Partitioning and Mapping for the hArtes European Project

F. Ferrandi L. Fossati M. Lattuada G. Palermo D. Sciuto A. Tumeo
Dipartimento di Elettronica e Informazione, Politecnico di Milano, Via Ponzio, 34/5 20133 Milano, Italy
{ferrandi,fossati,lattuada,gpalermo,sciuto,tumeo}@elet.polimi.it

Abstract

The hArtes - Holistic Approach to Reconfigurale real Time Embedded Systems - project has three main objectives: the development of a toolchain and a methodology supporting effective automatic or semi-automatic design of complex heterogeneous embedded systems, the design of a scalable heterogeneous and reconfigurable hardware platform and the validation of the tool chain on a set of innovative applications in the audio and video field.

This paper presents the ongoing works related to hArtes at Politecnico di Milano. Our role consists in the development of innovative methodologies and algorithms for software partitioning and for initial mapping of the resulting partitions on reconfigurable multiprocessor platforms. The development of these methodologies was integrated in PandA, our framework for hardware-software codesing; several other related were developed as an aid for the testing of the implemented technologies.

1. Introduction

The hArtes integrated project aims at laying the foundation for a new holistic (end-to-end) approach for complex real-time embedded system design, with the latest algorithm exploration tools and reconfigurable hardware technologies. The proposed approach addresses, for the first time, optimal and rapid design of embedded systems starting from high-level descriptions. hArtes will develop modular and scalable hardware platforms that can be reused and re-targeted by the tool chain to produce optimized real-time embedded products. The results will be evaluated using advanced audio and visual systems that support next -generation communication and entertainment facilities, such as immersive audio and mobile visual processing. Innovations of the hArtes approach include: (a) support for both diagrammatic and textual formats in algorithm description and exploration, (b) the implementation of a framework that contains novel algorithms for design space exploration, aiming at the automation of design partitioning, task transformation, choice of data representation, and metric evaluation for both hardware and software components, (c) a system synthesis tool producing near-optimal implementations that best exploit the capability of each type of processing element; for instance, dynamic reconfigurability of hardware can be exploited to support functionality upgrade or adaptation to operating conditions. From the application point of view, the complexity of future mobile devices is becoming too big to design monolithic processing platforms. This is where the hArtes approach with reconfigurable heterogeneous systems becomes vital.

The role of Politecnico di Milano in the hArtes project consists in the development of the mechanisms devoted to the automatic design partitioning and to the initial mapping of any application on the different processing elements on the hArtes target platform. This will be realized by exploiting and by further developing the PandA framework. The PandA framework is a toolchain designed to study different aspects of hardware-software codesign and, among other things, it already offers initial support for program parallelization targeted to specific hardware platforms and initial support for high level synthesis.

The remainder of this paper is organized as follows. Section 2 introduces some relevant past approaches in the area that hArtes project tries to address and some prominent works in the specific aspects that PoliMi focuses on. Section 3 illustrates the hArtes toolchain flow and where PoliMi solutions are positioned. Section 4 gives some details on the current developments of PoliMi's PandA framework for partitioning and initial mapping of C applications. Finally, Section 5 concludes the paper.

2. Related works

There already exists a set of approaches which aim at developing hardware/software co-design tools that also partially address some of the issues relevant in this research.

Some prominent examples from academia are the COSMOS tool [10] from TIMA laboratory, SpecC [6] from the UC Irvine, or the Ptolemy environment [5] and the Polis/Metropolis [3] framework from the UC Berkeley. There are also some commercial products, such as CoWare's ConvergenSC, N2C [1], and the meanwhile discontinued VCC environment from Cadence. These approaches reveal significant drawbacks since they mainly restrict the design exploration to predefined library-based components and focus on simulation and manual refinement.

PoliMi work in hArtes relates to the thread partitioning of the starting specification and to its initial mapping on the processing elements of the multiprocessor platform. This process can be decomposed in three main steps. First, the parsing of the initial specification to an intermediate graph representation used to explore the available parallelism is performed. Second, the partitioning of the intermediate graph representation into a task graph is executed. A task graph is a Directed Acyclic Graph (DAG) where each node describes a potentially concurrent code block. Third, the tasks are allocated on the available processors. This allocation is usually realized through a two-step process: clustering and cluster-scheduling (merging).

Most research works, dealing with partitioning of the initial specification, adopt specific intermediate representations. Among them, Girkar et al. [7] propose an intermediate representation, called Hierarchical Task Graph (HTG), which encapsulates minimal data and control dependences that can be used for extraction of task level parallelism. Most of their work focuses on simplification of the conditions for execution of task nodes. Luis et al. [13] extend this work by using a Petri net model to represent parallel code, and they apply some optimization techniques to minimize the overhead due to explicit synchronization.

Newburn and Shen [14], instead, present a complete flow for automatic parallelization through the PEDIGREE compiler. Their tool uses the Program Dependence Graph (PDG) as intermediate representation and applies a heuristic to create overlapping inter-dependent threads. Their approach searches the PDG for control equivalent regions (i.e., groups of statements depending from the same control conditions) and then partition these regions with a bottom up analysis. The resulting task graph is finally scheduled on subsets of processors of a shared memory multiprocessor architecture.

The clustering and merging phases have been widely discussed. Usually, these two phases are addressed separately. Well known deterministic clustering algorithms are dominant sequence clustering (DSC) by Yang and Gerasoulis [18], linear clustering by Kim and Browne [12] and Sarkar's internalization algorithm (SIA) [15]. On the other hand, many researches explore the cluster-scheduling problem with evolutionary algorithms [9, 17]. A unified view is given by Kianzad and Bhattacharyya [11], who modify some of the deterministic clustering approaches by introducing probability in the choice of elements for the clusters; they also propose an alternative single step evolutionary approach for both the clustering and cluster scheduling aspects.

Our approach starts from an intermediate representation which is not hierarchical like HTGs and PDGs, but instead flattens out all the dependence information at the same level. This produces larger structures but gives the opportunity to extract more parallelism as it allows more sophisticated explorations.

3. Project Requirements

hArtes target hardware platforms will be solutions composed by multiple DIOPSIS D940HF chips (each with an ARM9 processor and a mAgicV DSP) that can be joined together with multiple FPGAs containing a PowerPC hard core. This will offer very high computational power for audio and video applications.

Figure 1 illustrates the hArtes toolchain flow. The flow is separated in several stages, and most of them can be cyclic. The flow starts with annotated C code. The specification, after traversing a C2C merger that may add other annotations obtained in previous passes, is profiled. The profiled code is then partitioned in tasks taking into consideration the performance data just obtained. The resulting partitioned code, is further annotated in order to express an initial guess on the mapping of each task on the processing elements of the target platform. Each task can then be transformed to optimize it for the specific processing elements on which it has been mapped. To reduce the amount of hardware required for an operation, the number of bits used to represent data needs to be minimized. This goal is addressed by the Data Representation optimization stage. All these stages provide new information and can be repeated several times to optimize the resulting code. Each task is finally committed to each processing element before code generation. Code generation is performed by a specific back end for each target unit. The ELF objects for the software parts are merged to generate the executable code, while high level synthesis is performed and synthesizable VHDL is generated for the FPGA part.

The target platform supports a *Fork/Join*, non preemptive threading model, where a master processor spawns multiple software and hardware threads on the various processing elements (or on the FPGA) and retakes control after all of them terminate. This model fits well with the widely adopted OpenMP [16] standard; OpenMP is supported by many commercial compilers (Microsoft and IBM) and, in the OpenSource scene, by GCC, starting with the 4.2 release. The reasons behind the wide diffusion of OpenMP lie in the fact that it has a very powerful and complete syntax to express complex models of parallelism (e.g. for loops parallelism), but, at the same time, it remains simple and effective. A subset of the OpenMP pragmas is used in the context

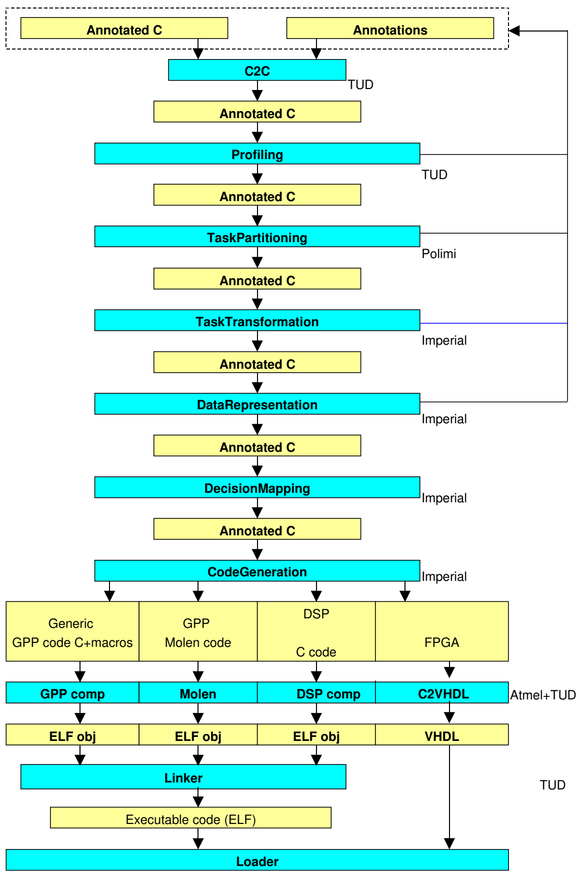


Figure 1. The hArtes toolchain

of the hArtes project: `#pragma omp parallel` is used to express parts of code that potentially runs in parallel and, inside it, the `#pragma omp sections` declares the single parallel parts. Each `#pragma omp parallel` block acts as a fork and it implicitly joins the spawned threads at its end. It is interesting to note that nested `#pragma omp parallel` are allowed, giving the possibility to support fork from children threads.

The annotations for the initial guesses on the mapping of the tasks extracted by the partitioner will instead adopt an ad-hoc, indepentent syntax. Using an independent syntax for mapping makes sense since it is a different problem from thread decomposition and it is tightly coupled with the target platform. Moreover, the parallel code produced by our tools could be in this way tested on different hosts that support OpenMP, simply ignoring the mapping directives. The initial task mapping will be implemented through an evolutionary approach guided by the high-level metrics obtained during the profiling and partitioning phase.

Figure 1 that PoliMi's work is right in the heart of the flow, where the partitioning in tasks of the input specification is performed. The partitioning and the initial guesses of mapping will be realised by exploiting and by extending our PandA framework in order to support the outlined requirements.

4. Toolchain Description

The PandA tool [4] takes an application, written in C, as input and produces a parallel version of the application as output. This tool behaves as a compiler in that it is composed of the *frontend*, which creates the intermediate representation of the input code, the *middle-end*, an internal part which manipulates the intermediate representation, and the *backend*, which prints executable C code annotated with OpenMP [16] and mapping directives. The remaining of this section is devoted to the description of the three parts composing the PandA tool.

4.1 Frontend

The PandA frontend does not directly read C source code, but it parses its Gimple-compliant tree-representation produced by a slightly modified version of GNU GCC compiler [2]. In this way it is possible to use the code optimizations performed by this compiler, eliminating the need to reimplement them in PandA. Note that only the target-independent optimizations, such as constant folding, constant propagation or dead code elimination, are chosen. In addition to the code optimizations, GCC also converts each original C instruction in one or more basic operations and it substitutes the loop control statements `while`, `do while` and `for` with constructs of type `if` and `goto`.

After parsing the Gimple-compliant tree-representation a Control Flow Graph (CFG) is created for each C function present in the source code. Every CFG node represents zero, one or more C basic operations. During graph construction a first analysis of the specification is performed and its results are annotated onto the graph. Information computed by this analysis is basically the type of each operation and which variables are read/written by each of them. Moreover at the end of this phase, PandA analyzes the produced Control Flow Graph to identify the edges which close a cycle in a path starting from the entry node[1]. This analysis and the annotations allow PandA to identify and rebuild cycles in the following phases.

4.2 Manipulation of the Intermediate Representation

As in traditional compilers, this phase could be considered the main part of the compilation flow implemented in

[1]entry node is a symbolic node which represents the begining of the computation flow in the specification

PandA. This phase is mainly composed of two steps: the Dependence Analysis and the Parallelism Extraction.

4.2.1 Dependence Analysis

This step consists of the analysis of the Control Flow Graph and of the tree in order to compute all the dependences between each pair of operations (nodes). Dependences between an operation A and an operation B are basically of three types:

Control Dependence : execution of operation B depends on the result of operation A or operation B has to be executed after operation A

Data Dependence operation B uses a variable which is defined by operation A

Anti-Dependence operation B writes a variable which is previously read by operation A (these dependences could be avoided by using the Static Single Assignment (SSA) technique)

All the identified dependences are represented into two different graphs (with or without feedback[2] edges). These graphs are produced for each C function of the original C specification. To extract as much parallelism as possible this dependence analysis must be very precise. A false dependence added between two operations indicates that they cannot be concurrently executed, so it eliminates the possibility of extracting parallelism among those instructions. On the other hand, all the true dependences have to be discovered otherwise the concurrent code produced by PandA could have a different functionality from the original one. The data dependence analysis is not based only on variables present in the original specification: a different variable is created for each field of a record and for each element of an array. This method significantly reduces the number of false positive data dependences.

Before computing data dependences and anti-dependences, alias analysis [8] has to be performed: this is necessary to correctly deal with pointers. An interprocedural alias analysis model is used. In this way an analysis less conservative than those produced by intraprocedural methods is obtained even if this costs in terms of computation time.

In addition to the analysis on the original specification, this phase of the PandA flow also partially manipulates the intermediate representation. For example PandA applies further dead code elimination made possible by alias analysis and by loop transformations.

[2]Feedback edges are the edges connecting operations A and B such that A and B are part of the same loop and B depends on the result produced by A in the previous iteration of the loop.

All the created graphs, together with the starting tree produced by GCC, are used as intermediate representation by the PandA tool.

4.2.2 Parallelism extraction

This phase aims at the division of the created graphs into subsets, trying to minimize the depencences among them; hence this phase is often identified as the *partitioning* phase.

The first step consists in the analysis of feedback edges in order to identify the loops and separate each of them from the other nodes of the graph; from now on, the partitioning steps will separately work on each of the identified subgraphs: parallelism can be extracted either inside a loop or by considering the nodes not part of any loop.

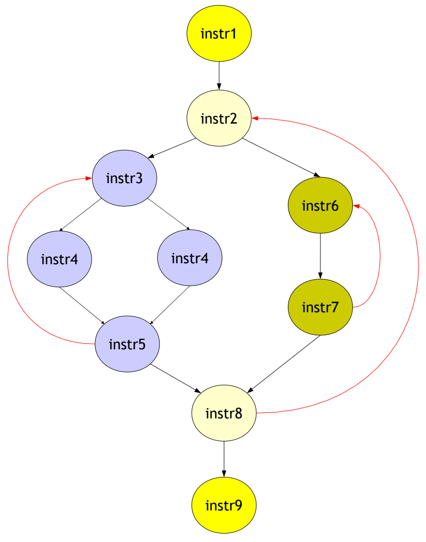


Figure 2. Nodes belonging to different loops are clustered in different subgraphs.

After computing the loop subgraphs, the core partitioning algorithm is executed. In a similar way to what performed in [14] the algorithm starts by examining the control edges to identify the *control-equivalent* regions: each of them groups together the nodes descending from the same branch condition (True of False) of predicated nodes; nodes representing `switch` statements are treated in a similar way. Data dependences and anti-dependences among the nodes inside each control-equivalent region are now analyzed to discover intra-dependent subgraphs: all the elements inside such subgraphs must be serially executed with respect to each other.

Each obtained "partition" (subgraph) represents a single block of instructions, with none, or minimal interdependence. Partitions that do not depend on each other contain blocks of code that can potentially execute in parallel.

Edges among partitions express data dependences among blocks of code, thus the data represented by in-edges of a partition must be ready before the code in that partition can start. Note that the identified partitions are just a first approximation of the tasks into which the input program is being divided.

Since OpenMP [16] is used to annotate the parallelism in the produced C specification, transformations to the task graph are necessary in order to make it suitable for the OpenMP programming model, the *Fork/Join* programming model (described in Section 3).

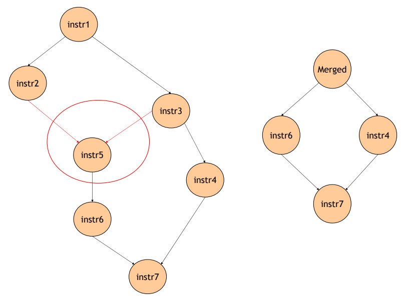


Figure 3. A task graph not compliant with the fork/join programming model (on the left) and one compliant.

The algorithm which transforms a generic task graph into one compliant with the fork/join programming model is composed of 3 main phases, which are iterated until the whole graph is compliant:

1. identification of the non fork/join compliances; they are always discovered in correspondence of *join* tasks (node 5 in the example)

2. computation of the nodes (from now on it will be used as a synonym for tasks) which need to be merged together to eliminate the non compliance just identified (in the example nodes 1, 2, 3 and 5)

3. reconstruction of the corrected task graph

Experiments show that the tasks, created with the presented algorithm, are usually composed of a limited number of instructions: on most systems (even the ones containing a lightweight operating systems) the overhead due to the management (creation, destruction, synchronization) of these small tasks could be higher than the advantages given by concurrent execution. The **optimization** phase tries to solve this problem by grouping tasks together. Two different techniques are used: optimizations based on control dependences and optimizations based on data dependences.

Control statements, such as `if` clauses, must be executed before the code situated beneath them, otherwise we would have speculation: it seems, then, reasonable to group together the small tasks containing the instructions which depend on the same control statement. Another optimization consists of grouping the `then` and `else` clauses in the same cluster: they are mutually exclusive, the parallelism is not increased if they are in separate clusters.

Data dependent tasks can be joined together when their weight is smaller than a predetermined number n. Those tasks, which are also neighbor of the same *fork* node, are joined together; if all the tasks of the fork group are to be joined together, then the whole fork group disappears and all its nodes (including the fork and join ones) are collapsed in the same task.

In order to complete the partitioning flow, as implemented in PandA, we need to produce the parallel executable C program representing the task graph.

4.3 Backend

OpenMP [16] is a library exposing APIs for the development of shared memory multi-threaded applications; its main strength lies in the fact that preprocessor directives are used to indicate parallel regions, thus even not OpenMP enabled compilers can still compile these applications.

As described in the previous paragraph, in order to be able to use this library, the task graph must strictly adhere to the *fork/join* programming model; this feature is used by the backend during the production of the concurrent C code. The operations composing this step are:

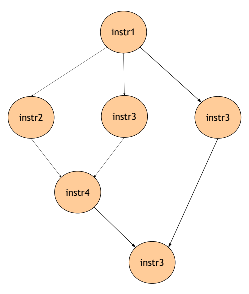
1. identification of all the fork/join groups; note that different groups may share the fork node (Figure 4(a)) or the join node (Figure 4(b)).

2. print of the C code; loops are represented through combinations of `if` and `goto` instructions.

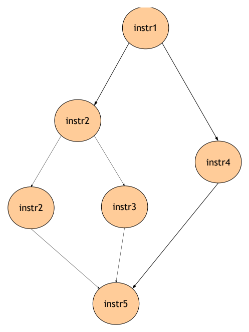
4.4 Initial guesses of mapping

Different approaches can be defined for mapping the application onto the target platform. The PoliMi approach for initial guesses on mapping is based on both standard statistical and more advanced data mining techniques aimed at: (i) developing the final partitioning and (ii) acquiring information regarding the system's critical areas for the given class of reconfigurable embedded systems.

As usually done in Data Mining applications, the model is viewed as a simple black box with an output C representing the set of cost functions of the input I representing a certain mapping and partitioning. Starting from the



(a) Fork node sharing



(b) Join node sharing

set of pairs $< Ij, Cj >$, statistical and data mining techniques can extract interesting relationships among inputs, i.e. among elements of partitioning configurations, trying also to extrapolate the system model. Note that, the number of pairs $< Ij, Cj >$ usually contains only a small subset of all the possible input-output configurations. The type of relations extracted depends on the technique used. For instance, some statistical techniques (e.g. linear regression), extract relations represented as linear combinations of the input values; whereas data mining techniques (e.g. regression trees, neural networks, or learning classifier systems) are able to extract highly non-linear relations among input-output configurations.

5. Conclusions and Ongoing Works

The role of Politecnico di Milano in the hArtes project consists in the production of innovative methods for task partitioning and initial mapping of a standard C application on the hArtes target architecture. Task partitioning is performed extracting all the parallelism available and then organizing the code in order to respect the fork/join threading model selected by the project. The resulting parallel code is annotated with OpenMP pragmas. Initial guesses of mapping are obtained through an evolutionary approach driven by high level metrics collected from profiling and partitioning. Ongoing works of the project are related to the optimization of the partitioning phase and to the modeling of the target architecture and algorithm selection for the mapping phase.

References

[1] CoWare Platform Architect, http://www.coware.com/products/platformarchitect.php.

[2] GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[3] Metropolis: Design Environment for Heterogeneous Systems, http://www.gigascale.org/metropolis/index.html.

[4] PandA framework, http://trac.elet.polimi.it/panda.

[5] Ptolemy Environment. http://ptolemy.eecs.berkeley.edu/.

[6] SpecC Reference Compiler. http://www.ics.uci.edu/ specc/reference/.

[7] M. Girkar and C. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, Mar. 1992.

[8] M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, New York, NY, USA, 2001. ACM Press.

[9] E. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5:113–120, 1994.

[10] T. B. Ismail, M. Abid, and A. Jerraya. Cosmos: a codesign approach for communicating systems. In *CODES '94: Proceedings of the 3rd international workshop on Hardware/software co-design*, pages 17–24, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[11] V. Kianzad and S. Bhattacharyya. Efficient techniques for clustering and scheduling onto embedded multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 17(17):667–680, July 2006.

[12] S. Kim and J. Browne. A general approach to mapping of parallel computation upon multiprocessor architectures. In *Int. Conference on Parallel Processing*, pages 1–8, 1988.

[13] J. Luis, C. Carvalho, and J. Delgado. Parallelism extraction in acyclic code. In *Parallel and Distributed Processing, 1996. PDP '96. Proceedings of the Fourth Euromicro Workshop on*, pages 437–447, Braga, Jan. 1996.

[14] C. Newburn and J. Shen. Automatic partitioning of signal processing programs for symmetric multiprocessors. In *PACT '96*, 1996.

[15] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, 1989.

[16] M. Sato. OpenMP: Parallel Programming API for Shared Memory Multiprocessors and On-Chip Multiprocessors. *isss*, 00:109–111, 2002.

[17] W. Wang. Process scheduling using genetic algorithms. In *IEEE Symposium on Parallel and Distributed Processing*, pages 638–641, 1995.

[18] T. Yang and A.Gerasoulis. Dsc: scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5:951–967, 1994.