OPTIMAL FUNCTIONAL MAPPING ONTO END-TO-END RECONFIGURABLE (E^2R) EQUIPMENT HARDWARE PLATFORM

Mirsad Halimic (Panasonic Broadband Communications Development Laboratory, Wokingham, Berkshire, United Kingdom; Mirsad.Halimic@eu.panasonic.com)
Didier Bourse (Motorola Labs, Paris, France; Didier.Bourse@motorola.com)
Eric Nicollet (Thales, Paris, France; Eric.Nicollet@fr.thalesgroup.com)

ABSTRACT

In reconfigurable systems very often a situation will occur where an application, unknown at the design time of equipment, will need to be deployed (integrated) onto hardware platform consisting of several and different processors. This deployment should be optimal or at least suboptimal in terms of speed of processing execution and power consumption. In this paper a process of optimal mapping of a software realisation of an application on E^2R equipment heterogeneous hardware platform is described.

1. INTRODUCTION

In an equipment design time, when an application and hardware platform are known, it is possible to associate application's modules with particular parts of the hardware platform in the best possible way in terms of certain performance metrics such as processing time, transport time, power consumption, etc.

However, in reconfigurable systems very often a situation will occur where an application, unknown at the design time of equipment, will need to be deployed (integrated) onto hardware platform consisting of several and different processors. Requirements for optimal or at least suboptimal mapping in terms of power consumption and performance will still be there. Applications like cellular phones draw their energy from a battery that has a limited amount of energy, and energy-conscious software mapping can lead to drastic reductions of energy dissipation of a whole mobile system. Therefore, there is a need for an entity that is capable of making the best possible integration of an application and its modules with processing fabrics of the available heterogeneous hardware platform.

Conceptually optimal software mapping entity consists of the elements as depicted in Figure 1.

Where:

- Input - System level description language, which is used throughout the whole design process, starting from algorithm design and evaluation to HW and SW implementation. The language has capability of a unified representation of a system that can be used to describe its functionality independent of type of processors employed in the hardware platform.
- Concurrency analyses, which is capable of finding and utilizing the inherent parallelism in various applications,
- Dependency analyses, which maintains the temporal scheduling,
- Profiling, which provides estimated or cycle accurate calculation of performance metrics,
- Mapping that evaluates different mapping solutions until it identifies the optimal one for the given working conditions,
- Output – Optimal scheduler, which presents integrated both temporal and spatial schedulers

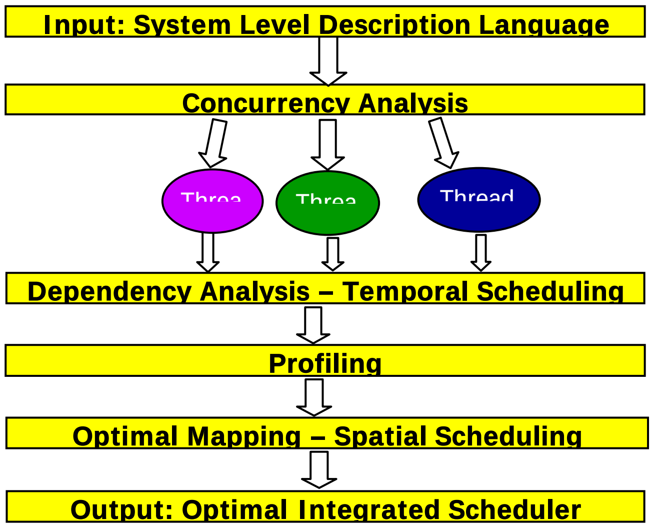

Input: System Level Description Language
Concurrency Analysis
Threa
Threa
Thread
Dependency Analysis – Temporal Scheduling
Profiling
Optimal Mapping – Spatial Scheduling
Output: Optimal Integrated Scheduler

Figure 1: Functional Components of an Optimal Integrated Scheduler

In this paper a process of optimal mapping of a software realisation of an application onto E^2R equipment heterogeneous hardware platform is described. Furthermore, performance investigation of optimisation algorithms utilised for E2R Optimal design mapper (Partitioner) is discussed.

2. E^{2}R RECONFIGURABLE RADIO EQUIPMENT (RRE) ARCHITECTURE

Reconfigurable Radio Equipments will be a key part of future End-to-End Reconfigurable systems [1]. Meeting the challenges imposed by the Seamless Experience vision of E²R will require radio devices with enhanced reconfiguration capabilities, to provide the customers (end-user or network operators) with flexible, modular and evolutive connectivity solutions.

In this work, by "Equipment" reference is made to any concrete equipment that is taking part to the overall E²R system. This can indistinctively refer to User equipment (e.g. flexible mobile phone), or Network equipment (e.g. flexible base station or flexible core network devices).

Thus, "Equipment" is not to be understood as the user terminal, this terminology does encompass base stations and/or access points.

By "Radio Equipment" is meant more specifically equipments capable to provide a radio connectivity (this excludes from the previous list flexible core network devices).

2.1. Three Architecture Areas

The internal architecture of the equipment is divided into a three-tier model, which describes the key architecture areas to be addressed for Reconfigurable Radio Equipment:

- Reconfiguration Management,
- Reconfiguration Control,
- Reconfigurable Elements.

Figure 2 gives a graphical view of such a breakdown into architecture areas.

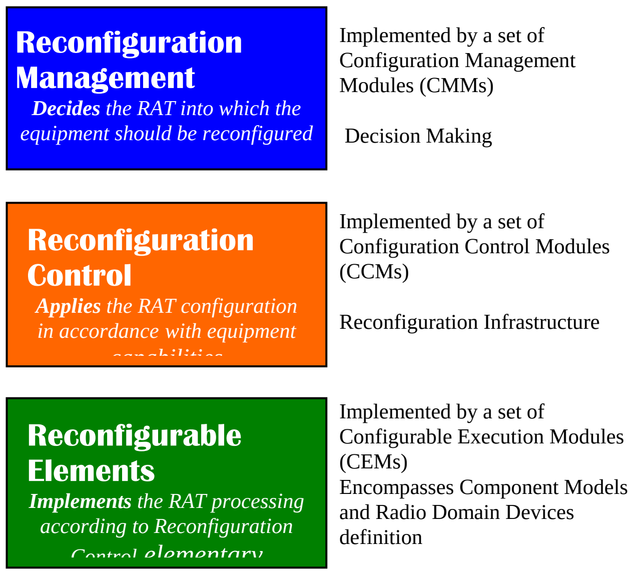

Reconfiguration Management
Decides the RAT into which the equipment should be reconfigured
Implemented by a set of Configuration Management Modules (CMMs)
Decision Making
Reconfiguration Control
Applies the RAT configuration in accordance with equipment
Implemented by a set of Configuration Control Modules (CCMs)
Reconfiguration Infrastructure
Reconfigurable Elements
Implements the RAT processing according to Reconfiguration
Implemented by a set of Configurable Execution Modules (CEMs)
Encompasses Component Models and Radio Domain Devices definition

Figure 2: Aspect of Reconfigurable Radio Equipment Architecture

2.1.1. Reconfiguration Management Definition

Reconfiguration Management covers all the means inside an Equipment, enabling it to contribute to take the appropriate decision concerning the RAT reconfiguration to be applied.

In E^{2}R context, those are the concepts that captured under the general wording of Configuration Management Module (CMM).

2.1.2. Reconfiguration Control Definition

Reconfiguration Control covers all the means inside an Equipment, enabling it to appropriately take advantage of the reconfigurable elements it is composed of.

In E^{2}R context, these concepts were identified as being the Configuration Control Module (CCM).

As depicted in Figure 3 Optimal mapping (Spatial Scheduler) is a part of Configuration Control Module (CCM):

2.1.3. Reconfigurable Elements Definition

Two main categories of Reconfigurable Elements are distinguished:

- Software programmable modules, which are processing units subject to host RAT software modules,
- Parametrical modules, which are reconfigured thanks to parameters adjustments instead of software installation and execution. Those can be of two sub-categories:
  - o Flexible hardware sub-systems (assembled with their driver software),
  - o Flexible software components (autonomous software modules proposing by themselves a certain number of tuneable parameters).

In E^{2}R context, the corresponding concepts were identified as being the Configurable Execution Modules (CEM).

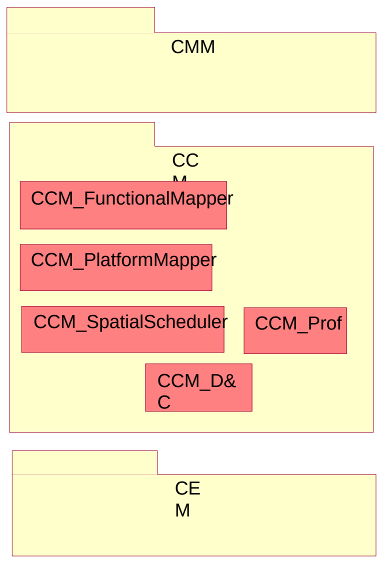


Figure 3: Positioning of Optimal mapping in E^{2}R Architecture

3. OPTIMAL MAPPING ENVIRONMENT

Reconfigurable systems have a hardware platform composed of a network of different architectural fabrics, such as ISA processor, DSP processor, FPGA, Accelerators and ASIC blocks. Instruction set architectures are often related to as soft programmable forms and field programmable logic as hard programmable forms. Therefore, spatial scheduling is frequently termed HW/SW partitioning and an entity performing partitioning is termed the Partitioner.

The Partitioner environment utilized in this work consists of three different commercially available development platforms representing a processor, DSP and FPGA. These platforms are briefly described below.

3.1. Tigersharc / VisualDSP++

The Tigersharc processor is a static superscalar DSP chip that supports 1-, 8-, 16-, and 32-bit fixed- and floating-point data types on the same chip. It is available at speeds up to 600MHz and with up to 24Mb of DRAM integrated.

VisualDSP++ is the integrated development environment from ADI. The environment contains an optimizing C/C++ compiler and a cycle-accurate simulator for the Tigersharc chip.

3.2. ARM / Armulator

The ARM is a processor core design that is available as IP from ARM Ltd. and exists in a wide range of different versions and packages from a number of manufacturers as well as for integration into custom silicon. There are also chips combining an FPGA with an ARM processor core available from Triscend.

The Armulator is a cycle-accurate simulator for the ARM cores available from ARM Ltd.

3.3. Xilinx Virtex II/ iSE

The Virtex II is Xilinx’s most advanced FPGA available at the moment, with a Virtex II Pro version available that contains embedded PowerPC processors.

4. THE HARNESS

4.1. Module Definitions

4.1.1. Tigersharc
In the case of the Tigersharc, the module is defined as a set of C files specified in the Partitioner program. A standalone test bench version of each module is required to get the initial performance data. After an initial partition and schedule has been determined the design can be run with full data interchange to verify the partitioning.

4.1.2. ARM
ARM modules are defined in exactly the same way as Tigersharc modules, allowing easy code sharing between these designs.

4.1.3. Xilinx
Xilinx modules are defined by creating the entire module as project in iSE. It is synthesized here and the Harness can execute the simulation in ModelSim and collect the data from it. Each Xilinx module will be contained in its own directory underneath the main project directory.

4.2. Module Interface

The module interface for C versions is in the file file_interface.c which has to be included in each module that makes use of the interface.

The template generation can be used to create an initial framework for each module with the necessary code for communicating on the channels used by the module.

Note that to allow simulation of each design module on multiple processing units it is necessary that there is only one module writing to each channel, though multiple modules may read from each channel.

4.3. Data Acquisition

4.3.1. C
Data Acquisition for C versions is done through a set of preprocessor macros which enable acquisition of timing data and disable collection around certain areas, for example the interface functions which are estimated separately.

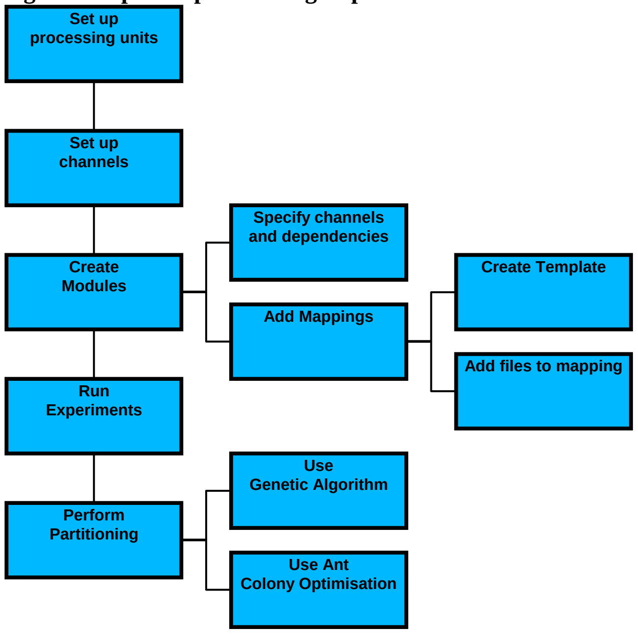
4.3.2. FPGA
For the FPGA parts the data is collected in the test bench for each module. The test bench also provides the module interface. This allows timing from the harness code to be automatically excluded in the Xilinx environment.

Furthermore, the harness collects information on the number of FPGA elements used on the Virtex chip by parsing the output of the synthesis runs.

The partitioner can create a test bench file that sets up a clock with a counter; when the clock is disabled, the profile information is written out to the data file read by the harness.

5. THE PARTITIONER DESIGN

Figure 4: Optimal partitioning steps



5.1. Overall structure

The partitioner uses an object oriented design using the following basic classes: processing units, communication channels, design modules and mappings.

A processing unit is a general purpose processor or FPGA device that code can be partitioned onto. Any number and type of these can be set up in the Partitioner.

A design module is a part of the program to be partitioned. It communicates with other modules on the communication channels specified for it. The Partitioner can create template code for the instantiation of a design module on a particular processing unit type.

A communication channel is a name for a channel that data can be exchanged on between different design modules. The Partitioner can create code for communicating on these channels as part of the template code generated for an instantiation of a design module on a processing unit type.

Mappings represent the implementation of a design module on a specific processing unit.

The Partitioner is separate from the Simulation harness and can be run independently once the simulation data has been acquired. Different scenarios can be tested without having to rerun the simulation.

5.2. GUI

The GUI allows entering the information on all the modules, their dependencies and different mappings. Several different mappings, even onto the same hardware part, can take place for each module in the design.
The GUI also allows hardware and design parameters to be set, such as the clock speed, for each hardware module.

5.3. Simulation

The partitioner will automatically link and compile all mappings of all modules and execute them in the appropriate simulator. An optional cygwin-based environment is available to provide quick tests outside of the simulators to allow fast tests of the module code.

5.3.1. Cost Function

The cost function depends on the following variables:

- Relative Power scale (set in the Partitioner's global settings dialog as “Power Cost Weighting”), determines how much value is given to the power consumption of a module in relation to the cost of the processing unit it is running on
- Absolute power scale is calculated by first finding the maximum cost of any processing unit and the maximum power consumption of any module. This way of calculating the absolute power scale is used to insure a well distributed cost function (if the cost function is not well distributed, the ant colony optimization algorithm does not perform optimally).
- The cost of instantiating a module on a processing unit is plus the cost of using the processing unit in the overall design (set in the processing unit properties) if it hasn't been used already.
- The overall cost function is the cost of all the module instantiations plus a time and area penalty if either of these limits have been exceeded.

5.3.2. FPGA

The FPGA has a unit cost as well as an “infinite” cost for exceeding the number of FPGA elements available on the chip.

The cost function is a weighted sum of the power and “fixed” costs for each module with a (usually) infinite penalty for exceeding the time or area limits.

5.3.3. ARM

The arm has a cost function similar to the FPGA one but without penalty for exceeding area limits as there are none (however a similar limit could be conceivably imposed on memory image size.

5.4. GA Partitioning Algorithm

The partitioning is performed using a genetic algorithm. Each possible partitioning is represented by a vector of numbers, where each number in each cell in the vector represents one possible mapping of the module represented by the cell onto a particular target device. The vector is called a “chromosome” in the language of genetic algorithms.

Initially a population of chromosomes is generated at random, then each chromosome is evaluated against the cost function. The most fit chromosomes are selected as parents for a new generation of chromosomes, and the cycle is repeated a chosen number of times.

The GA algorithm allows an almost completely free choice of cost function and tends to generate good results fairly quickly but does not give a guarantee of an optimal solution.

5.5. Scheduler

The scheduler finds the optimum schedule of design modules on processing units given a specified mapping of design modules onto processing units. It consists of a function to compute the execution time of a schedule and a recursive function to apply this function to all possible schedules in turn and determine the schedule with the shortest execution time. Given that the design modules are constrained to a single target processing unit the scheduling cost is not prohibitive.

5.5. Ant Colony Optimization Algorithm

The modified ant colony optimization algorithm created for this study combines partitioning and scheduling into one operation. The algorithm is based on the basic ant colony optimization algorithm described in [2]i.

The ant colony optimization algorithm works by simulating a colony of ants that attempts to find a path through the problem space, in our case attempt to find the optimum path (schedule) through the different instantiations of design modules onto processing units.

Unlike in the original algorithm each ant has two taboo lists, one for the design modules that have been scheduled and one for the particular mappings from design modules onto processing units that have been chosen.

Given a set of n design modules that have possible instantiations onto the processing units, we try to find the lowest cost schedule and instantiations of design modules onto the processing units available. That is, we try to find a path between all the possible mappings of design modules onto processing units such that each design module is only instantiated once. We call d_{ij} the cost of scheduling mapping j after mapping i was previously scheduled. In general, we use cost instead of the length of paths used in the original algorithm.

For each move of each ant in the algorithm, the ant considers all the possible mappings of all the possible design modules that haven't been scheduled yet. It determines the cost of scheduling each mapping given the already instantiated mappings. From this cost and the pheromone trail of each possible transition we determine the transition probability for each possible mapping, using equation (4) in [2].

After all ants have completed their tour the pheromone trails between possible mappings are updated based on the total cost of the tour each ant took.

An elitist strategy with 2 elitist ants according to section V.B in [2] is used; this means that the best path is given 3 times the usual intensity.

6. RESULTS

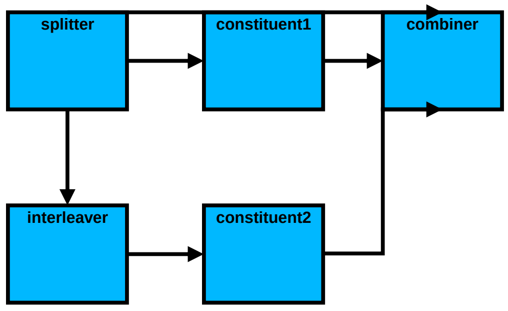


Figure 5: Turbo coder example

6.1. 3GPP channel coding example

The 3rd generation mobile phone channel coding example uses the turbo coder as it is the most computationally challenging parts of the channel coding.

The Turbo coder is made up of a splitter, an interleaver, two constituent encoders and a combiner.

Unfortunately only limited scope for parallelism exists in this example as the only possibility for parallel execution is “constituent1” in parallel with “interleaver” and “constituent2”. This limits the complexity of calculating the partitioning (the more options for executing code paths in parallel exist in the design, the bigger the solution space).

Optimizer	*Best Result*	*In Round*	*Time*
GA	**100.26**	**1**	**0.05**
ACO	100.3	**1**	0.11
Best:	**100.26**	**1**	**0.05**

In the simple 3GPP example little difference exists between the two partitioning algorithms. This example also illustrates that the ant colony optimization may not always find the very best result quickly (in fact, it will find a result very close to the optimal result very quickly but it can take a long time to find improvements to a near-optimal result as new potential solutions are chosen with a probability proportional to the potential change to the current best result).

6.2. Synthetic Benchmarks

6.2.1. Simple Synthetic Example

The "simple" example contains of 6 modules, 5 of which have mappings onto 3 possible ARM processing units and 1 which has a mapping onto a Tigersharc target. No dependencies are set between the modules.

Optimizer	*Best Result*	*In Round*	*Time*
GA	**386.47**	**1**	14.3
ACO	**386.47**	2	**0.7**
Best:	**386.47**	**1**	**0.7**

As can be seen, the GA is at a disadvantage in this case as it has to run the scheduler for each chromosome being evaluated, a lengthy process where a large number of possible schedules need to be evaluated.

The ACO converges on a good solution very quickly due to the good match of the ACO heuristic to the problem space.

6.2.2. Complex Synthetic Example

The "complex" example is the same as the "simple" example but it has dependencies between the different modules.

Optimizer	*Best Result*	*In Round*	*Time*
GA	**469.35**	11	8.9
ACO	**469.35**	**6**	**1.7**
Best:	**469.35**	**6**	**1.7**

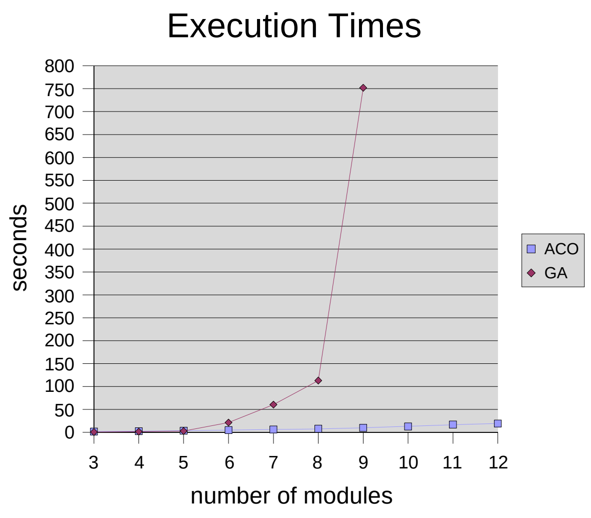
In this case the GA is better able to explore the complete problem space and is able to come up with a solution in less time than for the previous example. It is also much faster than on the simple example as far fewer different schedules need to be evaluated for each chromosome.

The ACO still finds a good solution though more slowly than on the simple example. It is still faster than the GA as it performs scheduling as part of the optimization.

6.2.3. Variable Synthetic Example

A synthetic example with a variable number of modules that need to be scheduled was created to compare the scalability of the ant colony versus genetic algorithm optimization algorithms. Figure 6 shows the results from this experiment:

Figure 6: Scalability comparison



The execution time for the genetic algorithm version of the partitioner takes exponential time to calculate a partitioning and it was prohibitively expensive to run it for more than 9 modules (this is as a result of the exhaustive search scheduler used in the genetic algorithm version of the Partitioner).

The ant colony optimization version of the partitioning algorithm executed in nearly linear time and should scale to very large problem sets. Both algorithms produced identical results for each problem tested in this example.

6.3. Conclusions

The ACO algorithm provides equally good results as the GA in a much more scalable fashion; however it is more sensitive to bad choices in the Partitioner parameters.

7. REFERENCES

[1] E2R II, Deliverable 4.1
[2] M. Dorigo, V. Maniezzo, and A. Colorni. *The Ant System: Optimization by a colony of cooperating agents*. IEEE Transactions on Systems, Man, and Cybernetics, 26:29--41, 1996.