2012 15th Euromicro Conference on Digital System Design

From Scilab To High Performance Embedded Multicore Systems – The ALMA Approach

(Invited Paper)

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

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

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

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

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

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

Diana Goehringer
Fraunhofer-Institute of Optronics, System Technologies
and Image Exploitation, Germany
diana.goehringer@iosb.fraunhofer.de

***Abstract*—The mapping process of high performance embedded applications to today's multiprocessor system on chip devices suffers from a complex toolchain and programming process. The problem here is the expression of parallelism with a pure imperative programming language which is commonly C. This traditional approach limits the mapping, partitioning and the generation of optimized parallel code, and consequently the achievable performance and power consumption of applications from different domains. 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 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 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.**

I. Introduction

Chips are needed that are efficient, flexible, and performant. 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 heterogeneity and parallel processing on a real hardware prototype. 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 having to know 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 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.

The ***A**rchitecture oriented para**L**lelization for high performance embedded **M**ulticore systems using scil**A**b* (ALMA, Greek for "leap") European project [3] intends to provide a full design framework for designing parallel and concurrent computing systems. The design framework will rely on Scilab, an open source language for developing high-level system models. Scilab will be extended so as to provide explicit parallel directives, which will allow high level optimization of Scilab system models, based on user defined cost functions and the constraints of the underlying architecture. The ALMA parallel software optimization environment will be combined with a fully functional SystemC simulation framework for multicore heterogeneous SoCs, which will be defined through generic SystemC interfaces/protocols to connect existent *Multiprocessor System-on-Chip* (MPSoC) simulation modules targeting multiple architectures.

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: Section II gives an overview of the ALMA toolset followed by in-depth descriptions of the individual components. The toolset is based on an *Architecture Description Language* (ADL) that is introduced in Section III. Section IV explains the ALMA frontend responsible for generating the *ALMA Intermediate Representation* (ALMA IR) out of the Scilab input language. The coarse grain parallelism extraction (Section V) partitions, maps, and schedules the tasks to the target processor cores while the fine grain parallelism

978-0-7695-4798-5/12 $26.00 © 2012 IEEE
DOI 10.1109/DSD.2012.65

extraction (Section VI) exploits data-level parallelism. Parallel platform code generation (Section VII) compiles the optimized ALMA IR to machine code that could be simulated by the multicore architecture simulator (Section VIII). In Section IX the ALMA application test cases are introduces and Section X concludes the paper.

II. ALMA Toolset Overview

The ALMA toolset provides an end-to-end tool chain from Scilab [4] code to executable code on embedded multicore systems. A typical end user that will use the ALMA toolset will start the application development by implementing Scilab code instrumented by comment-type annotations and specifying an abstract description of the target architecture using the ALMA *Architecture Description Language* (ADL). The end result will be parallelized executable code ready to run on the designated multicore embedded platform. In this user-driven perspective, two distinct phases can be identified. The first phase includes code transformations from Scilab to an *Intermediate Representation* (IR) and optimizations based on the IR and the architecture specification. The second phase is closer to the hardware and is responsible for transforming the IR produced by the first phase to executable code for the target embedded multicore architecture. The toolset workflow is presented in Figure 1. Throughout the two phases, the embedded multicore platform ADL and the parallel program IR are going to present the ALMA tools as an integrated toolset.

The first phase of the toolset operates in an intermediate code representation of the Scilab source code and performs optimizations based on a multicore ADL description (see Section III). This phase consists of three big steps: The frontend, the coarse grain parallelism extraction and optimization, and the fine grain parallelism extraction. The frontend representation converts the Scilab code into the ALMA IR and performs a set of preliminary compiler optimizations. Also, the frontend processes the ALMA defined Scilab language extensions and encodes them in the ALMA IR. The ALMA Scilab language extensions are included in the code as special comments – similar to OpenMP [5] pragmas – and are used to guide the parallelism extraction process. The coarse grain parallelism extraction and optimization starts with the ALMA IR produced by the frontend, which in effect is a *Control and Dataflow Graph* (CDFG) and produces a new ALMA IR with assignments of the tasks of the graph 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 fine grain parallelism that follows, implements local code optimizations in loops and small code constructs for each specific core. The optimization phases can be executed in an iterative manner, until no further optimizations are possible or a time limit has been reached.

The diagram in Figure 2 shows the ALMA approach from the perspective of the second phase. The figure distinguishes between hardware and software. On the bottom multicore parallel *System-on-Chip* (SoC) architectures are implemented, such as multicore SoCs based on Recore's reconfigurable DSP cores or Kahrisma [6], [7], [8] cores. The ALMA toolset from Figure 2 basically shows how the ALMA approach from Figure 1 is integrated with the multicore hardware/simulator. Figure 2 depicts that the output of the ALMA tools (e.g. Figure 1) is C-based code with

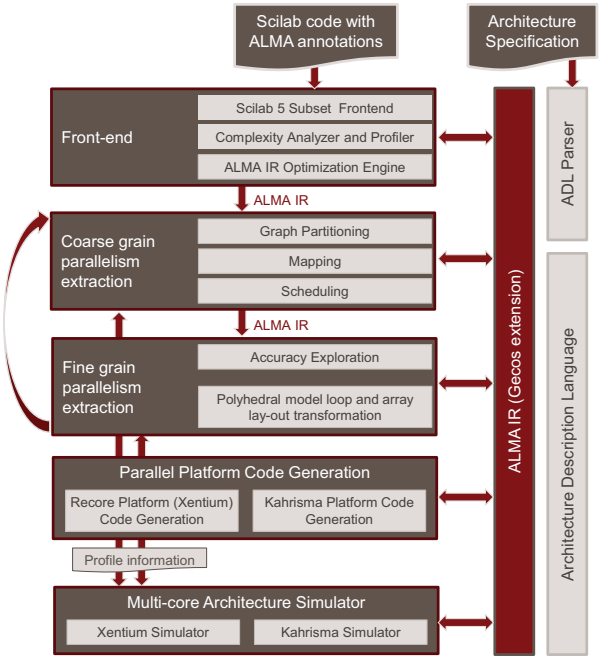

Fig. 1. ALMA Toolset Overview

parallel descriptions. This C-based code is taken as input for the multicore 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) Moreover, additional characteristics about the multicore hardware are defined which will be used during the optimization steps of the ALMA tools.

Figure 2 clearly indicates that the ALMA tools are built on top of the multicore hardware and their default compiler tools.

III. 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 data base to gather information about the current target architecture. 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. While there exists a couple of ADLs, no one 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

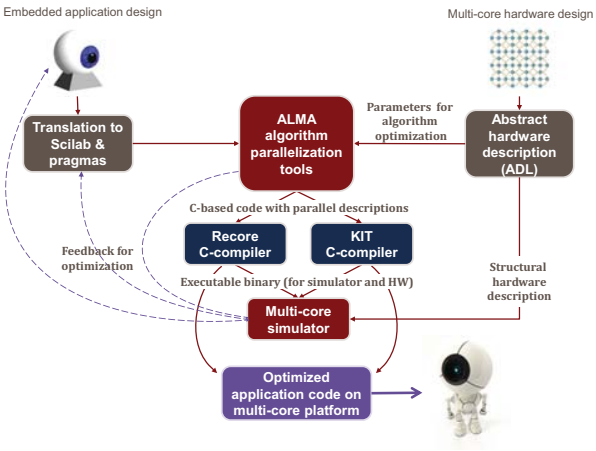

Fig. 2. ALMA Toolset From Second Phase Perspective

XML [9] and JSON [10] 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.

A. 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:

- the number of available cores
- which cores can communicate with each other
- the delay and throughput of any pairwise communication between two cores
- a list of all available memories including the connected cores, their access delay, and access width
- processor core information
  - supported data types
  - a list of supported SIMD instructions
  - unaligned memory access overhead

To structure the processing and information extraction the ADL Parser is organized in passes that subsequently parse and analyze the given target ADL description. The passes comprise markup parsing, ADL parsing, analyzing, and output generation:

Markup Parsing is the first pass and parses the markup language into a tree based data structure. Thereby, variable propagation, mathematical expression calculation, and for/if statement interpretation is performed.

ADL Parsing analyzes the content of this tree and converts its high-level content into internal data structures that are organized similar to the sections available within the ADL. Thereby, it is checked if the ADL is semantically correct, e.g. the references between the sections must be correct and the interfaces of the connected ports must match.

The **Analyzing** pass extracts high-level information from the structural ADL description for the coarse and fine grain parallelism extraction as well as parallel code generation.

Output Generation The high-level information that is extracted from the previous pass is outputted in a standard data serialization format that is used by the other tools as input, i.e. fine and coarse grain parallelization as well as parallel code generation.

IV. Scilab Frontend

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

- Frontend specification for parsing Scilab - annotations for parallelism
- Scilab 5 augmented subset frontend parser
- ALMA IR generator
- Complexity analyzer and profiler
- ALMA IR optimization engine

The Scilab subset frontend will support a large subset of the grammatical aspects of Scilab that are either directly or macro-compatible to MATLAB 5 [11]. Idiosyncratic elements of Scilab such as embedded C code blocks will not be supported. An open issue on Scilab support for practical end-user programs is the policy regarding Scilab intrinsic functions. Scilab uses two forms of intrinsic functions: a) “fundamental” ones written in a core language (C or Fortran depending on the case) and b) “derived” ones written in Scilab and accessible in source

files with a sci_* prefix. Typical examples of the first case are elementary functions (acos, atan, asin, exp) and of the second case pmodulo, primes and other numerical algorithms that are acceptable to be written in a higher level of abstraction. It should be noted that even the fundamental intrinsic functions have dependencies to the system math library (e.g. glibc or newlib) so their implementation at the most basic level is opaque. Certain exceptions are handled by external packages, for instance *Fast Fourier Transform* (FFT) functionality is provided by the FFTW library [12]. All these functions would naturally be treated as black boxes by the frontend. The end users should either avoid using such functions, or proper source implementations should be identified for them. Alternatively, a self-contained C library implementing these functions should be compiled by the native C compilers of the target platforms.

The ALMA frontend for Scilab will also support a preprocessing language for specifying static types and variable declarations that ensure the feasibility of certain static analyses. The annotation language for specifying parallelism will be a new formalism based on existing schemes such as OpenMP and GeCoS [13] pragmas. The annotation language will support empirical hypotheses of the end user regarding attainable task/loop parallelism, which can be surpassed by the coarse- and fine-grain parallelism extractors. The actual format of the language is a form of meta-comment pragma-like grammar.

The complexity analyzer/profiler will attempt early static and dynamic profiling on the Scilab source application with the help of a Scilab/MATLAB abstract machine. This abstract machine should be considered as a hypothetical processor directly executing ALMA MATLAB "bitcode". The operation set of the *PEloponnese ALMA Scilab/MATLAB BITcode* (PEMBIC) will be developed having in mind similar efforts such as McVM (however, intended for JIT compilation) [14] as well as other high/mid-level virtual machines. The PEMBIC will be defined as a linearized textual form closely matched to the actual internal ALMA IR, simply termed as *High-Level Intermediate Representation* (HLIR).

The profiler will provide statistics on the source input (block frequencies, number of operation classes), and a complexity analysis report highlighting hotspots of the applications and frequently executed patterns. Further, a list-based scheduler should be an integral part of the profiler for realistic analysis of performance on generic architectural templates. The ALMA frontend will produce a language-specific *Abstract Syntax Tree* (AST) that will then be converted to a neutral HLIR. The tree-based HLIR source representation can be converted to standalone, platform-independent ANSI/ISO C (C90) backend code.

The ALMA IR optimizer attempts to apply generally beneficial optimizations to the HLIR. Since the GeCoS fine-grain optimizer provides a comprehensive framework for optimizing loop-intensive SCoPs of programs, the frontend optimizations will focus on the following aspects:

- Aggressive as well as selective procedure inlining
- Graph-based procedure abstraction (a form of exlining)
- Algebraic optimizations such as constant multiplication and division optimization (forms of operator strength reduction)
- Loop invariant code motion
- Generic code canonicalizations
- Constant propagation and folding
- Copy propagation
- Common subexpression elimination
- Dead code elimination

Most of these optimizations will be developed as grammatical transforms using either a customized (written in C/C++) or a TXL-based term-rewriting engine [15].

V. Coarse Grain Parallelism Extraction and Optimization

The coarse grain parallelism extraction and optimization step of the ALMA toolset is responsible for a global, i.e. program wide optimization of the program.

The coarse grain parallelism extraction module has three inputs, the program representation in ALMA *Intermediate Representation* (IR), the *Architecture Description Language* (ADL) description, and the user optimization goals. Since the architecture representation may not fully represent the architecture behavior, the coarse grain parallelism extraction interacts with the architecture simulator modules through the ALMA toolset APIs in order to get accurate run-time information for the generated programs, in order to use them for fine tuning the optimization process.

The ALMA IR is an extended version of the GeCoS framework IR. The GeCoS IR offers the advantage to automatically maintain a synchronized AST with a CDFG and offers a simple and extensible basis for the project needs. Each node in the CDFG represents a task. Each task represents one or more Scilab statements and, for the coarse grain parallelism extraction is an indivisible amount of computational work. The CDFG defines control and data dependencies for each task. These dependencies impose constraints that the coarse grain parallelism extraction will actively use to identify how the task graph should be partitioned and what will be the cost of data transfers.

The coarse grain parallelism extraction module needs average timing information for typical instructions and memory operations from the architecture description (ADL) or a way to directly acquire this information from the architecture simulator for individual basic blocks. Moreover, the coarse grain parallelism extraction module could also optimize its cost-function to other parameters, such as power consumption. Also, the coarse grain parallelism extraction needs an abstract overview of the architecture, namely number of cores as well as an accurate memory subsystem organization.

The user preferences for optimization can be a combination of optimization goals for speed, number of cores, performance, and power consumption. The optimization goals can be combined in a user-defined formula of weight coefficients. In addition, the user may provide performance thresholds in the number of cores, power consumption and running time as well as other user defined characteristics that can be quantitatively derived from information provided to the optimization engine. In addition to the above inputs, it is possible for the user to provide a set of representative program inputs. Based on this input, profile specific optimization can be performed.

The first step for the coarse grain parallelism extraction is to partition the CFDG graph into sub graphs. The goal of this step is to generate sub graphs that can be combined to generate the whole original program and have minimal dependencies between them. The partitioning engine will use custom developed algorithms or use open source graph partitioning libraries (e.g.

Chaco [16], hMETIS [17], driven by a cost function). The task graph incorporates estimated task computation times (as derived by profiling) and loose/pessimistic estimates on the intertask communication timings.

Following partitioning, a mapping process can assign the produced sub graphs to different cores for execution of the target architecture in an efficient manner. Each task to core assignment may result in further communication costs as well as imbalance to the execution time and power consumption – due to control and data dependencies of the sub graph nodes. In order to enhance scheduling efficiency, the original CDFG may be enhanced with additional nodes. These additional nodes will include node duplicates and data transfer nodes. Node duplicates are used when it is cheaper to re-compute a result instead of transferring it from a distant memory. Data transfer nodes can transfer data at a convenient time, in order for future tasks to have it available. When it makes sense, nodes can be aggregated into super-nodes to simplify the scheduling phase.

From the above descriptions, it is apparent that the partitioning and scheduling phases would be performed in an iterative process or if the modeling of the problem and the performance of the algorithms permit it simultaneously. The coarse grain parallelism extraction will include an iterative process for these phases but also a combined process. From an optimization point view, splitting a process into phases generates biases to local optima based on each phase characteristics. This is not a desired behavior, but the division significantly reduces the combinatorial complexity in order to obtain high quality feasible solutions in a timely manner. Thus, the coarse grain parallelism extraction will investigate alternatives for both an iterative two-phase process but also attacking the problem as a whole in one phase.

The output of the coarse grain parallelism extraction phase is ALMA IR, which is an extended GeCoS IR, instrumented with the necessary artifacts in order to be ready for generation to the target architectures by the other ALMA toolset modules. It might not be ready for execution to the target embedded architectures, since certain code artifacts may not be addressed at coarse grain parallelism extraction phase but during the fine grain parallelism extraction phase. When the phases are executed in an iterative manner, the coarse grain parallelism extraction would be able to produce readily executable GeCoS IR.

VI. Fine Grain Parallelism Extraction

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: data type selection and memory access aware vectorization. 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 [18] including the RECORE and KAHRISHMA 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 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 2x32, 4x16 and 8x8 bits vector additions). 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 the SIMD extension (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 [19], and exposes complex performance/accuracy trade-off optimization problem that we want to automate in this component [20].

In our flow, the extraction of fine grain parallelism comprises two stages: data type binding and data parallelization. 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 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 exposed using intrinsic functions.

A. 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 type remains very limited, as it must correspond to a machine supported type (byte, word, double word), and therefore reduces the optimization search space.

The end user can guide the data type selection processor by pragmas annotations in the Scilab source code, which provides accuracy and dynamic range constraints to the component. The annotations used for fine grain optimization module are:

`//pragma alma_output VAL_DB`: This pragma specifies which variable holds the system output, along with an accuracy constraint corresponding to the quantization noise power expressed in dB. The value of the constraint is defined with VAL_DB.

`//pragma alma_dynamic var [MIN, MAX]`: This pragma specify through MIN and MAX the interval defining the dynamic range of the variable associated with the pragma.

B. 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 [21]. 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 a possible the need for unaligned memory access and vector packing/unpacking instructions [22].

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 also provided efficient code generation features.

VII. 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 instruction 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 and afterwards compiled to target binaries. The advantage is that we can reuse existing, target-specific C compilers generating optimized assembler code.

A. Parallel C Code Generation

The parallel C code generation 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 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. During execution the control code calls the functions implementing the tasks according to the schedule.

The communication primitives use an architecture independent API. As API we rely on a subset of the *Message Passing Interface* (MPI) [23] that can be efficiently supported by both target architectures. The communication on multiprocessor systems can present a huge impact on the whole system performance. The start-up cost of communication for distributed-memory architectures is typically greater than the per-byte transmission cost. That offers optimization potential that is used by the three following communication optimizations by combining messages in various ways to reduce the total amount of communication overhead [24]. Message coalescing [25] detects redundant communications and coalesced them into a single message, allowing the data to be reused rather than communicated for every reference. Separate communications for different references to the same data are avoided if the data has not been modified between uses. Message aggregation [26] reduces the number of messages between the same source and destination by aggregating them into a single larger message at the expense of copying them to a single contiguous buffer. Message vectorization optimizes array element accesses indexed within nested loops that can be vectorized into a single larger message. Dependence analysis is used to determine the outermost loop at which the combining can be applied. A number of optimizations seek to hide communication overhead by overlapping message with communication. Message pipelining attempts to hide message transfer time by separating send and receive primitives for element message. Message pipelining is supported by various architectures by non-blocking messages that hide the message copy time.

B. Single-Core Code Generation

The Xentium VLIW DSP processor [27] and the Kahrisma architecture are both ALMA target processors. Both target architecture are coming along with a software toolchain including a LLVM-based C compiler, an assembler, and linker [28]. Depending on the specified target architecture within the ADL description the appropriated 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.

The C compilers – based on the LLVM compiler infrastructure [29] – are separated into three parts: the front end, the middle end, and the back end. The front end checks whether the input program is correctly written in terms of the programming language syntax and semantics. It generates a LLVM *Intermediate Representation* (IR) of the source code for further processing by the middle end and back end. The middle end performs target independent optimizations on the LLVM IR. Typical transformations for optimization are removal of useless or unreachable code, discovery and propagation of constant values, or relocation of computation to a less frequently executed place (e.g. out of a loop). The back-end (also refereed as code generation) is responsible for translating the LLVM IR into assembly code. Instruction selection choses target instruction(s) for each IR instruction. Thereby, the intrinsic SIMD functions inserted by fine grain parallel extraction (see Section VI) are converted into target SIMD instructions. Register allocation assigns processor registers for the program variables where possible. The back end utilizes the hardware by figuring out how to keep parallel execution units busy, filling delay slots, and so on.

Besides instruction selection and register allocation, the compilation back end is responsible for scheduling the instructions of the target architectures. 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 rely on VLIW processor cores. 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. To improve the ILP several compilation techniques are known including superblock scheduling [30], trace scheduling [31], hyperblock scheduling [32], and modulo scheduling [33].

VIII. Multicore Architecture Simulation

The multicore architecture simulator will enable the simulation of all ALMA architectures. It uses as input the application binary generated by the parallel code generation as well as an ADL description. As output it will generate profiling information that is used by coarse and fine grain parallelism extraction for profile-based optimizations. The multicore architecture simulator will use one or more application binaries generated by the parallel code generation. It will allow starting on each processor core the same or a different executable file. The executable file will be available in *Executable and Linkage File Format* (ELF) [34].

Additionally, the multicore architecture simulator will use an ADL file as input to specify the target architecture to be simulated. The ADL file will contain a structural specification of the components of the target multicore hardware architecture. Based on the structural architecture description (i.e. ADL) the multicore simulation environment is established by instantiating modular simulation components. The ALMA multicore architecture simulator will rely on the SystemC/TLM [35] simulation language. SystemC is a set of C++ classes and macros that enable event-driven simulations within C++.

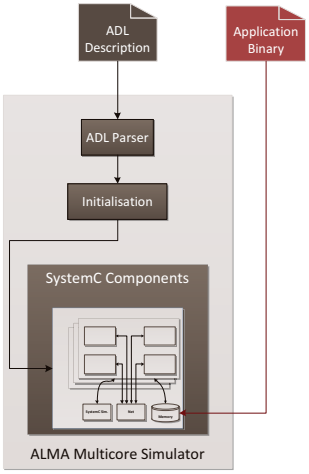

Fig. 3. ALMA Multicore Simulator

Figure 3 shows the components of the simulator. The ADL description is parsed by the ADL parser and stored into internal data structure. The ADL Parser will be embedded as a library into the simulator. Afterwards, the initialization uses the structural information stored in the internal data structure to instantiate and connect the SystemC modules according to the target architecture as described within the ADL file.

After initialization of the SystemC modules, the application binary is loaded into the available memory by using an ELF loader. 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 multicore 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. This information will comprise e.g.:

- Communication Network
  - Network usage/utilization
- Memory hierarchy
  - Cache miss/hit rates
  - Memory network usage/utilization
- Processor Cores
  - Instruction statistics
  - Profiling information on function and basic block level
  - Tracing information on function level
  - Trace file generation

IX. Application Test Cases

To ascertain that programmers can directly apply the programming tool chains that the ALMA consortium develops in the course of the project, ALMA targets two cases in market domains with different, 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 architectures 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 requires high-quality models that produce optimized code, and release 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 an image processing application in the area of multi-object detection and tracking. Multi-object tracking is often implemented on embedded systems with strong restrictions on power consumption and heating. The algorithms cope with a variable number of objects and must process data on each object within a small time frame. The computational load can vary heavily from image to image. For embedded systems offering low computational power, algorithms must be adapted and parallelized over multiple processors, often resulting in poor performance of the application. The tool supported, guided development that ALMA intends to deliver, which hides the complexity of the architecture of the embedded system and the algorithm from the developer result in an enormous increase of the system performance and definitely to a reduction of time consuming implementation.

X. Conclusion

In this paper we presented the ALMA toolset that aims to deliver 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 abstract 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 parallel platform code generation compiles the ALMA IR to executable machine code. The ALMA parallel software optimization environment 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 software 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

[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. Gei?ler, 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] "**A**rchitecture oriented para**L**lelization for high performance embedded **M**ulticore systems using scil**A**b (ALMA)," http://www.alma-project.eu.

[4] "Scilab," http://www.scilab.org.

[5] "OpenMP specification," http://www.openmp.org.

[6] 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.

[7] 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.

[8] ——, "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.

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

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

[11] "Mathworks Inc." http://www.mathworks.com.

[12] "FFTW," http://www.fftw.org.

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

[14] "McLAB project," http://www.sable.mcgill.ca/mclab/.

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

[16] "Chaco: Software for partitioning graphs," http://www.sandia.gov/ bahendr/chaco.html.

[17] "hMETIS - Hypergraph & Circuit Partitioning," http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview.

[18] 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.

[19] D. Menard, 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.

[20] D. Menard, 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.

[21] 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.

[22] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector simd architectures," in *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, ser. CC'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 225–245. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987237.1987255

[23] *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

[24] D. J. Palermo, E. Su, J. A. Chandy, and P. Banerjee, "Communication optimizations used in the paradigm compiler for distributed-memory multicomputers," in *Proceedings of the 1994 International Conference on Parallel Processing - Volume 02*, ser. ICPP '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 1–10.

[25] M. J. Koop, T. Jones, and D. K. Panda, "Reducing connection memory requirements of mpi for infiniband clusters: A message coalescing approach," in *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, ser. CCGRID '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 495–504. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2007.92

[26] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiling fortran d for mimd distributed-memory machines," *Commun. ACM*, vol. 35, pp. 66–80, August 1992. [Online]. Available: http://doi.acm.org/10.1145/135226.135230

[27] Recore Systems, "Xentium architecture," http://www.recoresystems.com/technology/xentium-technology.

[28] 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.

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

[30] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: an effective technique for vliw and superscalar compilation," *J. Supercomput.*, vol. 7, pp. 229–248, May 1993. [Online]. Available: http://dx.doi.org/10.1007/BF01205185

[31] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. 30, pp. 478–490, July 1981. [Online]. Available: http://dx.doi.org/10.1109/TC.1981.1675827

[32] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," *SIGMICRO Newsl.*, vol. 23, pp. 45–54, December 1992. [Online]. Available: http://doi.acm.org/10.1145/144965.144998

[33] B. R. Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops," in *Proceedings of the 27th annual international symposium on Microarchitecture*, ser. MICRO 27. New York, NY, USA: ACM, 1994, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/192724.192731

[34] TIS Committee, "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2," http://refspecs.freestandards.org/elf/elf.pdf.

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