# From Scilab to Multicore Embedded Systems: Algorithms and Methodologies

*(Invited Paper)*

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

Christos Gogos
Technological Educational Institute of Epirus, Greece
cgogos@teiep.gr

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

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

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

Michael Huebner
Ruhr-University of Bochum, Germany
michael.huebner@rub.de

Timo Stripf, Oliver Oey, Juergen Becker
Karlsruhe Institute of Technology, Germany
{stripf, oliver.oey, becker}@kit.edu

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

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

***Abstract*—While advances in processor architecture continues to increase hardware parallelism, parallel software creation is hard. There is an increasing need for tools and methodologies to narrow the entry gap for non-experts in parallel software development as well as to streamline the work for experts. This paper presents the methodology and algorithms for the creation of parallel software written in Scilab source code for multicore embedded processors in the context of the "Architecture oriented paraLlelization for high performance embedded Multicore systems using scilAb" (ALMA) EU FP7 project. The ALMA parallelization approach in a nutshell attempts to manage the complexity of the task by alternating focus between very localized and holistic view program optimization strategies.**

## I. Introduction

Software development has to exploit parallelism, regardless of the context at which it is considered. Parallelism appears everywhere from supercomputers over personal computers to embedded computing devices. Mobile devices like phones and tablets already exhibit multicore processors, while personal computer users enjoy multicore processors and manycore general computing accelerator cards in the form of graphics processing units. Although parallelism is an established fact, parallel software development is still difficult and substantially different from the sequential way of thinking and creating applications. On one hand there is a definite need for efficient tools for experts in parallel programming, on the other hand there is an increasing demand for tools to enable non-experts to produce efficient parallel software.

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

Within ALMA FP7 project [1], the focus is to bring automated parallelism extraction to engineering-related software developed in a subset of Scilab language. Scilab is an open source tool for numerical computation with a language similar to MATLAB [2]. The target hardware for ALMA consists of embedded *Multi-Processor System-on-Chip* (MPSoC) architectures, with two initial targets briefly introduced in the following section, KIT's Kahrisma and Recore's Xentium-based tiled multi-core architecture. The source input language for the applications is an extension of the Scilab language, with ALMA-specific source code annotations. ALMA is also extensible to other architectures, through a specially designed language for describing parallel platforms that is introduced, the *Architecture Description Language* (ADL). The ALMA parallelism approach tries to alternate focus between small program sections, mainly loops, and the broader program view. These two phases iteratively ameliorate the parallel program efficiency until no further improvement is observed.

The parallel algorithms and methodology are implemented as separate tools that form the ALMA toolset [3]. The ALMA toolset inputs are Scilab source code and the ADL description for the target hardware and the output is efficient parallel implementation ready to run on the designated architecture. The ALMA application drivers include one from the telecommunications industry and another from the image processing domains. The main phases of the ALMA toolset are the frontend for the transformation of the input sources to ALMA *Intermediate Representation* (IR), the coarse-grain parallelism extraction for optimizations applied on the global view of

978-1-4673-2297-3/12/$31.00 ©2012 IEEE

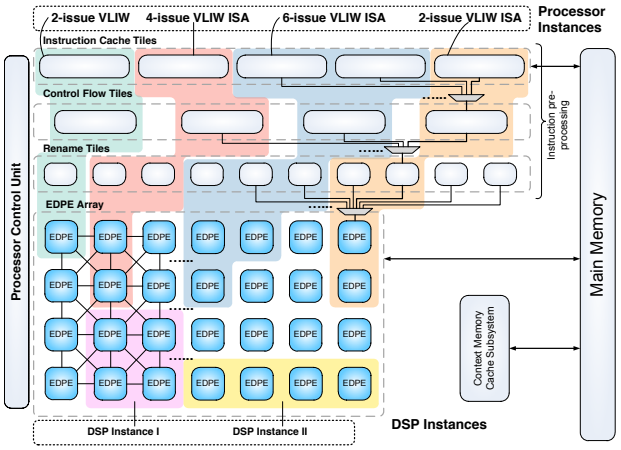


Fig. 1. Kahrisma Architecture

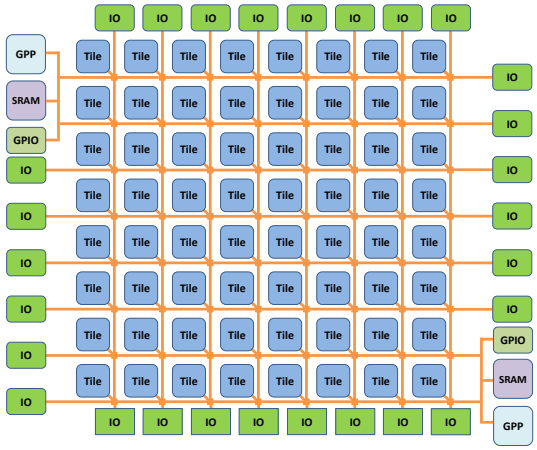


Fig. 2. Illustration of Recore Systems' tiled multicore architecture

the program, the fine-grain parallelism extraction for local optimizations mainly in loops and the parallel platform code generation to produce the final, architecture-specific implementation. The ALMA tool flow is supported by ADL-specific tools to provide common interfaces to access architecture description for exploiting architecture features and enabling architecture specific optimizations, as well as parallel platform simulators for the target architectures enabling application profiling and performance evaluation.

Similar to the ALMA toolset is the MEGHA compiler framework, which aims to compile MATLAB programs for parallel execution across CPU and GPU cores [4]. In the remaining of this paper, we present the initial ALMA embedded multicore target platforms, KIT's Kahrisma and Recore's Xentium-based tiled multi-core architecture. The ALMA approach to parallelism extraction is briefly outlined, followed by the ALMA toolset overview. Also, the algorithmic approaches for various ALMA toolset phases, the frontend, the coarse and fine-grain parallelism extraction are presented. At the end we draw a conclusion.

## II. Embedded Multicore Target Platforms

Some of the most exciting recent developments in embedded architectures are the advances made in multicore and reconfigurable architectures [5]. The ALMA toolchain targets two such novel architectures.

### A. KAHRISMA

The Kahrisma architecture [6], [7], [8], as shown in Figure 1, is a hypermorph reconfigurable processor architecture. It features dynamic reconfiguration of the instruction set as well as instruction format, e.g. switching between 2-issue and 8-issue *Very Long Instruction Word* (VLIW), to execute a configurable number of statically-scheduled operations in parallel. Multiple processor instances (each instance may be configured to execute a different instruction format) can coexist in parallel. Each instruction format requires a different amount of resources and also provides different peak performance characteristics.

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

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

The Kahrisma architecture comes along with a software toolchain [9] for the C programming language including an LLVM-based C compiler [10], an assembler, a linker, and a cycle-approximate single-core simulator [11]. The software toolchain supports code generation for and simulation of all configurable Kahrisma processor instances.

### B. Xentium® DSP multicore platform

Figure 2 shows an illustration of the Recore Systems' tiled multicore *System-on-Chip* (SoC) architecture template. The tiled architecture defines a *Network-on-Chip* (NoC) interconnecting a number of tiles with I/O and other blocks in a 2D mesh. The main tiles are Xentium® DSP cores and memory blocks but can also be small embedded *General-Purpose Processors* (GPP), DMA controllers, accelerators, etc.

The Xentium® processor [12] is a 32/40-bit fixed-point DSP core designed for high-performance embedded signal processing. The VLIW [13] architecture of the Xentium features 10 parallel execution slots and includes support for *Single Instruction Multiple Data* (SIMD) and zero-overhead loops. Instances of the SoC architecture have been implemented in *Integrated Circuit* (IC) [14] and *Field-Programmable Gate Array* (FPGA) [15].

In [16], a run-time resource manager is used to program a multicore platform with 45 Xentium cores and one ARM® processor. The mapping to platform resource is computed and deployed at run-time depending on the currently available resources. To map an application, the resource manager relies on an annotated task graph that specifies processing and communication requirements. 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. Currently, the task graphs are typically constructed mostly manually. The construction includes splitting, merging, and parallelizing tasks as well as task dependency analysis and calculation of processing and communication resource bounds.

## III. Overview of the ALMA Parallelization Approach

The ALMA parallelization process is initiated by user-provided sequential source code to produce an optimized parallel implementation of the original code for the target architecure. Source code instrumentation in the form of special comment annotations, similar to OpenMP [17] pragmas, allows the user to guide the parallelization process when relevant algorithms are not able to provide an optimal result. The comment annotations identify loop and task parallelization opportunities, but unlike OpenMP directives, ALMA source code annotations are hints to the parallelization process and are used to create an initial solution. The automatic parallelization problem is in fact an NP-hard combinatorial problem of optimizing the critical path of the *Control and Data Flow Graph* (CDFG) while ensuring that data dependencies are respected.

For automatic parallelization, it is natural to focus on loop structures, since typically most execution time is spent in such structures. However, it is important to be able to exploit parallelism opportunities at bigger granularity, when they are available. The ALMA parallelization strategy alternates the optimization focus between local and global scopes. Local techniques during the *fine-grain parallelism extraction* (Section VII) phase provide optimized parallel implementations for limited code blocks, like nested loops. The global view of the program is considered during the *coarse-grain parallelism extraction* phase(Section VI), where fine-grain parallelism extraction optimized sections are considered as opaque blocks. Coarse-grain parallelism extraction is a graph optimization process, similar to a project management problem where the critical path has to be optimized. In a nutshell, coarse-grain parallelism extraction clusters sequences of dependendent instructions to prepare for a graph partitioning step, creates graph partitions as processing core tasks, maps the tasks to cores to produce core-resident programs, and finally schedules these programs in order to optimize data transfers.

Both parallelism extraction phases operate directly on the CDFG *Intermediate Representation* (IR) of the code, while they exchange information through labels on the CDFG nodes and edges.

## IV. The ALMA Parallelization Toolset for Embedded Applications

The algorithms developed in ALMA are part of the ALMA toolset aiming to demonstrate an end-to-end user workflow from Scilab source code to embedded parallel code. The user provides source code, instrumented with comment-like annotations to guide the parallelization process and selects a target architecture. The toolset overview is illustrated in Figure 3. The selection of target architecture results in the second major input for the ALMA toolset, the target hardware platform description in the *Architecture Description Language* (ADL). Example input data form a third optional input, in order to allow for realistic platform simulator-driven optimization. The process that follows to produce optimized parallel binaries is analogous to a typical compiler infrastructure, like the LLVM [18], with front-end, middle-end and back-end phases. To summarize those phases, the front-end reads the code into the *ALMA Intermediate Representation* (ALMA IR), the middle-end provides optimizations on the IR based on superficial architecture features like number of cores, and the back-end provides hardware platform-specific optimizations and produces the final platform executables.

The front-end phase for ALMA transforms the user provided Scilab source code and produces an equivalent IR. All later steps operate on the produced *ALMA Intermediate Representation* (ALMA IR), which in effect is a GeCoS compiler infrastructure IR [19] with ALMA relevant extensions, mainly in the direction to support the comment-like source code annotations that guide the parallelization process. The ALMA IR is a CDFG. Besides IR production, this phase also performs sequential code optimizations before passing control to the the parallelism extraction phase.

Following the front-end, the parallelism extraction phase occurs, which is an iterative process involving coarse and fine-grain parallelism extraction, in order to alternate focus between local and global program view. The fine-grain parallelism extraction is mainly loop-oriented optimization while the coarse-grain parallelism extraction treats fine-grain blocks as black boxes and optimizes the entire CDFG. The two phases operate directly on the ALMA IR and communicate by special annotations on the IR nodes and edges according to the parallel code representation. The parallelism extraction phase, besides estimations based on ADL information, is supported by selective simulation of IR subgraphs. The result of the parallelism extraction phase is an IR partitioned in a manner

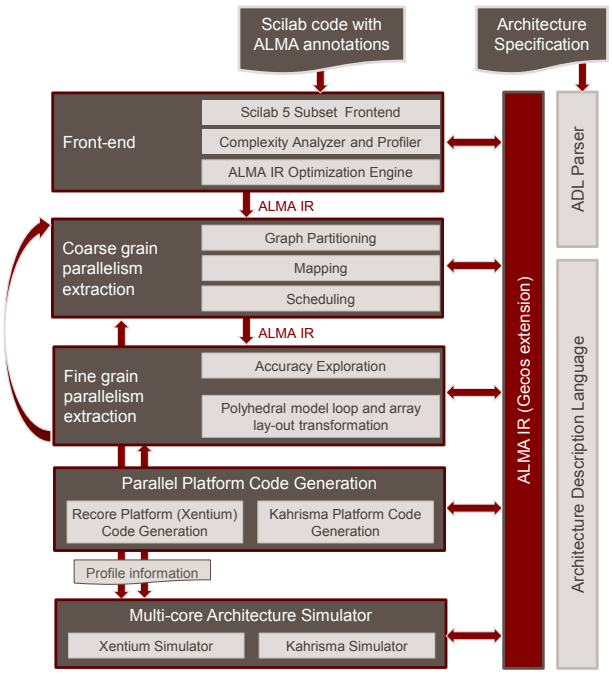


Fig. 3. ALMA Toolset Overview

suitable to be mapped to the available processors. Although this phase is analogous to a compiler middle-end phase, the parallelism extraction is guided by the abstract architecture description provided by the ADL. The parallel platform code generation is responsible to transform the IR to executables for the desired embedded hardware, based on the ADL description. The CDFG nodes are transformed to C functions while the edges may be transformed to communication primitives, when data communications occur. The generated C code is compiled, now using the specific platform compiler toolchain, in order to implement architecture-specific optimizations.

The parallelism extraction steps use the multicore architecture simulator on specific IR fragments to get accurate timing information for the generated code. The multicore architecture simulator gets input executables as *Executable and Linkage Format* (ELF) [20] binaries and is able to start different binaries on different cores. In addition, the multicore architecture simulator leverages on ADL descriptions to specify the details for the simulated processor, and provides SystemC modules for *Instruction Set Simulators* (ISS) including *Cycle-Accurate Simulators* (CAS) for the Recore Xentium and Kahrisma architectures. The path from the parallelism extraction to the simulation is very expensive, since it involves the creation of binaries and in effect, all parallel code generation steps. As a result, the parallelism extraction phase has to make predictions for code performance based on the ADL and be able to amend predictions with more accurate simulation results. In addition, since simulator and parallel code generation involve startup latencies, the parallel code generation aggregates various possible scenaria for examination into a single parallel program, in order to maximize the efficiency.

## V. Algorithms for the Front End

### A. Scilab Front-End

*Scilab Front-End* (SAFE) accepts Scilab input and produces C equivalent code. Rather than working as an interpreter, the SAFE is a compiler to facilitate compile time optimization and analyses. To bridge the gap between the usage of untyped vector data structures of Scilab language with the typed scalar data structures of the C language, SAFE accepts an additional declarative section before the Scilab source code, distinguishing declarative and Scilab source code with a special delimiter "//%%".

In order to avoid the definition of a new specialized declarative language for the Scilab data structures, SAFE leverages the C language rich in type expressiveness declarative syntax. Several additional semantics had to be incorporated in SAFE declarative syntax on top of C syntax, to adapt with the Scilab semantics. Scilab functions may accept multiple input parameters and return multiple results while C supports only one return parameter and from this, arrays are excluded, while pointers are allowed. To comply with this requirement, the Scilab declarative language adds two additional type qualifiers "in" and "out" to state input and output parameters.

Scilab language has a single namespace and only two types of scopes, global and function scope. To declare a variable into a function scope the function name should explicitly qualify the variable name using the resolution operator "::", while the absence of the resolution operator implicitly refers to the global scope.

The code section accepts Scilab code compliant 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. This language will be semi-transparent to SAFE, with its expressions enclosed to Scilab comments, and the information will be seamlessly attached to a specific Scilab segment with particular interest for the subsequent synthesis and parallelization stages.

### B. aprof (ALMA profiler)

The ALMA Scilab front-end includes a *High-Level Intermediate Represepntation* profiler, named *aprof* (ALMA profiler). Its purpose is to compute lower and upper performance bounds in order 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 basic steps involved in aprof are shown in Figure 4. aprof accepts Scilab *High-Level IR* (HLIR) as its input, produced by *SAFE*. Supporting Scilab matrix operators and library functions [2] as C builtins is required, given that these are not decomposed to sequential code at the ALMA IR level.

The HLIR-to-NAC generator produces code for the internal NAC IR. NAC operations specify $n$ ordered inputs producing $m$ ordered outputs. Declared objects are either a "globalvar" (global scalar or array), "localvar" (a local), "in", or "out" (input or output procedure argument). The memory access

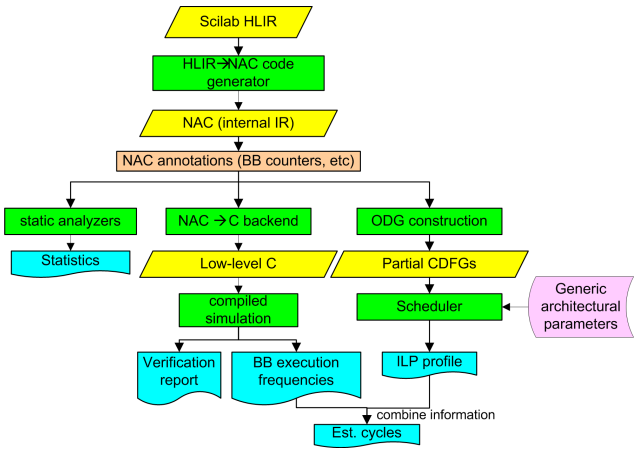


Fig. 4. ALMA Profiler (aprof) Overview

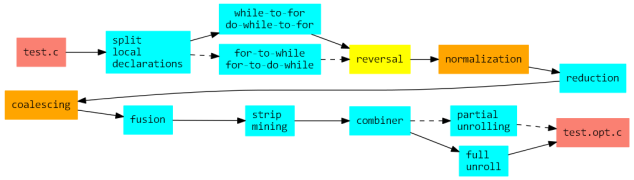


Fig. 5. Possible Optimization Flow Using TXL Passes

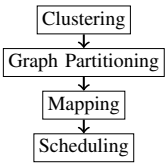


Fig. 6. Coarse-Grain Iteration Steps

model defines dedicated address spaces per array, so that both loads and stores require an explicit array identifier, while procedures are implemented as non-atomic operations.

At this stage, annotations can be placed as NAC operations for tracking events such as entering basic blocks. Then, static and dynamic analyses are ready to proceed: static analyzers are used for evaluating the static instruction mix and data type coverage for the application. aprof extracts *Operation Dependence Graphs* (ODG) in *Static Single Assignment* (SSA) [22] for each task/function of the application for estimating performance. A support form called pseudo-SSA does not need $\phi$-statement insertion since it applies intrablock variable renaming and out-of-SSA conversion. The schedulers account for generic architectural parameters to model a range of machines: a) a sequential machine, b) maximum intra-block parallelism (ASAP scheduling), c) ideal block, and d) ideal thread processor. The later two options investigate interblock parallelism, by identifying mutually independent basic blocks or functions in a given *Control Flow Graph* (CFG) or call graph, respectively. Reduced acyclic forms of these graphs must be examined, e.g. their spanning trees. Realistic constraints such as communication and synchronization effects are not modeled at this point.

Dynamic performance analysis uses a compiled simulator accepting back-translated C as its input. Host execution is used to collect basic block execution frequencies and extract the dynamic instruction mix. Then, these results are combined with the ILP profile produced by the scheduler to obtain an estimate on abstract machine cycles.

### C. High-Level Optimizer (ALMA HLO)

The ALMA *High-Level Optimizer* (HLO) is a collection of autonomous source-to-source code transformation passes that operate on ALMA HLIR for applying machine-independent optimizations. HLO is being 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, syntactic conversions among iteration schemes, and combining independent scalar statements. Loop-specific optimizations include enabling transformations such as loop normalization, reversal, extension, reduction, and fusion for assisting coalescing, strip mining, partial/full loop unrolling. Figure 5 summarizes a possible optimization flow using passes developed in TXL.

Interesting arithmetic optimizations include strength reduction of constant multiplications and divisions. HLO uses Bernstein's algorithm [24], [25] for replacing constant multiplications by a series of additions, subtractions and shifts. Integer constant division is optimized according to the multiplicative inverse technique [26] with the result adjusted by a series of compensation steps. A TXL transformation is used for introducing function calls to specialized constant multiplication routines are introduced. These routines are then generated by point tools (such as kdiv [27]), concatenated to the ALMA HLIR translation unit and finally inlined.

## VI. Coarse-Grain Parallelism Extraction

*Coarse-Grain Parallelism Extraction* (CGPE) 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 described later. CGPE is an iterative process of a sequence of internal steps, presented in Figure 6, 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.

The *clustering* step combines basic blocks into composite blocks. When basic blocks are not distant in the control flow and exhibit data dependencies, optimization steps that follow would map them to the same processing core in order to avoid the communication costs. This process is performed iteratively, with new composite blocks regarded like basic blocks, until

the heuristic can no longer group blocks together or composite blocks are becoming significantly large. A balance needs to be maintained in the clustering step, otherwise significantly large composite blocks can limit optimization options in later steps. While the sequence of blocks that form a composite block is going to be executed in a single core, the same composite block might be executed by multiple cores, i.e. when the composite block is part of the body of a parallelizable loop.

During the *graph partitioning* process, independent partitions of the CDFG, as it is modified by clustering, are generated. The graph partitioning step produces larger clusters of composite and basic blocks that exhibit minimal dependencies between them. Each graph partition is a *task* that is going to be executed by a single core, has specific dependencies and outputs, while it may share composite blocks with another graph partition. The requirement for graph partitioning is that each original CDFG node is covered at least one time, not exactly once. Blocks that participate to more than one graph partitions will be replicated to the relevant core programs when the IR is exported to platform code during the platform code generation phase.

After graph partitioning, *mapping* assigns tasks to processing cores for execution. In the context of the ALMA project, only homogeneous multicore architectures are addressed, which simplifies the mapping process. The outcome of the mapping process is complete per core intermediate program representations.

As graph partitions are assigned to cores, the *scheduling* step reorders instructions inside each graph partition in order to improve performance. The scheduling problem is modeled as a project scheduling problem with multiple workers. For each composite block, earliest and latest deadlines for begin and end are defined and each composite block must be scheduled to exactly one core. Several different optimization goals can be defined, as the smallest critical path, meaning fastest execution, or smallest average workload imbalance between worker cores, to produce a more stable schedule. In addition, when different schedules are identified as optimal depending on input parameters (i.e. input size), all static schedules are generated with a conditional on input parameters.

While the above flow is able to produce performant parallel code, each time an optimization process is split into steps, a bias towards specific solution characteristics is introduced. Therefore, it is an important exercise to employ common combinatorial optimization methods on the coarse-grain parallelism extraction problem. *Mixed Integer Programming* (MIP) modeling and solution, when it can describe the problem in its entirety, can provide solutions to optimality, but the combinatorial complexity imposes limits to the programs that can be parallelized. In fact, MIP has been used before for automatic parallelization of embedded software [28]. Single path metaheuristic methods like simulated annealing [29], tabu search [30], great deluge [31], and late acceptance hill climbing [32] provide stochastic decision methods to navigate the multidimensional search space towards a good solution. Population-based metaheuristic approaches, typically referred as evolutionary algorithms, like the genetic algorithm [33] variants and scatter search [34], maintain a population of problem solutions. The use of single-path and population-based metaheuristic methods for the task planing problem for heterogeneous multiprocessors is presented in [35]. Beg presents a heuristic for graph partitioning the data dependency graph in order to assign computational workload on cores of a multicore system, with the main feature to identify the critical path of the code [36]. Ferrandi et al. propose an ant colony optimization to optimize the hierarchical task graphs for MPSoC parallel applications [37]. Tournavitis et al. identify that traditional target-specific mapping heuristics are inflexible and propose a machine-learning based prediction mechanism for the mapping decisions [38].

## VII. 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 [39] including the Xentium 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 (multiplier, adder, shift) of word-length $W$ having SWP capabilities can be decomposed into $P$ operations in parallel on sub-words of $W/P$ length (e.g. a 64-bit SWP adder can execute 2 32-bit, 4 16-bit, or 8 8-bit additions in parallel). In [40], this technique has been used to implement a *Code Division Multiple Access* (CDMA) synchronisation loop in the TigerSharc *Digital Signal Processor* (DSP). The SWP capacities allow to achieve 6.6 *Multiply-and-Accumulate* (MAC) operations per cycle with two MAC units.

When moving from the initial floating-point Scilab implementation, the designer has the opportunity to choose shorter fixed-point data encodings to be able to benefit from SIMD extensions, provided that the software code exhibits enough fine-grain parallelism. This then comes at the cost of a loss of numerical accuracy in the computation due to quantization noise [41] and results in a complex performance/accuracy trade-off optimization problem that ALMA's toolflow will automate [42]. 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, that is *Static Control Parts* (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 are specified using intrinsics.

### 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. 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 $\mathbf{wl}$ be the word-length of all operations in the program. The optimization problem can then be modeled as

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

Our proposed methodology selects the set of SIMD instructions that respects the global accuracy constraint $P_{b_{max}}$ and minimizes the implementation cost. Such optimization requires on the 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 considered. This penalty can be obtained from the IR 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. Traditional methods [43][44] are based on fixed-point simulations. But these techniques lead to huge fixed-point optimization time. Thus, approaches based on analytical models will be favored over simulations to reach reasonable optimization time. The approach proposed in [45] determines automatically the analytical expression of the quantization noise power from the CDFG of the application. The proposed method is valid for systems made-up of smooth operations. An operation is considered to be smooth if the output is a continuous and differentiable function of its inputs, as in the case of arithmetic operations. In a fixed-point system, the output quantization noise is due to the propagation of the different noise sources generated during cast operations. The output noise power expression is computed from the noise source statistical parameters and the gains between the output and each noise source. These gains are impulse response between the output and the considered noise source, and can be time-varying.

An important aspect of the problem is that the number of data types remains very limited as they have to correspond to types supported by the processor (byte, word, double word), which therefore reduces the optimization search space. The data word-length optimization is achieved in two stages. First, an initial solution is obtained with a greedy algorithm using a steepest descent criteria to select at each iteration the best direction. This heuristic quickly provides an initial optimized solution, but does not always select the optimal solution. Thus, this optimized solution is refined with a *branch & bound* algorithm integrating different techniques to cut drastically the search space.

### 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 [46]. In addition to the control flow, we also explore complex array layout transformations to reduce the program memory footprint and also to limit as much as possible the need for unaligned memory access and vector packing/unpacking instructions [47].

This loop and array layout transformation framework is 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.

## VIII. Conclusion

In this paper, the automated parallelization methodology and algorithms of the ALMA project, from Scilab source code to embedded multicore system on chip platforms, are presented. The parallelization approach attempts to alternate focus between local and global program scope in order to iterarively progress towards an efficient parallel implementation of the input source. The main phases are the fine-grain parallelism extraction for the local optimizations and the coarse-grain parallelism extraction for the global scope. Fine-grain parallelism extraction focuses on loops, while the coarse-grain parallelism extraction considers the full source control and dependence graph. The methodologies described are going to be implemented within the ALMA toolset in the context of the ALMA project. The target architectures are KIT's Kahrisma and Recore Systems' Xentium processors, while the application drivers are from the telecommunication industry and image processing domains. For the future, new embedded multicore hardware architectures will be considered as ALMA targets, in order to further validate and adapt the parallelization methodology presented here.

## References


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

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

[3] T. Stripf, M. Huebner, J. Becker, G. Rauwerda, K. Sunesen, G. Goulas, P. Alefragis, N. S. Voros, D. Goehringer, S. Derrien, D. Ménard, O. Sentieys, N. Kavvadias, K. Masselos, and D. Kritharidis, "From Scilab To High Performance Embedded Multicore Systems — The ALMA Approach," in *Euromicro Conference on Digital System Design (DSD 2012)*, September 5th-8th, Cesme, Izmir, Turkey, 2012.

[4] A. Prasad, J. Anantpur, and R. Govindarajan, "Automatic compilation of matlab programs for synergistic execution on heterogeneous processors," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 152–163. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993517

[5] J. Cardoso and M. Hübner, *Reconfigurable Computing: From FPGAs to Hardware/Software Codesign*. Springer, 2011.

[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. 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.

[10] ——, "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.

[11] ——, "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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

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

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

[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, pp. 17–20, Apr. 1998.

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

[24] R. Bernstein, "Multiplication by integer constants," *Software— Practice and Experience*, pp. 641–652, Jul. 1986.

[25] P. Briggs and T. Harvey, "Multiplication by integer constants," Rice University, Technical report, Jul. 1994.

[26] H. S. Warren, *Hacker's Delight*. Addison-Wesley, Jul. 2002.

[27] "kdiv: Constant division routine generator," http://sourceforge.net/projects/kdiv/.

[28] D. Cordes, P. Marwedel, and A. Mallik, "Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES/ISSS '10. New York, NY, USA: ACM, 2010, pp. 267–276.

[29] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[30] F. Glover, "Tabu search — part I," *ORSA Journal on Computing*, vol. 1, no. 3, pp. 190–206, 1989.

[31] G. Dueck, "New optimization heuristics: The Great Deluge Algorithm and Record-to-Record Travel," *Journal of Computational Physics*, vol. 104, no. 1, pp. 86–92, Jan. 1993.

[32] E. Özcan, Y. Bykov, M. Birben, and E. K. Burke, "Examination timetabling using late acceptance hyper-heuristics," in *Proceedings of the Eleventh conference on Congress on Evolutionary Computation*, ser. CEC'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 997–1004.

[33] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[34] F. Glover, M. Laguna, and R. Martí, "Fundamentals of scatter search and path relinking," *Control and Cybernetics*, vol. 39, pp. 653–684, 2000.

[35] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, Jun. 2001.

[36] M. Beg, "Critical path heuristic for automatic parallelization," University of Waterloo, David R. Cheriton School of Computer Science, Technical Report CS-2008-16, Aug. 2008.

[37] F. Ferrandi, C. Pilato, D. Sciuto, and A. Tumeo, "Mapping and scheduling of parallel C applications with ant colony optimization onto heterogeneous reconfigurable MPSoCs," in *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ser. ASPDAC '10. Piscataway, NJ, USA: IEEE Press, 2010, pp. 799–804.

[38] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 177–187.

[39] A. Iranpour and K. Kuchcinski, "Evaluation of SIMD architecture enhancement in embedded processors for MPEG-4," in *Euromicro Symposium on Digital System Design (DSD 2004)*, 2004, pp. 262 – 269.

[40] D. Esftathiou, J. Fridman, and Z. Zvonar, "Recent developpements in enabling technologies for the software-defined radio," *IEEE Communication Magazine*, vol. 37, no. 8, pp. 112–117, august 1999.

[41] 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.

[42] 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.

[43] M. Coors, H. Keding, O. Luthje, and H. Meyr, "Fast Bit-True Simulation," in *Proc. ACM/IEEE Design Automation Conference (DAC)*, Las Vegas, june 2001, pp. 708–713.

[44] S. Kim, K. Kum, and S. Wonyong, "Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs," *IEEE Transactions on Circuits and Systems II - Analog and Digital Signal Processing*, vol. 45, no. 11, pp. 1455–1464, november 1998.

[45] R. Rocher, D. Menard, P. Scalart, and O. Sentieys, "Analytical accuracy evaluation of Fixed-Point Systems," in *Proc. European Signal Processing Conference (EUSIPCO)*, Poznan, September 2007.

[46] 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.

[47] 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.