation.

The ***A**rchitecture oriented para**L**lelization for high performance embedded **M**ulticore systems using scil**Ab*** (ALMA) FP7 project aims to bridge these hurdles through the introduction and exploitation of a Scilab- and *Architecture Description Language* (ADL)-based toolchain which enables the efficient mapping of embedded applications to reconfigurable multiprocessor platforms from high level of abstraction. The source input language for the applications is an extension of Scilab [3], a high-level, pointer-free, numerically-oriented programming language similar to the MATLAB language [4]. Scilab, together with ALMA-specific extensions, enables a simplified parallelism extraction.


Processor Control Unit

fines a 2D mesh *Network-on-Chip* (NoC) interconnecting a number of tiles with I/O blocks and a bus-based *General-Purpose Processors* (GPP) subsystem. The main tiles are Xentium® and Montium® DSP cores and memory blocks but can also be DMA controllers, accelerators, etc. The GPP is typically an embedded microprocessor such as an ARM® or LEON. The Xentium® processor [11] is a 32/40-bit fixed-point DSP core designed for high-performance embedded signal processing. The VLIW [12] architecture of the Xentium features 10 parallel execution slots and includes support for *Single Instruction Multiple Data* (SIMD) and zero-overhead loops. The Montium® is a coarse-grained reconfigurable 16-bit fixed-point DSP processor [13], [14] designed for low-power, compute-intensive signal processing. With its five execution units and 10 parallel memory banks, the Montium resembles a VLIW architecture. However, in contrast to conventional VLIW cores, the Montium does not have a fixed instruction set. Instead of fetching instructions, the execution units are thus explicitly (re)configured to perform the required functionality. Once configured the Montium resembles more ASIC than a DSP processor. Moreover, this coarse-grained reconfigurability of execution units is fast and done at run time. The Montium therefore provides an efficient balance between energy, performance, and reuse for embedded platforms for various streaming application with sustained high workloads. The Xentium® and Montium® DSP cores are cornerstones in Recore Systems' fully integrated platform ICs for embedded multi-media applications [15]. Instances of the tiled multicore SoC architecture have been demonstrated in *Integrated Circuit* (IC) [2], [16] and *Field-Programmable Gate Array* (FPGA) [17].

When programming this kind of novel heterogeneous reconfigurable multi-core platforms, some additional complexities are introduced. Each core type has its own optimized toolchain. To achieve efficient code generation, these point tools must be applied to generate the target code for each specific core. Even though DSP and GPP cores can often be programmed using the C programming language, the code generation support tends to vary substantially; providing different C runtime 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) support. Devicing 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. Instead, a toolchain is needed that can be built on-top of existing toolchains. The ALMA toolchain aims to achieve this by leveraging target-specific information about the target cores and toolchains using an abstract ADL.

The ADL enables the ALMA toolchain to directly support a wide class of processing architectures. Moreover, future multi-cores will benefit from this flexibility by providing several ADL architecture descriptions of (subsets of) the same hardware platform to improve application composability, adaptive *Quality-of-Service* (QoS) levels, and graceful degradation in case of permanent hardware faults. The CRISP NoC-based reconfigurable multi-core platform [2] can configure processing and communication resources such that independently programmed applications can be assigned resources and run in parallel with predictable behavior. The partitioning of the resources can be described by decomposing the ADL description of the full platform into several ADL descriptions of the part systems. Moreover, the same application can be compiled for systems using different types of cores, either to provide greater flexibility when mapping the application on the platform or to provide different levels of QoS.

In [18], a run-time resource manager is used to program a multi-core platform with 45 Xentium DSP cores and one ARM® core. Depending on the currently available resources, the mapping of applications to platform resources is computed and deployed at run time. The resource manager relies on an annotated task graph when mapping an application. The task graph annotations specifies processing and communication requirements for the application. The construction of annotated task graphs and the compilation of application code for the processing tasks are done before deployment. Code compilation is done using single-core toolchains for the processing tiles such as the Xentium toolchain for the C programming language. The task graph construction is typically mostly manually. The construction includes splitting, merging, and parallelizing tasks as well as task dependency analysis and calculation of processing and communication resource bounds. A (quasi) automatic toolset for this construction would extend futher the applicability of the approach.

III. ALMA Toolset

The ALMA toolset purpose is an end-to-end solution for the production of parallel code for embedded MPSoC from Scilab source code. The target embedded platform is presented as a selection for the user, but is actually an additional input for the toolset, which is open to MPSoC platforms that are described using ADL. An ALMA specific Scilab dialect is defined, which is a subset of the Scilab language, enhanced with comment-type annotations, similar to OpenMP [19]. An overview of the toolset is presented in Figure 3. During the Scilab Parser phase, the user code is converted to the ALMA *Intermediate Representation* (IR) which is an extension to the GeCoS compiler framework [20] IR, while initial performance estimations are produced by the Source-level profiler. Following that initial step, during the fine and coarse grain parallelism extraction phases, platform quasi-agnostic parallelization approaches are implemented. Then, platform-specific code generation occurs to perform platform specific optimizations and final code generation. The above phases are assisted by the ADL parser and relevant APIs, to provide platform information, and the platform simulators through ALMA specific APIs, to provide cycle accurate code simulations.

The ALMA parallelization approach alternates between localized loop-oriented optimization techniques and global, graph-oriented optimization for the input source program. Although it is natural for a parallelization approach to focus on the loop structures, local optimal solutions produced during this myopic approach could disable the possibility for other optimization opportunities. The local and global operations occur during the fine grain parallelism extraction and the coarse grain parallelism extraction phases respectively. These phases iteratively improve the program performance and communicate through the ALMA IR, which is a *Control and Data Flow Graph* (CDFG). Following the parallelism extraction phases, the parallel code generation produces parallel sources for the target architecture.

A. Scilab Frontend

The ALMA frontend tools that will be integrated in the ALMA toolset consist of the following components:

1) SAFE environment: Scilab Front-End (SAFE) accepts Scilab 5 input, exposes a textual tree-like *High-Level Intermediate Representation* (HLIR) and produces C code. Rather than working as an interpreter, the SAFE is a compiler to facilitate compile-time optimizations and analyses. To bridge the gap between the untyped Scilab to the strict typing of C, SAFE accepts data type

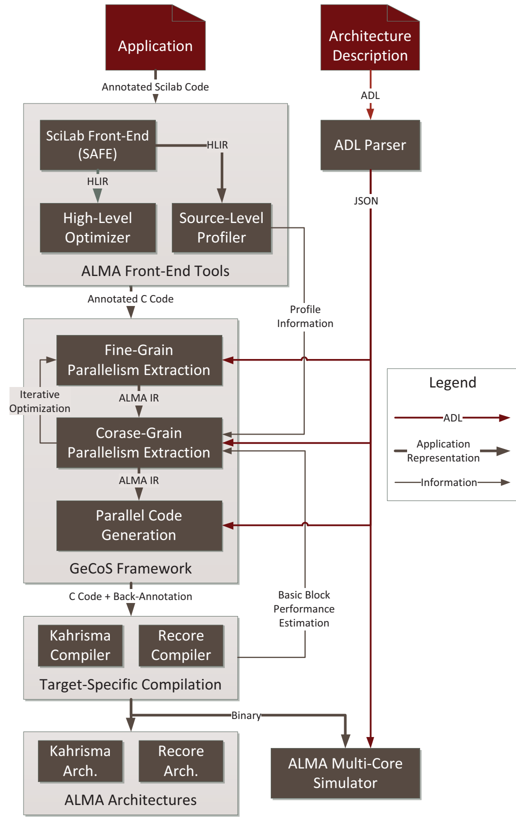


Figure 3. ALMA Toolset Overview

annotations in a separate declarative section which uses ANSI/ISO C notation.

2) aprof (ALMA profiler): The ALMA Scilab frontend includes an HLIR profiler, named *aprof* (ALMA profiler). Its purpose is to compute low and high performance bounds to assist end-users in identifying application hot-spots. It uses an internal IR called *N-Address Code* (NAC) [21] that is not exposed further in the ALMA flow.

The HLIR-to-NAC generator produces code for NAC, which specifies operations as ordered $n \rightarrow m$ mappings. Then, static and dynamic analyses are ready to proceed: static analyzers are used for evaluating the static instruction mix and data type coverage for the application. aprof extracts *Operation Dependence Graphs* (ODGs) in *Static Single Assignment* (SSA) [22] for each function of the application for estimating performance. The schedulers account for generic architectural parameters to model: a) a sequential machine, b) maximum intra-block parallelism (ASAP scheduling), c) ideal block, and d) ideal thread processor. The later two options investigate inter-block parallelism, by identifying mutually independent basic blocks or functions in a given *Control Flow Graph* (CFG) or call graph, respectively.

Dynamic performance analysis uses a compiled simulator accepting back-translated C as its input that ensures fast host execution for collecting basic block execution frequencies and extracting 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.

3) High-Level Optimizer (ALMA HLO): The ALMA *High-Level Optimizer* (HLO) is a collection of autonomous C-to-C code transformation passes for applying machine-independent optimizations, implemented in TXL [23].

The HLO supports *Generic Restructuring Transformations* (GRTs) and *Loop-Specific Optimizations* (LSOs). GRTs include code canonicalization for removing programming idioms, arithmetic optimizations, and combining independent scalar statements. Loop-specific optimizations include enabling transformations such as loop extension, reduction, and fusion for assisting strip mining, partial/full loop unrolling.

4) Scilab Code Section: The code section accepts Scilab code complaint with the grammar in Scilab 5.3.x release. To assist the subsequent parallelization and hardware synthesis tasks a new language will be devised that will convey additional information concerning specific parts of the Scilab code (i.e for loops). This language will be transparent to SAFE, with its expressions enclosed to Scilab comments, and the information will be seamlessly attached to a specific Scilab segment of a particular interested for the subsequent synthesis and parallelization stages.

B. Bitwidth-Aware Fine-Grain Parallelism Extraction

The aim of this module is to take advantage of SIMD or *Sub-Word Parallelism* (SWP) execution units to exploit data-level parallelism. SWP capabilities are available in most high-performance embedded and DSP processors [24] including Recore and Kahrisma platforms. 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 the fact that an operator 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 32-bit, 4 16-bit or 8 8-bit additions in parallel).

When moving from the initial floating-point Scilab implementation, the designer has the opportunity to choose shorter fixed-point data encoding to be able to benefit from SIMD extensions (provided that the software code exhibits enough fine-grain parallelism). This comes at the cost of a loss of numerical accuracy in the computation due to quantization noise [25] and results in a complex performance/accuracy trade-off optimization problem that ALMA's toolflow will automatize [26].

In our flow, the extraction of fine-grain parallelism comprises three stages: data type binding, data parallelization, and instruction selection. In the ALMA project, the 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 regenerates an expanded ALMA intermediate representation 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 specified using intrinsics.

1) Data Type Selection and Accuracy Estimation: 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. The goal of this step is therefore to minimize the implementation cost C under a given accuracy constraint $P_{b_{max}}$. The fixed-point accuracy is evaluated through the quantization noise power P_b. Let `wl` be the word-length of all operations in the program. The optimization problem can then be modeled as

$$\min\left(C(\texttt{wl})\right) \quad \text{subject to} \quad P_b(\texttt{wl}) < P_{b_{max}}. \qquad (1)$$

Our proposed methodology selects the set of SIMD instructions that enforces the global accuracy constraint $P_{b_{max}}$ and minimizes an implementation cost function.

An important aspect of the problem is that the number of data types remains very limited as they must match target processor supported types (byte, word, double word). This significantly reduces the solution of possible encodings, and allows to consider accuracy as a one constraint criterion in a more complex performance optimization problem.

2) Loop and layout transformations: 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 to improve spatial and/or temporal memory access locality [27]. In addition to the control flow, we also propose to 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 [28].

3) Toward Efficient and Retargetable Vectorization: The goal of the ALMA flow is to provide a retargetable vectorization flow that will leverage both the accuracy and program transformation stages described above. The goal is to regenerate a vectorized C code in which the target SIMD machine instructions are exposed to the back-end compiler through intrinsic function calls. In doing so, a flexible framework is provided that can easily be retargeted to different SIMD instruction sets, e.g. Recore Xentium and Kahrisma.

Deriving an retargetable vectorization framework is a difficult task, as such a tool needs a very accurate description of the target architecture to be efficient. To address this issue, we will rely on the ADL described in section IV. This ADL will be used to specify the SIMD instruction set available on the target architecture.

For each instruction, the ADL defines its operational semantics, its operands (registers, memory, immediate) along with their constraints on memory alignment and/or arithmetic behavior (overflow, saturation, etc). In addition, the ADL will also provide detailed information on the temporal behavior of the instruction (latency, throughput, possible resource conflicts in the pipeline, etc…).

The ALMA toolset will then use this information to build a performance model that is as realistic as possible, in which the penalties caused by unaligned memory accesses, cache misses likelihood, and packing/unpacking instructions will be considered. This model will then be used as an objective function of a complex optimization problem that we propose to address using two approaches: one based on a greedy algorithm and another one leveraging constraint-based programming.

C. Coarse-Grain Parallelism Extraction

Coarse-grain parallelism extraction attempts to provide parallel implementations for the input source code, with a global view of the source code, while ignoring specific code blocks to reduce search space. The ignored code blocks have been optimized by the fine-grain parallelism extraction step. The coarse-grain parallelism extraction is an iterative process of a sequence of internal steps, presented in Figure 4, which might be combined during the final implementation. The initial step is to cluster basic blocks into composite blocks that make sense to run on a single core. Next step, on the now clustered graph, is to generate graph partitions with minimal dependencies. These graph partitions are mapped to available cores to create graphs per core. Finally, the sequence of events are scheduled within the core processes in order to produce an optimal core schedule.

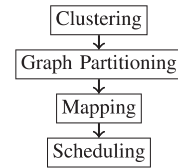


Figure 4. Coarse-Grain Iteration Steps

As the various graph transformations are evaluated, the optimization goal is the minimization of the execution time. In order for coarse-grain parallelism extraction to make an informed decision, accurate performance estimations are needed for various scenarios. The clustering step is performed solely based on information provided by the *basic block performance estimator*. While basic block performance estimation is very useful as a model, these estimations often are inaccurate even in one or two orders of magnitude due to cache and memory delays, chip network operations, branch prediction misses, etc. The hardware is designed to minimize the occurrence of such situations for most programs, but several code configurations can systematically reproduce these problems. The best alternatives produced by graph partitioning and mapping processes are packaged in benchmark-type programs. The alternatives are produced as sequences of benchmark procedures rather than sole programs, since a platform toolchain cycle through code building and simulation is a time-expensive process. These programs are executed in the simulator and the actual results are compared to the performance estimations. As the process continues, the coarse-grain parallelism extraction avoids the offending code configurations while trying to evaluate alternative implementations.

D. Parallel Platform 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 1- or 2-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) [29] 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. However, using the MPI API is coming along with an overhead resulting from the function calls and packing of Scilab variables into MPI messages. Thereby, the start-up cost of communication for MPI is typically greater than the per-byte transmission cost. 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 target-specific APIs to transfer Scilab variables between processors. The target-specific API must than be specified within the ADL.

E. Target-Specific Compilation

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 [8]. 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 [30]. 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.

F. Multi-Core Architecture Simulator

The ALMA multi-core simulator plays an important role for realizing a exploration of the available configuration space for a reconfiguration architecture. Only if a configuration could be evaluated to the resulting performance for an application, an efficient configuration could be selected. In contrast to a common architecture simulation, the ALMA simulator must be flexible in the way to simulate all ALMA target architectures that could be expressed within the ADL.

The multi-core architecture simulator enables the simulation of all ALMA architectures and configurations. It uses the application binary generated by the parallel code generation as input. As output it will generate profiling information that is used by coarse- and fine-grain parallelism extraction for profile-based optimizations.

Additionally, the multi-core architecture simulator will use an ADL file as input to specify the simulated target architecture and its configuration. The ADL file will contain a structural specification of the components of the target multi-core hardware architecture/configuration. Based on the structural architecture description (i.e. ADL), the multi-core simulation environment is established by instantiating modular simulation components. The ALMA multi-core architecture simulator relies on the SystemC/TLM [31] simulation language. SystemC is a set of C++ classes and macros that enables event-driven simulations within C++.

For initialization of the simulation environment, the simulator first parses the ADL description into internal data structures. Therefore, the ADL Parser 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.

Within the simulator a library of SystemC modules will be available. The ADL description references the SystemC modules of the library, e.g. network, memory/cache, processor cores, etc. The processor cores of the library could be either realized by *Instruction Set Simulators* (ISSs) or *Cycle-Accurate Simulators* (CASs) of the processor cores of Recore's and KIT's Kahrisma architecture. Hence, distinct single-processor simulators will be wrapped with a SystemC/TLM interface and integrated in the ALMA multi-core simulation environment.

During simulation the simulator will allow collecting 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.

IV. ALMA Architecture Description Language

The ALMA *Architecture Description Language* (ADL) is a central component of the toolset that is used by all other components as a central database to gather information about the current target architecture. It is used as an input for the ALMA approach and provides the following goals:

1) ADL defines an abstract hardware description of the multi-core hardware target. This abstract information is used to build a multi-core simulation environment for the multi-core hardware target.
2) Additional characteristics about the multi-core hardware are defined which will be used during the optimization steps of the ALMA tools.
3) The ADL is a key component of the ALMA approach to guarantee its target independence. Within the project, the architecture independence is gained by targeting two different architectures. Beyond that, the ADL-based approach enables the ALMA toolset to support other target architectures.
4) It enables support for reconfigurable architectures since it allows to flexible target all configurations of one architecture.

A. ADL Format

While there exists a couple of ADLs, none is suitable to fulfill the special needs of the ALMA toolset. Therefore, we develop

a novel ADL that is tailored for the special requirements of the ALMA project and the ALMA tools described within the following sections.

The ADL is based on a special markup language for coding hierarchical structured data in a text document. It is comparable to XML [32] and JSON [33] but offers the flexibility to use variables as well as constant mathematical expressions. Additionally, it allows to use for and if constructs. This enables the flexibility to describe regular MPSoC structure very abstract. E.g. central parameters could define the width and height of a MPSoC processor array and the processor array could be constructed by for loops dependent on the parameters. 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.

Based on the markup language, the structure of the ADL description is specified. The ADL is structured in various top sections that allow specifying the ALMA target architectures from a structural perspective including high-level information. Thereby, we rely on the concept of modules, instances, and connections as widely used by high-level description languages like VHDL and SystemVerilog but without describing the individual modules and connections on bit-level 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: *Modules* contains all available modules within the system. Each module has a list of ports, a set of parameters (e.g. the delay of an network component), high-level information (e.g. the module is a processor core, cache, network component, etc.), and simulation information (e.g. the SystemC class that simulates the module). The *TopLevel* top section contains all module instances and connections. It contains a list of module instances, each instance referring to a previously specified module. Each port of a module instance is connected to another port.

The modules could be extended by high-level information. A type specifies the kind of high-level information, e.g. memory, cache, processor core. Dependent on the module type, the additional information varies. E.g. for processor cores the SIMD instruction set is specified including throughput and delay information while for caches the size, line width, hit/miss delay, and associativity could be given.

B. ADL Parser

The ADL Parser is an utility of the ALMA toolchain that aims to parse and analyze the architecture description of the various architecture descriptions. Thereby, it prepares the structural ADL and extracts relevant information required by fine- and coarse-grain parallelism extraction as well as parallel code generation. Additionally, it is used as part of the ALMA architecture simulator.

In Figure 3 the usage of the ADL Parser is shown. After extraction of high-level information from the structural architecture representation, the ADL Parser output the information in JSON format. The JSON files are used by the GeCoS framework of ALMA to control the parallelism extraction and code generation phases. Thereby, the language barrier between Java and C++ and the code reuseability motivates the approach to generate JSON files and not directly parse the ADL within the GeCoS framework. Within the GeCoS framework, the information of the ADL are brought by a well-defined API to the individual phases:

`getCoreCount` Returns the maximum number of parallel processing cores.

`getCommunicationInformation(Core1, Core2)` Returns the communication information for transferring data between Core1 and Core2. This includes the delay in clock cycles, the bandwidth in bytes per cycle, and the start-up costs in cycles.

`getCoreMemoryInformation(Core)` Returns a list of memories for one core. Each memory contains the mapping into the address space, the delay, bandwidth, size, and information about the involved caches.

`getDataTypes(Core)` Returns a list of supported data types by the target architecture. Thereby, it is differed between basic and SIMD data types. The Kahrisma architecture e.g. natively supports 32-bit integer data type as well as 4 x 8-bit and 2 x 16-bit SIMD data types.

`getInstructions(Core)` Returns a list of supported assembly instructions by the target architecture. Each instruction is specified by its semantic, operands (including the operands data type), intrinsic function or C operator name, and resource consumption within the processor pipeline (e.g. the functional unit that can execute the instruction including its execution time in cycles on the unit). The semantic is specified by a pattern tree of target-independent instructions with special support for vector and saturation operations.

`getCoreArchitecture(Core)` Returns high-level information about the architecture of one core. In particular, an abstract description of the pipeline including e.g. the issue slots of a VLIW processor as well as the compiler toolchain that should be used for compilation together with some special compilation parameters.

The first three functions (getCoreCount, getCommunicationInformation, and getCoreMemoryInformation) are used within coarse-grain parallelism extraction by the *basic block performance estimator*. The basic block performance is estimated by the basic block execution time on the target core as well as the transfer delay of dependent Scilab variables from distinct cores. The execution time can be determined by a static analysis of the compiled assembler code of one block. The transfer delay can be estimated by start-up costs, delay, and bandwidth as well as the variable size. In Scilab, the size of array variables are not fixed at compile time and therefore profiling information including variable size are required from the ALMA profiler.

For fine-grain parallelism extraction, the getDataTypes and getInstructions functions are used to enable a retargetable vectorization flow. These functions return a list of available SIMD data types and SIMD instruction. Therefore, an as realistic as possible performance model is used that further requires a abstract pipeline description from getCoreArchitecture and memory delay/cache miss information from getCoreMemoryInformation.

SIMD instructions are not natively available within the C language. Therefore, the SIMD instructions within the ALMA IR are converted during parallel code generation with help of getInstructions into target-specific intrinsic functions that are supported and translated back by the compiler back-end to target instructions. For C code compilation, the appropriate compilation toolchain is than selected using the getCoreArchitecture function.

V. Conclusion

In this paper, we presented the ALMA toolset that aims to deliver an end-to-end solution for semi-automatic parallelization

of embedded applications to reconfigurable multi-core architectures. The solution tries to hide the software parallelism caused by multiple cores as well as the flexibility of reconfigurable architectures to the end user. The source input language for the applications is an extension of Scilab, a high-level, pointer-free, numerically-oriented programming language similar to the MATLAB language. Scilab, together with ALMA-specific extensions, enables a simplified parallelism extraction compared to conventional approaches that are based on the imperative C language. Besides Scilab, the ALMA software framework uses an *Architecture Description Language* (ADL) file as input containing an abstract architecture description of the target architecture. In that way, the ALMA toolchain is not tailored to one or more hardcoded architectures. Instead, it supports code generation for the ALMA architecture space, i.e. all architectures that can be expressed in the ADL and targeted by the toolchain. Furthermore, the ADL could be used to express configurations of reconfigurable architectures and thus the overall ALMA toolset enables configuration-space exploration of reconfigurable architectures.

The ALMA compilation flow is separated into the frontend, fine-grain parallelism extraction, coarse-grain parallelism extraction, and parallel code generation. The frontend generates the ALMA IR out of the annotated Scilab input language. Fine-grain parallelism extraction exploits the available data-level parallelism and selects appropriate data types. Coarse-grain parallelism extraction partitions, maps, and schedules the tasks to the target processors. The parallel platform code generation compiles the ALMA IR to executable machine code. Besides the compilation flow, the ALMA toolset comprises a fully functional SystemC simulation framework for multi-core architectures, which will be defined through generic SystemC interfaces/protocols to connect existent simulation modules targeting multiple architectures. The overall toolset is kept flexible towards the target architecture by using the ADL description as input. Thus, the software frameworks enables exploration of the configuration space covered e.g. by the introduced Kahrisma and Recore architectures.

ACKNOWLEDGMENT

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

REFERENCES

[1] F. Thoma, M. Kuhnle, P. Bonnot, E. Panainte, K. Bertels, S. Goller, A. Schneider, S. Guyetant, E. Schuler, K. Muller-Glaser, and J. Becker, "Morpheus: Heterogeneous reconfigurable computing," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, aug. 2007, pp. 409 –414.

[2] T. Ahonen, T. D. ter Braak, S. T. Burgess, R. Geißler, P. M. Heysters, H. Hurskainen, H. G. Kerkhoff, A. B. J. Kokkeler, J. Nurmi, G. K. Rauwerda, G. J. M. Smit, and X. Zhang, "CRISP: Cutting Edge Reconfigurable ICs for Stream Processing," in *Reconfigurable Computing: From Fpgas to Hardware/Software Codesign*, J. M. P. Cardoso and M. Hübner, Eds. London: Springer Verlag, 2011, pp. 211–238.

[3] Scilab Consortium (Digiteo), "Scilab," http://www.scilab.org.

[4] MathWorks, "MATLAB," http://www.mathworks.com/.

[5] R. Koenig, L. Bauer, T. Stripf, M. Shafique, W. Ahmed, J. Becker, and J. Henkel, "Kahrisma: A novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, march 2010, pp. 819 – 824.

[6] R. Koenig, T. Stripf, J. Heisswolf, and J. Becker, "A scalable microarchitecture design that enables dynamic code execution for variable-issue clustered processors," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, may 2011, pp. 150 –157.

[7] ——, "Architecture design space exploration of run-time scalable issue-width processors," in *Embedded Computer Systems (SAMOS), 2011 International Conference on*, july 2011, pp. 77 –84.

[8] T. Stripf, R. Koenig, and J. Becker, "A novel ADL-based compiler-centric software framework for reconfigurable mixed-ISA processors," in *Embedded Computer Systems (SAMOS), 2011 International Conference on*, july 2011, pp. 157 –164.

[9] ——, "A compiler back-end for reconfigurable, mixed-isa processors with clustered register files," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2012 IEEE International Symposium on*, may 2012.

[10] ——, "A cycle-approximate, mixed-isa simulator for the kahrisma architecture," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, march 2012, pp. 21 –26.

[11] "Xentium® DSP Core," http://www.recoresystems.com/technology/.

[12] J. A. Fisher, P. Faraboschi, and C. Young, "VLIW processors," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 2135–2142.

[13] G. K. Rauwerda, P. M. Heysters, and G. J. M. Smit, "Towards software defined radios using coarse-grained reconfigurable hardware," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 3–13, January 2008.

[14] "Montium® DSP Core," http://www.recoresystems.com/technology/.

[15] Recore Systems, "Product," http://www.recoresystems.com/products.

[16] E. Schuler, R. König, J. Becker, G. Rauwerda, M. B. van de, and G. J. Smit, "Smart chips for smart surroundings - 4s," in *Reconfigurable Computing: From FPGASs to Hardware/Software Codesign*, J. ao M.P. Cardoso and M. Hübner, Eds. London: Springer Verlag, 2011, pp. 117–148. [Online]. Available: http://doc.utwente.nl/78380/

[17] K. H. Walters, S. Gerez, G. Smit, G. Rauwerda, S. Baillou, and R. Trautner, "Multicore SoC for on-board payload signal processing," in *NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2011*. USA: IEEE Computer Society, July 2011, pp. 17–21. [Online]. Available: http://doc.utwente.nl/77933/

[18] T. D. ter Braak, H. A. Toersche, A. B. Kokkeler, and G. J. Smit, "Adaptive resource allocation for streaming applications," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2011*, L. Carro and A. Pimentel, Eds. USA: IEEE Circuits & Systems Society, July 2011, pp. 388–395.

[19] OpenMP Architecture Review Board, "OpenMP specification," http://www.openmp.org.

[20] "GeCoS," https://gforge.inria.fr/projects/gecos/.

[21] N. Kavvadias and K. Masselos, "NAC: A lightweight intermediate representation for ASIP compilers," in *Proc. Int. Conf. on Engin. of Reconf. Sys. and Applications (ERSA11)*, Las Vegas, Nevada, USA, Jul. 2011, pp. 351–354.

[22] A. W. Appel, "SSA is functional programming," *ACM SIGPLAN Notices*, vol. 33, no. 4, pp. 17–20, Apr 1998.

[23] J. Cordy, "Specification and tools for the TXL (Turing eXtension Language)," http://www.txl.ca.

[24] A. Iranpour and K. Kuchcinski, "Evaluation of simd architecture enhancement in embedded processors for mpeg-4," in *Digital System Design, 2004. DSD 2004. Euromicro Symposium on*, aug.-3 sept. 2004, pp. 262 – 269.

[25] D. Ménard, R. Rocher, and O. Sentieys, "Analytical fixed-point accuracy evaluation in linear time-invariant systems," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 55, no. 10, pp. 3197 –3208, nov. 2008.

[26] D. Ménard, D. Chillet, and O. Sentieys, "Floating-to-fixed-point conversion for digital signal processors," *EURASIP J. Appl. Signal Process.*, vol. 2006, pp. 77–77, January 2006.

[27] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen, "Polyhedral-model guided loop-nest auto-vectorization," in *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, sept. 2009, pp. 327 –337.

[28] T. Henretty et al., "Data layout transformation for stencil computations on short-vector simd architectures," in *Proc. of the 20th intl. conf. on Compiler construction: part of the joint European conf. on theory and practice of software*, ser. CC'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 225–245.

[29] *MPI: A Message-Passing Interface Standard Version 2.2*, Message Passing Interface Forum, September 2009. [Online]. Available: http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

[30] "LLVM Compiler Infrastructure," http://www.llvm.org.

[31] "Open systemc initiative," http://www.systemc.org/, 2006.

[32] T. Bray, J. Paoli, C. M. Sperberg-McQueen, "Extensible markup language (XML) 1.0 W3C recommendation. Technical report," 2008.

[33] D. Crockford, "JSON: the fat-free alternative to XML," in *Proceedings of XML 2006, USA*, 2006.