Chapter 2
The hArtes Tool Chain

Koen Bertels, Ariano Lattanzi, Emanuele Ciavattini, Ferruccio Bettarelli, Maria Teresa Chiaradia, Raffaele Nutricato, Alberto Morea, Anna Antola, Fabrizio Ferrandi, Marco Lattuada, Christian Pilato, Donatella Sciuto, Roel J. Meeuws, Yana Yankova, Vlad Mihai Sima, Kamana Sigdel, Wayne Luk, Jose Gabriel de Figueiredo Coutinho, Yuet Ming Lam, Tim Todman, Andrea Michelotti, and Antonio Cerruto

This chapter describes the different design steps needed to go from legacy code to a transformed application that can be efficiently mapped on the hArtes platform.

2.1 Introduction

The technology trend continues to increase the computational power by enabling the incorporation of sophisticated functions in ever-smaller devices. However, power and heat dissipation, difficulties in increasing the clock frequency, and the need for technology reuse to reduce time-to-market push towards different solutions from the classic single-core or custom technology. A solution that is gaining widespread momentum consists in exploiting the inherent parallelism of applications, executing them on multiple off-the-shelf processor cores. Unfortunately, the development of parallel applications is a complex task. In fact, it largely depends on the availability of suitable software tools and environments, and developers must face with problems not encountered during sequential programming, namely: nondeterminism, communication, synchronization, data partitioning and distribution, load-balancing, heterogeneity, shared or distributed memory, deadlocks, and race conditions.

Since standard languages do not provide any support for parallel programming some effort has been devoted to the definition of new languages or to the extension of the existing ones. One of the most interesting approaches is the OpenMP standard based on pragma code annotations added to standard languages like C, C++ and Fortran [45].

K. Bertels (✉)
Fac. Electrical Engineering, Mathematics & Computer Science, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: k.l.m.bertels@tudelft.nl

K. Bertels (ed.), *Hardware/Software Co-design for Heterogeneous Multi-core Platforms*,
DOI 10.1007/978-94-007-1406-9_2,

The aim of the hArtes toolchain is to have a new way for programming heterogeneous embedded architectures, dramatically minimizing the learning curve for novice and simultaneously speed up computations by statically and transparently allocating tasks to different Processing Elements.

Currently the hArtes toolchain addresses three platforms, based on Atmel Diopsis SOC. This SOC embeds an ARM9EJS and MAGIC, a floating point DSP. The OMAP family will be supported soon. These target platforms support 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 clearly fits well with the OpenMP standard. In fact, the OpenMP standard has been adopted in this project since it is well supported and accepted among parallel application developers and hardware vendors. It is supported by many commercial compilers (Microsoft and IBM) and, in the OpenSource scene, by GCC [13], starting with the 4.2 release. The reasons behind the wide diffusion of OpenMP lie in the fact that it is considered a good mean to achieve portable parallel execution on shared-memory multiprocessor (SMP) systems, it allows the incremental parallelization of existing codes and 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 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 on the target platforms will instead adopt an ad-hoc, independent 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. In this way the parallel code produced can be tested on different hosts that support OpenMP, simply ignoring the mapping directives.

For these reasons, differently from CUDA or other architectures that address the problem of "parallel computing", the hArtes toolchain doesn't impose to write applications by using new syntaxes or new libraries. In fact, the programmer can express parallelism as he always did on a PC platform (threads, OpenMP), or he may leave the toolchain to extract the parallelism automatically. Then, at the end of the compilation process the harts toolchain produces a single executable with all the symbols and debugging information for each processing element. The programmer has only to execute/debug it on the targeted heterogeneous platform as he was dealing with a single processor architecture (just like a PC).

The following sections of the chapter will first introduce the overall toolchain structure and then present the different tools composing it, concluding with the introduction of the overall framework user interface, showing how it can be used by the toolchain users for mapping their applications.

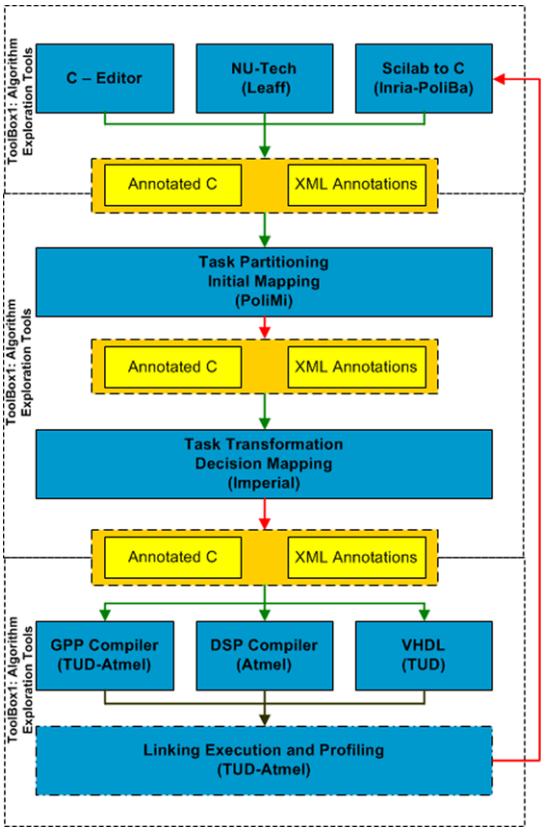


Fig. 2.1 The hArtes toolchain

2.2 Tool Chain Structure

Figure 2.1 illustrates the overall hArtes toolchain flow. The flow is constituted by several stages, and each stage can be iterated more than once. Information produced in the execution of the different phases is transferred from one phase to the next using C pragma annotations added to the application source code and XML annotations contained in a dedicated file that is modified and enriched by each tool. The pragma annotations are used to specify partitioning, mapping and profiling of the application that is currently analyzed by the toolchain. The XML annotations contain information about the target architecture (i.e., the available processing elements and their interconnections) and about the characterization of the implementations for each function to be mapped onto that architecture. In conclusion, all the information about the structure of the application is contained in the application source code, while the architecture description file provides all the information on the characteristics of the execution on the target architecture. The flow is composed of the following three main blocks:

1. the Algorithm Exploration and Translation (AET) toolbox (detailed in Sect. 2.4)
2. the Design Space Exploration (DSE) toolbox (detailed from Sect. 2.5 to Sect. 2.7.6)
3. the System Synthesis (SysSyn) toolbox (detailed from Sect. 2.8 to Sect. 2.10)

The Algorithm Exploration Tools (AET) are used in the first phase for the algorithm design and exploration. Its output is the algorithm C code. The available graphical or textual tools allow the designer to explore different algorithm solutions, until the final correct C code of the application (plus possible notations) has been obtained. The Design Space Exploration (DSE) tools are used in the second phase to manipulate and optimize the C code. The profiled C code is then partitioned in tasks taking into consideration the performance data just obtained. The resulting partitioned code, is further annotated to provide 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. Finally, the System Synthesis tools are used in the third (final) phase, to perform the compilation, the linking and loading of the application onto the target hardware. In particular, the 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 hArtes project includes also the development of the hArtes IDE, which is the human interface to the workspace. The integration in a unified toolchain of a set of tools going from the very high level of the application development to the lowest levels is an original contribution of hArtes and the integration methodology adopted is generic enough to conceive the hArtes framework as the basis on which other toolchain instances for embedded system development can be built. The rest of the chapter describes the functionalities expected from each tool represented in the toolchain.

2.2.1 The Algorithm Exploration Tools

The user has three different options to enter the hArtes toolchain:

1. Describe the application with Scilab
2. Describe the application with NU-Tech
3. Describe the application directly in C language

The AET Tools have the aim of translating in C language the application described with NU-Tech or with Scilab. Starting from an application described as

a NU-Tech network of Satellites (NUTSs), NU-Tech is able, thanks to the GAE Tool feature, to produce a C description of the application and pass the results to the toolchain DSE tools. In order to produce a C description of the application the source code of each functional block is needed. The GAE Tool feature handles the task of describing how each block interacts with each other and with the system I/O. The ready-to-use NUTS library allows the user to concentrate on the core of his/her own algorithm and write it down according to the GAE Tool specifications. Specifications are few and very easy to follow:

- fixed sampling frequency for each streaming session: any streaming session should work at a fixed sample rate.
- Init/Deinit/Process architecture: any algorithm should consist of these three functions.

As an alternative option, the application can be described using the scripting language of Scilab. A tool to translate the Scilab description into the C language description is developed and is integrated with a library of functions to support the DSP applications. The initial set of functions is selected according to the application partners requirements. Addition of new library functions is possible if the developer provides the underlying description in C, supporting all the different required implementations associated with the data types supported by Scilab (e.g. implementation for scalar variables, operating on individual items like float variables, or implementation for vectorial variables operating on array of data mono or two dimensional). In order to optimize the implementation on the embedded platform, dynamic memory allocation and de-allocation shall be avoided. The last option is directly writing the C description of the application. The code must respect some coding guidelines and it is possible to add some annotations to control the behaviour of the tools that will be subsequently called. The output of the AET tools is C code with optional pragma annotations added by the user.

2.2.2 The DSE Tools

The Design exploration tools are composed of a set of tools aiming at first at collecting information on the application, then on proposing a task partitioning based on cost estimations and applying transformations to these tasks in order to make them more suitable for their implementation on the given target platform. Finally, a phase of task mapping is performed to identify the most suitable hardware/software assignment of the different tasks, in order to optimize specific figures of merit.

At first a profiling of the application is performed. The Code Profiler provides information necessary for the subsequent tasks since it generates information on the CPU cycles of the different functions to identify computational hotspots and analyzes the memory usage and the access patterns. CPU usage is computed by pro, that has been modified to produce its output in the required format. The memory usage and access patterns provide as output a quantitative usage graph containing

information on the amount of data exchanged between two functions. The tools output their results by annotating the C code of the application. The computation time of a function will be provided as an annotation in the C-code with # pragma_profile directives. The two measures num_calls and time are provided for each function. num_calls indicates the number of times the function is called and the time expresses the running time of the program.

Task Partitioning

The task partitioning tool identifies the parallelism available in the application by analyzing the tree structure of the code and groups the identified independent operation clusters at the appropriate granularity level. Moreover, the task partitioning tool automatically proposes an initial guess on the tasks mapping, defining which parts of the application should be implemented in hardware, or which parts should be executed in software and on which kind of processing element available in the target system. The mapping is based on an initial cost estimation taken into account together with the performance. The requirement analysis shows that the problem can be clearly separated in two distinct sub-problems:

1. in the first phase the tool identifies the parallelism in the application, i.e. shall annotate, using task partitioning C pragma notations, how it can be decomposed in separate threads, presenting the parallel tasks within
2. the annotated code; in the second phase the tool will indicate, using specific code annotations, an initial guess concerning which processing element (GPP, DSP or programmable logic) should execute the identified tasks, i.e. where each task should be mapped in the system. The overview of panadA automatic parallelization process is shown in Fig. 2.11.

The inputs are the C description of the application and an XML file containing information about the architecture and the performance of the application on each processing element. When the XML file does not provide any performance estimation, the task partitioning tool performs an internal estimation of the performance of the application. Feedback from the toolchain can improve the accuracy of the performance estimations and therefore the quality of the partitioning and of the initial mapping identified by the partitioning tool. The result of Task partitioning is a code annotated with Task Partitioning annotations and possibly with HW assignment annotations.

Task Mapping

The task mapping tool (**hArmonic**) has two main inputs: (1) the source code, supporting an arbitrary number of C source files and (2) the XML platform specification. The platform specification describes the main components of the heterogeneous system, including the processing elements, the interconnect, storage components, and system library functions. The output of the task mapping tool is a set of C

sources. Each source is compiled separately for each backend compiler targeting a processing element, and subsequently linked to a single binary. The hArmonic tool is divided in four stages:

- **C Guidelines Verification**. Automatically determines whether the source code complies with the restrictions of the mapping process, allowing developers to revise their source in order to maximize the benefits of the toolchain.
- **Task Filtering**. Determines which processing elements can support each individual task in the application in order to generate feasible mapping solutions on the hArtes platform.
- **Task Transformation**. Generates several versions of the same task in order to exploit the benefits of each individual processing element, and maximize the effectiveness of the mapping selection process.
- **Mapping Selection**. Searches for a mapping solution that minimizes overall execution time.

Cost Estimation

The cost estimation process includes the contribution of Imperial, PoliMi and TUD. Each contribution solves a different problem in the hArtes toolchain. Imperial's cost estimator is used for the task mapping process (**hArmonic**) to determine the cost of each task when executed on a particular processing element. In this way, the task mapper is able to decide, for instance, whether there is a benefit to run a particular task on a DSP instead of a CPU. PoliMi's cost estimator is a module integrated in the partition tool (**zebu**) which assists in deriving a better partition solution and reduces the design exploration time. Finally, the cost estimator from TUD is used to aid early design exploration by predicting the number of resources required to map a particular kernel into hardware. Its output is a set of information added to the XML file.

2.2.3 The System Synthesis Tools

The System Synthesis tools purpose is to create a unified executable file to be loaded on the HW platform starting from the C codes produced by the task mapping tool. The System Synthesis tools are:

- a customized version of the C compiler for the GPP (**hgcc**),
- a DSP compiler enriched with tools to generate information required by the hArtes toolchain,
- an RTL generator from the C code provided as input,
- a linker to generate the unified executable image for the hArtes HW platform, containing the executable images associated with the GPP, DSP and FPGA,
- a loader to load the executable image sections into the associated HW platform memory sections,

- additional tools and libraries for format conversion, metrics extractions, interaction with the operating system running on the platform.

The **hgcc Compiler** for the target GPP is extended to support pragma annotations to call the selected HW implementation. The effect of these pragmas is to expand functions into a sequence of Molen APIs (see later). This customized version of the C compiler for the GPP is called hgcc since it is a customization for hArtes (h) of the gcc compiler. The **DSP Compiler** is the compiler available for the DSP. It must be completed by adding an executable format conversion in order to create a unified elf file for the entire application (see later mex2elf) and a tool to add the cost information to the XML file (see later DSP2XML). The **VHDL Generator** tool (C2VHDL) generates an RTL HDL from the C code provided as input. This tool include two steps. First, the input C-description of the algorithm shall be presented as a graph, which shall be then transformed into a set of equivalent graphs, introducing different graph collapsing techniques. Second, a metric base decision on the optimal graph representation shall be made and this graph representation shall be translated into RTL level and then to a selected HDL. The **DSP2XML** tool collects information about the code compiled on the DSP. The collected information concerns the optimization achieved by the task allocated on the DSP, such as code size and execution speed. The **mex2elf:** tool converts the executable format generated by the DSP C compiler into the object format used by the GPP linker, i.e. ARM little endian elf format. The **MASTER GPP Linker** creates an elf executable ready to be loaded by the target OS (Linux). It links Molen Libraries, GPP and DSP codes. The FPGA bitstream is included as a binary section. It is composed of a set of customization of the scripts controlling the code production, and customization to the linker itself is avoided in order to reuse the linker tool from the GNU toolchain. The **MASTER GPP Loader** is in charge of loading DSP and MASTER codes on the Target platform. The FPGA bitstream is loaded into a portion of the shared memory accessible by the FPGA. The loading process is performed in the following steps:

1. Loading of the code sections following the default Linux OS strategy, including the m-mapping of the hArtes memory sections associated with the HW platform components (DSP, FPGA).
2. Customized C Run Time (CRT) performing the additional operations required by the hArtes HW/SW architecture, such as copy of the memory sections to the physical devices using the appropriate Linux drivers developed to access the HW platform components (DSP, FPGA).

Dynamic FPGA reconfiguration is performed by copying the required configuration bitstream to the FPGA from the system shared memory where all the available configuration bitstreams were loaded during the loading phase.

2.3 hArtes Annotations

The high level hArtes APIs are POSIX compliant. Software layers have been added to make uniform C, thread and signal libraries of PEs. The POSIX compliance

makes easy the porting of application across different architectures. Low level and not portable APIs are available to the application in order to access directly particular HW resources like timers, audio interfaces. The hArtes toolchain requires an XML architecture description, where main HW and SW characteristics are described. Beside the XML description (provided by the HW vendors), the toolchain uses source annotations via pragmas. From the developer point of view source annotations are optional because they are generated automatically by the toolchain. Both source and XML annotations can be used by the developer to tune the partitioning and mapping of the application.

2.3.1 XML Architecture Description File

The XML Architecture Description File aims at providing a flexible specification of the target architecture and it is used for information exchange between the tools involved in the hArtes project. First we provide a brief overview of the XML format. Next we present the organization of the XML file for architecture description. Then we describe the interaction of the hArtes tools with the presented XML file.

There are many advantages of using the XML format for the Architecture Description File. The XML format is both human and machine readable, self-documenting, platform independent and its strict syntax and parsing constraints allow using efficient parsing algorithms. In consequence, the XML format is suitable for the structured architecture description needed by the tools involved in the hArtes project.

The structure of an XML document relevant for the Architecture Description File is shortly presented in this section. The first line of the XML file usually specifies the version of the used xml format and additional information such as character encoding and external dependencies (e.g. `<?xml version="1.0" encoding="ISO-8859-1"?>`).

The basic component of an XML file is an element, which is delimited by a start tag and an end tag. For example, the following element:

```
<name>hArtes</name>
```

has `<name>` as the start tag and `</name>` as the end tag of the “name” element. The element content is the text that appears between the start and end tags. Additionally, an element can have attributes, such as:

```
<name id="12">hArtes</name>
```

An attribute is a pair of name-value, where the value must be quoted.

Finally, we mention that every XML document has a tree structure, thus it must have exactly one top-level root element which includes all the other elements of the file.

The root element of the XML description is named `ORGANIZATION`. It contains the following elements:

- HARDWARE: which contains information about the hardware platform;
- OPERATIONS: which contains the list of operations (i.e., C functions) and implementations;
- PROFILES: which contains additional information generated by tools.

```
<ORGANIZATION>
   <HARDWARE>
      ...
   </HARDWARE>
   <OPERATIONS>
      ...
   </OPERATIONS>
   <PROFILES>
       ...
   </PROFILES>
</ORGANIZATION>
```

The HARDWARE XML Element

The HARDWARE element contains the following elements:

- NAME: name of the board/hardware;
- FUNCTIONAL_COMPONENT: which describes each processing element;
- STORAGE_COMPONENT: which describes the storage (memory) elements;
- BUS_COMPONENT: which describes architectural interconnection elements;
- VBUS_COMPONENT: which describes virtual (direct) interconnection elements.

Each FUNCTIONAL_COMPONENT is composed of:

- NAME: the unique identifier for the FUNCTIONAL_COMPONENT, such as *Virtex4* or *ARM*. It should be a valid C identifier as it can be used in pragmas;
- TYPE: the class of the functional component. Valid values are *GPP*, *DSP* and *FPGA*. Based on this information, the proper compiler will be invoked the related stitch code will be generated;
- MODEL: represents the specific model for the processing element (e.g., *XILINX VIRTEX XC2VP30*);
- MASTER: whether the functional component is the master processing element or not. The element contains *YES* if the component is the master or *NO* otherwise. Only one functional component can be defined as master.
- SIZE: which is relevant only for reconfigurable hardware and represents the number of available Configurable Logic Blocks (CLBs).
- FREQUENCY: is the maximum frequency of the functional component, expressed in MHz;
- START_XR: the starting range of the transfer registers associated with that functional component.
- END_XR: the ending of the range of the transfer registers.

- HEADERS: contains the list of all headers to be included when the C code is split to the corresponding backend compilers. It is composed of:
  - NAME: (multiple) filename name of the header.
- DATA: contains C data (storage) specification. In particular, it specifies the following elements:
  - MAXSTACKSIZE: the maximum stack size.
  - DATA_TYPE: is of list of the C basic types. Note that a **typedef** can be treated as a basic type if included here. In this case, it does not matter if the hidden type does not resolve to a basic type.
    * NAME: name of the basic type;
    * PRECISION: is the precision of the data type.

Example:

```
<FUNCTIONAL_COMPONENT>
  <NAME>Arm</NAME>
  <TYPE>GPP</TYPE>
  <MODEL>ARMv9</MODEL>
  <MASTER>YES</MASTER>
  <START_XR>1</START_XR>
  <END_XR>512</END_XR>
  <SIZE>0</SIZE>
  <FREQUENCY>250</FREQUENCY>
  <HEADERS>
    <NAME>my_header.h</NAME>
    <NAME>my_header2.h</NAME>
  </HEADERS>
  <DATA>
    <MAXSTACKSIZE>1000</MAXSTACKSIZE>
    <DATA_TYPE>
      <NAME>int</NAME>
      <PRECISION>32</PRECISION>
    </DATA_TYPE>
  </DATA>
</FUNCTIONAL_COMPONENT>
```

Similarly, each STORAGE_COMPONENT contains the following elements:

- NAME: for the name of the component. E.g. MEM1. One storage component, connected to FPGA must be named XREG.
- TYPE: type of the memory. E.g., SDRAM.
- SIZE: the size of the memory in kilobytes. E.g., 16.
- START_ADDRESS: is the starting range of memory addresses in the shared memory. These should be hexadecimal number and thus you must use “0x” prefix.
- END_ADDRESS: is the ending range of memory addresses in the shared memory. These should be hexadecimal number and thus you must use “0x” prefix.

Example:

```
<STORAGE_COMPONENT>
  <NAME>MEM1</NAME>
  <TYPE>SDRAM</TYPE>
  <SIZE>128</SIZE>
  <START_ADDRESS>0</START_ADDRESS>
  <END_ADDRESS>0xFFFFFFFF</END_ADDRESS>
</STORAGE_COMPONENT>
```

The BUS_COMPONENT contains:

- NAME: used to identify the bus.
- BANDWIDTH: the size of one memory transfer in kbytes/sec (for example: 1024).
- FUNCTIONAL_COMPONENTS: functional components that can access this bus
  – NAME: the name of the functional component.
- STORAGE_COMPONENTS: storage components connected on this bus
  – NAME: the name of the storage component.
- ADDR_BUS_WIDTH: The size in bits of the address bus to the corresponding storage component. Used by the DWARV toolset for CCU interface generation, if the bus is not connected to a FPGA, the element is optional.
- DATA_BUS_WIDTH: The size in bits of the read and write data busses for the corresponding storage component. Used by the DWARV toolset for CCU interface generation. If the bus is not connected to an FPGA, the element is optional.
- READ_CYCLES: The number of cycles to fetch a word from the corresponding storage element. If the storage component runs at different frequency than the CCU, the number of access cycles has to be transferred as CCU cycles. For example, if the storage component is clocked at 200 MHz and requires 2 cycles to fetch a word and the CCU operates in 100 MHz, the cycles in the component description are reported as 1. If the storage component is clocked at 50 MHz, the CCU at 100 MHz, the fetch cycles are 2, then the component description contains 4 as read cycles. If the storage component has non-deterministic access time, the cycles element shall be set to “unavailable”.

Example:

```
<BUS_COMPONENT>
  <NAME>InternalFPGA</NAME>
  <TYPE>INTERNAL</TYPE>
  <BANDWIDTH>1024</BANDWIDTH>
  <ADDR_BUS_WIDTH>32</ADDR_BUS_WIDTH>
  <DATA_BUS_WITH>64</DATA_BUS_WITH>
  <READ_CYCLES>2</READ_CYCLES>
  <FUNCTIONAL_COMPONENTS>
    <NAME>FPGA</NAME>
  </FUNCTIONAL_COMPONENTS>
  <STORAGE_COMPONENTS>
    <NAME>MEM1</NAME>
```

```
  </STORAGE_COMPONENTS>
</BUS_COMPONENT>
```

The VBUS_COMPONENT element represents the direct interconnection between two functional elements

- NAME is the bus interconnect identifier. Example: VBUS1;
- FUNCTIONAL_COMPONENT_NAME (multiple) name of the functional component inside this bus;
- BITSPERUNIT the number of bits exchanged per unit;
- BANDWIDTH the number of units transferred per unit of time.

```
<ORGANIZATION>
  <HARDWARE>
    <VBUS_COMPONENT>
      <NAME>id</NAME>
      <FUNCTIONAL_COMPONENT_NAME>component 1</FUNCTIONAL_COMPONENT_NAME>
      <FUNCTIONAL_COMPONENT_NAME>component 2</FUNCTIONAL_COMPONENT_NAME>
      <BITSPERUNIT>21</BITSPERUNIT>
      <BANDWIDTH>102</BANDWIDTH>
    </VBUS_COMPONENT>
  <HARDWARE>
<ORGANIZATION>
```

The OPERATIONS XML Element

The OPERATIONS element is used for the description of the operations that are implemented on the hardware components. It contains a list of OPERATION elements which contain the associated functional components and describe the hardware features of each specific implementation.

```
<ORGANIZATION>
   <OPERATIONS>
      <OPERATION>
           ...
      <OPERATION>
      <OPERATION>
           ...
      <OPERATION>
   </OPERATIONS>
</ORGANIZATION>
```

The OPERATION element structure is:

- NAME, the name of the operation in the C file.
- multiple COMPONENT elements containing:
  - NAME, this has to be a name of an existing FUNCTIONAL_COMPONENT.
  - multiple IMPLEMENTATION elements containing
    - ∗ ID, an unique identifier (for all the XML) for each hardware implementation.

- SIZE, for the implementation size. For each type of component this will have a different meaning as follows: for the FPGA it will be the number of 100 slices, for the GPP and for the DSP it will be the code size in KB.
- START_INPUT_XR, START_OUTPUT_XR, for the first XRs with the input /output parameters.
- SET_ADDRESS, EXEC_ADDRESS, for the memory addresses of the microcode associated with SET/EXEC, can be omitted if not FPGA.
- SET_CYCLES, EXEC_CYCLES, for the number of component cycles associated with hardware configuration/execution phase, can be omitted if not FPGA.
- FREQUENCY—the frequency of the implementation on FPGA in MHz. If the functional component is not FPGA, this can be omitted.

Example:

```
<OPERATION>
  <NAME>SAD</NAME>
  <COMPONENT>
    <NAME>Arm</NAME>
    <IMPLEMENTATION>
      <ID>11</ID>
      <SIZE>100</SIZE>
      <START_INPUT_XR>3</START_INPUT_XR>
      <START_OUTPUT_XR>10</START_OUTPUT_XR>
      <SET_ADDRESS> 0X00000000 </SET_ADDRESS>
      <EXEC_ADDRESS> 0X00000000 </EXEC_ADDRESS>
      <SET_CYCLES> 100 </SET_CYCLES>
      <EXEC_CYCLES> 200 </EXEC_CYCLES>
    </IMPLEMENTATION>
  </COMPONENT>
</OPERATION>
```

In the presented Architecture Description File, there is a clear delimitation about the hardware/software features of the target architecture/application. The information contained in an OPERATION element (such as size, frequency) has to be provided by the automatic synthesis tools that generates a specific implementation for a specific operation. Finally, the information for the HARDWARE element (such as Memory sizes, GPP type) is general and should be introduced by the architecture designer.

The PROFILES XML Element

The PROFILES element stores the result of some tools in the hArtes toolchain.

HGPROF is a profiling program which collects and arranges statistics of a program. Basically, it captures performance information about specific elements of the program such as functions, lines, etc. Currently, HGPROF captures the following information which are stored in the PROFILES element:

- NAME: is the name of a function;
- STIME: is an average execution time (ms) of each function per call without subroutine calls;
- CTIME: is an average execution time (ms) of each function per call with subroutine calls;
- NCALLS: is the number of times a function is called;
- LINE element containing
  - NUMBER is the line number;
  - NCALLS is the number of time a line is executed.

Example:

```
<PROFILE>
    <HGPROF>
        <FUNCTION>
            <NAME> function1</NAME>
            <STIME>134</TIME>
            <CTIME>1340</TIME>
            <NCALLS>32</NCALLS>
            <LINE>
                <NUMBER> 34</NUMBER>
                <NCALLS> 350</NCALLS>
            </LINE>
        </FUNCTION>
    </HGPROF>}
</PROFILE>}
```

As HGPROF collect this information on the associated tag the same happened for QUIPU and QUAD.

2.3.2 Pragma Notation in hArtes

Pragmas are used to embed annotations directly into the source-code. The advantage of pragmas over XML is that the code itself carries the annotations; however it can also clutter the code and make it less easy to read. The biggest benefit of using #pragmas is that they can be used to annotate constructs such as assignment and loop statements, which, differently from functions, do not have obvious identifiers. Moreover, it is worth noting that the program structure has to be revisited when there are functions that interact outside the program itself. In fact, if the program contains IO directives or supervisor calls, it is problematic to extract parallelism from them since memory side-effects cannot be controlled or predicted. For this reason, functions are classified into two different types:

- data interfacing functions
- data processing functions

The former ones should be excluded from the parallelism extraction. In particular, when a data-interfacing function contains calls to functions communicating with the external world (e.g. IO or supervisor calls), it shall be marked with a specific directive to be excluded from the partitioning. In this way, a better performing code will be obtained since the parallelism extraction operates only on the data processing region, where no interfacing with external world occurs. For this reason, the source code has to be compliant with this structure to allow the parallelism extractor tool to concentrate its effort on meaningful parts of the application. Note that how functions exchange data is not a constraint, but the distinction between interfacing and processing functions is.

Few common format directives will be shared among the different pragma notations used in hArtes. The common directives will refer to the format of a single line pragma notation. No common rules will be defined to describe grammar expressions built with the composition of multiple pragma lines. The notation is case sensitive. Pragma notation lines will be specified with the following format:

```
pragma ::= #pragma <pragma_scope> [<pragma_directive>] [<clauses>]
new_line
```

White spaces can be used before and after the '#' character and white spaces shall be used to separate the *pragma_scope* and the *pragma_directive*. The *pragma_scope* is composed of a single word; multiple words, separated by white spaces, may compose a *pragma_directive*.

```
pragma_scope ::= <word>
pragma_directive ::= <word> [<word> [<word>...]]
```

The *pragma_scope* identifier specifies the different semantic domain of the particular *pragma_directive*. The specific hArtes pragma scopes are the following:

- **omp** for the OpenMP domain;
- **profile** for the profiling domain;
- **map** for the hardware assignment domain;
- **issue** for generic issues not related to the previous scopes, but mandatory for hArtes toolchain.

They are used to reduce the potential conflict with other non-hArtes pragma notations. The *pragma_directive* identifies the role of the pragma notation within the scope of the specific semantic domain. For the different directives, see the paragraph related to the specific scope.

<clauses> represents a set of clauses, where each clause is separated by white space:

```
clauses ::= <clause> [<clause> [<clause>]]
```

The clause may be used to define a parameter or a quantitative notation useful for a pragma_directive. Each clause is comprised by a name and a list of variables separated by commas:

```
clause ::= <name> [(<variable>[,<variable>...])]
```

The order in which the clauses appear is not significant. The *variable* identifier's, if any, shall be used to define only variable parameters.

OpenMP Domain

We adopt a subset of the OpenMP pragmas to describe parallelism (task partitioning and thread parallelism). OpenMP pragmas are denoted by the *pragma_scope* **omp**. Note that the OpenMP syntax is used to annotate how tasks are partitioned. Inter-thread synchronization is not used except for barriers, as the hArtes specifications explicitly ask for extraction of independent threads with a load-execute-commit behaviour.

Parallel Construct Program code supposed to run in parallel is introduced by the pragma:

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
{
      structured-block
}
```

This #pragma creates a team of threads. The thread that first encounters this pragma becomes the master of the new team. Note that all the spawned threads execute the code in the structured-block. Code inside square brackets [ ] denotes optional elements.

Even if the OpenMP specs provide support for different clauses, we only consider the *num_threads* and *default(shared)* clauses: the former to express the number of threads spawned for the parallel region, the latter to explicit the only supported variable management among the spawned threads. Other clauses are not supported. It is worth noting that if unsupported clauses are present in the original source code, the parallelism extraction analysis is aborted, since the original semantics could be changed. Note also that without the *num_threads* clause we cannot a priori determine the number of threads that will be created, since the behaviour is compiler dependent. Nested *#pragma omp parallel* constructs are supported: each thread that reaches this nested pragma becomes master of a new team of threads. Note that at the closing bracket } of the *#pragma omp parallel* there is an implicit barrier that joins all the spawned threads before returning control to the original master thread.

Worksharing Constructs OpenMP allows the use of work-sharing constructs inside a parallel region to distribute the execution of the code in the parallel region to the spawned threads. A work-sharing region must bind to an active parallel region: worksharing constructs are thus introduced only inside *#pragma omp parallel* constructs. OpenMP defines the following work-sharing constructs:

- loop (**for**) construct
- **sections** construct
- **single** construct

At the moment, we only support the **sections** worksharing construct. This is used to express the sections of code running in parallel. Each structured block is executed once by one of the threads in the team. The other construct are considered as invalid. The sections worksharing constructs is declared with the following pragma:

```
#pragma omp sections [clause[[,] clause] ...] new-line
{
    structured-block
}
```

Note that at the moment these clauses are ignored by the partitioning tool. Inside the brackets { } the different parallel blocks of code are declared with the pragma:

```
#pragma omp section
{
    structured-block
}
```

Notice the lack of the s at the end of the section clause.

This is how *#pragma omp sections* combines with *#pragma omp section*:

```
#pragma omp sections [clause[[,] clause] ...] new-line
{
   [structured-block]
   #pragma omp section new-line
   {
      [structured-block]
   }
   #pragma omp section new-line
   {
      [structured-block]
   }
   ...
}
```

Note that the first *#pragma omp section* can be omitted, since it is always assumed that the following structured code is executed by one thread only. The following *#pragma omp section* constructs must instead be present to signal that the structured blocks of code below will be executed by other threads. Note that nested parallel regions are always possible. So a nested *#pragma omp parallel* can be declared in the structured code belonging to a *#pragma omp section*.

Barrier Annotation Synchronization points (barriers) for parallel region and for sections are implicit at their end. This means that a barrier is present at the closing bracket } of each *#pragma omp parallel*, *#pragma omp sections* and *#pragma omp parallel sections*.

Anyway the explicit pragma is supported by our tool for additional information inside parallel regions and sections. The format is:

```
#pragma omp barrier new-line
```

Note that OpenMP allows the use of this pragma at least inside a first level parallel region. Of course it can be used inside *#pragma parallel sections* (which binds to a parallel region) to wait for the termination of certain structured blocks of code before starting the others. When this construct appears all the threads spawned to execute the binding parallel region must reach the barrier before proceeding. Note anyway that since implicit barriers are present at the end of parallel regions and sections, the code can be written without using this pragma, without losing expressive power. Since nested parallel regions are possible, the barrier construct binds to the innermost region.

Profiling Domain

The profiling domain is used to capture performance information about specific elements of the program (functions, loops, etc.). In general, the `#pragma` profiling directive must appear in the line before the construct it refers to. Note that white spaces (newlines, spaces and tabulation characters) may exist between the pragma and the corresponding C construct.

The notation used for this directive is: *#pragma profile* data where data contains the performance measures, defined by the following regular expression:

```
data := measure_name(mean,variance)
        [measure_name(mean,variance)]*
```

Profiling information is structured in clauses. These clauses contain statistical measures on the performance of the application, in the sense that both mean and standard deviation values should be given for each measure. This is due to the fact that application behaviour is, in general, dependent on the input data.

So far two measures are used:

- *num_calls*
- *time*

The former indicates the number of times the function is called while the latter expresses the running time of the program which was spent in the function in micro seconds.

The *#pragma profile* clause must appear immediately before the function it refers to; at most there can be blank spaces (newline, spaces or tabulation characters) among them. This clause can either be inserted before the function prototype or before the function body; in case both of the notations are present, only the one before the prototype will be considered Example:

```
#pragma profile num_calls(5,0.8)
void f1();
```

means that function f1 is called an average of 5 times and the standard deviation of the measurements is 0.8.

```
#pragma profile time(30,0.02)
void f1();
```

means that 30 micro seconds of the execution time of the program is spent is executing function f1; the standard deviation is 0.02

```
#pragma profile time(30,0.02) num_calls(5,0.8)
void f1();
```

is just the combination of the previous two notations; note that the order between time and num_calls is not important.

```
#pragma profile time(30,0.02) num_calls(5,0.8)
void f1()
{
.........
.........
}
```

is the same notation seen before, but this time used before the function body and not the function prototype. The *#pragma profile* clauses are not mandatory, but the information they carry may be useful to improve the partitioning process. The information depends of course on the platform used. The profile information is given for the GPP processor, which is the processor that contains the MASTER element in the architecture description file.

Mapping Domain

The mapping domain is used to instruct the backend compilers how tasks are mapped to different processing elements. We use two pragma directives:

- **generation pragmas**, used to mark that, an implementation needs to be built for one particular functional component (like FPGA or DSP)
- **execution pragmas**, used to indicate which functional component or which specific implementation a particular function is mapped for being executed.

The generation pragma is placed immediately before the definition of the function to be compiled by one of the hardware compilers. The syntax is:

```
#pragma  generate_hw  impl_id
```

The <*impl_id*> corresponds to the implementation identifier in the XML file.

The hardware compilers (DWARV, the Diopsis compiler) will read the XML and determine if they need to generate bitstream/code for that function.

The execution pragma is placed immediately before the definition of the function to be executed or offloaded to the hardware component and the corresponding *pragma_directive* is **call_hw**. The syntax is:

```
#pragma call_hw  <component_name> [<impl_id>]
```

The *<component_name>* is the unique identifier of the hardware component where the function will be executed. The *<impl_id>* is the identifier of the chosen implementation associated with the function on the specified component. Additional information about the implementation is in the XML architecture file. If the *<impl_id>* is not specified, it means that no implementations are still available and the mapping gives information only about the target processing element and not the implementation (e.g., specific information about area/performance is not available).

Examples:

```
#pragma call_hw ARM
void proc1 (int* input, int* output1, int* output2)
{
   ...
}

#pragma call_hw FPGA 2
void proc2(int* input, int* output1, int* output2)
{
   ...
}

#pragma call_hw ARM 0
void proc3 (input1, &output1, &output);
```

The function *proc1* will be executed by the ARM component, but no information about a specific implementation is available. Note that, since this pragma has been put before the declaration, it means that all the instances of the *proc1* function will be mapped in this way. The function *proc2*, instead, will be executed on the component named FPGA, with the implementation identified by the id 2. In this case, the synthesis toolchain has also information about the corresponding implementation that has to be generated. In the other hand, if you desire to specify a different mapping for each call of the function, you have to insert the pragma immediately before the related function call. In this example, only the specified *proc3* function call will be implemented on ARM with the implementation with id 0 and no information is given about the other calls of the same function in the program.

Issue Domain

These clauses contain general information or issues for the application. An issue could be the desire for the programmer to exclude a routine from the partitioning process. The notation used for this directive is: *#pragma issue* and a list of directives to be applied to the function that follows:

```
#pragma issue [directive[[,] directive] ...] new-line
```

So far only the *blackbox* directive has been considered. This directive forces the related routine to be excluded from the partitioning process. It can be used by the

programmer to indicate that the function will not been partitioned. This directive is often applied to functions containing I/O since it is problematic to extract parallelism when they are involved. For this reason the IO regions (data input and data output) should be marked with the *#pragma issue blackbox* directive. In this way, these two regions will be excluded from parallelism extraction and a better performing code will be obtained.

The *#pragma issue* clause must appear immediately before the function it refers to; at most there can be blank spaces (newline, spaces or tabulation characters) among them. This clause can either be inserted before the function prototype or before the function body; in case both of the notations are present, only the one before the prototype will be considered.

Example:

```
#pragma issue blackbox
void proc4 (int* input, int* output1, int* output2)
{
   ....
}
```

In this case, the routine *proc4* will be considered as a black-box and no partitioning is tried to be extracted from it.

2.4 Algorithm Exploration

The Algorithm Exploration Toolbox (AET) has the role of enabling the high level description of the application, which shall be translated in C code. The main aim of the AET is to provide tools with two basic functionalities:

- Assist the designers in tuning and possibly improving the input algorithm at the highest level of abstraction in order to easily obtain feedback concerning the numerical and other high-level algorithmic properties.
- Translate the input algorithms described in different formats and languages into a single internal description common for the tools to be employed further on. In the hArtes tool-chain, this internal description is done in C language.

The Algorithm Exploration Toolbox, in particular, deals with high-level algorithms design tools. Its main goal is to output a C description of the input algorithm complemented by optional specification directives reflecting the algorithmic properties obtained from the tools.

This ToolBox translates the multiple front-end algorithmic entries considered into a single unified C code representation.

In these terms the most simple AET Tool may be just a traditional text editor, for C programming, possibly with the typical features such as syntax highlighting, automatic indenting, quick navigation features, etc. The user which is writing C code to be processed by the hArtes tool-chain, differently from other C programmers, has the opportunity to enrich the code with additional information that may be useful

in the activity of task partitioning, code profiling and code mapping. Nevertheless, the syntax that has been defined in the hArtes project, shall be based on C pragma notation, and then compatible with the use of a typical C text editor.

C code programming, anyway, is often not the most efficient way of designing the application algorithms. The hArtes AET toolbox offers the opportunity of describing the application also using both graphical entry and computational-oriented languages.

The hArtes Consortium decided, since the beginning, to adopt NU-Tech as Graphical Algorithm Exploration (GAE) solution and Scilab as computation-oriented language. Later on this chapter the specific issues concerning the integration of these tools in the hArtes ToolChain will be analyzed.

2.4.1 Scilab

Scilab is a scientific software package for numerical computations providing a powerful open computing environment for engineering and scientific applications.

Scilab is an open source software. It is currently used in educational and industrial environments around the world.

Scilab includes hundreds of mathematical functions with the possibility to add interactively programs from various languages (C, C++, Fortran...). It has sophisticated data structures (including lists, polynomials, rational functions, linear systems...), an interpreter and a high level programming language.

Scilab supports the following features:

- 2-D and 3-D graphics, animation
- Linear algebra, sparse matrices
- Polynomials and rational functions
- Interpolation, approximation
- Simulation: ODE solver and DAE solver
- Xcos: a hybrid dynamic systems modeler and simulator
- Classic and robust control, LMI optimization
- Differentiable and non-differentiable optimization
- Signal processing
- Metanet: graphs and networks
- Parallel Scilab
- Statistics
- Interface with Computer Algebra: Maple package for Scilab code generation
- Interface with Fortran, Tcl/Tk, C, C++, Java, LabVIEW

Overview

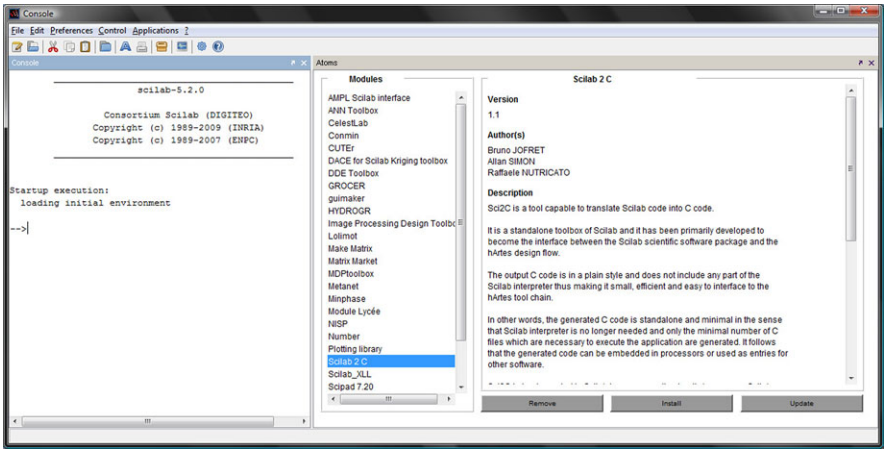
Scilab2C is a tool capable to translate Scilab code into C code.

It is a standalone toolbox of Scilab and it has been primarily developed to become the interface between the Scilab scientific software package and the hArtes design flow.

The output C code is in a plain style and does not include any part of the Scilab interpreter thus making it small, efficient and easy to interface to the hArtes tool chain.

In other words, the generated C code is standalone and minimal in the sense that Scilab interpreter is no longer needed and only the minimal number of C files that are necessary to execute the application are generated. It follows that the generated code can be embedded in processors or used as entries for other software.

Scilab2C is a Scilab toolbox available through Atoms (AuTomatic mOdules Management for Scilab) and can be directly installed/used in the Scilab Development Environment.



From Scilab Script to C Code

The aim of the tool is to give an easy path, from Scilab code down to C code, to a user who may not have any skills on low level language programming but who wants to have an accelerated execution of its high level representation of the algorithm.

Scilab takes advantage from:

- an easy high level programming and testing environment.
- C code for quicker execution and possibly other optimizations (hardware acceleration, parallelization, ...).

Scilab Scripting Language Scilab provides a powerful language to exploit those capabilities:

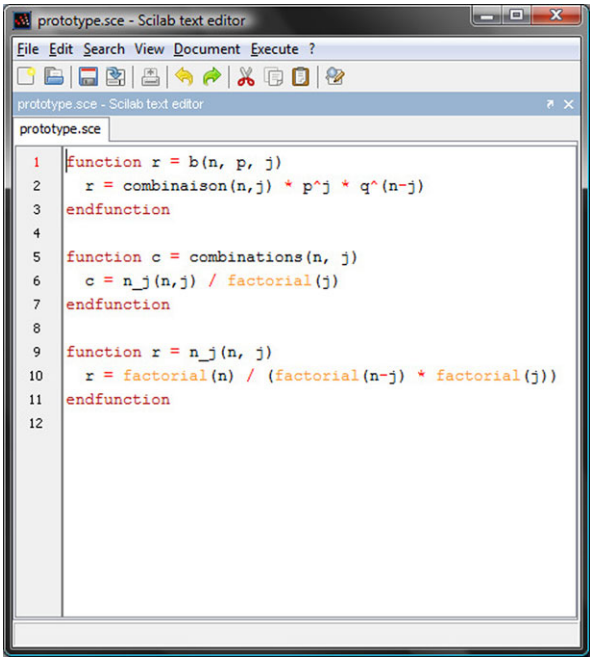
- High level, non-typed and non-declarative programming language: the user does not need to take care about memory allocation, variable type declaration and other programming habits C programmers are used to. Moreover Scilab is a non-declarative language which means the user is allowed to use variables without declaring (nor typing) them before.
- Lazy syntax and control structure instructions: with respect to other programming languages Scilab has a different way to write control structures (if/then, for, func-

tion) that will give the user some freedom writing their scripts/function avoiding the syntax strictness of C code. Nevertheless Scilab language is as rich as other programming language regarding control capabilities: if, for, while, select, try, ...

- Matrix oriented language with an extended choice of standard mathematics computations: Scilab provides and extended set of standard mathematics capabilities, allowing the user to apply mathematical algorithms and will be able to easily manipulate the obtained results.

To sum it up the user can write any algorithm, found in a publication or any paper, using Scilab scripting language, and the result will "look like" the original except that it can physically run on a PC and then be tested, improved, etc. (see figure below).

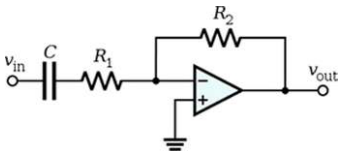
$$
\begin{aligned}
b(n, p, j) &= \binom{n}{j} p^j q^{n-j} \\
\binom{n}{j} &= \frac{(n)_j}{j!} \\
(n)_j &= n \cdot (n-1) \cdots (n-j+1) \\
&= \frac{n \cdot (n-1) \cdots 1}{(n-j) \cdot (n-j-1) \cdots 1} \\
&= \frac{n!}{(n-j)!j!}
\end{aligned}
$$



Example: High-pass filter

Let us consider a simple example like this high-pass filter, the following equations can be extracted:

$$\begin{cases} V_{out}(t) = I(t) \cdot R \\ Q_c(t) = C \cdot (V_{in}(t) - V_{out}(t)) \\ I(t) = \frac{dQ_c}{dt} \end{cases}$$

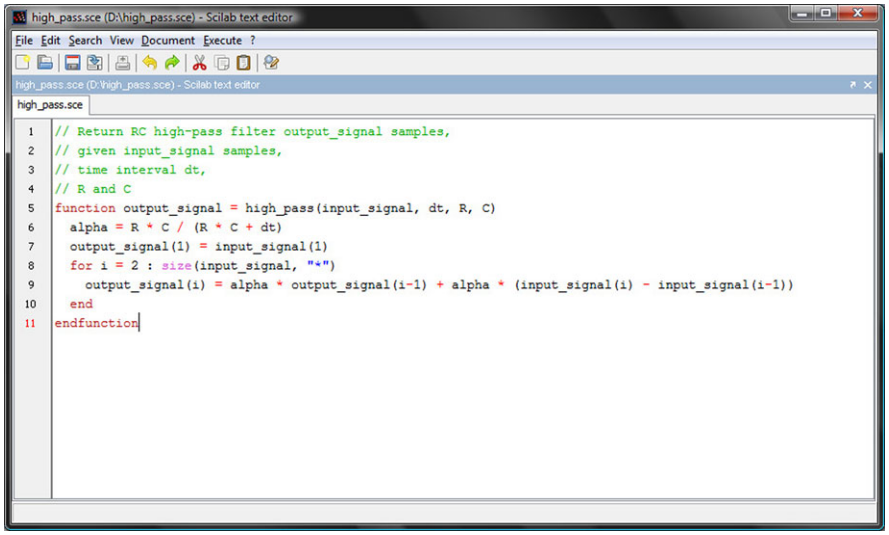


This equation can be discretized. For simplicity, assume that samples of the input and output are taken at evenly-spaced points in time separated by Δ_t time.

$$V_{out} = RC\left(\frac{V_{in}(t) - V_{in}(t-\Delta_t)}{\Delta_t} - \frac{V_{out}(t) - V_{out}(t-\Delta_t)}{\Delta_t}\right)$$

$$\begin{cases} V_{out}(i) = \alpha V_{out}(i-1) + \alpha\,(V_{in}(i) - V_{in}(i-1)) \\ \alpha = \frac{RC}{RC+\Delta_t} \end{cases}$$

We can now easily implement this high-pass filter using Scilab:

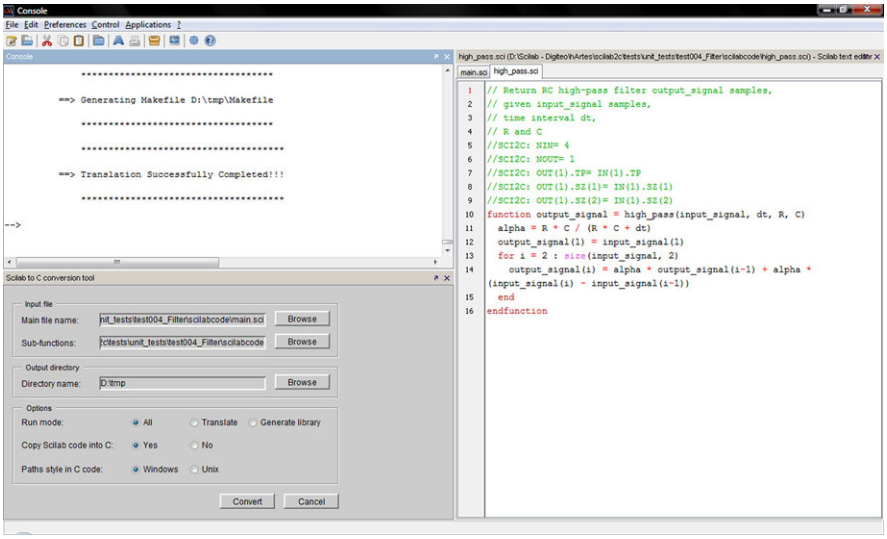


C Code Generator The output C code produced by the tool is written in plain C style, so it can be easily manipulated by optimizing tools available in the hArtes toolchain.

The generated code is standalone and minimal in the sense that Scilab interpreter is no longer needed and only the minimal number of C files, which are necessary to execute the application, are generated and linked. The generated code can then be embedded in processors or used as entries for other software.

As indicated in the following figure, from Scilab the user only has to:

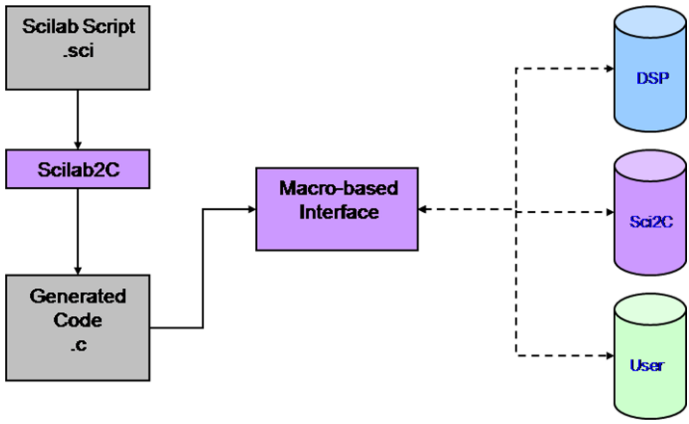
1. annotate the Scilab functions in order to specify the size and type of the output arguments;
2. (optional) specify the precision to be used for the data;
3. launch a GUI to generate the corresponding C-Code.



Using Generated C Code The generated code follows some clear naming rules that will allow the user to link this code with the given library or with a dedicated one (DSP, GPU, . . .).

Each time Scilab2C finds a function call to translate, it automatically generates an explicit function name containing information about input and output types and dimensions.

According to the naming rules, the user will be able (see figure below) to link its generated code with the “implementation” he considers as the best:



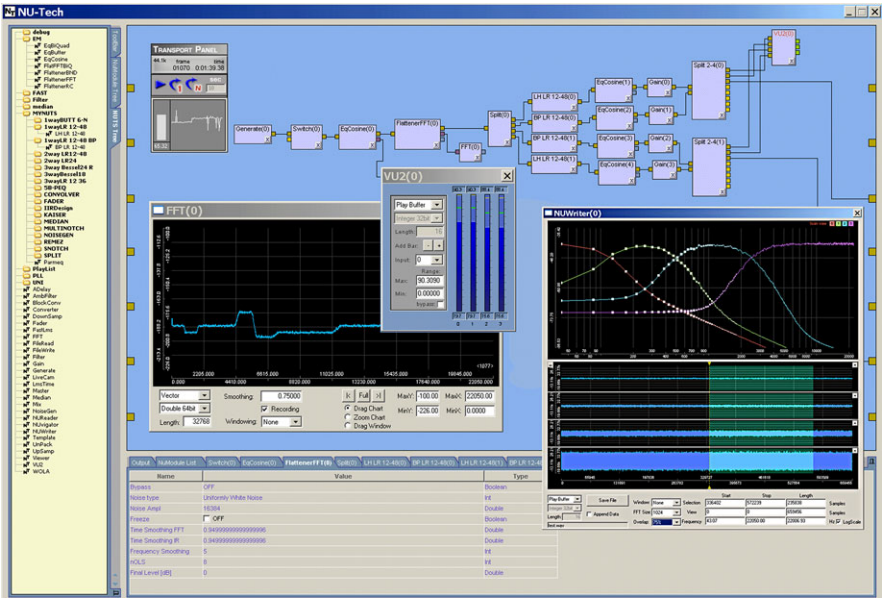


Fig. 2.2 NU-Tech graphical user interface

2.4.2 GAETool

NU-Tech Overview

Starting from the Leaff NU-Tech core the GAE Tool has been designed to play a central role as a starting point in the hArtes tool chain.

NU-Tech is a platform aimed to real-time scenarios, where a strict control over time and latencies is paramount. It represents a powerful DSP platform to validate and real-time debug complex algorithms, simply relying on a common PC. If the goal is implementing a new algorithm, no matter what kind of hardware the final target will be, NU-Tech offers all the right benefits.

During the early stage of the development, it is possible to evaluate the feasibility of a project saving time and money.

The user interface is simple and intuitive, based on a plug-in architecture: a work area hosts the NUTSs (NU-Tech Satellites) that can be interconnected to create very complex networks (limited only by the power of the computer). The graphical representation of a NUTS is a white rectangle with a variable number of pins, and it is the elementary unit of a network. From a developer point of view it is nothing but a plug-in, a piece of code compiled as a DLL. The overview of NU-Tech graphical user interface is show in Fig. 2.2.

A basic set of logical and DSP NUTS is provided and an SDK shows how to develop user's own Satellites in few steps in C/C++ and immediately plug them into the graphical interface design environment. That means the user can strictly focus on his piece of code because all the rest is managed by the environment.

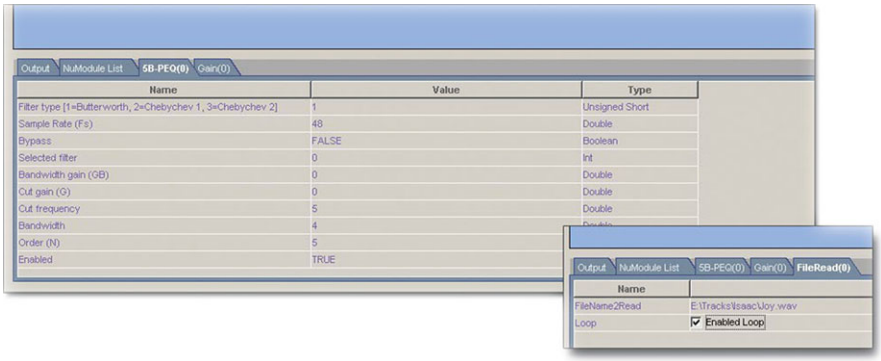


Fig. 2.3 NU-Tech RealTime Watch

The core of NU-Tech has been thought in a modular way so to *connect* to the external world by means of interchangeable drivers. For audio real-time applications ASIO 2.1 has been adopted, providing compatibility with all soundcards supporting the protocol.

A switch to a different driver turns NU-Tech into another tool but NUTSs are still there and the user can always take advantage of their functionalities. New drivers will be continuously added in the future giving the platform even more flexibility.

NUTSs are not compelled to provide a settings window in order to change algorithm parameters and for hArtes needs they should not. To ease the developer in quickly creating new NUTSs without having to deal with GUI programming, NU-Tech provides a window called "RealTime Watch" to be associated to each NUTS. In brief, the developer can choose, by code, to *expose* some NUTSs' internal variables on this window, and effectively control his plug-in. The window is nothing but a tab on the bottom Multitab pane that automatically pops up when at least one parameter is exposed by the developer. When the user double-clicks on a NUTS, the associated tab (if any) immediately jumps in foreground.

A *Profiling Window* (see Fig. 2.3) has been designed in NU-Tech indicating:

- Time Consumption of all the NUTSs Process Functions
- Percentage of time slot (ex. FrameSize/Fs) used for Processing

NU-Tech profiling Window is shown in Fig. 2.4. NU-Tech distinguish between audio and video processing. This could be useful in a scenario where audio and video processing are performed by two different dedicated hardware.

GAETool Overview

The Graphical Algorithm Exploration Tool is a new feature of the NU-Tech Framework. Its main purpose is to produce a C description of an application designed in NU-Tech as a network of functional blocks (NUTSs) interconnected with each other and interacting with the PC inputs/outputs. The general overview of GAETool is shown in Fig. 2.5.

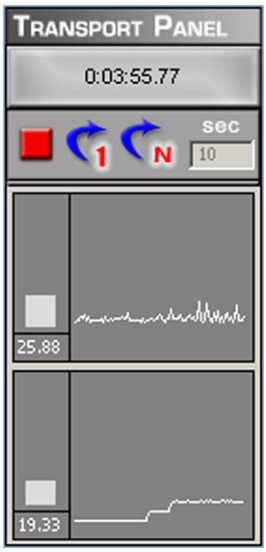


Fig. 2.4 NU-Tech Profiling information

Developing an application with NU-Tech plus GAETool feature is the most straightforward way to take advantage of the hArtes toolchain benefits. The user can use the library of CNUTS to build the core of his application, use the guidelines to code his own algorithm and benefit by the graphical NUTS to get visual feedbacks, plot diagrams and, most of all, real-time tune his parameters in order to obtain satisfactory results and start the porting to the hArtes hardware.

In order to produce a C description of a network of NUTS the source code of each block should be available to the tool. Moreover some specific guidelines should be followed when coding a NUTS in order to let the tool work properly.

CNUTS blocks have been introduced for this purpose. A CNUTS is nothing but a NUTS coded following the hArtes CNUTS specifications. A CNUTS is a .dll that must export some functions in order to be considered by NU-Tech a valid CNUTS. Very simple and easy to follow guidelines have been given to those willing to code a CNUTS. Each CNUTS is characterized by three main functions:

- Init(...): executed during CNUTS initialization phase.
- Process(...): the main processing, starting from input data and internal parameters produces output values.
- Delete(...): frees CNUTS allocated resources.

Any algorithm should consist of these three functions. NUTSs and CNUTSs programming therefore relies on the following structure:

- LEPlugin_Init: Called by the host when streaming starts. Initialization code should be here included.
- LEPlugin_Process: Called by host during streaming to process input data and pass them to the output. Processing code should be placed here.

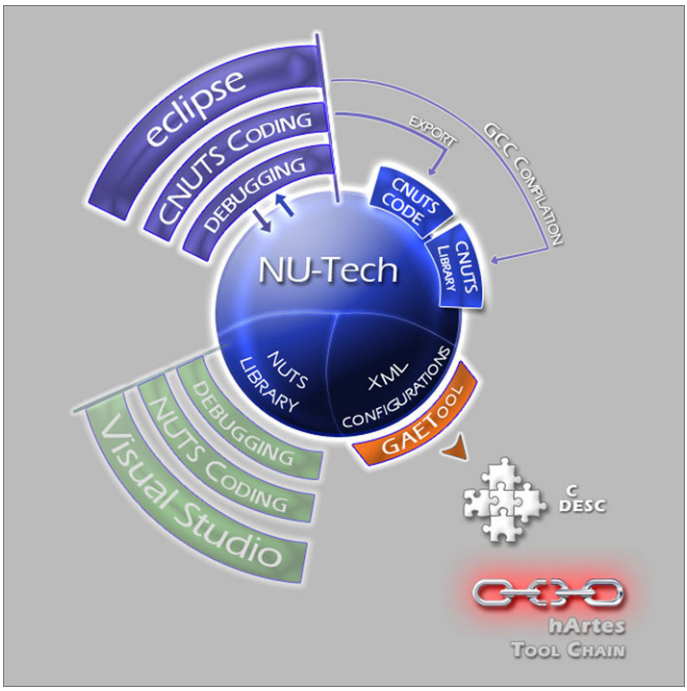


Fig. 2.5 GAETool

Parameters:

– Input[H]: Pointer to an array of pointers each identifying a PinType structure. Each structure contains information about the type of data of the related input.
– Output[P]: Pointer to an array of pointers each identifying a PinType structure. Each structure contains information about the type of data of the related output.

- LEPlugin_Delete: called by the host when streaming is stopped. It contains deinitialization code.

GAETool is based on:

- Eclipse Ganymede version
- GNU Toolchain (MinGW 5.1.4 + GDB 6.8)
- GCC 3.4.5 compiler to compile CNUTS

GAETool is mainly composed by three different features:

- an Eclipse Wizard: to help the user to create a CNUTS from scratch;
- an Eclipse Feature: to export coded CNUTS to NU-Tech environment;
- a NU-Tech feature: to generate C code of CNUTS applications.

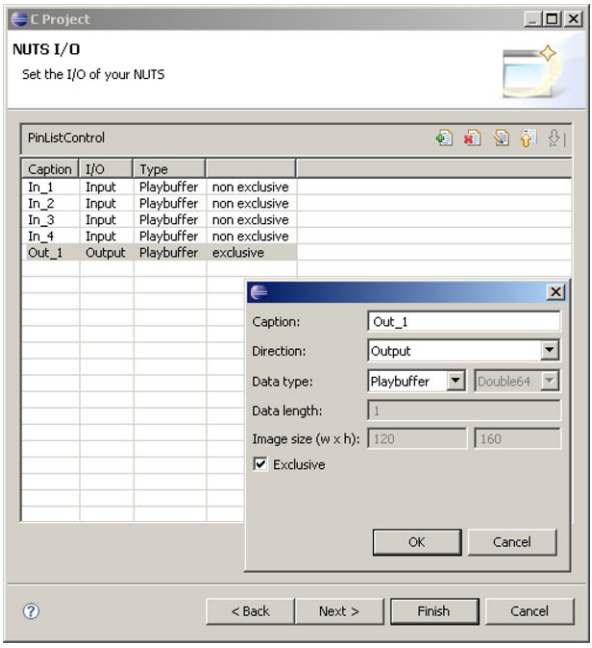


Fig. 2.6 GAETool Eclipse Wizard

GAETool Eclipse Wizard

A snapshot of GAETool Eclipse wizard is shown in Fig. 2.6. When creating a new C Project CNUTS template can be used to start a new CNUTS from scratch. The wizard follows the user during the creation phase and helps him defining:

- basic Settings (name, author, etc.);
- CNUTS mode: NUTS can work in ASIO, DirectSound, Trigger and offline mode;
- processing settings: which kind of processing flow to use (Audio, Video or MIDI)
- NUTS I/O: defining each single input/output of the CNUTS, its name, type and properties
- RTWatch settings: defining which variables should be monitored in realtime

As a result the wizard generates a complete CNUTS projects with all source files needed to correctly compile a CNUTS. All needed functions to handle all NUTS capabilities are included. The project is ready to host the programmer's code and properly compile using MinGW toolchain.

GAETool Eclipse Feature

Once the programmer compiled his project with no errors he can then export it to NU-Tech environment through the GAETool Eclipse feature. A "Export to GAE-Tool" button is included in Eclips IDE. Pressing the button resulting in a transfer to NU-Tech appropriate folders of CNUTS source code and compiled DLL. The user can now run NU-Tech being able to use his new CNUTS.

GAETool NU-Tech Feature

Once the new CNUTS has been ported to NU-Tech it can be used in a configuration together with other NUTS and CNUTS. At this stage of development one can debug and tune his CNUTS in order to get desired behaviour. Graphical NUTS can be also used to get visual feedback, for example: plotting a waveform or a FFT can really help understanding if an audio algorithm is correctly working. Once the user is satisfied with results he can drop all non-CNUTS and prepare his code to be passed to the next stage: the hArtes toolchain. All he has to do is to click on the GAETool icon in NU-Tech toolbar in order to generate the code of the whole configuration. The code is now ready to be used by the hArtes toolchain.

2.5 Application Analysis

This phase aims at providing information on the current status of the application in order to increase the effectiveness of the design space exploration, both for the partitioning and the mapping. In fact, it can be executed not only at the beginning of the exploration, to provide initial information, but also after the partitioning, to support the decision mapping. Moreover, when the toolchain is iteratively executed, the resulting application from each iteration can be further analyzed to provide an additional support to lead the exploration towards an efficient solution. In particular, two different analyses can be performed to support the toolchain:

1. profiling information can improve the partitioning and reduce the computation time focusing the analysis and the transformations only on the most executed paths;
2. cost estimation for the hardware implementations has to be provided to support the decision mapping in deciding which tasks can fit into the reconfigurable device.

2.5.1 HGPROF: Profiling Applications

In the context of hArtes, Hgprof is used as a preliminary profiling tool. Hgprof uses GNU profiling tools such as gprof [15] and gcov[17] tools. Gprof and gcov are an example of a software profiler that provides functional level profiling. Gprof analyzes the program at functional level and provides various information on the functions. Gcov is a coverage tool and analyzes a program to provide information such as how often each line of code executes. Towards this end, the profiling information from gprof and gcov has been processed to achieve various functional characteristics. Hgprof takes applications (C sources) as an input and it generates profiling information as XML output. Figure 2.7 shows the Hgprof tool flow.

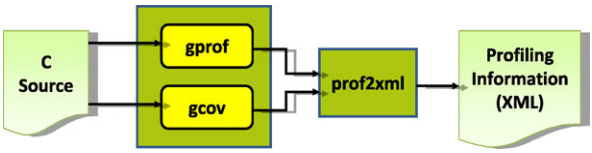


Fig. 2.7 Hgprof Tool Flow

The tool provides the following profiling information:

- Number of times a function is called,
- Average execution time (ms) of each function per call without subroutine calls,
- Average execution time (ms) of each function per call with subroutine calls,
- Number of times a line is executed per function call.

An example of the profile information provided by the tool is shown below. This information is provided in the .xml format under <HGPROF> tag under <PROFILE>.

```
<PROFILE>
     ....
     <HGPROF>
          <MODULE  Filepath = "C:/application" >
               <FUNCTION>
                    <NAME> function1 </NAME>
                    <STIME>134</TIME>
                    <CTIME>1340</TIME>
                    <NCALLS>32</NCALLS>
                    <LINE>
                      <NUMBER> 34 </NUMBER>
                      <NCALLS> 350 </NCALLS>
                    </LINE>
               </FUNCTION>
          </MODULE>
     </HGPROF>
       ....
</PROFILE>
```

where,

<MODULE>: is the header of the module section;
<NAME>: is the name of a function;
<STIME>: is an average execution time (ms) of each function per call without subroutine calls;
<CTIME>: is an average execution time (ms) of each function per call with subroutine calls;
<NCALLS>: is a number of times a function is called;
<NUMBER>: is a line number;
<NCALLS>: is a number of time a line is executed.

2.5.2 Cost Estimation for Design Space Exploration: QUIPU

During the implementation on reconfigurable heterogeneous platforms, developers need to evaluate many different alternatives of transforming, partitioning, and mapping the application to multiple possible architectures. In order to assist developers in this process the hArtes project uses the Design Exploration (DSE) ToolBox to analyze the application at hand and subsequently transform and map the application. The profiling and analysis tools in this toolbox identify compute-intensive kernels, estimate resource consumption of tasks, quantify bandwidth requirements, etc.

The Quipu modeling approach is part of the DSE ToolBox and generates quantitative prediction models that provide early estimates of hardware resource consumption with a focus on the Virtex FPGAs that are used within the hArtes project. Usually estimation of hardware resources is performed during high-level synthesis, but in contrast Quipu targets the very early stages of design where only C-level descriptions are available. By quantifying typical software characteristics like size, nesting, memory usage, etc. using so-called Software Complexity Metrics, Quipu is able to capture the relation between hardware resource consumption and C-code. These software complexity metrics are measures like the number of variables, the number of loops, the cyclomatic complexity of the CFG, and so on. These metrics can be determined in a short time, which makes fast estimates possible. Especially in the early stages of design where many iterations follow each other in a short amount of time, this is an advantage.

The estimates of the quantitative models that Quipu generates help drive system-level simulation, when reconfigurable architectures are targeted. Such modeling and simulation frameworks require estimates for FPGA resource consumption like area and power. Furthermore, Quipu models can provide valuable information for task transformations, e.g. large tasks can be split, while small tasks can be aggregated, etc. Also, task partitioning on reconfigurable heterogeneous platforms is possible when early estimates are available. And of course, manual design decisions benefit from resource estimates.

Quipu Modeling Approach

The Quipu modeling approach uses software complexity metrics (SCM) to represent code characteristics relevant to hardware estimation. Based on a library of kernels, it then extracts a set of SCMs for each every kernel, as well as a calibration set of corresponding actual resource consumption measurements. The dependence of the two datasets is then quantified using statistical regression techniques. The outcome of this process is a (linear) model that can be used to predict the resource consumption of kernels in applications that are being targeted to a heterogeneous platform. There are two tools and a set of scripts that implement the modeling approach as depicted in Fig. 2.8:

- *Metrication Tool* This tool parses the source code and calculates the software complexity metrics needed for prediction. The tool is off-line, although Quipu might also incorporate metrics from run-time in the future.

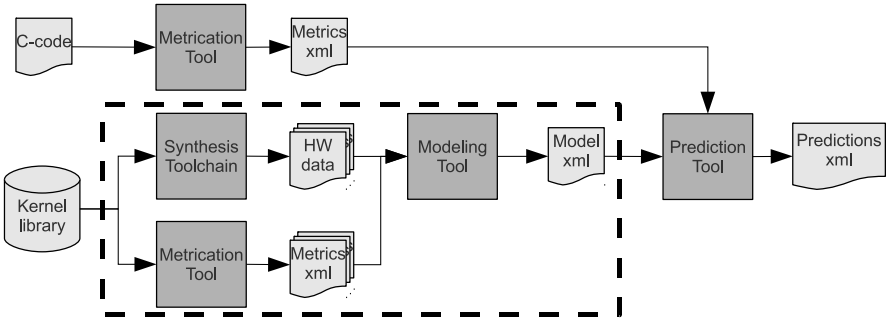


Fig. 2.8 Interaction of tools, scripts, and intermediate results in the Quipu modeling approach

- *Prediction Tool* This tool reads in an earlier generated model file and a set of metric values in order to make predictions for the modeled hardware characteristics.
- *Modeling scripts* These scripts generate calibration sets using a kernel library and help choose and tune the correct regression techniques. However, the process itself remains largely manual as building a good statistical model requires creativity and insight in the problem at hand.

As Quipu is a modeling approach instead of only a model, it is able to generate models for different combinations of architectures, tools, parameters, and hardware characteristics. As an example, Quipu could generate an area estimation model for Xilinx Virtex-5 XC5VLX330 FPGAs using the DWARV C2VHDL compiler and Xilinx ISE 11.1. However, other models are possible, i.e. estimating interconnect or power, targeting Altera Stratix or Actel IGLOO FPGAs, considering optimizing area or speed-up, assuming Impulse-C or Sparc, etc. As long as the predicted measure is to some extent linearly dependent on software characteristics, Quipu can generate specific models. Within the hArtes framework, Quipu has generated models for the Virtex2pro and Virtex4 FPGAs using the DWARV C2VHDL compiler and Xilinx ISE 10.1. We have shown that in those instances Quipu can produce area and interconnect models that produce estimates within acceptable error bounds. Indeed, Quipu trades in some precision for speed and applicability at a high level, but these properties are very important in the highly iterative and changing context of the early design stages. In the following we will discuss Quipu in more detail.

Statistics

In order to model the relation between software and hardware, Quipu utilizes statistical regression techniques. Regression captures the relation between two data sets. Observations of the dependent variables are assumed to be explained by observations of the independent variables within certain error bounds. Consider the following equation:

$$y_i = F(x_{i1}, \ldots, x_{in}) = \hat{F}(x_{i1}, \ldots, x_{in}) + \epsilon_i \tag{2.1}$$

where y_i is the i'th dependent variable, x_{ij} are the independent variables, $F()$ is the relation between the variables, $\hat{F}()$ is the estimated relation, i.e. the regression model, and ϵ_i is the error involved in using the regression model. The linear regression techniques employed by Quipu generate regression models that are of the form:

$$\hat{y}_i = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n + \epsilon_i \tag{2.2}$$

where β_i are the regression coefficients that were fitted to the model. It is clear from this equation that predictions using such models requires the measurement of the independent variables and a fixed amount of multiplications and additions. This is one of the main reasons for the speed of Quipu models.

In order to perform linear regression, a calibration set of measurements of the dependent and independent variables is needed translating to the following equation:

$$\bar{y}_i = \mathbf{X}_i \bar{\beta}_i + \bar{\epsilon}_i \tag{2.3}$$

Using linear regression on this equation the vector of linear coefficients ($\bar{\beta}_i$) can be estimated using the design matrix ($\mathbf{X}_i$) with all SCM observations and the vector of dependent variable observations ($\bar{y}_i$) i.e. the hardware measurements. There are several different regression techniques to choose from. Up to now, we have used Principal Component Regression, Generalized Linear Regression, Partial Least Squares Regression, LEAPS Regression Subset Selection, etc. in the context of Quipu.

A clear advantage of linear regression modeling is that the final prediction consists of a few additions and multiplications in addition to measuring the required SCMs for the kernel. On the other hand linear models are not capable of capturing the non-linear aspects of the transformation of C to hardware, nor can it predict run-time behavior without factoring in other variables that relate to e.g. input data. These issues translate into a relatively large error for QUIPU models. However, at the very early stages of design even a rough indication of resource consumption can be invaluable.

Kernel Library

In order to perform regression we need to build a calibration set of measurements for the dependent and independent variables. For this purpose we have built a library of software kernels that represents a broad range of applications and functionalities. We have shown that a model calibrated using this kernel library can be used for area prediction with acceptable error bounds. Currently, this library consists of over a hundred kernels. Table 2.1 shows the details on the composition of this library. There are no floating point kernels in the library at the time of writing, because when the library was constructed the C2VHDL tools at hand were restricted to integer arithmetic.

Table 2.1 Number of functions in each domain with the main algorithmic characteristics present in each application domain

Domain	Kernels	Bit-based	Streaming	Account-keeping[1]	Control-intensive
Compression	2	x		x	x
Cryptography	56	x	x		x[2]
DSP	5	x	x		x[b]
ECC	6	x	x		x
Mathematics	19				
Multimedia	32	x[b]	x		x
General	15			x[b]	x
Total	135				

[1]Non-constant space complexity

[2]Only some instances in that domain express this characteristic

Software Complexity

In order to quantify the software characteristics of a certain kernel, our approach utilizes software complexity metrics (SCM). The SCMs applied in Quipu capture important features of the kernel source code that relate to the hardware characteristics that it wants to predict. Some examples of SCMs are: Halstead's measures (# of operators, operands), McCabe's cyclomatic number, counts of programming constructs (# of loops, branches, etc.), nesting level, Oviedo's Def-Use pairs, and so on. At the moment, Quipu employs more than 50 SCMs in several categories: code size, data intensity, control intensity, nesting, code volume, etc. The more complex SCMs come mostly from the field of Software Measurement and were originally intended for planning source code testing and project cost estimation. However, we also have introduced some new metrics that specifically try to quantify features related to hardware implementation.

Results

The Quipu modeling approach has been demonstrated to generate usable models for FPGA area and interconnect resources on Xilinx Virtex FPGAs. In Fig. 2.9 we see the predicted versus the measured number of flip-flops. The predictions were made by a Quipu model generated for the Virtex-4 LX200 [59] combined with the DWARV C2VHDL compiler and Xilinx ISE 10.1 Synthesis toolchain optimizing for speed-up. We Observe that the predictions are fairly accurate, however non-linear effects seem to affect smaller kernels more severely. The overall expected error of this model is 27%.

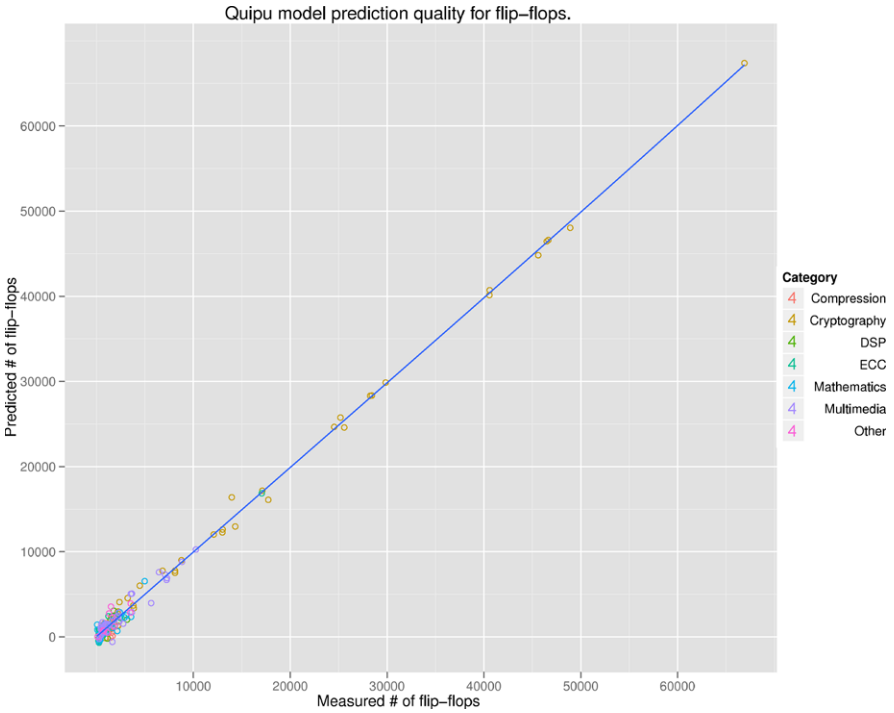


Fig. 2.9 Predicted versus measured number of flip-flops on a Virtex-4 LX200 using the Quipu prediction model. These are validated predictions using leave-one-out validation

2.6 Application Partitioning

MultiProcessor systems are becoming common, not only in the high performance segment, but also in the consumer and embedded markets. Developing programs for these architectures is not easy: the developer needs to correctly decompose the application in order to enhance its performance and to exploit the available multiple processing elements.

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 [22] from TIMA laboratory, SpecC [62] from the UC Irvine, or the Ptolemy environment [7] and the Polis/Metropolis [2] framework from the UC Berkeley. There are also some commercial products, such as CoWare's ConvergenSC, N2C [6], and the meanwhile discontinued VCC [46] 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.

Many of the proposed methodologies for parallelism extraction have been implemented by reusing an existing compiler framework and not by creating a new one,

even if this fact could limit the potential of the proposed approach. For example Jin et al. extend CAPTools [21] to automatically generate OpenMP directives with one [23] or two [24] levels of parallelism; the implementation of their methodology can only be applied to Fortran 77 because this is the only code language that CAPTools can take as input.

Banerjee et al. [3] present a valid overview of the different techniques adopted for parallelization both at the instruction and coarse grained levels.

Most research works, dealing with partitioning of the initial specification, adopt specific intermediate representations. Task graphs are used as intermediate representation by various methodologies that aim at extracting parallelism and consider target architectures different from Shared Memory Multiprocessors. For example Vallerio and Jha [50] propose a methodology for building Task Graphs starting from C source code. These task graphs are then used as input for HW/SW co-synthesis tools that do not necessarily address a fork-join programming model, so they have fewer restrictions in the construction of the Task Graphs.

Girkar et al. [14] propose an intermediate representation, called Hierarchical Task Graph (HTG), which encapsulates minimal data and control dependences and which can be used for extraction of task level parallelism.

Similarly to [14] our intermediate representation is based on loop hierarchy even if we use a more recent loop identification algorithm [43]. Moreover, as in [14] and in [11] we use data and control dependences (commonly named Program Dependency Graph—PDG) as defined in [11] but we target the explicit fork/join concurrency model. In fact, since they do not use this model to control the execution of tasks their work mainly focuses on the simplification of the condition for execution of task nodes (they define a sort of automatic scheduling mechanism).

Luis et al. [33] extend this work by using a Petri net model to represent parallel code, and they apply optimization techniques to minimize the overhead due to explicit synchronization.

Franke et al. [12] try to solve the problems posed by pointer arithmetic and by the complex memory model on auto-parallelizing embedded applications for multiple digital signal processors (DSP). They combine a pointer conversion technique with a new modulo elimination transformation and they integrate a data transformation technique that exposes to the processors the location of partitioned data. Then, thanks to a new address resolution mechanism, they can generate programs that run on multiple address spaces without using message passing mechanisms.

Newburn and Shen [35] present a complete flow for automatic parallelization through the PEDIGREE compiler; this tool is targeted to Symmetric Multi-Processor Systems. They work on assembly code thus their tool can exploit standard compiler optimizations. They are also independent of the high level programming language used to specify the application. The core of PEDIGREE works on a dependence graph, very similar to the CDG graph used by our middle end; this graph encodes control dependences among the instructions. Parallelism is extracted among the instructions in control-equivalent regions, i.e. regions predicated by the same control condition. Applying PEDIGREE to the SDIO benchmark suite, the authors show an average speed-up of 1.56 on two processors.

The work proposed in [39] focuses on extracting parallelism inside loops: each extracted thread acts as a pipeline-stage performing part of the computation of the original loop; the authors fail to specify how synchronization among the threads is implemented. By considering only loops inside benchmarks, a speed-up of 25% to 48% is obtained.

Much work on thread decomposition has been done also for partitioning programs targeted to speculative multiprocessor systems. Speculative multiprocessors allow the execution of tasks without absolute guarantees of respecting data and control dependences. The compiler is responsible for the decomposition of the sequential program in speculative tasks. Johnson et al. [25] propose a min-cut-based approach for this operation, starting from a Control Flow Graph (CFG) where each node is a basic block and each edge represents a control dependence. In their implementation the weights of every edge are changed as the algorithm progresses in order to account for the overhead related to the size of the threads being created.

The clustering and merging phases have been widely researched. Usually, these two phases are addressed separately. Well known deterministic clustering algorithms are Dominant Sequence Clustering (DSC) by Yang and Gerasoulis [60], linear clustering by Kim and Browne [27] and Sarkar's Internalization Algorithm (SIA) [44]. On the other hand, many researches explore the cluster-scheduling problem with evolutionary algorithms [16, 53]. A unified view is given by Kianzad and Bhattacharyya [26], 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.

The approach proposed in hArtes is very similar to the one presented by Newburn and Shen: GNU/GCC is used to generate the input to the partitioning tool thus there is no need to re-implement standard compiler optimizations. In addition to concentrating on extracting parallelism inside control-equivalent regions, several Task Graph transformations have been implemented to improve efficiency and to adapt it to OpenMP needs. In fact, as previously mentioned, we consider the explicit fork/join model of concurrency efficiently implemented by the OpenMP standard; this concurrency model does not require to explicitly define the conditions for which a task can execute.

2.6.1 Partitioning Toolchain Description

The Task Partitioning tool is named **Zebu** and it is developed inside PandA, the HW/SW co-design framework shown in Fig. 2.10 and currently under development at Politecnico di Milano. It aims at identifying the tasks in which the application can be decomposed to improve its performance. It also proposes an initial mapping solution to be refined by the mapping tool.

In particular, the requirement analysis shows that the problem can be clearly separated into two distinct sub-problems:

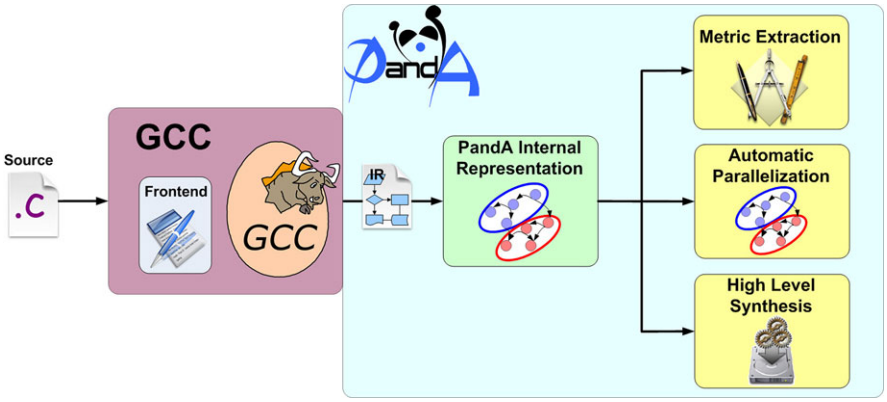


Fig. 2.10 PandA framework

1. in the first phase the tool identifies the parallelism in the application, i.e. it annotates, using task partitioning pragma notations, how it can be decomposed in separate threads, presenting the parallel tasks within the annotated code;
2. in the second phase the tool indicates, using specific mapping pragma annotations, an initial guess concerning which processing element (GPP, DSP or programmable logic) should execute the identified tasks, i.e. where each task should be mapped in the system. The pandA automatic parallelization overview is shown in Fig. 2.11.

The input of the tool is the C source code of the application and the XML file containing information about the target architecture and available data of the performance of the different parts of the application on each processing element. It also contains information about library functions and the header files where they are defined. It can also represent a feedback from the toolchain, that can improve the accuracy of the performance estimations and therefore the quality of the solution identified by the Zebu tool.

The output of Zebu is a C source code annotated with pragmas representing the task partitioning and the mapping suggestions. Note that the tasks identified by the tool will be represented as new functions. Moreover, it also reproduces the XML file, augmented with these new functions. It is worth noting that the internal performance estimations are not reported into the output XML since they are almost related to the internal representation of the tool and, thus, they could not exploited by the rest of the toolchain.

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, creates an efficient partitioning also exploiting internal performance estimation techniques and suggesting a initial guess of mapping, and
- the **backend**, which prints the executable C code annotated with OpenMP [45] and mapping directives.

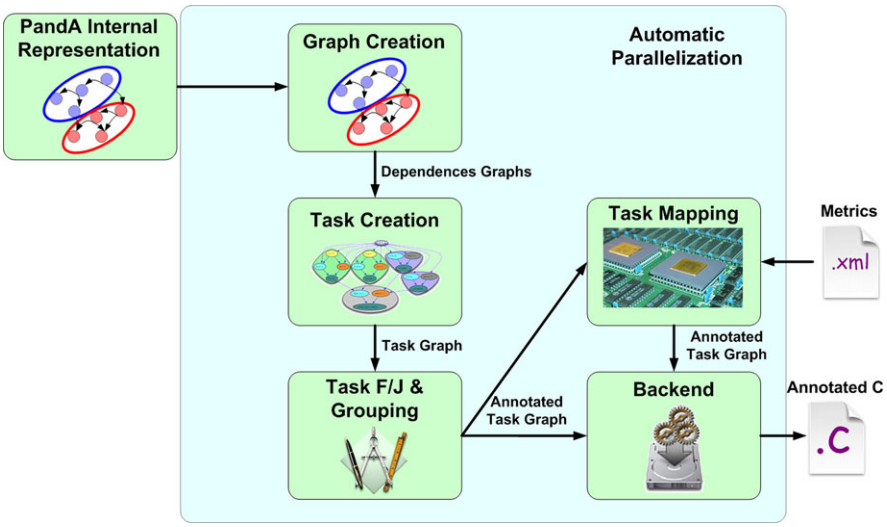


Fig. 2.11 PandA automatic parallelization

Frontend

The frontend does not directly read C source code, but it invokes a slightly modified version of the GNU/GCC compiler and parses the resulting GIMPLE compliant tree-representation. In this way it is possible to control the code optimizations performed by this compiler, avoiding to re-implement them into Zebu. Note that, the analysis performed by Zebu can only take into account target-independent optimizations, such as constant folding, constant propagation or dead code elimination, that have a direct impact on the GIMPLE representation. In fact, since the different parts of the application could be executed by any of the processing elements, based on mapping decisions, target-dependent optimizations cannot be considered at this stage. These optimizations can be enabled or disabled directly passing to Zebu the same optimization flags that would be given to the GNU/GCC compiler. Besides the code optimizations, each original C instruction is converted by GNU/GCC in one or more basic operations and the loop control statements `while`, `do while` and `for` are replaced with `if` and `goto`, and have to be reconstructed in the following.

Parsing: since the Zebu frontend does not directly parse the source code, this part is mainly composed of a wrapper to an internally modified version of the GNU/GCC 4.3 compiler, that exploits the Single Static Assignment (SSA) form as representation. Currently, only few patches have been applied to the original GCC version and most of them concern how the tree, which holds the GCC internal representation, is written to file: what has been done is to make GCC dump, in its debug files, as much useful information as possible. This was necessary since part of its intermediate representation (virtual operands [36]—used for tracking dependencies among aggregate variables and for serialization purposes—, the structure of

```
b[1] = 0
a = b[1] + c;
b[1] = a - 1;
c = d + a;
if (c > 0)
{
   d = b[1] * c;
}
else
{
   d = b[1] - c;
}
b[1] = 2;
printf("%d", d);
```

(a)

```
A:  b[1] = 0;
B:  a_1 = b[1] + c_1;
C:  b[1] = a_1 - 1;
D:  c_2 = d_1 + a_1;
E:  if (c_2 > 0)
    {
F:     d_2 = b[1] * c_2;
    }
    else
    {
G:     d_3 = b[1] - c_2;
    }
H:  d_4 = phi(d_2, d_3);
I:  b[1] = 2;
L:  printf(%d, d_4);
```

(b)

Fig. 2.12 Example of program in original format (**a**) and in the corresponding SSA-form (**b**)

the basic block control flow graph, operands of some special operators, etc.) wasn't originally printed out.

GCC is thus invoked to produce the ASCII file containing the intermediate representation. We chose to dump the intermediate tree representation after as many optimizations as possible have been performed on it by GCC. Ideally the last created tree, after all GIMPLE optimizations are applied to it, should be used. However the tree that satisfies our needs is the tree dumped after the last phase of the phi-optimization since it is still in SSA-form [37]. The SSA form is based on the assumption that program variables are assigned in exactly one location in the program. Multiple assignments to the same variable create new versions of that variable (see Fig. 2.12 for an example). Naturally, actual programs are seldom in SSA form initially, thus the compiler modifies the program representation so that every time a variable is assigned in the code, a new version of the variable is created. Different versions of the same variable are distinguished by subscripting the variable name with its version number. Variables used in the right-hand side of expressions are renamed so that their version number matches that of the most recent assignment. SSA-form is used in the GCC GIMPLE tree both for scalar and aggregated variables, but not in the same way: scalar SSA variables are used directly as operands of GIMPLE operations replacing original operands, while aggregated SSA variables are only used in annotation of GIMPLE statements to preserve their correct semantic and they will be referred as *virtual operands*.

The optimizations that GCC performs on the input code are controlled by our tool for mainly three reasons:

1. To evaluate the improvement of every single optimization on the parallelization flow;
2. Some optimizations could reduce the available parallelism; an example is the common subexpression elimination. In fact, suppose there are two code fragments which do not have any inter-dependences but have a common subexpression; at the end of the partitioning flow it is possible that these two segments

are assigned to two different parallel tasks. This is true only if common subexpression elimination is not applied. Otherwise, the two segments would share the same subexpression: in this way an inter-dependence has been created and this prevents the two segments from being assigned to parallel tasks.

3. Some optimizations and the precision of some analyses are parameterized. For example it is possible to change the size threshold which controls if a loop should be unrolled or not or it is possible to change the maximum number of virtual operands used during alias analysis. Even if these parameters could impact significantly on GCC compilation time, it should be considered that this time is usually only a little fraction of the total time spent in performing the parallelization, so direct impact of these changes on execution time could be ignored. Nevertheless, the choice of these parameters impacts indirectly on the total execution time of the tool since it could change very much the size of the data on which the tool works; these effects on timing must be taken into consideration when deciding which optimizations have to be enabled. For example, loop unrolling increases the number of statements of the specification. In this way there is an obvious trade-off between total execution time and analysis precision; this means a trade-off between the transformation time and quality of the results (in terms of amount of parallelism which could be extracted and therefore in terms of speed-up of the produced code).

After parsing the GIMPLE-compliant tree-representation, two different collections of graphs are built for each function of the original specification. In the first one, a vertex corresponds to a basic block, as defined by GCC itself. In particular, according to this definition, a function call does not start a new basic block. In the second one, instead, a Control Flow Graph (CFG) is created for each C function. Every node of these graphs corresponds to one simple GIMPLE operation and can represent zero, one or more C basic operations. Phi-nodes of real variables or of virtual operands are treated as if they were statements, since, in this way, the sequential behavior can be easily maintained. In the first type of graphs an edge represents dependences or precedences between two basic blocks, while in the second type precedences or dependences between two operations. The type of dependency or of precedence depends upon which particular graph inside the collection is considered. During this preliminary phase, the operations inside each basic block are also annotated with some of the information present in the GCC tree such as the real and virtual operands read and written by each operation (as shown in Fig. 2.13).

A first analysis of the specification is also performed and its results are annotated onto the graphs. This analysis basically retrieves information about the type of each operation and which variables are read/written by each of them. Moreover at the end of this phase, Zebu analyzes the produced CFGs to identify those edges that close a cycle in a path starting from the entry node (entry node is a symbolic node which represents the beginning of the computation flow into each function). The results of these analyses allow Zebu to correctly identify and rebuild cycle constructs in the following phases.

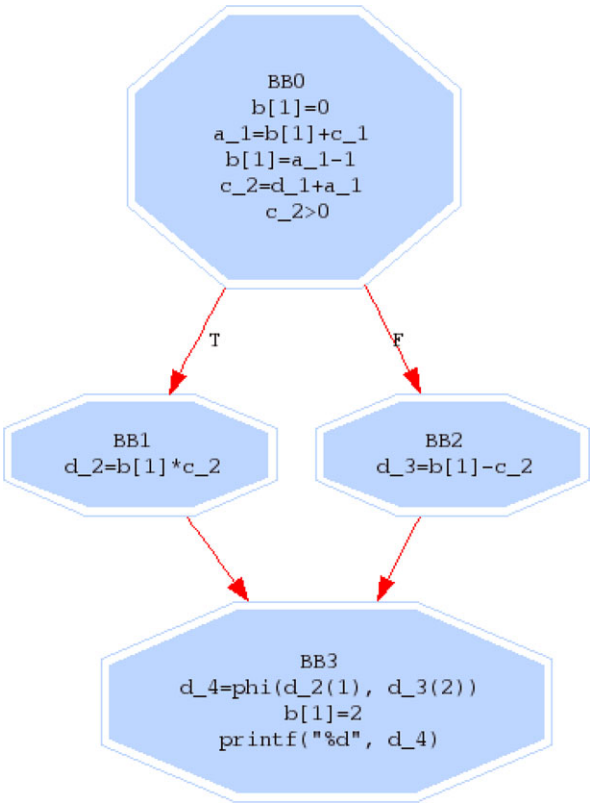


Fig. 2.13 Example of Control Flow Graph of basic blocks

Middle-End

As in traditional compilers, the middle-end can be considered the main part of the compilation flow implemented in Zebu. This phase is mainly composed of three steps:

- the dependence analysis and the manipulation of the intermediate representation;
- the performance estimation and partitioning;
- the mapping onto the target platform.

Dependence Analysis and Manipulation of the Intermediate Representation
The dependence analysis step consists of the analysis of the Control Flow Graphs and of the tree to compute all the dependences between each pair of operations (nodes). Dependences between an operation A and an operation B can be basically of one or more of the following 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. Even if we exploit the Static Single Assignment (SSA) form, these dependences can still occur between pointer accesses and they are necessary to maintain the correct semantic of the application;
- **Feedback Dependence**: operation B depends on the execution of A in the previous iteration of the loop which both the operations belong to.

Additional graphs are thus built to represent these dependences. All these graphs are produced into two different versions, i.e., with or without feedback dependences.

These graphs are produced for each C function of the original 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 Zebu could have a different functionality from the original one.

Before computing data dependences and anti-dependences, alias analysis has to be performed: this is necessary to correctly deal with pointers. An inter-procedural alias analysis model is used. In this way less conservative results than those produced by intra-procedural methods are obtained, despite an overhead in computation time.

In addition to the analyses of the original specification, Zebu also partially manipulates the intermediate representation by applying different transformations. For example it applies further dead code elimination made possible by alias analysis and by loop transformations, such as loop interchange.

Dependence extraction: loop detection is the first analysis performed. The Control Flow Graph of basic blocks is analyzed to detect loops which are classified into reducible or irreducible ones. A loop forest is also built using a modified version of the Sreedhar-Gao-Lee algorithm [43]. Then the Control Flow Graph of GIMPLE operations is built from the Control Flow Graph of Basic Blocks simply by creating sequential connections among the operations inside each basic block and among the operations ending and starting two connected basic blocks. The next step is the creation of the dominator and post-dominator tree of the Control Flow Graph of the Basic Blocks. In particular the second of these trees is used to compute the **Control Dependence Graph** (CDG) of Basic Blocks from which the Control Dependence Graph of operations is easily built. Edges in this class of graphs could be of two types: normal edges or feedback edges. The latter indicates which operations control the execution of the iterations of a loop execution. Moreover, each edge in the CDG is annotated with the condition on which a conditional construct operation controls other operations. Values of these labels can be “true” or “false” for edges going out conditional constructs of type if, or a set of integers for edges going out from switch constructs (each integer represents a case label). In this way all information about control dependences between each pair of operations is extracted and annotated; this information will be used by the backend to correctly write back the intermediate representation to C code.

After the control dependences analysis, the tool starts the computation of data dependences. This is based on the information of variables in SSA-form read and written by the different GIMPLE statements, annotated in the corresponding nodes of the Control Flow Graph. For each pair of definition and use of a variable in SSA-form (either real or virtual), an edge in the **Data Dependence Graph** is added starting from the node of operation which defines the SSA variable and going to the target node where the variable is used. Since also virtual operands are considered, in some cases, additional data flow edges are inserted even if there are no real data dependences. This happens, for example, when a fixed ordering between pair of operations is required to preserve the semantic of the original C source code. For example, the order of the `printf` calls between all functions of a program has to been maintained to force the output of our produced code to be the same of the original one. In this case the serialization of these function calls is forced by adding reading and writing of special fake variables in all corresponding nodes with the same purpose of GCC with *virtual operands*.

Thanks to the fact that scalar SSA-form variables are directly embedded into the GIMPLE code, if different SSA versions of the same scalar variable are dumped back in the produced source code, they are actually different variables thus the anti-dependences (dependences of type write after read and write after write) can be ignored during reordering of the operations: this type of dependences never occurs in pure SSA code. On the other hand, dumping back aggregated variables in SSA-form is more complex; the reasons are the impossibility of writing back directly the phi-nodes of array of variables and the difficulty in managing situations where pointers could point to different variables (so operations could represent define of multiple virtual operands). We decided to maintain the original formulation of GIMPLE operations in all cases: scalar variables will be printed out considering their SSA version (each SSA form of a variable is treated as a different variable), while aggregated variables are printed back using base form and using the SSA-form of virtual operands only to track the dependences. As a consequence, simple data dependences of type read after write are not enough to describe all possible correct orderings between the operations. Consider for example the fragment of code shown in Fig. 2.12(b) and suppose that no optimizations have been performed by GCC. In this case, according to the previous definition, the assignments F and G use the virtual definition of the element of the array defined in the assignment C; then the assignment I redefines that definition.

Hence, according to simple data dependences, it could be possible to execute the assignment I before the F and G ones, that results in a wrong run-time behaviour. In fact, it could happen that, in the following phases of the partitioning flow, these two assignments are clustered into different and parallel threads and so they could be executed in whatever order. This simple example proves that data flow analysis based only on read after write dependences is not sufficient to catch all the actual dependences between operations. For this reason we build the **anti-dependence graph** to take correctly into account all the dependences. As in the previous graph computations, this type of dependences can be derived from the GCC GIMPLE tree. In fact, GIMPLE annotates a virtual definition of a variable with an additional

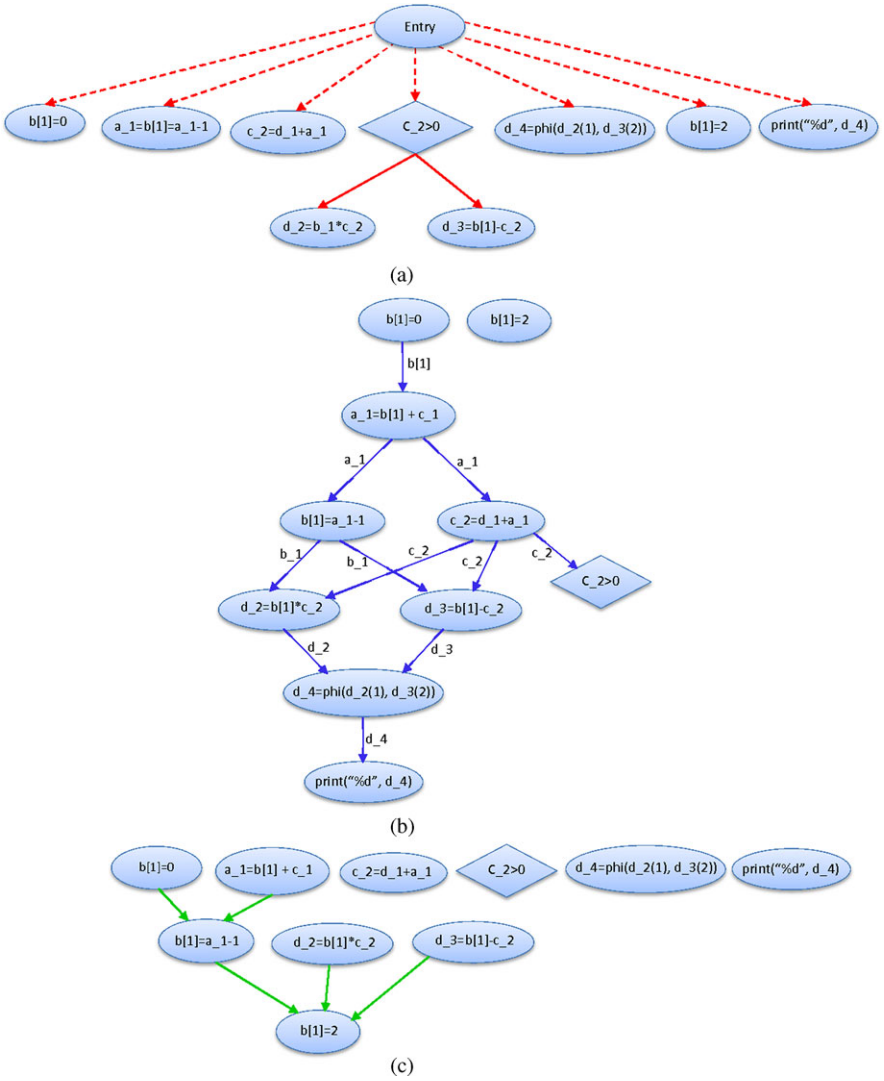


Fig. 2.14 Control dependence graph (**a**), data dependence graph (**b**), anti-dependence graph (**c**)

operand that is the related *killed* definition. In this way, in the previous example, the assignment I is annotated with the information on the killed definition, that is the one defined by the assignment C and used by the F and G ones. Therefore anti-dependence edges can be added between the C and the I assignments (write after write dependence) and between F (and G) and the I (write after read dependence) ones. In this way, the correct behavior can be preserved (see Fig. 2.14).

The next phase of our flow tries to exploit the scalar replacement of aggregates techniques presented in [5] by applying it to the data dependences analysis just performed. The GCC *SRA* optimization is not applied directly because it causes an

increase of the size of the produced source code (indeed it is applicable only to arrays with very few elements) and it reduces its readability. Moreover, in this way it is also possible to select where this replacement is performed: for example this technique could be applied only in critical points where the elimination of some false dependences (added by the compiler in a conservative approach) could help to extract more parallelism.

Combining the Data Dependence, the Anti-Dependence and the Control Dependence Graph, we build the Program Dependence Graph which will be used in the following phases of the task partitioning process (see Fig. 2.15).

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

The first step consists in the analysis of the 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, between loops or by considering the nodes not part of any loop (see Fig. 2.16).

After computing the loop subgraphs, the core partitioning algorithm is executed (refer to Fig. 2.17).

In a similar way to what performed in [35] 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* or *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.

The analysis starts from a generic node in a control-equivalent region with a depth-first exploration. A node is added to the cluster being formed if it is dependent from one and only one node or if it is dependent from more than one node, but all its predecessors have already been added to the current cluster. Otherwise, the cluster is closed and the generation of a new set starts. These operations are iterated until all the nodes in the control-equivalent partition are added to a set.

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.

The result of this partitioning is thus represented by Hierarchical Task Graphs (HTGs). HTGs introduce the concept of hierarchy, and provide an easy representation of cyclic task graphs and function calls.

Fork/Join Task Creation and Task Grouping Since OpenMP [45] is used to annotate the parallelism in the produced C specification, transformations to the

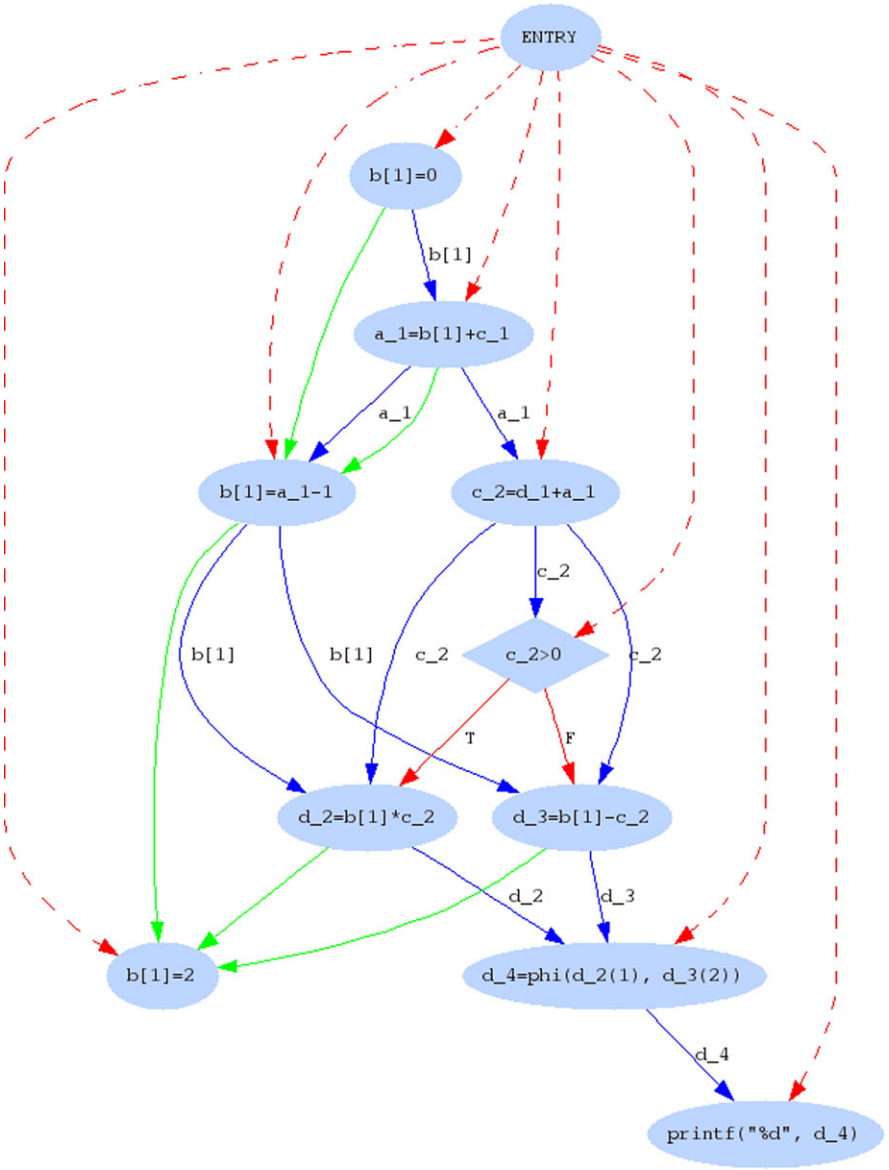


Fig. 2.15 Program dependence graph

task graph are necessary in order to make it suitable for the OpenMP programming model, that is the Fork/Join programming model. This programming model requires that each task spawning threads (called *fork* task) has a corresponding *join task*, which can be executed only after all the created threads have completed their execution. Figure 2.18 shows the leftmost task graph compliant with the fork/join programming model and the one on the right not compliant.

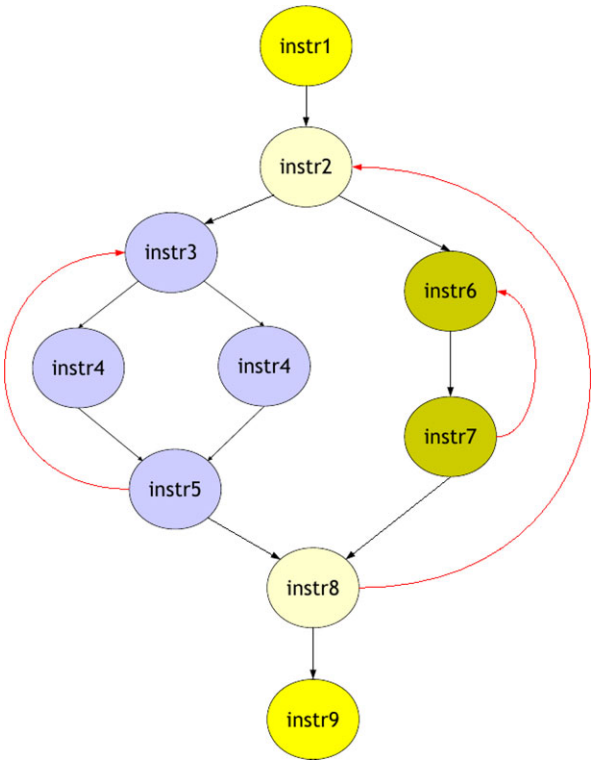


Fig. 2.16 Nodes belonging to different loops are clustered in different subgraphs

The algorithm which transforms a generic task graph into one compliant with the fork/join programming model is composed of two 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. reconstruction of the corrected task graph, that satisfies the tasks' dependences but with a compliant structure.

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 obtained by their concurrent execution.

The **optimization** phase (task grouping) 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

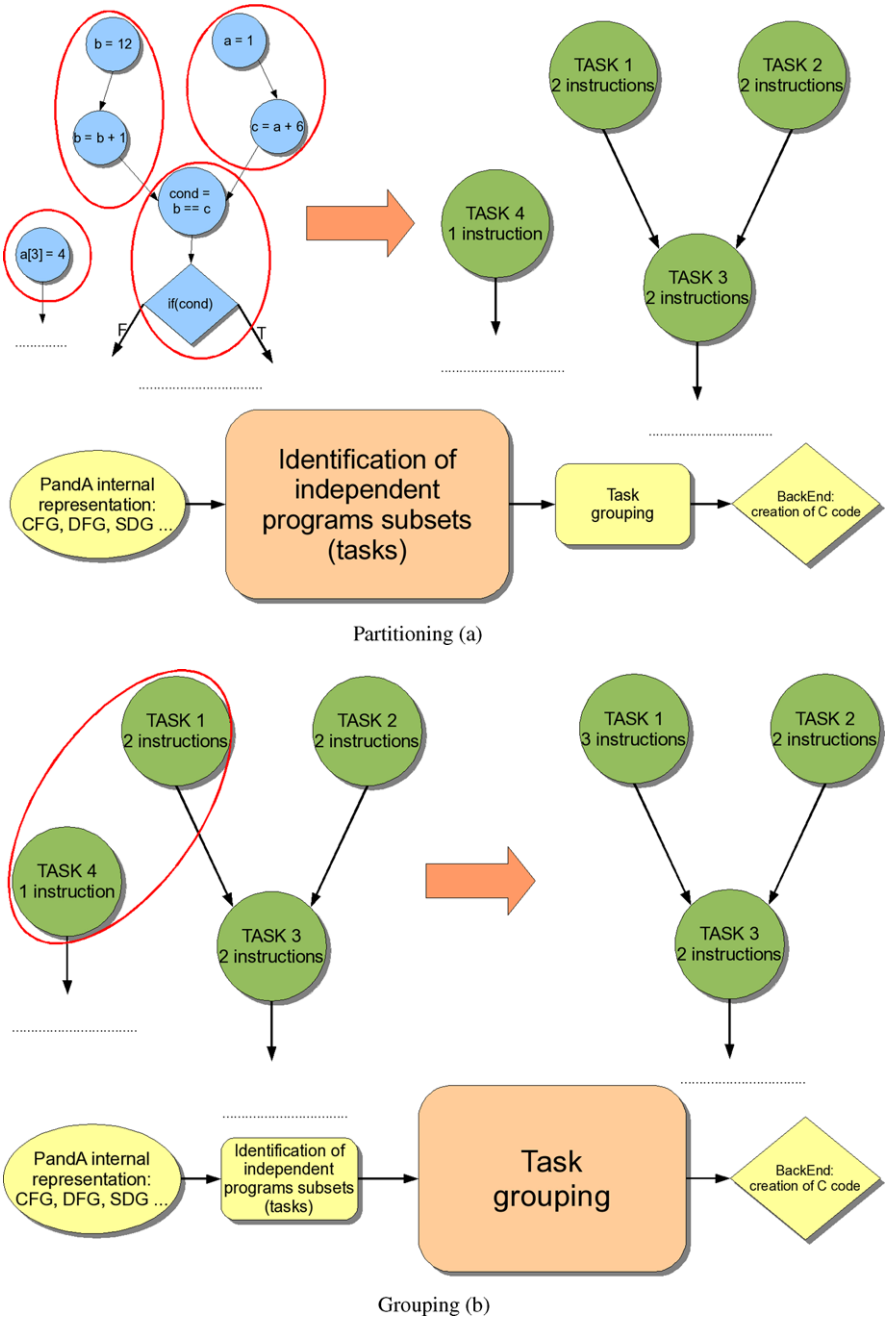


Fig. 2.17 Phases of the parallelism extraction algorithm

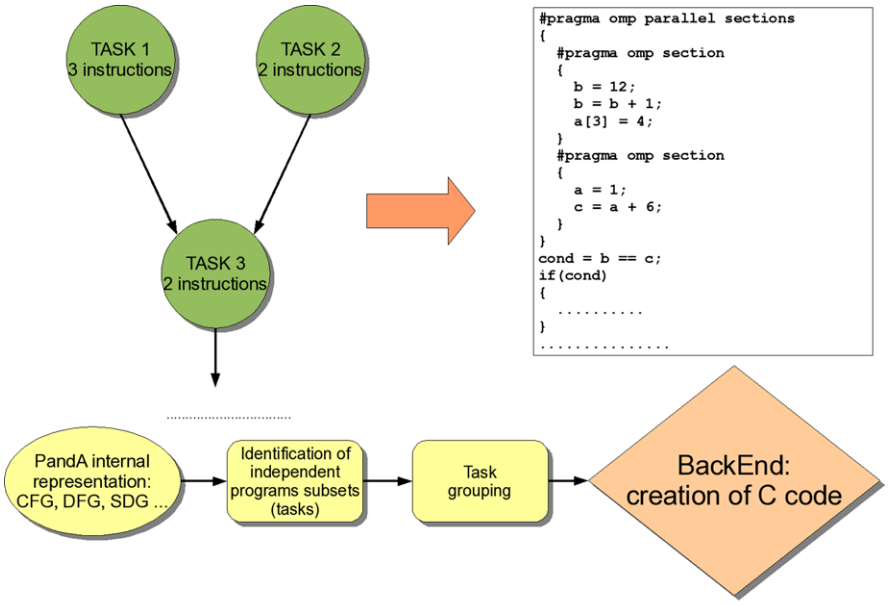


Fig. 2.17 (Continued)

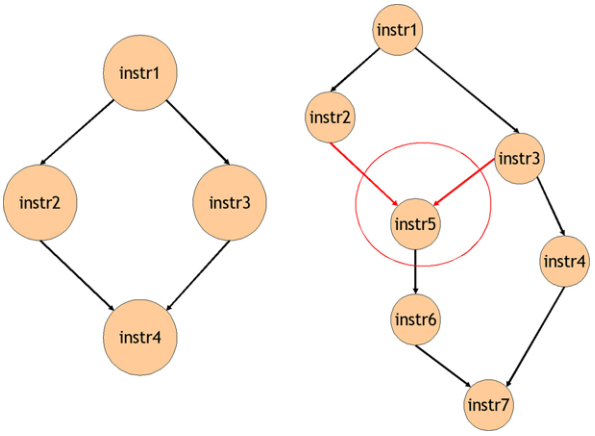


Fig. 2.18 A task graph compliant with the fork/join programming model (on the *left*) and one not compliant

`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 neighbors 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.

Besides the optimization with respect to the fork-join compliance also the estimated target platform overhead is considered. In addition to parallel tasks, sequential ones are also created in order to exploit the heterogeneity of the architecture and obtain an overall speed-up of the application.

Initial Mapping of the Application onto the Target Platform Different approaches have been proposed for mapping the application onto the target platform. In particular, a heuristic optimization methodology that has recently gained interest in dealing with such class of problems for its performance and the quality of the results is the Ant Colony Optimization (ACO) approach.

Starting from the HTG intermediate representation, we apply an Ant Colony Optimization (ACO) approach that explores different mapping combinations. Differently from previous approaches, the proposed solution iteratively builds different combinations of mapping, evaluating the global effects and identifying the decisions that improve the overall performance of the application. Furthermore, it allows the share of the information at the different levels of the hierarchy and it is able to support a large set of constraints that can be imposed by the target architecture, such as the available area for the hardware devices.

We compared [10] this solution with traditional techniques for mapping parallel applications on heterogeneous platforms and the results are shown in Table 2.2. In details, we compared the execution time needed to generate the solution (Cpu column) for the different approaches, namely the proposed ACO, the Simulated Annealing (SA) and the Tabu Search (TS). The quality of the results, in terms of execution time of the emulation platform, has also been reported for the proposed approach, along with the percentage difference for the other solutions with respect to this one. A solution obtained with a dynamic scheduling and allocation has also been reported to demonstrate the need for such methodology on heterogeneous platforms.

Note that the proposed approach systematically outperforms existing methodologies by at least 10% in average and it is also much faster than the other approaches to converge to a *stable* solution, as represented by the differences in terms of computation time.

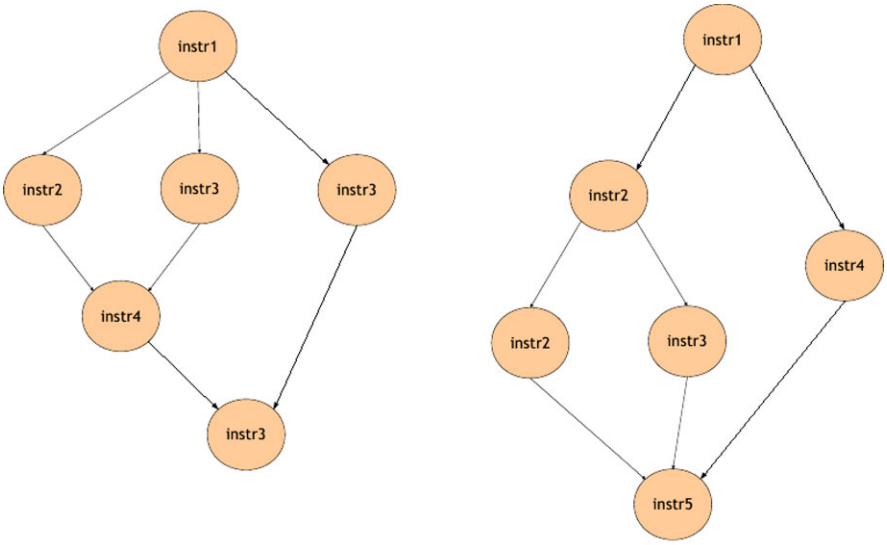
Backend

In the last phase, **Zebu** needs to produce a **parallel executable C** program containing the results of the partitioning and of the initial mapping. In particular, the code is annotated with hArtes pragma notations that express the partitioning in a set of tasks and, for each task, the target processing element suggested for the execution. In the same way, the other annotations, contained in the original specification, are reproduced without modifications in the produced code.

As described in the previous paragraph, in order to be able to use this library and address the MOLEN paradigm of execution, the task graph must strictly adhere to the fork/join programming model and the OpenMP formalism. For this reason, during the production of the concurrent C code, the backend has to perform the following tasks:

Table 2.2 Comparison of results of Ant Colony Optimization (ACO), Simulated Annealing (SA) and the Tabu Search (TS), along with a dynamic policy

Benchmarks	ACO				SA		TS		Dynamic scheduling
	Proposed approach	Cpu (s)	Only mapping	Only scheduling	Mapping scheduling	Cpu (s)	Mapping scheduling	Cpu (s)	
sha	1.72 msec	4.20	2.28	12.14%	8.23%	5.18	6.71%	7.11	29.44%
FFT	13.41 sec	8.12	103.57	108.38%	31.11%	11.89	27.84%	17.20	257.21%
JPEG	0.46 sec	10.67	0.12	5.15%	1.13%	14.63	4.57%	13.07	27.64%
susan	9.31 sec	6.08	0.15	21.96%	4.41%	7.16	7.58%	9.18	21.30%
adpcm coder	1.42 msec	0.20	0.15	7.08%	9.10%	0.22	4.33%	0.25	7.08%
adpcm decoder	1.76 msec	0.19	0.05	4.65%	9.24%	0.23	8.96%	0.21	5.56%
bitcount	0.15 sec	0.10	1.12	1978.77%	11.02%	0.10	14.12%	0.11	2024.77%
	1.14 sec	0.34	0.07	178.07%	35.30%	0.62	29.89%	0.58	178.77%
rijndael	0.81 sec	2.58	2.12	6.30%	3.12%	3.36	1.01%	4.32	3.40%
	8.36 sec	2.72	2.02	6.20%	0.09%	2.91	0.74%	3.10	3.39%
Avg. difference			+11.17%	+232.87%	+11.28%	+27.53%	+10.58%	+45.21%	+255.86%



(a) Fork node sharing (b) Join node sharing

Fig. 2.19 Fork (**a**) and Join (**b**) node sharing

1. identification of all the fork/join groups; note that different groups may share the fork node (Fig. 2.19.a) or the join node (Fig. 2.19.b);
2. identification of the task functions to be created and the related parameters that have to be given to them;
3. printing of the C code that respects the original semantic of the application.

Finally, the resulting C code is provided to the rest of the toolchain for the decision mapping phase.

Note that not only phi-nodes of virtual operands, but also phi-nodes which define scalar SSA-form variables, are not printed back. We could print back them as assignments with "conditional expression", but in this way a significant overhead to the application execution occurs; moreover, this technique is difficult to be used as it is for phi-nodes with more than two *uses*. For these reasons a phi-node of a scalar variable is replaced by as many assignments as the *variable uses* present in that node, placed immediately after the related variable definition. For example, suppose you have a phi-node `a_1 = PHI <a_4(3), a_5(4)>`, where `a` is a variable and the suffix `_<index>` identifies its SSA version; note that the number inside the parentheses represents the basic block where the definition occurs. According to SSA-definition, variable `a_4` should be defined only in one statement of the function and according to phi-node definition, this statement should be mutually exclusive with the others in the same phi-node. Therefore, the assignment `a_1 = a_4` can be added after the statement that defines `a_4`, that will be in the basic block numbered as 3. Then the same procedure is repeated for definition of variable `a_5`, that will be in the basic block numbered as 4. In this way phi-nodes are not necessary anymore and the consistency of the sequential model is maintained.

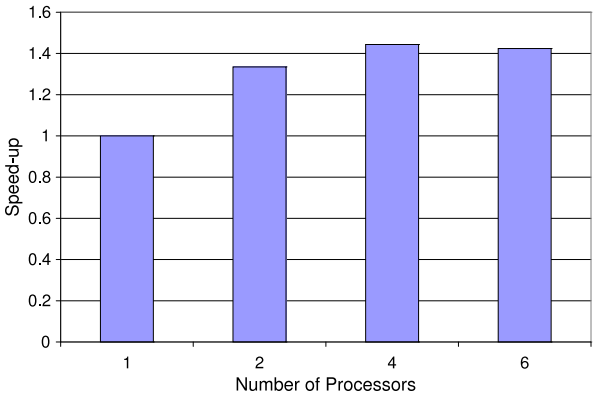


Fig. 2.20 Parallelization of JPEG encoder

Case Studies: The JPEG Encoder and the DRM Decoder

In this section, we analyze two different case studies for our partitioning methodology: the JPEG encoder and the core of the Digital Radio Mondiale (DRM) decoding process, that was one of the benchmark applications proposed in the hArtes project.

The kernel of the JPEG encoder contains a sequence of data transformations applied to the raw image, as shown on the left side of Fig. 2.20: Color Space Transformation, Downsampling, Block Splitting, Discrete Cosine Transformation, Quantization and Entropy Encoding. Among these phases, Color Space Transformation (RGBtoYUV), Discrete Cosine Transformation (DCT) and Quantization are the most computationally intensive. The analysis performed by Zebu is able to identify some Single Instruction Multiple Data parallelism in the first two functions. Then, after applying the different transformations to reduce the overhead, Zebu produces a hierarchical task graph representing which parts can be executed in parallel, as shown on the right side of Fig. 2.20. The task graph of the RGBtoYUV function contains a single parallel section with four parallel tasks. The task graph of the DCT function has a similar structure, but with only three tasks. This structure is then reproduced in the output C code through the corresponding OpenMP pragmas compliant with the hArtes guidelines.

The DRM refers to a set of digital audio broadcasting techniques designed to work over shortwave. In particular, different MPEG-4 codecs are exploited for the transmission and the core of this decoding process is mainly composed of the Viterbi decoder. Exploiting GNU/GCC optimizations, Zebu is able to identify and extract some data parallelism from the proposed implementation of the algorithm, as shown in Fig. 2.21. A task graph with four parallel tasks describing the introduced parallelization is then reproduced annotating the output C code through OpenMP as well.

Note that, in both the cases, the parallelism identified among loop iterations is expressed through `omp parallel sections` instead of `omp parallel for` pragmas. In fact, in the hArtes toolchain, the different tasks have to be represented as functions to support the mapping that is performed at compile time and not during the execution of the code. For this reason, we explicitly represent the partitioning of the loop as `omp sections`.

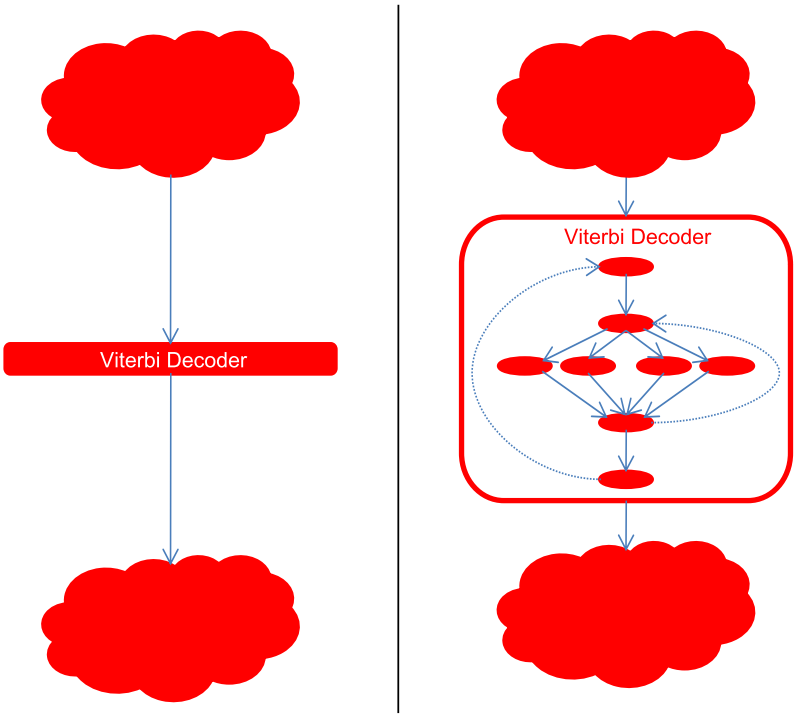


Fig. 2.21 Parallelization of DRM decoder

To evaluate the obtained partitioning, we executed the resulting applications on two different simulators: a modified version of the SimIt-ARM cycle accurate simulator [41] and ReSP [4], a MPSoC simulation platform developed at Politecnico di Milano. Both the simulators have been configured to simulate a homogeneous multiprocessor architecture composed of four ARM processors and a shared memory, connected through a common bus. We decided to target a homogeneous architecture to verify the potential benefits introduced by the partitioning, without being affected by the mapping decisions.

In details, we modified the original version of SimIt-ARM in order to support a multi-core simulation, with private caches and shared memory. The OpenMP pragmas are substituted by code instrumentations that communicate with a co-processor that manages the counting of the simulation time. In this way, our version of SimIt-ARM is able to take into account the concurrent execution of the different tasks. On the contrary, ReSP is able to model a complete architecture where the OpenMP pragmas are translated into proper `pthread` functions supported by a very light operating system. As a result, ReSP allows an accurate simulation of the complete behaviour of a multiprocessor system also considering the overhead due to bus contention and system calls.

Both the sequential and the partitioned versions of the two case studies have been executed on these simulators and the resulting speed-ups are reported in Table 2.3.

Table 2.3 Speed-ups for the parallel versions of JPEG and DRM applications as estimated by Zebu and measured on SimIt-ARM and ReSP

Application	Speed-up		
	Zebu	SimIt-ARM	ReSP
JPEG encoder	2.17	2.56	2.36
DRM decoder	1.72	1.87	1.40

Since SimIt-ARM does not consider the delays due to the contention on accessing the resources (e.g., limit number of processors or concurrent accesses to the bus), it systematically produces an overestimation of the speed-up with respect to the one obtained with ReSP that simulates a more complete model of the target architecture. For this reason, it can represent the maximum speed-up that can be obtained with the current parallelization.

It is worth noting that Zebu completely parallelizes the whole Viterbi decoder that takes most of the execution time of the DRM application. However, the overheads due to thread management and synchronization costs reduce most of the potential benefit in terms of parallelization.

Finally, we can conclude that Zebu is able to statically estimate the speed-up with a good accuracy with respect to methods that require a dynamic simulation of the entire application and the architectures. The corresponding methodology will be detailed in the following section.

2.6.2 Cost Estimation for Partitioning

In all the phases of Zebu, information about performance of the different parts of the application on the different processing elements of the architecture is crucial. Indeed, Zebu needs this type of information to correctly create tasks of proper granularity, to manipulate the task graphs removing inefficient parallelism (i.e., a parallelism which slows down the processing because tasks creation/synchronization/destruction overhead nullifies the gain) and to compute the initial mapping of the application onto the target platform.

Such information could be provided by the tools that precede Zebu in the hArtes toolchain, but, most of the time this information is incomplete and it is limited to the execution time of existing functions, such as library ones. In particular, there is no detailed information about the execution time of an arbitrary piece of code (i.e., a candidate task) clustered by Zebu. For this reason, additional performance estimation techniques have been necessarily included into the tool.

Two types of performance estimation techniques have been implemented in Zebu:

- techniques for estimating the performance of a single task on each processing element of the architecture;
- a technique for accurately estimating the performance of the whole task graph, given a partitioning solution.

The first ones are based on the building of a linear performance model of the different processing elements (GPP and DSP) exploiting a regression technique both for software and hardware solutions.

Linear Regression Analysis

The approach for internal performance estimation is based on both standard statistical and more advanced data mining techniques aimed at identifying some correlations between some characteristics of the piece of code to be analyzed and the actual execution time on the given processing element. 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 task and its mapping. Starting from the set of pairs $\langle Ij, Cj \rangle$, 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 $\langle Ij, Cj \rangle$ 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. These models are then used to evaluate the performance of arbitrary pieces of code during the early phases of the partitioning. To improve the accuracy of the estimation, the performance models are combined with profiling information (e.g., branch probability, loop iteration numbers) retrieved by executing the application on the host machine with a significant dataset for the input application.

The linearity of the models is known to introduce an approximation in the performance estimation, but it simplifies the transformations performed on the task graphs, speeding up the design space exploration. For example, in this way the performance of a new task resulting from the merging of other two existing ones is computed by summing their performance. Finally, the available performance information provided via XML is used during this phase to evaluate the performance of library functions and of not-partitioned functions.

FPGA Area Estimation Model

When the target architecture contains also hardware components, we need an estimation not only for the execution time, but also for the area occupation, to deal with the architectural constraints (e.g., the limited area into the device) in the mapping phase. For this reason, given a candidate task, we first analyze if it can be synthesized by the available synthesis tool (e.g., DWARF in the hArtes toolchain). In particular, if it contains constructs that cannot be synthesized, the estimation is

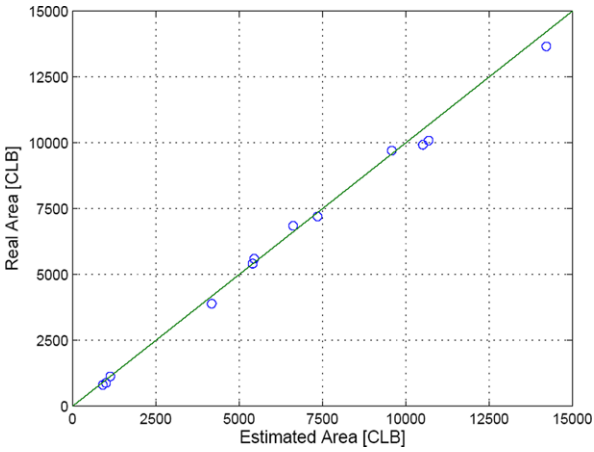


Fig. 2.22 Validation of the FPGA area model

not performed and a mapping to this component will be forbidden for this task. On the other hand, if the implementation is possible, the area model we use for fast estimation is obtained starting from the results of a *fast* high-level synthesis (HLS) of the task, while the performance is related to the worst case execution time of the controller synthesized by the HLS. The total area estimated by the model is composed of two distinct parts: the sequential part (FF) and the combinatorial part (LUT). While the FF part is easy to be estimated, since it is composed of the data registers used in the data-path and of the flip flops used for the state encoding of the controller finite state machine, the LUT part is a little more complex to estimate. Four contributions define the LUT estimation value: FU, FSM, MUX and Glue. The FU part corresponds to the contribution of the functional units and so its value is the sum of the area occupied by each functional unit. The other three parts (FSM, MUX, Glue) are obtained by using a linear regression-based approach and they are defined as follows:

1. the FSM contribution is due to the combinatorial logic used to compute the output and next state;
2. the MUX contribution is due to the number and size of multiplexers used in the data-path;
3. the Glue contribution is due to the logic to enable writing in the flip flops and to the logic used for the interaction between the controller and the data-path.

The coefficients of the model have been obtained starting from a set of standard high-level synthesis benchmarks and the comparing the values with the ones obtained with actual synthesis tools (i.e. Xilinx ISE). The resulting FPGA area model shows an average and a maximum error of 3.7% and 14% respectively and Fig. 2.22 shows the such a comparison where the bisector represents the ideal situation (i.e., the estimated value is equal to the actual one). Further details can be found in [40].

Software Cost Model

The software cost estimation methodology is mainly based on linear regression and it consists of two different steps: construction of a cost estimation model for each processing element and estimation of the performance of the application under analysis using the built cost models. Given a processing element, the model is produced starting from a set of embedded system benchmarks significant for the class of applications which should be estimated. These benchmarks are then analyzed to extract a set of features from each of them that are used as input training set for the building of the estimation model.

There are several different types of features which could be extracted by the analysis of the application without executing it directly onto the target platform. However, only a part of these features is suitable to be used as input for building the cost model because some of them have no correlation with the total performance cost and they can be ignored. Moreover, the larger is the number of input features selected for building the model, the larger should be the size of the training set. To choose among them, some characteristics of the estimation problem have to be analyzed, such as the aspects of the architecture that are usually difficult to be caught by a model. Considering the class of processors and of the target applications of this methodology, the more relevant aspects are the presence of the cache, the characteristics of the pipeline and the type of possible optimizations introduced by the compiler.

The building of the model does not require information about the overall target architecture but only information about the execution cost of a set of significant benchmarks on the target processing element. This information could be retrieved by executing the instrumented code directly onto the element or by using a cycle-accurate simulator of it. This is necessary only during the training to produce the cost model, that will be then used for estimation. In particular, to estimate the performance of an application onto a processing element, only its source code and the model related to the component are used. The source code is compiled and translated into the intermediate representations, that are then analyzed and profiled to extract the selected features for each function. These features are then given as input to the model obtained by the previous phase which quickly produces the performance estimation for each function of the application. By combining the information about cost and the number of executions of each function the total performance cost of the application can be easily estimated.

Besides the features described above, the following metrics have been also considered:

- number of loop iterations;
- static metrics for tasks communication evaluation to be used during the mapping to select an efficient assignment of tasks onto different processing elements.

The number of loop iterations can be easily extracted from the source code for countable loop. For uncountable loops, an estimation of the average number is obtained by an internal dynamic profiling of a proper annotated source code. Our metrics on communication are based on an abstract model that formalizes most of the

Table 2.4 Performance estimation error on single processing element

Processing element	Opt. level	Mean error	Standard deviation of error
ARM	O0	16.9%	15.6%
	O1	14.6%	14.2%
	O2	16.7%	15.8 %
DSP		18.0%	15.9%

information of the communication aspects. Upon this model, we defined a set of metrics that provide information useful for architectural mapping and hardware-software co-design. Further details can be found in [1].

Concerning the software cost model, we built models for the ARM processor and the DSP MAGIC on a suitable set of benchmarks (see Fig. 2.23 for a representation of the dataset). Since these performance estimations are obtained by applying regression techniques and in order to evaluate the effective accuracy of the resulting models, we exploit the cross-validation technique of them. Cross-validation is a technique for assessing the performance of a static analysis on an independent data set and can be performed with different methods. We used the *K-fold cross-validation* where the initial data set is randomly divided into K subsets (in our case, $K = 10$). The model building process is repeated K times and, at each iteration i, a performance model is built by analysing all the subsets but the i-th, which is used to test the model. At the end, the average error across all the K trials is computed and a unique model is built by combining the K models. The obtained results are reported in Table 2.4. Note that different optimization levels have been also considered in building performance models of the ARM processor.

Task Graph Estimation Model

Even if the created tasks are of proper granularity, the parallelism introduced with the initial task graphs cannot produce an actual benefit because of the overhead introduced by the threads creation and synchronization. For this reason, we introduce a methodology to estimate the performance of the whole task graphs highlighting the inefficient parallelism which has to be removed. Such methodology has been introduced since computing the overall performance by simply combining the performance of the single tasks can produce wrong estimations, in particular when the execution time of a task heavily depends on control constructs, on the particular execution path that results to be activated and on its frequency.

Consider for instance the example shown in Fig. 2.24.

Annotating the tasks with their average performance with respect to the branch probability and computing the task graph performance by considering the worst-case execution time on them would lead to a wrong estimation ($1000+500+1000 =$ 2500 cycles) with respect to the actual execution time (3000 cycles in all the cases).

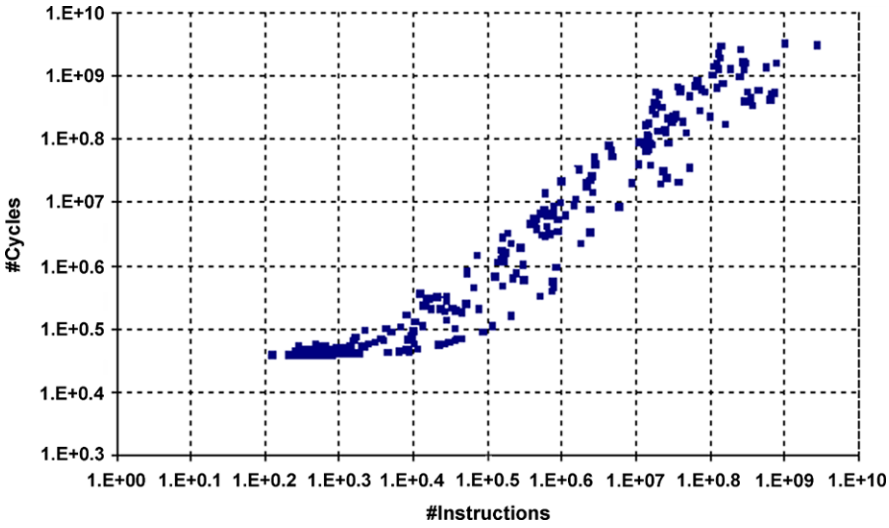


Fig. 2.23 Distributions of the benchmarks dataset with respect to the number of instructions and the number of cycles

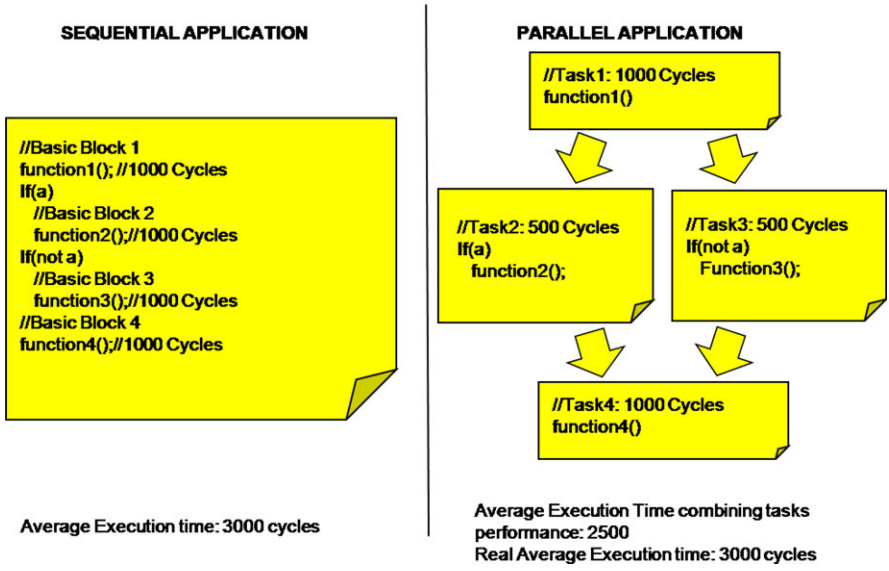


Fig. 2.24 Example of unprofitable parallelism

For this reason, we have proposed [9] a task graph performance estimation technique which combines the performance models of the target processing element, the execution paths, the dynamic profiling information and the structure of the task graph to produce a more accurate estimation of the performance of a partitioned

Table 2.5 Average and Maximum Speed-up Estimation Error on whole task graph compared with techniques based on Worst Case and Average Case

Optimization level	Techniques					
	Worst Case		Average Case		Zebu	
	Avg.	Max.	Avg.	Max.	Avg.	Max.
O0	25.3%	85.4%	24.3%	85.4%	3.9%	12.6%
O2	20.9%	85.7%	21.6%	85.7%	4.3%	11.5%

application. In particular we exploit the processing element performance models described above to estimate the performance of the single pieces of code (e.g.: `function1()`). For each application execution path (e.g., Basic Block 1–Basic Block 2–Basic Block 4) identified by the profiling of the code, we evaluate its contribution to the overall execution time of the task graph (e.g.: 3000 Cycles for the considered path). This combination takes also into account the cost for creating and synchronizing the parallel tasks. These costs have been retrieved by averaging a set of measures obtained running parallelized and automatically instrumented code on the target platform [8]. Finally we obtain the overall performance of the task graph combining the contribution of each execution path, that is averaged by its relative frequency measured executing the application on the host machine on a set of significant input datasets. These estimations are then used to evaluate the effects of the different parallelizations and accordingly transform the task graph to improve the performance. In particular, all the unprofitable parallel sections are removed by applying different transformations, such as merging the corresponding parallel tasks. Moreover, since OpenMP is used to annotate the parallelism in the produced C specification, transformations of the task graph are necessary to make it suitable for the OpenMP paradigm. The task graph is thus arranged to accomplish this model and balance the number of parallel tasks in each section.

Note that when different transformations are available, we choose the one that introduces the greatest benefit (or the lowest degradation) in terms of performance, according to the estimation techniques described above.

The methodology has been validated on a set of applications parallelized either by hand or by Zebu. In particular, the parallel code has been executed on the modified version of SimIt-ARM [41] described in the previous section. Then, this cycle-accurate simulation has been compared with the estimation obtained with the proposed methodology and the results that can be achieved by traditional techniques based on the computation of Worst Case and Average Case. The results about the error on speed-up estimation are reported in Table 2.5.

These results show that the proposed methodology is able to obtain an estimation of the speed-up much closer to the actual simulation with SimIt. It has been thus integrated into Zebu since it better drives the exploration process about the benefits that can be obtained with different parallelizations, as described above.

2.7 Task Mapping

This section describes the task mapping process of the hArtes toolchain [34]. The goal of task mapping is to select parts of the C application to execute on specialized processors (DSP or FPGA), in order to minimize the overall execution time of the application. The task mapping process is designed with three novel features. The first is that near optimal solutions can be generated by a heuristic search algorithm to find the optimal solution. The second is that developers can guide the decision process by providing directives to constrain the mapping solution. Finally, a transformation engine has been devised to perform source-level transformations that can improve the execution of each individual task on the selected processing element, and as a consequence helps the mapping process generate a better solution. In this section we cover the following topics:

1. **Task Mapping Approach (Sect. 2.7.1)**. We provide an overview of the task mapping process including the basic design flow and its main components.
2. **Task Filtering (Sect. 2.7.2)**. The task filtering process is responsible for constraining the mapping solution and ensuring that it complies to the hArtes platform.
3. **Task Transformation (Sect. 2.7.3)**. The task transformation is part of the task mapping approach which enables developers to select, describe and apply transformations to individual tasks, taking into account application-specific and platform-specific requirements.
4. **Mapping Selection (Sect. 2.7.4)**. The mapping selection is responsible for assigning a processing element to each task in the program in order to exploit as much as possible the full capabilities of the heterogeneous system, while providing a feasible solution for the hArtes platform.
5. **Experimental Features (Sect. 2.7.5)**. We report three research experiments that have been performed in conjunction with the task mapping approach, and show that they improve the quality of the solution.
6. **hArmonic Tool (Sect. 2.7.6)**. We illustrate some of the main features of the hArmonic tool, which implements the task mapping process in the hArtes toolchain. In addition, we summarize the key results.

2.7.1 Mapping Approach

The task mapping design flow and its main components are shown in Fig. 2.25. There are two main inputs: (1) the source code, which supports an arbitrary number of C source files, and (2) the platform specification. The platform specification describes the main components of the heterogeneous system, including the processing elements, the interconnect, storage components, and system library functions.

The first stage of task mapping is the automatic C guidelines verification, which determines whether the source code complies with the restrictions of the mapping process. Some of these limitations correspond to language elements and coding

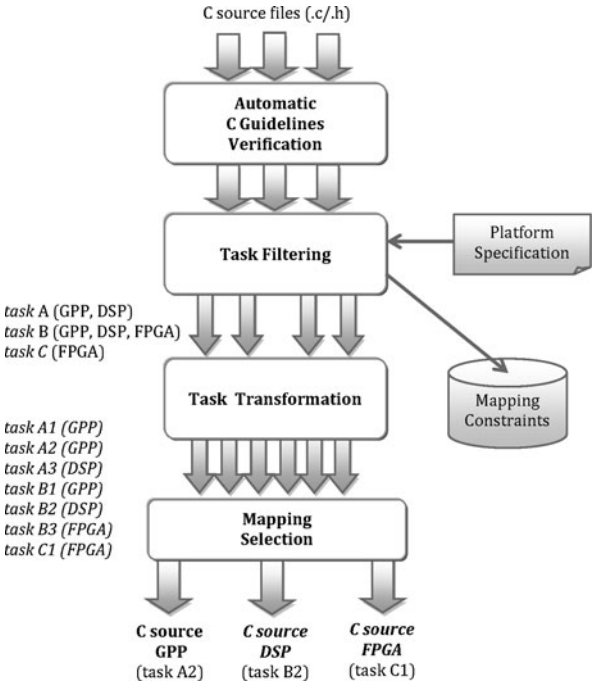


Fig. 2.25 The task mapping approach is comprised by four main modules: (1) an automatic C guidelines verification process that automatically determines if the code conforms to the restrictions of the task mapping process, (2) a filtering engine that determines which processing elements can support each task, (3) a task transformation engine which produces one or more implementations of a given task that can potentially exploit the capabilities of the processing elements in the system, and (4) the mapping selection process which assigns a processing element implementation to each task of the application

styles that can hinder the efficiency of the task mapping process, such as the use of function pointers. Other limitations include forbidding language elements not supported by our approach, for instance static function definitions. Hence, the C guidelines verification can provide hints to improve the task mapping process, enabling developers to revise their code in specific parts of the program. Furthermore, any violation of the C guidelines found by the automatic verification process will terminate the task mapping process.

The second stage of task mapping is task filtering. The objective of task filtering is to determine which processing elements can support each individual task in the application, and under what conditions. The filtering engine has two main outputs. The first is a list of supported processing elements for each task in the application. For instance, in Fig. 2.25, the filtering engine determines that task A can only be supported by the GPP and DSP processors, whereas task C can only be synthesized to the FPGA. The second output of the filtering engine is a set of mapping constraints which ensure that the mapping result would comply with the hArtes platform requirements and limitations. For instance, one mapping constraint could state that task B and task C must be mapped to the same processing element. Another constraint could state that task C can only be mapped to an FPGA. When combining these two constraints, the mapping solution should map task B to an FPGA to satisfy all constraints, otherwise the mapping solution is infeasible. Section 2.7.2 provides more details about the task filtering process.

The third stage of task mapping is task transformation [49]. Once each task has been assigned to a list of feasible processors by the previous stage, the transforma-

tion engine generates one ore more implementations of each task according to an established portfolio of transformations for each processing element. The transformation engine has two main benefits. First, it can potentially uncover optimizations that exploit the properties of each individual processing element in the system, and can consequently improve the performance of the resulting mapping solution. Second, it enables the mapping process to target a wide range of implementations for each task, making the selection process less biased towards a particular processing element. Section 2.7.3 presents the task transformation engine in more detail.

The fourth and final stage of task mapping is mapping selection [31]. At this point, we have a set of tasks and associated implementations generated by the task transformation engine, a set of mapping constraints produced by the filter engine to comply with the hArtes platform as well as constraints provided by the developer, and a cost estimator which computes the cost of each task. The search process is geared towards finding a solution that minimizes as much as possible the cost of the application and satisfies the mapping constraints. Once a solution is found, and tasks have been assigned to their corresponding processing elements, we generate the code for each processing element. The generated code is compiled separately by the corresponding backend compilers, and linked afterwards to produce a single binary. Each individual compilation unit corresponds to a subset of the application designed to realize the overall mapping solution.

2.7.2 Task Filtering

The task filtering process relies on individual filters to determine what processing elements can support each application task in order to find a feasible mapping solution. A total of 15 different filters are employed, a sample of which are shown in Table 2.6.

Each individual filter is responsible for computing the list of processing elements to which each task can be mapped. Since the result of one filter can affect the result of other filters, they are run sequentially in a loop until there are no changes in the result. In addition to assigning a list of processing elements to each task, each filter can derive a set of mapping constraints which can, for instance, specify that a task mapping is valid only if other task mappings are part of the solution (e.g. mapping task A to ARM $\Rightarrow$ mapping task B to ARM). By constraining the solution, the task filtering process can considerably reduce the mapping selection search time as well as ensuring that the resulting mapping solution can be realized on the hArtes platform.

2.7.3 Task Transformation Engine

The task transformation engine (Fig. 2.26) applies pattern-based transformations, which involve recognizing and transforming the syntax or dataflow patterns of design descriptions, to source code at task level. We offer two ways of building task

Table 2.6 List of filters employed to find mappings

Filter	Description
FRMain	Ensures that the main function is always mapped to the master processor element (GPP)
FRCall	Ensures that a function that calls another is either mapped to the master processing element, or both functions are mapped to the same processing element
FRDwarvFPGA	Ensures that a function is not mapped to FPGA if it does not comply with the restrictions of the DWARV compiler, such as not supporting `while` loops
FRTargetDSP	Ensures that a function is not mapped to DSP if it does not comply with the restrictions of the DSP compiler, such as the use of array declarations with nonconstant sizes
FRGlobal	Ensures that functions that share one or more global variables are mapped to the same processing element
FRLibrary	Ensures that a function cannot be mapped to a processing element (other than the master processing element) if it invokes a library function not specified in the platform specification file
FRRemoteFn	Ensures that if one function calls another, they are either mapped to the same processing element or to different processing elements if the parameter types can be used in a remote call
FRTypes	Ensures that a function can only be mapped to a processing element if it supports the basic data types specified in the platform specification file

transformations: using the underlying compiler framework, ROSE [42], to capture transformations in C++; this is complex but offers the full power of the ROSE infrastructure. Alternatively, our domain-specific language CML simplifies description of transformations, abstracting away housekeeping details such as keeping track of the progress of pattern matching, and storing labeled subexpressions.

CML is compiled into a C++ description; the resulting program then performs a source-to-source transformation. For design exploration, we also support interpreting CML descriptions, allowing transformations to be added without recompiling and linking. Task transformations could be written once by domain specialists or hardware experts, then used many times by non-experts. We identify several kinds of transformations: input transformations, which transform a design into a form suitable for model-based transformation; tool-specific and hardware-specific transformations, which optimize for particular synthesis tools or hardware platforms.

Each CML transformation (Fig. 2.26) consists of three sections: (1) pattern, (2) conditions, and (3) result. The pattern section specifies what syntax pattern to match and labels its parts for reference. The conditions section typically contains a list of Boolean expressions, all of which must be true for the transformation to apply. Conditions can check: (a) validity, when the transformation is legal; (b) applicability: users can provide additional conditions to restrict application. Finally, the result section contains a pattern that replaces the pattern specified in the pattern section, when conditions apply.

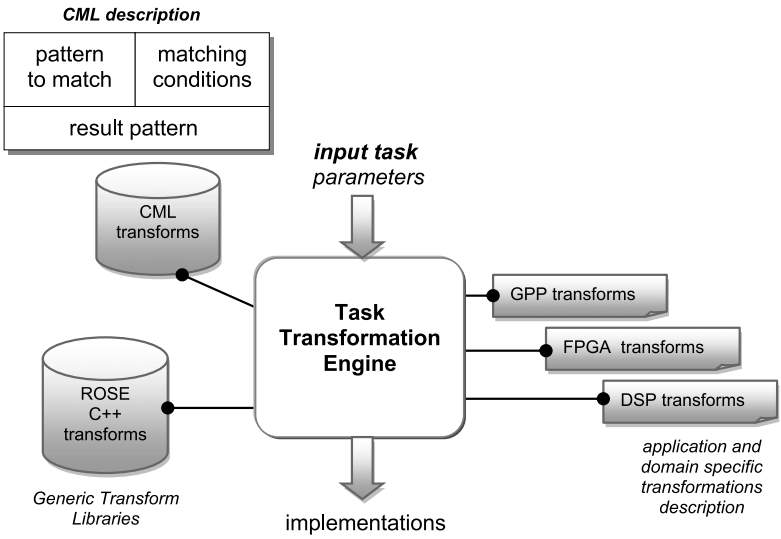


Fig. 2.26 An overview of the task transformation engine. The task transformation engine receives as input a task and additional parameters such as the processing element that we wish to target, and generates a set of implementations. The set of transformations to be applied to each processing element is provided by the user. The implementations of transformations are stored as shared libraries for ROSE transformations, and as text files for CML-based transformations. A CML description consists of three sections: the pattern to match, the matching conditions, and the resulting pattern (Listing 2.1)

```
1     pattern {
2        for (var(a)=0;var(a)<expr(e1);var(a)++){
3           for (var(b)=0;var(b)<expr(e2);var(b)++)
4              stmt(s);
5        }
6     }
7     conditions {

9     }
10    result {
11       for (newvar(nv)=0;
12          newvar(nv)<expr(e1)*expr(e2);
13          newvar(nv)++)
14       {
15          var(a) = newvar(nv) / expr(1);
16          var(b) = newvar(nv) % expr(1);
17          stmt(s);
18       }
19    }
20 }
```

Listing 2.1 CML description of the loop coalescing transformation

A simple example of a CML transformation is loop coalescing (Listing 2.1), which contracts a nest of two loops into a single loop. Loop coalescing is useful

in software to avoid loop overhead of the inner loop, and in hardware to reduce combinatorial depth. The transformation works as follows:

- Line 1: LST 2.1 starts a CML description and names the transformation
- Lines 2–7: LST 2.1 gives the pattern section, matching a loop nest. CML patterns can be ordinary C code, or labelled patterns. Here `var(a)` matches any value and labels it "a". From now on, each time `var(a)` appears in the CML transform, the engine tries to match the labelled code with the source code.
- There is no conditions section (lines 8–10: LST 2.1), as coalescing is always valid and useful.
- Lines 11–20: LST 2.1 gives the result section. The CML pattern `newvar(nv)` creates a new variable which is guaranteed unique in the current scope. The resulting loop behaves the same as the original loop nest. The values of the iteration variables, `var(a)` and `var(b)`, are calculated from the variable `newvar(nv)` in the transformed code. This allows the original loop statement, `stmt(s)`, to be copied unchanged.

When the transformation engine is invoked, it triggers a set of transformations that are specific to each processing element, which results in a number of implementations associated with different tasks and processor elements. The implementation description of ROSE transformations are stored as shared libraries, and the CML definitions as text files. Because a CML description is interpreted rather than compiled, users can customise the transformation by using a text editor, and quickly evaluate the effects of the transformation without requiring an additional compilation stage.

To show the effect of our transformation engine, we apply a set of transformations to an application that models a vibrating guitar string (provided by UNIVPM). These transformations have been described in both CML and ROSE, and allow the user to explore the available design space, optimizing for speed and memory usage. We modify the application for a 200 second simulated time to show the difference between the various sets of transformations. The set of transformations includes:

- **S**: simplify (inline functions, make iteration variables integer, recover expressions from three-address code)
- **I**: make iteration bounds integer
- **N**: normalise loop bounds (make loop run from 0 to N-2 instead of 1 to N-1)
- **M**: merge two of the loops
- **C**: cache one array element in a temporary variable to save it being reread
- **H**: hoist a constant assignment outside the loop
- **R**: remove an array, reducing 33% of memory usage (using two arrays instead of three)

Figure 2.27 shows how the design space can be explored by composing these transformations. Transformation S provides an almost three-fold improvement, mostly by allowing the compiler to schedule the resulting code. Transformation I gives nearly another two-fold improvement, by removing floating-point operations from the inner loop. Transformation N gives a small improvement after transformation I. Transformation M slows the code down, because the merged loop uses

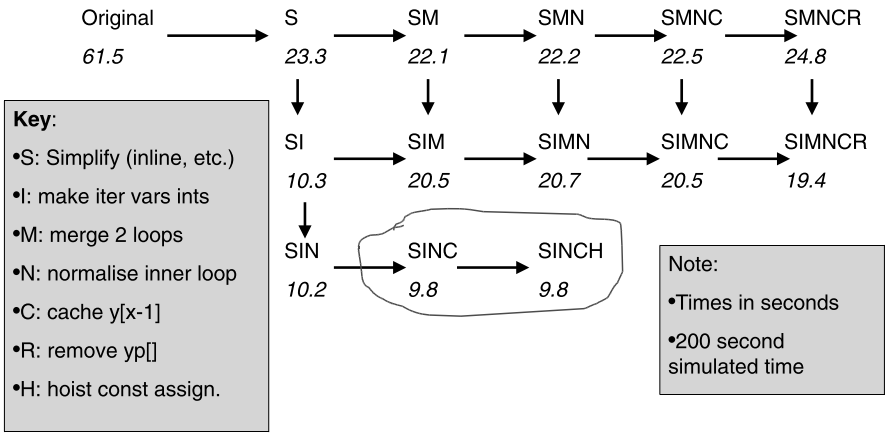


Fig. 2.27 Starting with the original code for the application that models the vibration of a guitar string, we explore ways of using seven different transformations to attempt to improve the run time and memory usage. Much of the speedup comes from simplifying the code and making iteration variables integer, while the remainder comes from caching to prevent repeat memory access and removing a constant assignment from the loop body. The caching also enables one array to be eliminated (about 33% reduction in memory usage), possibly at the expense of performance

the GPP cache badly. Transformation C improves the integer code (I) but leaves the floating point version unimproved. Finally, transformation R gives a small improvement to the integer version, but actually slows down the floating-point version. Overall, we have explored the design space of transformations to improve execution time from 61.5 seconds to 9.8 seconds, resulting in 6.3 times speedup.

2.7.4 Mapping Selection

Our mapping selection approach is unique in that we integrate mapping, clustering and scheduling in a single step using tabu search with multiple neighborhood functions to improve the quality of the solution, as well as the speed to attain the solution [30]. In other approaches, this problem is often solved separately: a set of tasks are first mapped to each processing element, and a list scheduling technique then determines the execution order of tasks [54], which can lead to suboptimal solutions.

Figure 2.28 shows an overview of the mapping selection approach. Given a set of tasks and the description of the target hardware platform, the mapping selection process uses tabu search to generate different solutions iteratively. For each solution, a score is calculated and used as the quality measure to guide the search. The goal is to find a configuration with the lowest score, representing the execution time.

Figure 2.29 illustrates the search process. At each point, the search process tries multiple directions (solid arrows) using different neighborhood functions in each move, which can increase the diversification to find better solutions. In the proposed

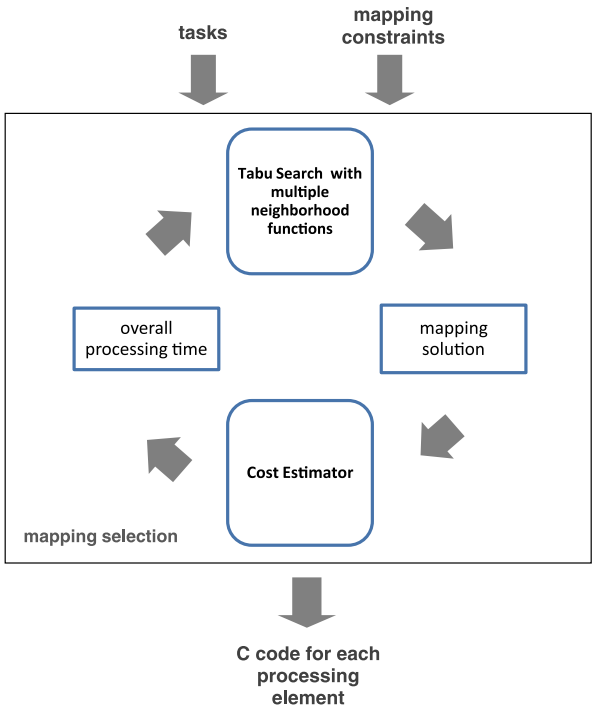


Fig. 2.28 An overview of the mapping selection process

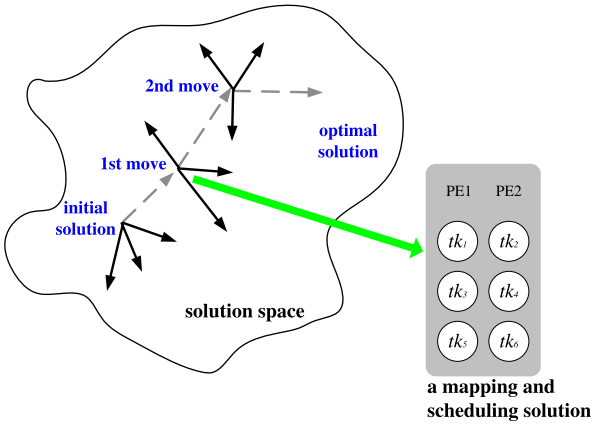


Fig. 2.29 Searching for the best mapping and scheduling solution using multiple neighborhood functions. The *solid arrows* show the moves generated by different neighborhood functions. The *dotted arrows* denote the best move in each iteration of the search. PE: processing element, tk: task

technique, after an initial solution is generated, two neighborhood functions are used to generate neighbors simultaneously. If there exists a neighbor of lower cost than the best solution so far, and it is not in the tabu list, this neighbor is recorded. Otherwise a neighbor that cannot be found in the tabu list is recorded. If all the above conditions cannot be fulfilled, a solution in the tabu list with the least degree, i.e. a solution resident in the tabu list for the longest time, is recorded. If the recorded solution has a smaller cost than the best solution so far, it is recorded as the best

solution. The neighbors found are added to the tabu list, and solutions with the least degree are removed. This process is repeated until the search cannot find a better configuration for a given number of iterations. An advantage of using multiple neighborhood functions is that the algorithm can be parallelized, and therefore the time to find a solution can be greatly reduced.

Another important component of the mapping selector is the cost estimator (Fig. 2.28). The cost estimator computes the overall processing time, which is the time for processing all the tasks using the target computing system including data transfer time between processing elements. The processing time of a task on a processing element is calculated as the execution time of this task on the processing element plus the time to retrieve results from all of its predecessors. The data transfer time between a task and its predecessor is assumed to be zero if they are assigned to the same processing element.

Our approach for estimating the cost of a task running on a particular processing element currently exploits rule-based techniques. Our rule-based estimator makes use of linear regression to estimate the cost based on a set of metrics:

$$EstTime = \sum_{i=1}^{N} T_{P_i}$$

where N is the number of instructions, P_i is the type of instruction i, T_{P_i} is the execution time of instruction P_i. Each processing element contains one set of T_{P_i} for each type of instruction. Instructions include conditionals and loops, as well as function calls.

2.7.5 Experimental Features

In this section we provide a brief overview of three experiments that have been developed independently from the task mapping approach. They have been proved to enhance the task mapping process.

- **Automatic Verification**. A verification framework has been developed in conjunction with the task transformation engine [48]. This framework can automatically verify the correctness of the transformed code with respect to the original source, and currently works for a subset of ANSI C. The proposed approach preserves the correct functional behavior of the application using equivalence checking methods in conjunction with symbolic simulation techniques. The design verification step ensures that the optimization process does not change the functional behavior of the original design.
- **Model-Based Transformations**. The task transformation engine (Sect. 2.7.3) supports pattern-based transformations, based on recognizing and transforming simple syntax or dataflow patterns. We experiment with combining such pattern-based transformations with model-based transformations, which map the source

code into an underlying mathematical model and solution method such as geometric programming. We show how the two approaches can benefit each other, with the pattern-based approach allowing the model-based approach to be both simplified and more widely applied [32]. Using a model-based approach for data reuse and loop-level parallelization, the combined approach improves system performance up to 57 times.

- **Data Representation Optimization**. Another approach used in conjunction with the task transformation engine is data representation optimization for hardware tasks. The goal of data representation optimization is to allow developers to trade accuracy of computation with performance metrics such as execution time, resource usage and power consumption [38]. In the context of reconfigurable hardware, such as FPGAs, this means exploiting the ability to adjust the size of each data unit on a bit-by-bit basis, as opposed to instruction processors where data must be adjusted to be compatible with register and memory sizes (such as 32 bits or 64 bits).

  The data representation optimization approach has two key features. First, it can be used to generate resource-efficient designs by providing the ranges of input variables and the desired output precision. In this case, the optimization process analyzes the code statically and derives the minimum word-lengths of variables and expressions that satisfy user requirements. In addition, a dynamic analysis can be employed to automatically determine the input ranges and output requirements for a particular set of test data.

  Second, this approach can be used to generate power-efficient designs using an accuracy-guaranteed word-length optimization. In addition, this approach takes into account library functions where implementation details are not provided. Results show power savings of 32% can be achieved by reducing the accuracy from 32 bits to 20 bits.

2.7.6 The hArmonic Tool

This section describes some of the features of the hArmonic tool which implements the functionality described in Fig. 2.25. The hArmonic tool receives as input an arbitrary number of C sources, and generates subsets of the application for each individual processing element according to the derived mapping solution. The list of features include:

- **System and Processor Driver Architecture**. hArmonic uses a driver-based architecture. Each driver captures the specific elements of the platform (system driver) and the processor elements (processor driver) and interfaces with the mapping process which is generic. For instance, the system driver is responsible for estimating the cost of the whole application for the corresponding platform. The processor driver, on the other hand, shows whether a particular task can be supported by associated processing elements. This way, hArmonic can be customized to support different hardware platforms and processing elements by adding new drivers, without the need to change the interfaces or the core mapping engine.

Guidelines	Description
Avoid function pointers	Detects the use of function pointers
Avoid union types	Detects the use of union types
No static functions	Detects the use of static functions
No comma operators	Detects the use of complex expressions using comma operators
No implicit function calls	Detects the use of function calls which were not previously declared

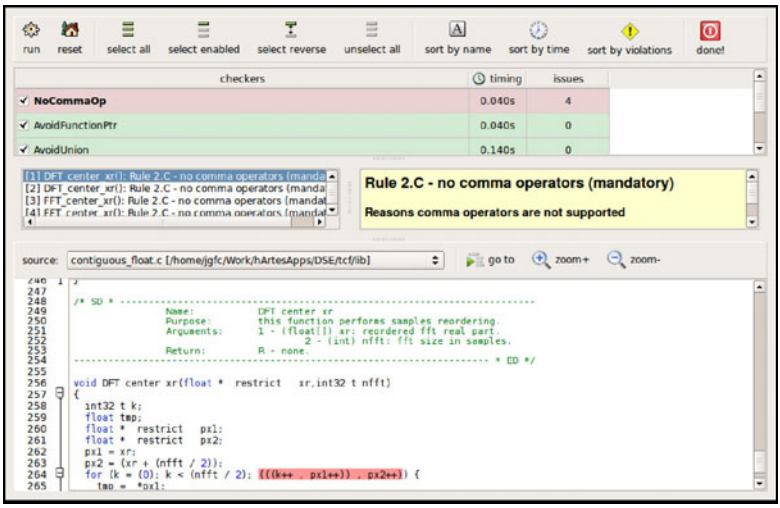


Fig. 2.30 A description of C guidelines automatically detected by hArmonic, and a screenshot of its graphical user-interface

- **Automatic C Guidelines Verifier**. The automatic C guidelines verifier analyzes the input C source and highlights any guideline violation and makes recommendations about the code, to ensure conformance and integration between the tools. For instance, it automatically detected instances of static functions in code generated by SCILAB-to-C tool (AET), and the use of function pointers in the x264 source-code. The former is a violation because static functions are constrained to the source-file where they are located, whereas hArmonic must move functions to different source-files according to the mapping solution. Avoiding the use of function pointers, on the other hand, is a recommendation, as hArmonic does not support dynamic mapping and therefore the quality of the solution can be diminished. Developers are therefore invited to change their sources or tools in a way that can maximize the efficiency of the toolchain and the mapping solutions.

  The C guidelines supported by hArmonic are shown in Fig. 2.30. This figure also presents a screenshot of the corresponding graphical user interface, which illustrates how hArmonic automatically pinpoints potential problems in the source-code that prevent either good mapping or a feasible solution.

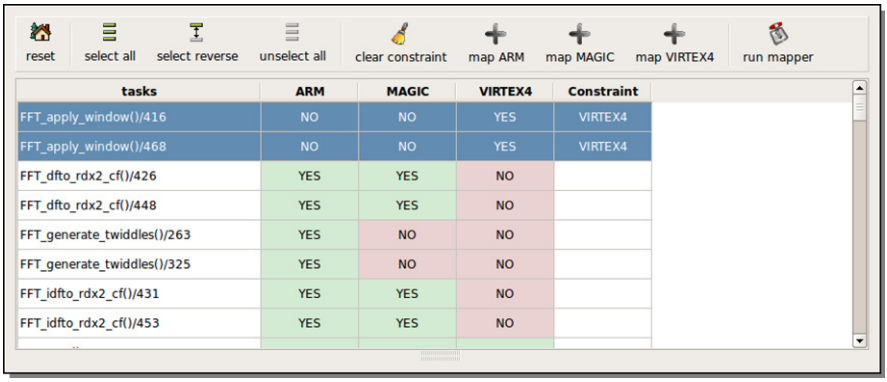
- **Multiple Source Files Support**. To support full C projects where applications can span multiple source files and contain preprocessing macros and directives, hArmonic requires a complex analysis of the code to reconcile multiple symbols in each compilation unit and to obtain a complete picture of the application.
- **Mapping Constraint System**. To facilitate design exploration and to allow tools and developers to influence the task mapping process, hArmonic supports a mapping constraint system that can be used to guide the mapping process. For instance, a tool can automatically generate C code specifically for DSP and instruct hArmonic to map those functions to DSP. There are three ways in which developers can define mapping constraints:
  - **Source Annotation**. In this case, a #pragma annotation can be optionally placed next to a function declaration or a function call to set the final mapping for that task. In the example below, any task relating to function f ( ) will be mapped to MAGIC (DSP) or to VIRTEX4 (FPGA).

```
#pragma map call_hw MAGIC VIRTEX4
void f() {...}
```

  - **Constraint File**. Rather than annotating the source which can span through many files and require parsing the code, the mapping constraints can be placed in a separate text file making it easier for tools to interact with hArmonic. In the example below, all tasks related to function f( ) are mapped to either ARM or MAGIC, task g( ) with id 12 must be mapped to VIRTEX4, and all tasks defined in source-file dsp_fn.c are to be mapped to MAGIC.

```
f():= ARM,MAGIC
g()/12:= VIRTEX4
dsp_fn.c:=MAGIC
```

  - **Graphical User Interface**. Additionally, developers can use hArmonic's graphical user-interface to set the constraints directly by selecting one or more tasks and associating them to the desired processing element as shown below:



- **OpenMP Support**. OpenMP directives can be introduced to indicate that two or more tasks can be parallelized. When the OpenMP mode is enabled, hArmonic is able to provide a mapping solution that exploits parallelization. In the example below, depending on the cost estimation of f( ) and g( ), these functions may be mapped to different processing elements (such as DSP and FPGA) to reduce the execution time of the application.

```
#pragma omp parallel sections
...
{
 #pragma omp section
 f();
}
#pragma omp section
{
 g();
}
```

- **Source Splitting**. In the initial versions of hArmonic, the mapping process would generate a single C source file with source annotations for each function indicating to which processing element they were assigned, and therefore which compiler to use. However, this approach turns out to be infeasible because (a) compilers would have to be made more complex so that they could compile selectively, (b) there are subtle and not so subtle differences between the C languages supported by the compilers, which mean that it would be difficult to generate one single source that all parsers could support, (c) there are specific headers required for each processing element which can be incompatible when used together in a single source. To avoid these problems, later versions of hArmonic generate one source file for each processing element that is a part of the mapping solution. Each compilation unit is a subset of the application and includes all the necessary function definitions that are mapped to a particular processing element, as well as every type and symbol definitions required for a correct compilation and execution. Furthermore, hArmonic supports C customizations in the source for each processing element to conform to the hArtes platform, backend compilers and system headers.

Table 2.7 summarizes the results of the complete automatic mapping performed by the final version of the hArmonic tool on all hArtes applications. For the largest application (Audio Enhancement) with more than 1000 tasks, it takes a single minute to complete the whole task mapping process, resulting in a 30 times speed-up. The speed-up corresponds to the ratio between the application running solely on the GPP (no mapping) and execution of the application where selected tasks have been automatically assigned to the available accelerators on the hArtes platform. The hArtes constraints column corresponds to the number of mapping restrictions generated by the mapping filter rules (see Sect. 2.7.4) in order to comply with the hArtes platform and ensure that the mapping solution is feasible.

Table 2.7 Evaluation of the main hArtes applications

hArtes application	Total number of tasks [mapped to DSP]	hArtes constraints	Time to compute solution (sec)	Speed up
From UNIVPM				
Xfir	313 [250]	1180	8	9.8
PEQ	87 [39]	261	3	9.1
FracShift	79 [13]	251	24	40.7
Octave	201 [137]	692	6	93.3
Audio enhancement	1021 [854]	4366	59	31.6
From Thales				
Stationary noise filter	537 [100]	3101	17	5.2
From FHG-IGD				
Beamformer	168 [38]	677	6	7.9
Wave-field synthesis	145 [24]	455	5	9.2

In addition, the hArmonic tool automatically achieves speed-ups that ranges from 5.7 times to more than 90 times by mapping selected tasks to available processing elements in the hArtes platform. Several factors contribute to the quality of the solution: (a) the hArtes applications have been developed with the recommended C guidelines and best practices to exploit the toolchain, (b) the performance and efficiency of the hArtes toolchain synthesis tools (GPP, DSP and FPGA compilers), (c) the use of specialized libraries such as DSPLib, and finally (d) the architecture and efficiency of the hArtes hardware platform. Further descriptions of the hArtes applications and the automatic mapping approach can be found in Chaps. 4 and 5.

2.8 Compiling for the hArtes Platform

In order to execute the produced application on the hArtes target platform, the following steps have to be performed:

- compiling the code for the GPP processing element and generating the additional code to address the MOLEN paradigm;
- compiling the code for the DSP processor;
- generating the hardware components;
- linking and creating the effective executable.

2.8.1 HGCC Compiler and MOLEN Programming Paradigm

The MOLEN programming paradigm is a paradigm [52] that offers a standard model of interaction between the processing elements of a heterogeneous systems.

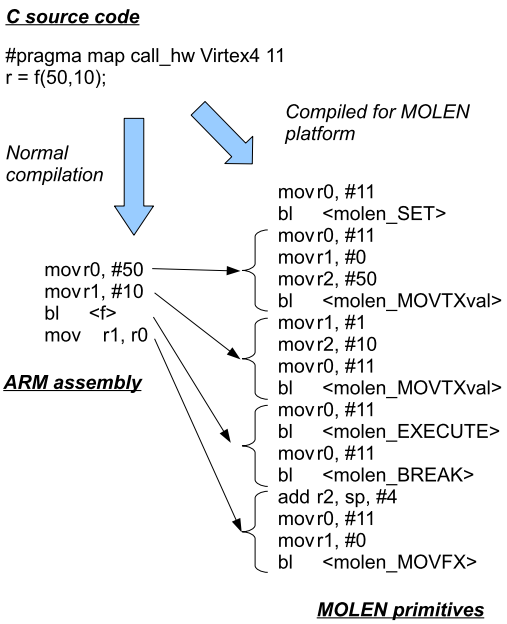


Fig. 2.31 Example of MOLEN

By extending the run-time system and the compiler with 6 primitives a theoretically unlimited number of different processing elements can be managed transparently. New types of processing elements can be added without complex modifications to the existing infrastructure—the only thing needed being the specific implementation of the 6 primitives for the new element.

The paradigm is based on the processor/coprocessor model and the basic computation unit is a function. We consider that the application is running on the General Purpose Processor (**GPP**) and the computationally intensive functions (named from this point on **kernels**) are executed by different accelerators or processing elements (**PE**). The memory is considered to be shared between the GPP and all the PE-s. The parameters have to be placed in special transfer registers that have to be accessible by both the GPP and the PE.

The modified GPP compiler, namely **HGCC**, will replace a normal call to the function with the 6 primitives as shown in Fig. 2.31.

Each of the primitive has to be implemented for each processing element. The list of the primitives for one processing element is the following:

- SET(int *id*)—performs any initialization needed by the processing element. In the context of hArtes it means: reconfiguration for FPGA, load of binary executable file for Diopsis DSP.
- MOVTXval(int *id*, int *reg*, int *val*)—moves the value *val* to the specific transfer registers *reg* that will be used by the kernel identified by *id*
- EXECUTE(*id*)—starts the kernel identified by *id* on the processing element processing element. The GPP can continue execution as this call as asynchronous.

- BREAK(*id*)—used as a synchronization primitive, will not return until the PE execution the kernel identified by *id*, has not finished execution.
- MOVFXval(int *id*, int *reg*, int &*val*)—does the reverse of MOVTXval. Will transfer the value from the register *reg*, used by kernel identified by *id* and will store it at *val*.

These functions are provided in a library, and are linked with the binary generated by the GPP compiler.

To mark a function is a kernel that has to be accelerated a pragma is used. The syntax of the pragma is the following:

```
#pragma map call_hw Virtex4 1
```

The third element in the pragma is the processing element, while the fourth element is the unique identifier of the kernel. This pragma can be placed at the following points in the program:

- function declaration—which means all the calls to that function will be replaced with MOLEN primitives.
- function call—which means that just that call will be called accelerated.

Parallel Execution and MOLEN

Even if it is not explicitly specified MOLEN is a asynchronous paradigm, i.e. the PE can execute in parallel with the GPP. To describe this in the application we use the OpenMP parallel pragma, because it is a well established syntax. We do not use any other feature of the OpenMP specification so far.

Let's assume we have two independent kernels: *kernelA* and *kernelB*. Without taking into account the parallelism the compiler will generate the code and schedule in Fig. 2.32.

Assuming *kernelA* and *kernelB* are put in an OpenMP parallel section the compiler will generate the optimized code and schedule as in Fig. 2.33.

An important aspect is that even though it uses OpenMP syntax, the parallelism is obtained without any threading on the general purpose processor, which can save a lot of execution time. To obtain such an effect the compiler does the following steps:

- it replaces the calls with the MOLEN primitives;
- when it generates the assembly code, the compiler starts processing the parallel regions one by one. For each of them, it copies to the sequential output all the code before the last BREAK primitive of the section;
- copies the rest of each section.

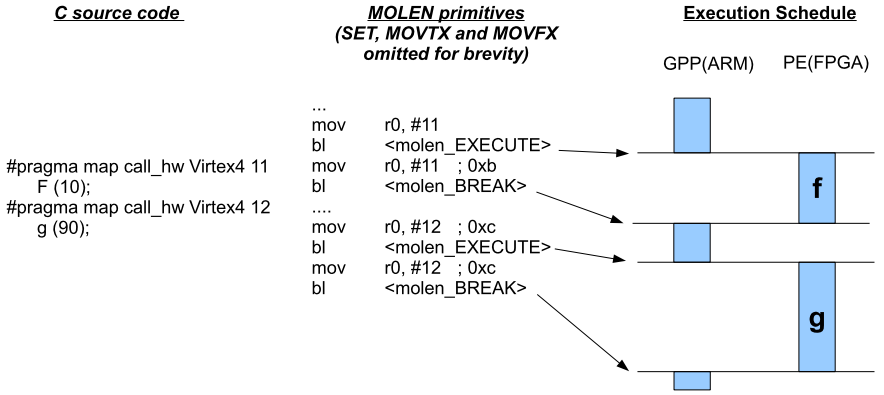


Fig. 2.32 Molen without parallelism

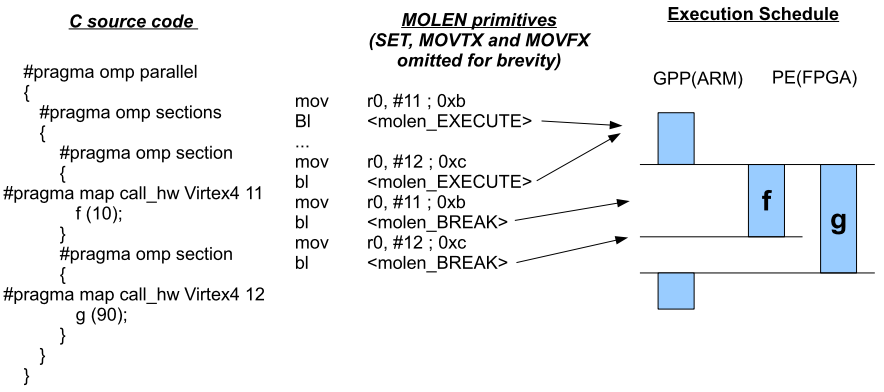


Fig. 2.33 Molen with parallelism expressed with OpenMP syntax

2.8.2 DSP Compiler

For the DSP Processing Element, the hArtes toolchain uses a commercial toolchain from Target™. This toolchain has its own binary format, the C-compiler is not fully C99 compliant and has particular syntaxes to optimize C code. One of the goal of the hArtes toolchain is to behave exactly as the compilation chain of a GPP where one single executable is created containing all symbol and debug information. So the main integration task has been to convert this proprietary image into something that could be linked together with the GPP compiler.

The integration of this tool inside the hArtes toolchain has been made possible through the *mex2elf* tool that translates Target's DSP proprietary format into an ARM ELF compliant format. A more complex task has been to re-create debugging information following DWARF2 specifications, from logs and disassembler outputs of the Target compiler. There is some debug information still missing such as: frame

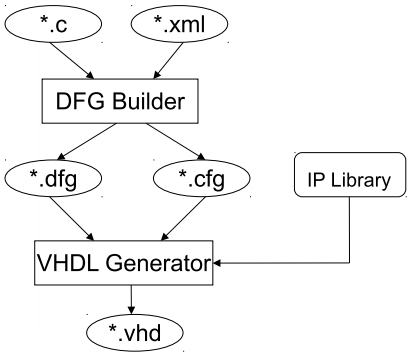


Fig. 2.34 The DWARV toolset

and variable info. However debugging location information has been produced, so it's possible put line/file or function breakpoint.

2.8.3 Hardware Generation

In this section, we present the Delft Workbench Automated Reconfigurable VHDL Generator (DWARV) C-to-VHDL generation toolset. The purpose of the compiler is to exploit available parallelism of algorithms and generate designs suitable for hardware/software co-execution. This chapter is structures as follows: First, an overview of the toolset and description of the modules it is composed of is given (Sect. 2). An explanation of the current C subset it accepts and a brief overview of possible transformation needed to make c-functions DWARV-compatible is given in Sect. 3. Finally, in Sect. 4, we will present results obtained when using the toolset with the Molen computing organisation.

DWARV Compiler

This subsection is focused on the design and implementation of the DWARV toolset for automatic generation of VHDL designs from C code considering execution on a hardware prototyping platform. We present the design flow of the DWARV toolset and explain the main components in details.

The main objective of the DWARV toolset is generation of VHDL designs from C code considering execution on a real hardware prototyping platform and without limitations on the application domain. The toolset consists of two modules, namely Data-Flow Graph (DFG) Builder and VHDL Generator, depicted in Fig. 2.34.

The input of the toolset is a pragma annotated C code. The pragma annotation identifies the C code segments to be translated into VHDL code. Currently, the translation is performed at function-level. Hence, the pragma annotation is placed just prior the function signature.

The input C code is processed by the DFG Builder. This module is implemented as a pass within the SUIF2 compiler framework [47]. Its current purpose is to transform the C code into a data-flow representation suitable for hardware generation. The SUIF2 front-end is used to generate SUIF intermediate representation (IR) of the input file. A series of code transformations are applied by the DFG Builder:

1. The code is transformed into static single assignment (SSA) form;
2. Scalar replacement is performed. During, the scalar replacement pass, the memory access expressions are analyzed and if necessary and possibly, un-aliased;
3. A second SSA pass goes over the newly generated scalar variables.

After the code is normalized, the actual data-flow graph construction is performed. For that purpose, the SUIF IR is traversed and for each operation, a corresponding DFG node is created. The edges, corresponding to the data-dependencies between the operations are also constructed. During this traversal, if-conversion is performed. In addition, the precedence order between the memory accesses as well as between the operations within the loop bodies is analyzed and the corresponding edges are inserted. As a final optimization, common sub-expression and dead code elimination are performed. The output of the DFG Builder module is a Hierarchical Data-Flow Graph (HDFG), serialized into a binary file (*.dfg).

The HDFG, used as intermediate representation, is a directed cyclic graph $G\langle V, E\rangle$. The vertices of the graph are divided into two major types: simple and compound. The simple nodes correspond to the arithmetic and logic operations, the memory transfers, the input/output parameters transfers, the constants, and the registers. Simple nodes corresponding to registers in the graph represent only the function parameters and the local variables used to transfer values produced in one loop iteration and consumed in the next iteration. The compound nodes correspond to the loops in the function body. These nodes contain the sub-HDFG of the loop body.

The edges of the graph are also two types: data-dependency edges and precedence edges. A data-dependency edge $e(v \rightarrow u), v \in V, u \in V$, indicates that the value produced by node v is consumed by node u. A precedence edge $e(v \rightarrow u), v \in V, u \in V$, indicates that the operation(s) in node v has to complete before the operation(s) in node u is initiated. The precedence edges prevent a value, prepared for the next loop iteration to be consumed in the same loop iteration. These edges are also used to order the (possibly) dependent memory accesses.

The HDFG is the input of the second module in the toolset called VHDL Generator. This module is implemented as a stand-alone Linux executable. Its current purpose is to schedule the input graph and to generate the final VHDL code. The performed scheduling is As-Soon-As-Possible. During the scheduling, each executable node, except the memory and parameters transfer and loop nodes, is assumed to take one cycle. An operation is scheduled at a given level (cycle) if all node inputs are already available (scheduled at preceding cycles) and there are no precedence constraints. The scheduling of the loop nodes is merged with the scheduling of the rest of the graph. These nodes are considered as normal nodes, until they are ready to be scheduled. When a loop node is ready to be scheduled, first all other ready nodes

are scheduled. Then the scheduling of the upper-level graph nodes is suspended and the scheduling of the compound node sub-graph is started. When the loop body is completely scheduled, the scheduling of the upper-level nodes is resumed.

The number of cycles, necessary for the memory and parameters transfers is provided as an additional input to the DFG Builder. These data are specified in the configuration file (*.xml) shown in Fig. 2.34. This file also containing the memory and the register banks bandwidth and the address size of the corresponding busses. As additional configuration parameters, the endianess of the system and the sizes of the standard data types are also specified in this file. The xml file is transformed in a text file (*.cfg) that is more suitable for processing by the VHDL Generator.

The last block in Fig. 2.34 constitutes the IP Library. The purpose of this is to provide VHDL components (primitives and cores) that are functionally equivalent to C code fragments which can not be translated automatically to a very efficient VHDL code. As an example of what is described in this library, consider the floating point division of two variables in c code. This operation would be translated to VHDL by instantiating the **fp_sp_div** core that is described in the IP Library, e.g. number of cycles the operation takes, port names of the inputs and output, size of the operands.

The generated output VHDL code represents a FSM-based design. Only one FSM is instantiated for the entire design. The transition between the states is consecutive, unless a loop-back transition is performed. The generated VHDL code is RTL as specified in IEEE-Std 1076.6-2004 [18] and uses the numeric_std synthesis packages [19]. In addition, the RTL designs are generated with the MOLEN CCU interface [28, 29, 51], which allows actual execution on a real hardware prototype platform.

C Language and Restrictions

An objective of the toolset described above is to provide support for almost all standard C-constructs that can be used in the input C code. Nevertheless, in the current version of the toolset several syntax limitations are imposed. These limitations are listed below. A note should be made that the listed restrictions apply only to the functions translated into VHDL code:

- Data types: the supported data types are integer (up to 32-bit) and one-dimensional arrays and pointers. There is not support for 64-bit data types, multi-dimensional arrays and pointers, structures, and unions;
- Storage types: only auto local storage type is supported with limitation for arrays initialization;
- Expressions: there is full support for the arithmetic and logic operations. One-dimensional array subscripting and indirection operation as memory access are also supported. There is no support for function calls, field selection, and address-of operation;
- Statements: The expression statement is limited to the supported expressions, the selection statements are limited to if-selection, and the iteration statements are

Table 2.8 C language support—data types

Data types	Current support	Future support	Future work
Integer types	long long not supported	Full	Use other compiler framework
Real FP types	Supported	Full	N/A
Complex types	Not supported	Not supported	N/A
Pointer types	1D, memory location	ND, local un-aliasing	Pointer analysis
Aggregate types	1d arrays	Full	Data manipulation extension
Unions	Not supported	Full	Data manipulation extension

Table 2.9 C language support—storage

Storage	Current support	Future support	Future work
Auto/local	No array initialization	Full for supported data types	VHDL model extension
Auto/global	Not supported	Constant	VHDL model extension
Static	Not supported	Not supported	N/A
Extern	Not supported	Not supported	N/A
Register	Ignored	Ignored	N/A
Typedef	Full	Full	N/A

Table 2.10 C language support—expressions

Expressions	Current support	Future support	Future work
Arithmetic and logic	FOR & IF	Full	Expression restore analysis extension
Function calls	Not supported	Partial	Function analysis and VHDL extension
Array subscripting	1D	Full	Data manipulation extension
Field selection	Not supported	Full	Data manipulation extension
Address-of	Not supported	Full	Data manipulation extension
Indirection	1D mem	ND mem	Pointer analysis

limited to for-statements. There is no support for jump and labeled statements. The compound statements are fully supported.

Although currently limitations on the input C code are imposed and no sophisticated optimizations are implemented, the toolset is able to translate kernels from different application domains that exhibit different characteristics. In addition, performance improvement over pure software execution is also observed [61]. For a detailed overview of the current C language support, the reader is referred to Tables 2.8, 2.9, 2.10, 2.11.

Although the DWARV toolset does not restrict the application domain, based on the presented C Language limitations above (see Table 2.8), some transformations still have to be performed on the original c-code to make it compliant with the cur-

Table 2.11 C language support—statements

Statements	Current support	Future support	Future work
Expression	Limited to supported expr	Limited to supported expr	N/A
Labeled	Not supported	Switch-case	Data-flow analysis extension
Jump	Not supported	(Switch-) break & continue	Data-flow analysis extension
Selection	If-statement	Full	Data-flow analysis extension
Iteration	For-loop	Full	Data-flow analysis extension
Compound	Full	Full	N/A

rent version of the DWARV compiler. In the future, these restrictions will be relaxed. The transformations needed to make a function DWARV-compliant are summarized below:

- The code to be transformed to VHDL has to be extracted as annotated C function.
- All external data have to be passed as function parameters.
- Multi-dimensional array accesses have to be transformed to 1D memory accesses.
- Pointers are interpreted as memory accesses (hence, no pointers to local data).
- Switch statements have to be re-written as series of if-statements.
- While and do-while loops have to be re-written as for-loops.
- Function calls have to be inlined.
- Structures or unions have to be serialized as function parameters.

Results

For the evaluation of the DWARV compiler, kernels from various application domains identified as candidates for acceleration were used. These generated designs were implemented in Xilinx's VirtexII Pro XC2VP30 FPGA and Virtex-4 XC4VFX100 and the carried experiments on the MOLEN polymorphic processor prototype [28, 29] suggest overall application speedups between 1.4x and 6.8x, corresponding to 13% to 94% of the theoretically achievable maximums, constituted by Amdahl's law [61]. For a more detailed explanation of the carried experiments and results obtained, the reader is referred to the applications chapter.

Hardware Cores for Arithmetic Operations

Many complex arithmetic operations and functions can be more efficiently implemented in hardware than software, when utilizing optimized IP cores designed from FPGA vendors. For example, Xilinx provides a complete IP core suite [58] for all widely used mathematical operations, ranging from simple integer multipliers and dividers, to customizable double precision division and square root. These IP cores are finely-tuned for speed or area optimization, thus providing robust solutions that can be used as sub-modules for larger designs. Based on this fact, we developed

a hardware components library that consists of all widely-used mathematical functions. The latter are built using various Xilinx IP cores. In the next section, we describe which functions are supported by pre-designed hardware modules and elaborate on their specifications.

Operations Supported by the Library Components Table 2.12 shows all the operations that are supported by the library components. The latter can be divided into three main categories, based on the operands used:

- IEEE Floating-Point Arithmetic Single Precision Format (32 bits);
- IEEE Floating-Point Arithmetic Double Precision Format (64 bits);
- Integer (32 bits).

As it is shown in Table 2.12, all mathematical operations are supported for both floating point formats. The reason we decided to support these operations with hardware modules, is that they require less cycles when executed with custom hardware modules, than executed on an embedded PowerPC processor in emulation mode. In addition, we developed hardware modules that perform various types of comparisons between two floating point numbers, like $>$ or $>=$. All these hardware accelerators can be designed to be also pipelined, thus reducing even more the application total execution time [55].

Except from all mathematical operations, we developed custom accelerators for functions that are commonly used in software applications written in C. These functions are also shown in Table 2.12, while Table 2.13 describes each one of them.

Implementation of the Hardware Modules In order to implement all library elements we use the Xilinx ISE 9.2i CAD tools. Xilinx provides dedicated IP cores [56] that utilize the IEEE 754 Standard [20] for all basic mathematical operations, like addition, subtraction, multiplication, division and square root. We used these IP cores as the fundamental building block for all single and double precision hardware accelerators. During the development of each library element, we tried to keep a trade-off between its latency and maximum operating frequency.

The hArtes hardware platform accommodates a Virtex4 FX100 FPGA with an external clock of 125 MHz, thus all hardware accelerators should be able to operate at least to that frequency when mapped onto the FPGA. On the other hand, increasing an element maximum operating frequency much more than 125 MHz, would introduce additional latency cycles. In this case, the element performance would be degraded, since it would never have to operate at a frequency more than 125 MHz. Based on these facts, all hardware accelerators were designed with an operating frequency up to 150 MHz, in order to make sure that they would not introduce any implementation bottlenecks when mapped with other hardware sub-modules. Furthermore, every design is fully pipelined, where a new instruction can be issued every clock cycle.

Finally, regarding the FPGA resource utilization, we tried to exploit as much as possible the dedicated XtremeDSP slices [57]. This way, we achieved two major goals: The first one is that, because of the hardwired XtremeDSP slices, all accelerators could easily operate at the target frequency of 150 MHz. The second one is

Table 2.12 hArtes library components specifications

Precision	Operation
single	$x+y, x-y, x*y, x/y, \sqrt{x}, x\%y, x>y, x<y, x>=y, x<=y, x==y,$ $x!=y$, neg(x), round(x), floor(x), ceiling(x), x2int, x2short, x2char, zero(x)
double	$x+y, x-y, x*y, x/y, \sqrt{x}, x\%y, x>y, x<y, x>=y, x<=y, x==y,$ $x!=y$, neg(x), round(x), floor(x), ceiling(x), x2int, x2short, x2char, zero(x)
int	x/y, x2fp_sp, x2fp_dp, $x\%y$
short	$x/y, x\%y$
char	$x/y, x\%y$

Table 2.13 hArtes library components specifications

Function	Explanation
round(x)	If the decimal part of x is >=0.5, then round(x) = $\lceil x \rceil$, else round(x) = $\lfloor x \rfloor$
floor(x)	floor(x) = $\lfloor x \rfloor$
ceiling(x)	ceiling(x) = $\lceil x \rceil$
x2int(x)	Keep x integer part and convert it to integer format
x2short(x)	Keep x integer part and convert it to short format
x2char(x)	Keep x integer part and convert it to char format
zero(x)	If $x == 0.0$ or $x == -0.0$ then zero(x) = 1, else zero(x) = 0

that we leave more FPGA slices available to map other sub-modules, that are automatically generated by the DWARV C-to-VHDL tool. In practise, the majority of the library elements occupies only 1% of the hArtes platform Virtex4 FPGA regular slices.

2.8.4 Linking

The hArtes linker produces a single executable, linking all the contributes coming from the different PE' compilation chains.

We use the standard GNU Linker (LD) targeted for ARM Linux and a customized hArtes Linker script (*target_dek_linux.ld*) that instructs the Linker on how to build the hArtes executable from the different PE's input sections. Each PE has an associated *text* (program section), *bss* (not initialized data), *data* (initialized data) section, plus some additional sections that correspond to platform's shared memories (if any).

The Linker and the customized linker scripts have been integrated inside the hArtes framework as last hArtes compilation pass, as shown in Fig. 2.35. The Linker script, at linking time, generates additional global GPP variables in the hArtes executable that describe the PE sections that must be loaded by the hArtes runtime. Once the executable starts the hArtes runtime uses these variables to retrieve addresses and size of the sections that must be loaded.

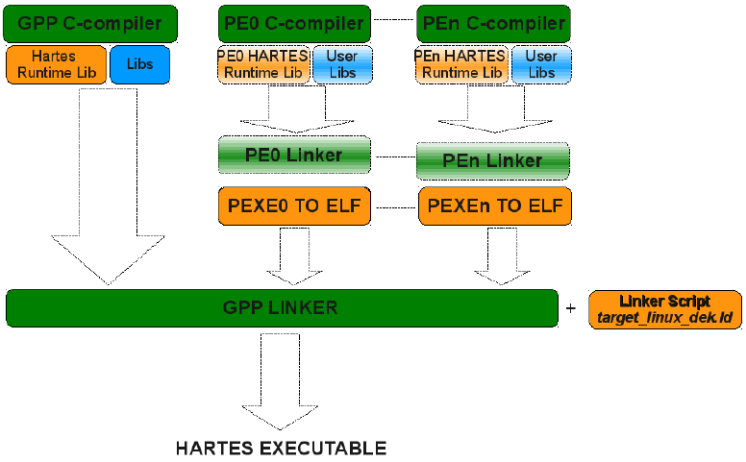


Fig. 2.35 Representation of the linking process in the hArtes toolchain

2.9 Runtime Support

The *hArtes Runtime* provides a simple device driver interface for programs to connect to the underlying Processing Elements and GPP Operating System. The hArtes runtime it is also integrated with the GPP ANSI C standard library and provides hosted I/O services to the PE (printf, fopen.).

Different functionalities have been implemented inside the hArtes Runtime to support the runtime analysis of the application on the target platform. The main features are:

- memory allocation
- profiling support
- debug support

2.9.1 Memory Allocation

Memory allocation is performed by the runtime support. It keeps trace of allocations performed by the PEs and the GPP, in order to figure out how to move and optimize data to/from GPP and PEs. It also performs basic checking on pointers passed in remote calls.

2.9.2 Profiling and Benchmarking Support

The aim of the hArtes toolchain is to optimize an application by partitioning it into pieces and mapping it onto the heterogeneous PEs of a given architecture. Profiling and benchmarking activities are thus fundamental to understand to improve harts toolchain outputs by analyzing the execution of the current status of the application.

Manual Instrumentation of the Code

The hArtes runtime provides a very accurate function to get time, that is *hget_tick*. This function can be used in the code to compute the actual time expressed in ticks (order on NS). This function cannot be used to measure time greater than 4 s (32 bit counter). Use standard timer.h functions to measure seconds.

Automatic Instrumentation of the Code

The HGCC compiler provides options to automatically instrument code, currently supporting:

- `-dspstat` that instruments all DSP remote calls.
- `-fpgastat` that instruments all FPGA calls.
- `-gppstat` that instruments all GPP calls.

Measure Power Consumption

Some HW targets have the support to measure currents. For instance the DEB board can measure power consumption via a SPI current sensor on the board.

2.9.3 Debug Support

We used **`Gnu DeBugger`**(GDB) targeted for ARM Linux, plus an additional hArtes patch to support multiple PE. This patch essentially adds an additional signal handler coming from the hArtes Runtime. This signal is treated as a standard ARM Linux breakpoint trap, interrupting the application flow execution, meanwhile the hArtes Runtime suspends the execution of other PEs (if the HW provides support). In this way a "soft" synchronization is realized and the user can inspect a stopped application.

In the current implementation we are supporting ARM + MAGIC(DSP) debugging, that is because FPGA does not support, at the moment, any kind of debugging facilities (debugging symbols, HW mechanism to suspend execution), so remote FPGA calls behave as a black box, providing the possibility to inspect only inputs and outputs before and after remote execution.

At the moment it is possible:

- To inspect the hArtes address space, including PE I/O spaces;
- to add DSP breakpoints;
- to inspect DSP program frame;
- to read/write and decode DSP registers;
- to provide GPP/DSP inter block (a breakpoint or an event on the GPP blocks the DSP and vice versa)

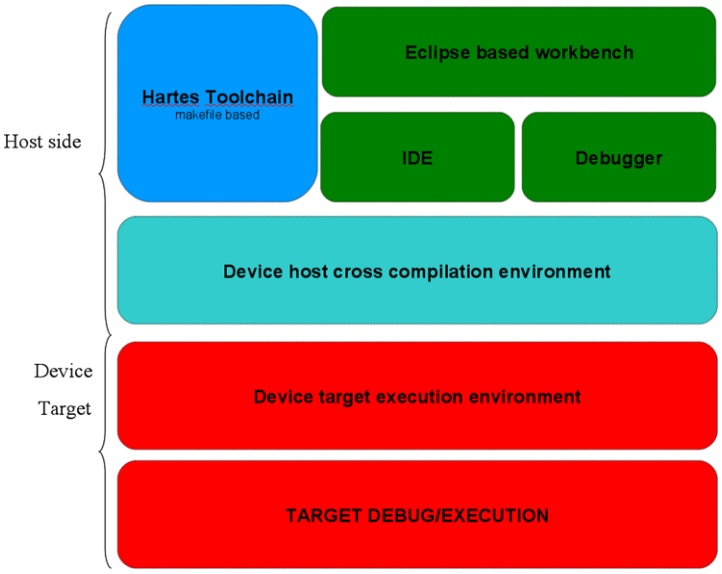


Fig. 2.36 Hartes framework

The hArtes runtime keeps trace of every thread and remote function is being executed. The user can dump the status of a running hArtes application just sending a HUP signal (see POSIX signals) to it. Exceptions or abnormal situations caught by the runtime generate an immediate dump of the process status.

2.10 hArtes Framework

The hArtes Framework makes it easy to develop Linux applications for GPP-based platforms (in our case ARM). It reduces the learning curve, shortens the development and testing cycle, and helps building reliable applications quickly.

In the hArtes view, the program flow is executed on the GPP, PEs are used to speed up computing intensive tasks. The GPP runs Linux. For the GPP the hArtes toolchain uses a standard GNU toolchain plus an additional hArtes patch, in order to support hArtes XML and pragma annotations. An overview of hArtes framework is shown in Fig. 2.36.

2.10.1 Text Based Workbench

The hArtes projects are makefile based, the makefile contains all the information needed to build an hArtes application, starting from sources. Once the hArtes framework is installed, the user has to provide application sources and a simple makefile that essentially lists the files to be passed through the hArtes toolchain.

hArtes Project Compilation The creation of an hArtes project requires a very simple very simple makefile.

The following example shows a typical hArtes makefile for a system composed of a GPP and two PEs (DSP and FPGA):

```
PROJECT = hArtesProject

## put here the common includes -I<DIR> and defines
## -D<your Define> for the project
HARTESCOMMON =
################ DSE TOOLS SETTINGS ##################
## put here the sources that you want to pass to the hArtes
## toolchain
DSECOMPILESRC =
## put here the flags for zebu (partioner)
DSEPARTITIONFLAGS =
## put here the flags for hArmonics (mapper)
DSEMAPPINGFLAGS =
#######################################################


################ Synthesis TOOLS SETTINGS ##################

####### HGCC COMPILER [GPP] #######
## put here the source you want compile for the ARM
COMPILESRC =
## GPP Compiler flags (include directories, defines..)
CFLAGS =
## GPP Linker flags (include libraries, additional libraries)
LDFLAGS =

####### CHESS COMPILER [DSP]#######
## put here the source you want compile for the DSP
DSPCOMPILESRC =
## DSP Compiler flags (include directories, defines..)
DSPCCFLAGS =
## DSP Linker flags (include libraries, additional libraries)
DSPLDFLAGS    =

####### DWARV COMPILER [FPGA]#######
## put here the source you want compile for the FPGA
FPGACOMPILESRC =
## FPGA Compiler flags (include directories, defines..)
FPGACCFLAGS =

include $(GNAMDIR)/config.mak
```

The above makefile can be generated by the `hproject_create.sh` command, available once installed the hArtes framework.

The same project can be compiled in three different configurations:

- No mapping (low effort, low results)
- Manual mapping (medium effort, best results)
- Completely automatic mapping (low effort, good/best results)

No Mapping This means that the application is compiled entirely for the GPP.

This pass will provide a working reference and allow to evaluate the baseline performance.

Manual Mapping (Martes) The user does a manual partitioning and mapping, by using `#pragma` for mapping and by putting the sources on the appropriate PE section. This pass is useful to evaluate the maximum performance of the application on the system.

Completely Automatic Mapping This kind of project is entirely managed by the hArtes toolchain that will partition and will map the application on the available PEs.

This pass is useful to evaluate the performance obtained with a very low human effort.

2.10.2 hArtes Execution

This pass is very intuitive, since it requires just to copy to the target board the single executable image produced by the toolchain and then execute it.

2.10.3 Graphical Based Workbench

The hArtes Eclipse workbench is built on the standard Eclipse development environment providing outstanding windows management, project management, and C/C++ source code editing tools. We provided additional plugins to manage hArtes projects. This plugin allows the creation of hArtes projects with two possible configurations: no mapping and completely automatic mapping.

The Eclipse IDE's fully featured C/C++ source editor provides syntax checking.

- Outline view which lists functions, variables, and declarations
- Highlights syntax errors in your C/C++ source code
- Configurable syntax colorization and code formatting for C/C++ and ARM/Thumb/Thumb2 assembly
- Full change history which can be integrated with popular source code control systems, including CVS and SVN
- Graphical configuration of parameters in the source code via menus and pull-down lists

File Transfer to Target The hArtes Eclipse distribution includes a Remote System Explorer (RSE) perspective for easy transfer of applications and libraries to the Linux file system on the target.

RSE enables the host computer to access the Linux file system on hardware targets.

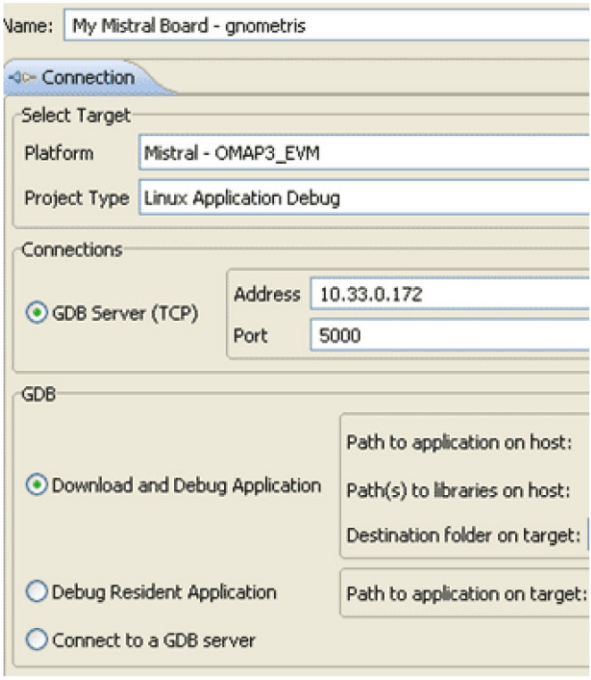
- FTP connection to the target to explore its file system, create new folders, and drag & drop files from the host machine
- Open files on the target's file system by double-clicking on them in the FTP view. Edit them within Eclipse and save them directly to the target's file system
- Shell and terminal windows enable running Linux commands on the target system without a monitor and keyboard
- Display of a list of processes running on the target

Window Management The flexible window management system in Eclipse enables the optimal use of the visual workspace.

- Support for multiple source code and debugger views
- Arrange windows: floating (detached), docked, tabbed, or minimized into the Fast View bar
- Support of multi-screen set-ups by dragging and dropping detached windows to additional monitors

Debugger Overview The Eclipse debugger is a powerful graphical debugger supporting end-to-end development of GPP Linux-based systems. It makes it easy to debug Linux applications with its comprehensive and intuitive Eclipse-based views, including synchronized source and disassembly, memory, registers, variables, threads, call stack, and conditional breakpoints.

Target Connection The hArtes debugger automates target connection, application download and debugger connection to *gdbserver* on supported platforms.



- The debugger connects to a *gdbserver* debug agent running on the target, using an Ethernet cable;
- A launcher panel automates the download of Linux applications to the hardware target by using a telnet or ssh connection;
- The debugger can connect to GPP Linux target. The Remote System Explorer (RSE) is used to manually transfer files to the target, open a terminal window, and start *gdbserver*.

Run Control Control the target's execution with high (C/C++) and low (assembler) level single-stepping and powerful conditional breakpoints.

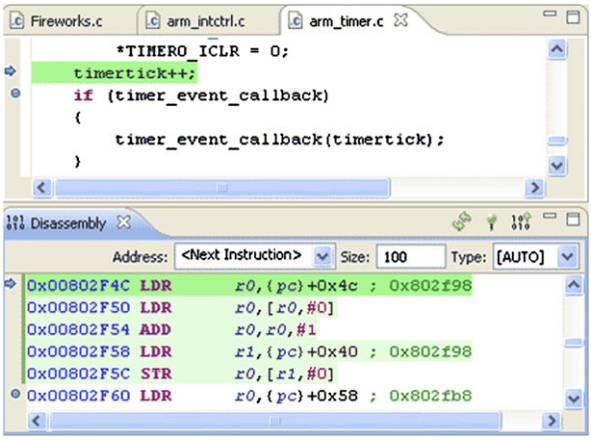
All aspects of CPU operation can be controlled from Eclipse.

- Run control: run, stop, step through source code and disassembly
- Set an unlimited number of software breakpoints by double clicking in the source or disassembly views
- Conditional breakpoints halt the processor when hit a pre-defined number of times or when a condition is true
- Assignment of actions to breakpoints, allows message logging, update views, or output messages
- If the debugger detects a slow target connection it disables the system views until the user stops stepping. This enables fast single-stepping operation

System Views The hArtes Eclipse debugger provides access to the resources inside the target device, including processor and peripheral registers, code, memory, and variables.

Synchronized source code and disassembly views provide easy application debug since they provide the possibility to

- Open as many system views at the same type as necessary. Freeze them for easy comparison of their contents over time
- Color code synchronized source code and disassembly for easy debug of highly optimized C/C++ code
- View and modify C variables and C++ classes, whether local to a function or global
- View a list of current threads and the call stack for each thread. Click on a thread or a call stack entry to focus the debugger views on that frame
- Use expressions in C-syntax on any of the system views. For example, write to a memory location the contents pointed at by pointer ptr by typing =*ptr



2.10.4 hArtes Debugging Customization

Debugging an hArtes application it's not easy. However we added to **GDB**, the capability to debug a GPP + DSP + (FPGA) hArtes application, as indicated in the Runtime support section.

In the following chapters will be described techniques to debug an hArtes application for GPP + DSP.

For the GPP and DSP we use the GNU GDB targeted for ARM and customized for hArtes. Both local (target) or remote debugging configuration are possible.

Local debugging is performed directly on the Target HW (HHP or DEB) by running `gdb` , while remote debugging is performed by running *`gdbserver`* on the target HW and **gdb** on the local host.

New GDB Commands For the DSP new commands have been added:

- info hframe that gives information about the current hArtes frame
- mreset that performs a DSP HW reset

2.11 Conclusion

This chapter aimed at presenting a global overview of the hArtes methodology to map a C-application onto an heterogeneous reconfigurable platform composed of different processors and reconfigurable logic. The different features provided by the different tools provide a large set of options to the designer to explore the design space, from the algorithm down to the mapping decisions. The main restrictions of the actual design flow is the assumption of a shared memory model and a fork/join programming model, on which OpenMP is based.

References

1. Agosta, G., Bruschi, F., Sciuto, D.: Static analysis of transaction-level models. In: DAC 03: Proceedings of the 40th Conference on Design Automation, pp. 448–453 (2003)
2. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.: Metropolis: an integrated electronic system design environment. Computer **36**, 45–52 (2003)
3. Banerjee, U., Eigenmann, R., Nicolau, A., Padua, D.A.: Automatic program parallelization. Proc. IEEE **81**(2), 211–243 (1993)
4. Beltrame, G., Fossati, L., Sciuto, D.: ReSP: a nonintrusive transaction-level reflective MPSoC simulation platform for design space exploration. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **28**(12), 1857–1869 (2009)
5. Carr, S., Kennedy, K.: Scalar replacement in the presence of conditional control flow. Softw. Pract. Exp. **24**(1), 51–77 (1994)
6. Coware N2C: http://www.coware.com/products/
7. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. Proc. IEEE **91**, 127–144 (2003)
8. Ferrandi, F., Lattuada, M., Pilato, C., Tumeo, A.: Performance modeling of parallel applications on MPSoCs. In: Proceedings of IEEE International Symposium on System-on-Chip 2009 (SOC 2009), Tampere, Finland, pp. 64–67 (2009)
9. Ferrandi, F., Lattuada, M., Pilato, C., Tumeo, A.: Performance estimation for task graphs combining sequential path profiling and control dependence regions. In: Proceedings of ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2009), Cambridge, MA, USA, pp. 131–140 (2009)
10. Ferrandi, F., Pilato, C., Tumeo, A., Sciuto, D.: Mapping and scheduling of parallel C applications with ant colony optimization onto heterogeneous reconfigurable MPSoCs. In: Proceedings of IEEE Asia and South Pacific Design Automation Conference 2010 (ASPDAC 2010), Taipei, Taiwan, pp. 799–804 (2010)
11. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987)
12. Franke, B., OBoyle, M.F.P.: A complete compiler approach to auto-parallelizing C programs for multi-DSP systems. IEEE Trans. Parallel Distrib. Syst. **16**(3), 234–245 (2005)
13. GCC, the GNU Compiler Collection, version 4.3: http://gcc.gnu.org/
14. Girkar, M., Polychronopoulos, C.D.: Automatic extraction of functional parallelism from ordinary programs. IEEE Trans. Parallel Distrib. Syst. **3**(2), 166–178 (1992)
15. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: a call graph execution profiler. SIGPLAN Not. **17**(6), 120–126 (1982)
16. Hou, E.S.H., Ansari, N., Ren, H.: A genetic algorithm for multiprocessor scheduling. IEEE Trans. Parallel Distrib. Syst. **5**, 113–120 (1994)
17. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html
18. IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, IEEE Std 1076.6-2004, available online at http://ieeexplore.ieee.org/servlet/opac?punumber=9308
19. IEEE Standard VHDL Synthesis Packages, IEEE Std 1076.3-1997, available online at http://ieeexplore.ieee.org/servlet/opac?punumber=4593
20. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008, vol. 29, pp. 1–58 (2008)
21. Ierotheou, C.S., Johnson, S.P., Cross, M., Leggett, P.F.: Computer aided parallelisation tools (CAPTools)—conceptual overview and performance on the parallelisation of structured mesh codes. Parallel Comput. **22**(2), 163–195 (1996)
22. Ismail, T.B., Abid, M., Jerraya, A.: COSMOS: a codesign approach for communicating systems. In: CODES'94: Proceedings of the 3rd International Workshop on Hardware/Software Co-design, pp. 17–24 (1994)
23. Jin, H., Frumkin, M.A., Yan, J.: Automatic generation of OpenMP directives and its application to computational fluid dynamics codes. In: Proceedings of the Third International Symposium on High Performance Computing (ISHPC 00), London, UK, pp. 440–456. Springer, Berlin (2000)

24. Jin, H., Jost, G., Yan, J., Ayguade, E., Gonzalez, M., Martorell, X.: Automatic multilevel parallelization using OpenMP. Sci. Program. **11**(2), 177–190 (2003)
25. Johnson, T.A., Eigenmann, R., Vijaykumar, T.N.: Min-cut program decomposition for thread-level speculation. In: Programming Language Design and Implementation, Washington, DC, USA (2004)
26. Kianzad, V., Bhattacharyya, S.S.: Efficient techniques for clustering and scheduling onto embedded multiprocessors. IEEE Trans. Parallel Distrib. Syst. **17**(17), 667–680 (2006)
27. Kim, S.J., Browne, J.C.: A general approach to mapping of parallel computation upon multiprocessor architectures. In: Int. Conference on Parallel Processing, pp. 1–8 (1988)
28. Kuzmanov, G.K., Gaydadjiev, G.N., Vassiliadis, S.: The Virtex II Pro MOLEN processor. In: Proceedings of International Workshop on Computer Systems: Architectures, Modelling, and Simulation (SAMOS), Samos, Greece, July. LNCS, vol. 3133, pp. 192–202 (2004)
29. Kuzmanov, G., Gaydadjiev, G.N., Vassiliadis, S.: The Molen media processor: design and evaluation. In: WASP'05 (2005)
30. Lam, Y.M., Coutinho, J.G.F., Luk, W., Leong, P.H.W.: Integrated hardware/software codesign for heterogeneous computing systems. In: Proceedings of the Southern Programmable Logic Conference, pp. 217–220 (2008)
31. Lam, Y.M., Coutinho, J.G.F., Ho, C.H., Leong, P.H.W., Luk, W.: Multi-loop parallelisation using unrolling and fission. Int. J. Reconfigurable Comput. **2010**, 1–10 (2010). ISSN 1687-7195
32. Liu, Q., et al.: Optimising designs by combining model-based transformations and pattern-based transformations. In: Proceedings of International Conference on Field Programmable Logic and Applications (2009)
33. Luis, J.P., Carvalho, C.G., Delgado, J.C.: Parallelism extraction in acyclic code. In: Proceedings of the Fourth Euromicro Workshop on Parallel and Distributed Processing (PDP'96), Braga, January, pp. 437–447 (1996)
34. Luk, W., Coutinho, J.G.F., Todman, T., Lam, Y.M., Osborne, W., Susanto, K.W., Liu, Q., Wong, W.S.: A high-level compilation toolchain for heterogeneous systems. In: Proceedings of IEEE International SOC Conference, September 2009
35. Newburn, C.J., Shen, J.P.: Automatic partitioning of signal processing programs for symmetric multiprocessors. In: PACT 96 (1996)
36. Novillo, D.: A new optimization infrastructure for GCC. In: GCC Developers Summit, Ottawa (2003)
37. Novillo, D.: Design and implementation of tree SSA. In: GCC Developers Summit, Ottawa (2004)
38. Osborne, W.G., Coutinho, J.G.F., Luk, W., Mencer, O.: Power and branch aware word-length optimization. In: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 129–138. IEEE Comput. Soc., Los Alamitos (2008)
39. Ottoni, G., Rangan, R., Stoler, A., Bridges, M.J., August, D.I.: From sequential programs to concurrent threads. IEEE Comput. Archit. Lett. **5**(1), 2 (2006)
40. Pilato, C., Tumeo, A., Palermo, G., Ferrandi, F., Lanzi, P.L., Sciuto, D.: Improving evolutionary exploration to area-time optimization of FPGA designs. J. Syst. Archit., Embed. Syst. Design **54**(11), 1046–1057 (2008)
41. Qin, W., Malik, S.: Flexible and formal modeling of microprocessors with application to retargetable simulation. In: DATE 03: Proceedings of Conference on Design, Automation and Test in Europe, pp. 556–561 (2003)
42. Quinlan, D.J.: ROSE: compiler support for object-oriented frameworks. In: Proceedings of Conference on Parallel Compilers (CPC2000) (2000)
43. Ramalingam, G.: Identifying loops in almost linear time. ACM Trans. Program. Lang. Syst. **21**(2), 175–188 (1999)
44. Sarkar, V.: Partitioning and scheduling parallel programs for multiprocessors. In: Research Monographs in Parallel and Distributed Processing (1989)
45. Sato, M.: OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors. In: Proceedings of ISSS'02, pp. 109–111 (2002)

46. Schirrmeister, F.: Cadence virtual component co-design: an environment for system design and its application to functional specification and architectural selection for automotive systems. Cadence Design Systems, Inc. (2000)
47. SUIF2 Compiler System: available online at http://suif.stanford.edu/suif/suif2
48. Susanto, K., Luk, W., Coutinho, J.G.F., Todman, T.J.: Design validation by symbolic simulation and equivalence checking: a case study in memory optimisation for image manipulation. In: Proceedings of 35th Conference on Current Trends in Theory and Practice of Computer Science (2009)
49. Todman, T., Liu, Q., Luk, W., Constantinides, G.A.: A scripting engine for combining design transformations. In: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), US, May 2010
50. Vallerio, K.S., Jha, N.K.: Task graph extraction for embedded system synthesis. In: Proceedings of the 16th International Conference on VLSI Design (VLSID 03), Washington, DC, USA, p. 480. IEEE Comput. Soc., Los Alamitos (2003)
51. Vassiliadis, S., Wong, S., Cotofana, S.: The MOLEN $\rho\mu$-coded processor. In: Proceedings of International Conference on Field-Programmable Logic and Applications (FPL). LNCS, vol. 2147, pp. 275–285. Springer, Berlin (2001)
52. Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K.L.M., Kuzmanov, G.K., Moscu Panainte, E.: The MOLEN polymorphic processor. IEEE Trans. Comput. **53**(11), 1363–1375 (2004)
53. Wang, W.K.: Process scheduling using genetic algorithms. In: IEEE Symposium on Parallel and Distributed Processing, pp. 638–641 (1995)
54. Wiangtong, T., Cheung, P.Y.K., Luk, W.: Hardware/software codesign: a systematic approach targeting data-intensive applications. IEEE Signal Process. Mag. **22**(3), 14–22 (2005)
55. Xilinx Inc: Virtex-5 APU Floating-Point Unit v1.01a, April 2009
56. Xilinx Inc: Floating-Point Operator v5.0, June 2009
57. Xilinx Inc: XtremeDSP for Virtex-4 FPGAs, May 2008
58. Xilinx Inc: IP Release Notes Guide v1.8, December 2009
59. Xilinx, Inc: Virtex-4 FPGA Configuration User Guide, Xilinx user guide UG071 (2008)
60. Yang, T., Gerasoulis, A.: DSC: scheduling parallel tasks on an unbounded number of processors. IEEE Trans. Parallel Distrib. Syst. **5**, 951–967 (1994)
61. Yankova, Y., Bertels, K., Kuzmanov, G., Gaydadjiev, G., Lu, Y., Vassiliadis, S.: DWARV: DelftWorkBench automated reconfigurable VHDL generator. In: FPL'07 (2007)
62. Zhu, J., Gajski, D.D.: Compiling SpecC for simulation. In: ASP-DAC'01: Proceedings of the 2001 Asia and South Pacific Design Automation Conference, Yokohama, Japan, pp. 57–62 (2001)

http://www.springer.com/978-94-007-1405-2

Hardware/Software Co-design for Heterogeneous Multi-core Platforms
The hArtes Toolchain
Bertels, K. (Ed.)
2012, XXII, 234 p., Hardcover
ISBN: 978-94-007-1405-2