View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Archivio istituzionale della ricerca - Politecnico di Milano

POLITECNICO
MILANO 1863

Data Transfers Analysis in Computer Assisted Design Flow of FPGA Accelerators for Aerospace Systems

M. Lattuada, F. Ferrandi, M. Perrotin

M. Lattuada, F. Ferrandi, and M. Perrotin. Data transfers analysis in computer assisted design flow of fpga accelerators for aerospace systems. *IEEE Transactions on Multi-Scale Computing Systems*, PP(99):1–14, 2017

The final publication is available via http://dx.doi.org/10.1109/TMSCS.2017.2699647

©2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or list, or reuse of any copyrighted component of this work in other works

Data Transfers Analysis in Computer Assisted Design Flow of FPGA Accelerators for Aerospace Systems

Marco Lattuada, Fabrizio Ferrandi, and Maxime Perrotin

Abstract—The integration of Field Programmable Gate Arrays (FPGAs) in an aerospace system improves its efficiency and its flexibility thanks to their programmability, but increases the design complexity. The design flows indeed have to be composed of several steps to fill the gap between the starting solution, which is usually a reference sequential implementation, and the final heterogeneous solution which includes custom hardware accelerators. Among these steps, there are the analysis of the application to identify the functionalities that gain advantages in execution on hardware and the generation of their implementations by means of Hardware Description Languages. Generating these descriptions for a software developer can be a very difficult task because of the different programming paradigms of software programs and hardware descriptions. To facilitate the developer in this activity, High Level Synthesis techniques have been developed aiming at (semi-)automatically generating hardware implementations of specifications written in high level languages (e.g., C). With respect to other embedded systems scenarios, the aerospace systems introduce further constraints that have to be taken into account during the design of these heterogeneous systems. In this type of systems explicit data transfers to and from FPGAs are preferred to the adoption of a shared memory architecture. The first approach indeed potentially improves the predictability of the produced solutions, but the sizes of all the data transferred to and from any devices must be known at design time. Identifying the sizes in presence of complex C applications which use pointers can be a not so easy task.
In this paper, a semi-automatic design flow based on the integration of an aerospace design flow, an application analysis technique, and High Level Synthesis methodologies is presented. The initial reference application is analyzed to identify which are the sizes of the data exchanged among the different components of the application. Next, starting from the high level specification and from the results of this analysis, High Level Synthesis techniques are applied to automatically produce the hardware accelerators.

Index Terms—Space Systems, FPGA, High Level Synthesis, Code Analysis

1 Introduction

The evolution of the aerospace systems is characterized by the improvement of the on-board sensors which are able to capture larger and larger amount of data. However, the transmission bandwidth between them and the Earth stations has not equivalently grown. For this reason, to actually exploit the availability of larger amount of data, more and more pre-processing has to be executed directly on the aerospace system to reduce the data to be sent to the Earth. Traditional high performance computing devices such as general purpose processors could provide the required computational power, but they are not suitable for these systems because of the requirements in terms of low power and high dependability. On the contrary, micro processors developed for space environment meet such requirements but cannot provide the required computational power. The solution which has instead been identified for solving this issue is the use of Rad-Hard Field Programmable Gate Array (Rad-Hard FPGA). This type of devices indeed has good characteristics in terms of reliability and power consumption and guarantees a significant amount of computational power.

There are unfortunately two issues which arise during designing of systems which include FPGA devices: the generation of the hardware accelerators and their integration with the rest of the system. The first issue consists of the difficulty of designing the hardware accelerators implemented on the FPGA devices which will be included in the final solutions. Their design indeed requires specific skills, such as the knowledge of the HDL languages, which are often not owned by software designers or by aerospace engineers. To overcome this problem, High Level Synthesis techniques [1] have been proposed. They aim at automatically generating hardware implementations starting from high level specifications by means of (semi-)automatic design flows. While state of the art techniques well address the issue about the generation of hardware accelerators, their integration requires extra effort to be covered.

One of the most critical problems of the integration of hardware accelerators in aerospace is the management of the data transfers. FPGAs are indeed characterized by local memories, where data have to be moved in order to efficiently exploit their computational power. The design of solutions for heterogeneous systems characterized by distributed memory can be a hard task. For this reason, infrastructures like Heterogeneous System Architecture (HSA) [2] have been developed to hide the complexity of distributed memory systems and to avoid the necessity of performing explicit data transfers. They make designing solutions simpler, since some architectural details can be ignored by the designer, but they introduce a potential issue. This type of mechanisms indeed, by hiding the actual position of

• *M. Lattuada and F. Ferrandi are with the Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy (e-mail: {marco.lattuada, fabrizio.ferrandi}@polimi.it)*
• *M. Perrotin is with ESTEC, European Space Agency (e-mail: {Maxime.Perrotin}@esa.int)*

the accessed data, can introduce significant unpredictable variance in the time required to perform the memory accesses. Since predictability is a mandatory constraint in space systems scenario, the architectures characterized by explicit data transfers are preferred because they do not have this sources of unpredictability. These solutions require precise information at design time about the amount of data transferred during the execution of the application to correctly predict the delay of the communication in the worst case. A conservative approach (e.g., transferring more data than required) would allow a correct predictability analysis, but it would imply an increment of the required resources both in terms of memory and computational power. Indeed, not only larger FPGA memories would be necessary to store the input and the output data, but also faster processing elements could be required to compensate the longer delay of the data transfers. In an heterogeneous embedded system there are also other causes of unpredictability (e.g., cache misses), but analyze and remove all of them is out of the scope of this paper.

The identification of the sizes of the transferred data in applications which are designed from scratch is usually very easy, but most of the times this approach cannot be considered, since the development starts from existing reference sequential applications. Moreover, if the reference applications are written in C source code, the analysis of their data transfers can be a very hard task because of the presence of pointers which can partially or totally hide the data characteristics and in particular their size. A possible approach consists of requiring the designer to directly provide this information, but there are two possible drawbacks. The first one is that a significant knowledge of all the parts of the analyzed application and of all the used libraries is required. The second is that this type of approach can be subject to errors performed by the designer.

In [3] a design flow which partially addresses the issues of including FPGA hardware accelerators in a space system is presented. The design flow is based on the integration of High Level Synthesis methodologies in a framework [4] for the development of critical systems, but it is still semi-automatic: there are several steps that have to be performed by hand by the designer during the development of the solution. In this paper the design flow is extended with a technique for the analysis of the applications aimed at automatically identifying the size of transferred data. In this way one of the most complex steps which was demanded to the developer is automatized, simplifying the design process. This is performed by statically analyzing the structure of a C source code application, identifying for each pointer parameter of each function what is the maximum size of the data which can be pointed by it. Moreover this analysis technique allows the High Level Synthesis to be more integrated with the rest of the framework resulting in a new design flow which will be presented in this paper.

The rest of the paper is organized as follows. Section 2 discusses the related works, Section 3 presents the proposed analysis technique which is integrated in the design flow as described in Section 4. Its evaluation is discussed in Section 5, where the results about its applicability and its accuracy are presented. Finally, Section 6 draws the conclusion of this paper.

2 Related Works

Several approaches have been proposed to support the development of heterogeneous space embedded systems. For example, Ludtke et al. [5] proposed a reconfigurable system composed of different processing elements (including a FPGA device), an ad-hoc operating system, and a middle-ware to allow the exploitation of the same architecture for the implementation of different specifications. The focus of this work is mainly on the characteristics that the hardware/software architecture must have, while no specific framework or tool are proposed to program it. On the contrary, the framework proposed in [6] and [7] aims at helping the designer in choosing the best combination of FPGA device and fault-tolerant strategy, but the design of the hardware accelerators for the particular combination is still demanded to the developer. Another possible approach to help the designer in the exploitation of FPGA devices was proposed by Greco et al. in [8]. The USURP framework was extended adding a Hardware Abstraction API, very similar to the GNU Scientific Library API, which provides transparent access to a set of already implemented hardware accelerators. This approach has the advantage of not requiring knowledge of hardware design nor of HDL languages, but only a limited set of hardware accelerators are available: the functions not already included in the hardware library cannot be accelerated by the FPGA. A whole framework for the design of space systems was also proposed by Deshmukh et al. [9]. The authors proposed a new Domain Specific Language for describing the different components of an application targeting space systems and provided a framework to automatically generate the code to implement them. FPGA devices are however not considered in this work and the designer is forced to use a new unique language for developing the whole application. The TASTE framework [4], which is exploited by the proposed design flow, instead hides the modeling language allowing the developer to use different languages (HDL included) to describe and implement the single components of the application.

The problem of computing the size of the exchanged data is usually not addressed by High Level Synthesis tools [10]. Legup [11] for example adopts a shared memory architecture with the processor and the accelerators sharing off-chip memory. Since the accelerators exchange the data with the rest of the system by reading and writing this memory, the information about the size of the data is not required. The accelerators generated by Vivado HLS [12] can access external data into two main ways: by means of bus interfaces or by means of local memories where the exchanged data are temporary stored. In the former case, the information about the size of the input data is not required, in the latter case this information is mandatory and must be provided by the user since it is not computed by the tool. Since this information can be computed by the analysis technique proposed in this paper, the technique can also be integrated as pre-processing step of other High Level Synthesis tools. Finally, FPGA hardware accelerators can be designed to work on data streaming (e.g., [13]): for this type of applications the overall size of exchanged data is not significant.

The size of the exchanged data is not relevant only in space systems including FPGAs, but it has to be considered whenever there are heterogeneous systems characterized by explicit data transfers. The problem of computing these quantities is typically not addressed by means of automatic techniques: the usually adopted solution consists of requiring the designer to provide the necessary information. In the TASTE framework this is accomplished by exploiting ASN.1 [14] which must be used by the designer to specify the characteristics (e.g., the sizes) of the data exchanged between the different components of the designed application. In Section 4.2 it will be shown how ASN.1 descriptions can be generated starting from the results of the proposed analysis technique. Another possible approach consists of embedding the information directly in the source code of the application. This objective can be obtained with newly proposed language extensions and annotation formats designed to facilitate the data transfers to accelerator memories. In some cases the proposals start from OpenMP API [15] which was originally designed for shared memory multiprocessor systems and which did not allow designers to specify explicit data transfers. For this reason, different extensions to OpenMP API were proposed (e.g., [16], [17]) to include support to accelerators like DSPs and GPUs, some of them explicitly aimed at specifying the size of transferred data. Native support for accelerators has been integrated in OpenMP API Version 4.0, released in July 2013: new pragma directives have been introduced which allow the designer to specify the offloading of tasks and data to accelerators. Like in the previously proposed extensions, the designer has to annotate the data transfers with pragmas explicitly specifying the size of pointer parameters.

In a similar way other programming infrastructures, even if natively designed with support to heterogeneous architectures, require the designer to explicitly define the size of data transfers. For example, OpenCL API [18], typically exploited for programming heterogeneous systems including FPGAs [19], includes a function (`clSetKernelArg`) to explicitly define an input data parameter of an offloaded kernel and its size. The OpenACC [20] standard instead requires to describe the input and the output of each offloaded kernel by means of pragma annotations (`copyin` and `copyout`). All these approaches need the direct interaction with the designer while on the contrary the static analysis proposed in this paper solves this problem automatically.

The problem addressed by the proposed analysis technique is quite close to the alias analysis problem, but it has some particular features which require an ad-hoc technique to solve it. Moreover, perfect alias analysis is a very time-consuming activity [21] which can be non-affordable in case of very complex applications: an ad-hoc technique which covers exactly the pointed data size problem has to be preferred [22]. A similar problem was addressed by Yong et al. in [23] where static analysis of source code is performed to compute a safe approximation of the range of memory locations which can be accessed through a pointer. Even if it has a different aim, its results could be useful to solve the problem addressed by the proposed analysis technique, but its limitations are so significant to reduce its actual applicability. As stated by the authors, their range analysis relies on the presence of constants in the source code to derive meaningful ranges and does not record information about the relationship between variables, significantly limiting the set of analyzable code patterns. Furthermore, the range analysis is not inter-procedural, so it is not applicable in presence of data transferred across different functions. On the contrary, the analysis technique proposed in this paper has a very limited set of preconditions, smaller than the limitations usually applied to critical software, so, how shown by the experimental results of Section 5, it can be applied to a large set of embedded applications, without any further interaction with the designer, producing meaningful and accurate results.

3 Proposed Analysis Technique

In this section, the proposed analysis technique is presented. In particular, Section 3.1 gives its overview and describes the limitations of its applicability, then Section 3.2 briefly describes the produced output. The details of the technique are presented in Section 3.3, while Section 3.4 discusses its time and space complexity.

3.1 Overview of the Analysis Technique

The proposed analysis technique aims at identifying at compile time in a C source code application how much large can be the data structures pointed by each pointer, and in particular by each pointer parameter, during an application execution. In the rest of this paper, this quantity will be identified as *Pointed Data Size (PDS)*. This information is mandatory for the design of hardware accelerators in the TASTE flow, but the results of the analysis can be exploited in whatever type of design flow targeting critical heterogeneous embedded systems characterized by explicit data transfers.

The proposed analysis can be applied on the source code or on the intermediate representation adopted by a compiler, but for the sake of readability and generality, its application to C source code will be presented. In Section 4.2 an example of its implementation applied to GCC intermediate representation will be presented. The produced information ideally should have the following properties:

- *correctness*: the real PDS cannot be larger than the computed, otherwise the generated code could produce wrong results;
- *accuracy*: the computed PDS has to be as much close as possible to the real value to minimize the data transfer delays and the memory usage;
- *completeness*: the PDS of every pointer should be computed.

The accuracy of the obtained results can be more critical than in the alias analysis problem. Accepting conservative solutions could indeed imply too long data transfers and too high usage of memory, which can be a very critical resource in an space embedded system. The proposed methodology addresses this issue by relaxing the requirement of the completeness: on a limited set of rare known code patterns, the analysis is not able to produce a result. Note that unsupported patterns can be easily identified during the analysis: in this case the technique does not produce any result. Moreover, inapplicability can be limited to a subset of an application (i.e., only to one or few functions), while the remaining code can be correctly analyzed. The missing information can be integrated by the designer (e.g., in the TASTE flow by directly specifying the ASN.1 types of the input and output parameters which cannot be analyzed). The experimental results presented in Section 5 will show how the unsupported patterns are actually rare in embedded systems applications.

The patterns, whose presence limits the applicability of the analysis, are the following:

- a pointer is assigned in a function `foo` and then used in a function `bar` which calls `foo`; this can happen if the pointer is passed by address from `bar` to `foo` or if the pointer is a global variable:

```
void foo(int ** arg) {
   *arg = malloc(10);
}

void bar() {
   int * temp;
   foo(&temp);
   /*some usage of temp */
}
```

- a function returns a pointer to a memory space dynamically allocated inside the function or to a global variable:

```
int global[10];
int * custom_malloc(int size) {
   return malloc(size * sizeof(int));
}
int * get_global() {
   return &global;
}
```

Moreover the analysis assumes that the application cannot provoke any memory access error such as out of bound access, invalid memory access, buffer overflow, etc. Note however that the absence of dynamic memory allocations and the correctness of the memory accesses are usually already verified in critical applications, so that these requirements do not actually limit the applicability of the proposed technique in the considered design flow.

There is instead mainly one pattern which introduces inaccuracy in the analysis results: pointer variables with multiple assignments:

```
char * string;
if(condition)
   string="title";
else
   string="-";
process(string);
```

Note that this problem is not solved by adoption of Static Single Assignment form [24] : in this case the approximation would be introduced by the *Phi* instructions.

The proposed analysis technique has been mainly designed for the analysis of sequential applications. Nevertheless, it can be easily extended to applications parallelized by means of OpenMP or pthread library. In the former case the instances of the private variables of the different sections have to be considered as different variables. In the latter case, the calls to pthread library functions have to be managed in an ad-hoc way.

3.2 Output of the proposed analysis

The output of the proposed analysis consists of the PDS of each pointer present in the analyzed application measured in terms of number of pointed elements (i.e., the overall size of the pointed data divided by the size of one element), which is a target independent quantity. The independence of this information from a particular target can be useful during design space exploration for generic heterogeneous systems, but it is not necessary during development of applications in the TASTE framework since the exact size of all the basic types must be explicitly specified in ASN.1 descriptions. For example, if the type to be described by means of ASN.1 is an array of 10 integer, not only it is necessary to specify the number of elements of the array, but it is also necessary to explicitly specify the precision of the single integer. A PDS can be a non-negative integer number, if it can be analytically computed at compile time, or an expression, when its value is not fixed but depends on the particular input of the application. In the following example PDS(`p`)=3 and PDS(`q`)=`atoi(argv[1])`:

```
int main(int argc, char ** argv){
   int * p, * q;
   int var = atoi(argv[1]);
   p = malloc(sizeof(int) * 3);
   q = malloc(sizeof(int) * var);
   ...
}
```

Note that in case of the TASTE design flow, the PDSs of the pointer parameters must be necessarily constant non negative numbers. Nevertheless, supporting expression in PDSs can extend the applicability of the proposed technique to different embedded systems design scenarios.

In some cases copying an amount of data equal to the PDS of a pointer starting from its current value does not guarantee that all the necessary data are transferred. For example if the address of an intermediate cell of an array is assigned to a pointer:

```
int array[5];
int * p = &array[2];
```

thanks to pointer arithmetic through `p` it is possible to access all the cells of `array` (including `array[0]` and `array[1]`). A flag associated with the PDS of a pointer is used to signal that some memory locations can be accessed through this pointer and a negative offset, so that the whole range [`p - PDS(p), p + PDS(p)`} has potentially to be transferred. Note that as a consequence the size of the corresponding type would be $2 \cdot sizeof(array)$.

The output of the proposed analysis is not limited to the PDS of each formal pointer parameter of each function. Indeed, storing the PDSs also of the actual parameters of each function call allows to optimize the data transfers in each call site. Furthermore, intermediate results (e.g., PDSs of all local and global variables) could be exploited during application partitioning in hardware/software co-design flows.

Because of the characteristics of the C language, defining which are all the pointers for which the PDS has to be computed is not trivial. A pointer can be part of a more complex structure (like a structure or an array). To identify each pointer, including the ones contained in complex structures, the analysis associates a string with it. This string, which is called *Pointer Path (PP)*, is recursively computed applying the rules listed in Table 1 (for the sake of brevity only the most significant rules are reported). Differently from *Access Path*s [25], PPs refer to the pointers and not to the memory locations pointed by them. The set of PPs obtained analyzing the application is the set of pointer expressions whose PDS is computed. For example, in the following code the identified PPs are `variable.nested.pointer` and `variable.pointer`:

```
typedef struct {
  int * pointer;
  int value;
} s3;
typedef struct {
  s3 nested[10];
  int * pointer;
} s4
s4 variable[10];
```

Different pointers can have the same PP and so the same computed PDS even if the actual ones are different, potentially introducing inaccuracy in the produced results. In case of recursive structures (e.g., linked lists), only one element is considered.

3.3 Details of the proposed analysis

Algorithm 1: Code Analysis.

```
1  BuildCallGraph()
2  foreach Strongly Connected Component of the call graph in top-down order do
3      repeat
4          foreach function $f_i \in$ Strongly Connected Component do
5              foreach statement $s_j$ of function $f_i$ do
6                  CheckPointerAssignment($s_j$)
7              end
8              PropagateAssignment()
9              foreach call point $c_j$ in function $f_i$ do
10                 ActualToFormal($c_j$)
11             end
12         end
13     until PDSs do not change
14 end
```

Algorithm 1 sums up the proposed analysis technique. It starts by computing the call graph of the application (line 1) and then its code is analyzed in detail function by function. The information about the PDSs of the formal parameters has to be already available at the beginning of the analysis of a function because these parameters can be assigned to local variables or used as parameters of called functions. For this reason, caller functions have to be analyzed before called functions: the analysis of the whole application starts from `main` function (or from the entry point specified by the developer) and then all the other functions of the application are examined following one of the topological orders induced by the call graph (lines 2-14). In presence of recursive calls, the functions in the corresponding strongly connected component have to be iteratively analyzed until the PDSs of their formal parameters do not change. Note however that recursion is usually forbidden in critical embedded systems.

The analysis of each function consists of several phases which store results in four main data structures:

- `MayAlias`: stores for each PP all the other PPs with which it may alias;
- `PDS`: stores for each PP its PDS in pointed element size;
- `SymPDS`: stores for each PP the expression containing its PDS;
- `Offset`: stores for each PP if it can be used with negative offset.

At the beginning of the analysis of a function the PDS of all the PPs included in function parameters are always available, while the PDSs of the PPs included in global variables are available only if these are globally initialized or are initialized by an already analyzed function. Information about global variables assigned in called functions would not be available and causes the abort of the technique. All the statements of the function are analyzed looking for pointer assignments (line 6); since the performed analysis is flow-insensitive, the order of the analysis of the single statements is not relevant. `MayAlias`, `SymPDS`, `PDS` and `Offload` are updated according to the rules of Table 2 (for the sake of brevity only the most significant rules are reported). The following phase is the elaboration of the collected information (line 8): `SymPDS`, `PDS` and `Offset` are updated on the basis of `MayAlias`. For example, if a pointer `p` is assigned to a pointer `q` with smaller PDS, the PDS of the latter is updated. The complete update of the data structures can be done in linear time if the propagation is performed starting from PPs with largest PDS. Finally, during the last phase of the analysis of the function, the PDSs of all the PPs contained in formal parameters of called functions are eventually updated. In particular the PDS of a PP contained in a formal parameter is updated if the PDS of the actual parameter in one of the calls is larger than it. In this way, information about the PDSs of formal parameters will be available when the called functions will be analyzed.

3.4 Time and Space Complexity of the Proposed Analysis

In this section the time and the space complexity of the proposed analysis are discussed. In the following S identifies the set of statements of the analyzed application, S_i is the set of statements of function f_i. The time complexity of the analysis of a function f_i (lines 5-10) is $O(|S_i|)$. `PropagateAssignment` can be indeed performed in $O(|S_i|)$ if the assignments are considered starting from the ones relative to the PPs with largest PDS. If the application does not contain any recursion (i.e., its call graph is acyclic), each function is examined once, so the overall complexity of the analysis is $O(|S|)$. If the application contains recursive calls, some functions can be analyzed more than once. In the worst case during each iteration of the analysis of a strongly connected component of the call graph the PDS of only one PP contained in a formal parameter is updated to its final value. The PDS of each PP contained in formal parameters can be updated only once: from the current value to the largest value of the corresponding actual parameters discovered during the analysis of functions included in the strongly connected component. For this reason, the number of repeated analysis of a same function is bounded by the number of PPs in its parameters, so the overall time complexity of the proposed analysis is $O(|S| \cdot pn)$ where pn is the maximum number of PPs in the parameters of a function.

TABLE 1
Rules for the construction of *PP(exp)*.

1	PP(variable) = variable
2	PP(*expression) = PP(expression)*
3	PP(expression[index]) = PP(expression) if expression is an array
3	PP(expression[index]) = PP(expression)* if expression is a pointer
4	PP(expression operator expression2) = PP(expression) if expression is of pointer type
5	PP(expression.field) = PP(expression).field if expression is of struct type
6	PP(expression.field) = PP(expression) if expression is of union type

TABLE 2
Rules for data structures updating.

Id	Structure of the assignment	Updates	Description
1	`p1=p2`	`MayAlias[PP(p2)].insert(PP(p1));`	Assignment between pointers.
2	`p1=p2+constant`	`MayAlias[PP(p2)].insert(PP(p1));` `Offset[PP(p1)] = true;`	Assignment between pointers with offset. `Offset` is set for the left-side pointer since it points to something inside some data structure.
3	`p1=&p2[index]`	`MayAlias[PP(p2)].insert(PP(p1));` `Offset[PP(p1)] = true;`	
4	`p1=p2+variable`	`MayAlias[PP(p2)].insert(PP(p1));` `Offset[PP(p1)] = true;`	
5	`p1=malloc(constant)`	`if constant > PDS[PP(p1)]` `PDS[PP(p1)] = constant;`	Dynamic allocation of memory of known size: if the allocated memory is larger than current known `PDS` of left-side pointer, this is update.
6	`p1=malloc(expression)`	`if SymPDS[PP(p1)] != empty` `Abort();` `SymPDS[PP(p1)] = expression;`	Dynamic allocation of memory of unknown size: if a memory space of unknown size has already allocated to the pointer the analysis aborts, otherwise the expression storing the unknown size is associated with the pointer.
7	`p1=&variable`	`if 1 > PDS[PP(p1)]` `PDS[PP(p1)] = 1`	Address assignment: if the `PDS` of left-side pointer is 0, this is updated to 1.
8	`p1=&array[index]`	`if sizeof(array) > PDS[PP(p1)]` `PDS[PP(p1)] = sizeof(array)` `Offset[PP(p1)] = true`	Array address assignment: if the size of the variable in the right side is larger than `PDS` of left-side pointer, this is updated; moreover, `Offset` is set for the left-side pointer since it can point inside the array.
9	`p1=function(args)`	`foreach pointer argument arg` `MayAlias[PP(arg)].insert(PP(p1));` `Offset[PP(p1)] = true;`	The result of a function is assigned to a pointer; since the called function has not yet been analyzed, it is assumed that whatever input pointer can be returned by the function; for library functions, ad-hoc rules can be defined.

`p1` and `p2` are pointers or expressions whose type is a pointer. `constant`, `variable` and `index` are an integer constant, an integer variable and an integer variable or constant. `array` is an array of generic elements.

The space complexity is determined by the size of `PDS`, `SymPDS` and `Offset` which, at the end of the execution of the analysis, store information about each PP. For this reason, the space complexity of the proposed methodology is $O(|PPs|)$ where PPs is the set of PP of the application.

4 Integration of the Proposed Analysis Technique with the TASTE Framework

This section presents how the proposed analysis technique has been integrated in a design flow based on the TASTE framework. Section 4.1 briefly describes the TASTE Framework, providing the background of the design flow presented in this paper while Section 4.2 describes the design flow integrating the proposed analysis technique.

4.1 The TASTE Framework

TASTE (*The ASSERT Set of Tools for Engineering*) [4] is a development framework for the design of applications for real time safety-critical embedded systems. The framework was originally created in 2008 as the final result of ASSERT (*Automated proof based System and Software Engineering for Real-Time applications*), a research project co-founded by the Sixth Framework Programme for Research and Technology Development of the European Union, which was coordinated by the European Space Agency and which involved about 30 industrial and accademic partners. In the following years European Space Agency has continued to support the framework by funding several follow-up activities to maintain and extend it.

The TASTE framework is composed of a collection of tools, most of which released under GPL/LGPL license, aimed at building in a semi-automatic way a distributed real time system. It supports different operating systems (RTEMS, Linux and Linux with Xenomai), different processors (x86, x86-64, LEON2, LEON3, and ERC32 BSP) and, by means of the integration of device drivers as functional models, external devices (e.g., ethernet network interfaces, serial ports, Spacewire interfaces, etc.). The main aim of the framework is to allow the system designers to focus their attention on the design of the algorithms composing the specification and not on the implementation details

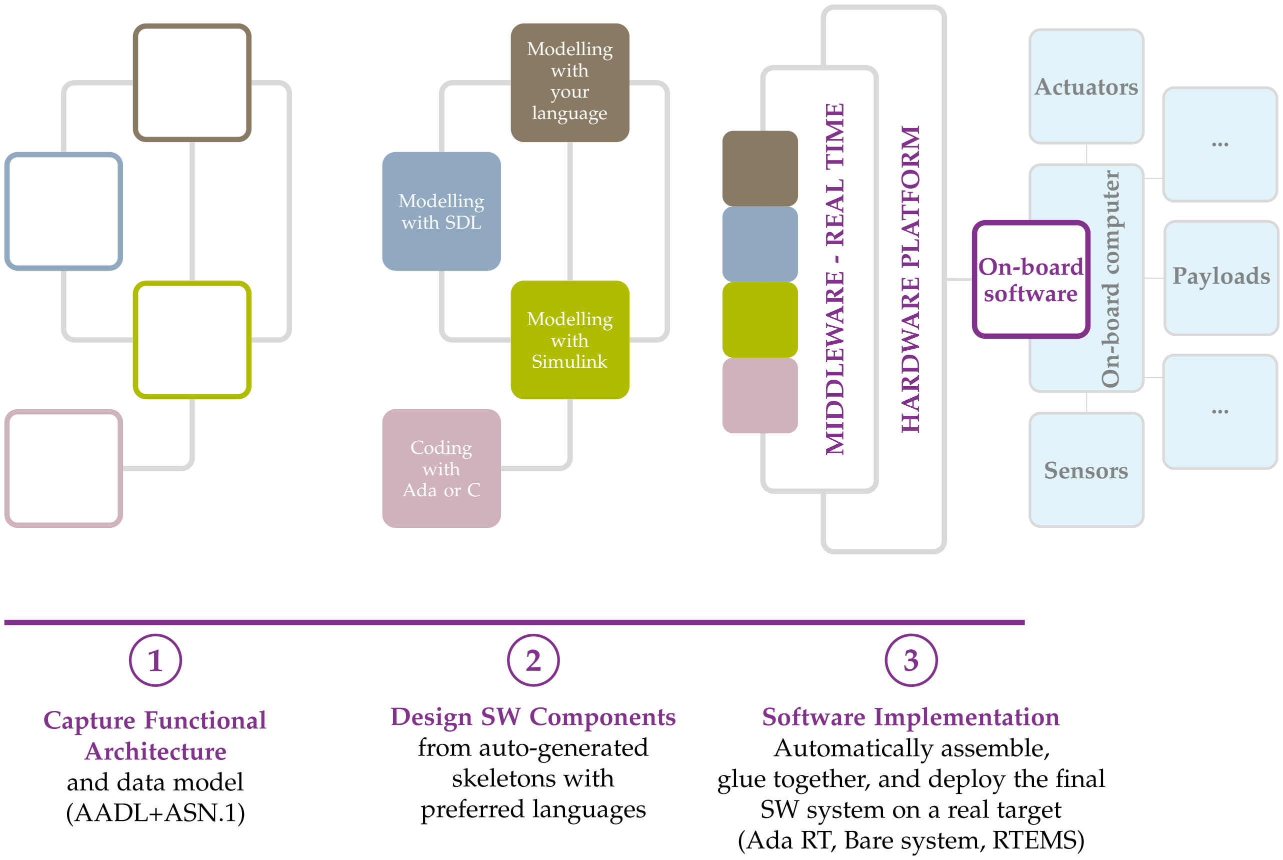
Modelling with your language
Modelling with SDL
Modelling with Simulink
Coding with Ada or C
MIDDLEWARE - REAL TIME
HARDWARE PLATFORM
Actuators
...
On-board software
On-board computer
Payloads
...
Sensors
1
Capture Functional Architecture
and data model (AADL+ASN.1)
2
Design SW Components
from auto-generated skeletons with preferred languages
3
Software Implementation
Automatically assemble, glue together, and deploy the final SW system on a real target (Ada RT, Bare system, RTEMS)

Fig. 1. The design flow of the TASTE framework.

related to the particular combination of operating system and hardware components. Nevertheless, the adoption of a single unique language (e.g., UML) for modeling each aspect of a potentially very heterogeneous system has been considered unrealistic. On the contrary, the TASTE framework does not try to remove or reduce the heterogeneity of the system, but tries to hide this heterogeneity to the designer without requiring to adopt a unique formal modeling language. The single parts of the systems can be described by the designer in the preferred language: TASTE currently supports Matlab/Simulink, SDL, C, Ada, and VHDL as input description languages. Starting from very different languages, the TASTE framework hides all the details about communication among heterogeneous subsystems, but allows the designer to verify at design time in a formal way the critical properties of the generated system.

The generic design flow implemented in the TASTE framework is presented in Figure 1. The main steps are:

- *Capture Functional Architecture*: in this phase the designer builds the *Interface view* of the application. This view defines the skeleton of the application: which are the software components (i.e., collections of functions) which compose the application and which are their exposed interfaces. For each interface the input and the output data have to be specified by means of ASN.1 types. The TASTE framework provides a graphical tool (*Interface view editor*) to perform this activity.
- *Design SW Components*: the designer fills the skeleton by implementing all the software components; each software component can be implemented with one of the languages supported by the TASTE framework.
- *Software Implementation*: the framework automatically generates the glue code which allows the different software components to interact and exchange data; finally the software system is deployed on the target.

Beside the tools aimed at implementing these steps, the TASTE framework integrates also a set of tools for analyzing the design solution and in particular to perform schedulability analysis (MAST [26] and CHEDDAR [27]).

During the different steps of the design flow, the TASTE framework exploits two languages to formally describe the characteristics of the designed application:

- *AADL* [28], which is used to describe the *Interface view*; note that AADL descriptions are automatically generated by the tools of the TASTE framework, so its knowledge is not required.
- *ASN.1* [14], which is used to specify the types of the data exchanged between the different components of the designed application. The framework only provides the ASN.1 descriptions for the basic types (e.g., integer, float): the description of more complex types used in function interfaces has to be provided

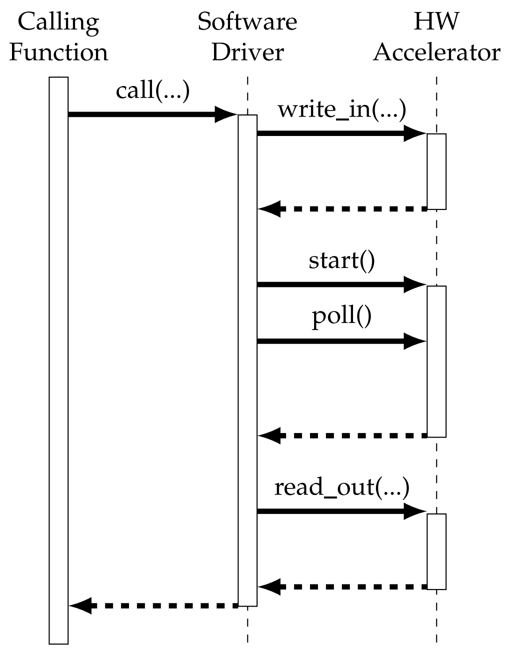

Fig. 2. The Sequence Diagram of Hardware Function Invocation.

by the designer or it has to be generated by other tools as proposed in this paper. The adoption of this standard in the TASTE framework guarantees the correct interaction among functions assigned to different processing elements with different endianness and different data size.

Aggregate data types (i.e., structures and arrays) can be specified in *ASN.1*, but their size must be fixed and known at design time. If an interface will be implemented by means of a C function, the sizes of all the input and output parameters must be known. If this information is not directly known by the designer, for example because the C implementation is legacy code, it can be automatically extracted by means of the proposed analysis technique.

The only FPGA board supported by the current version of the TASTE framework is the GR-CPCI-XC4V board [29]. This board has been developed as a cooperation between Aeroflex Gaisler and Pender Electronic Design. It is a compact PCI board containing a Virtex4 XC4VLX FPGA, 16 Mbyte of FLASH prom and up to 256 Mbyte of SDRAM. The access to the GR-CPCI-XC4V board is provided in the TASTE platform by means of the PCI bus. Each input and output parameter of each synthesized function is mapped in a memory address. The steps performed during the invocation of a function mapped on the FPGA are shown in Figure 2. When a software function has to invoke a function implemented in hardware, it calls its driver (`call(...)`) which writes (`write_in(...)`) in the opportune FPGA memory locations the input parameters through the PCI bus. The information collected by the PDS analysis allows the driver to know the exact amount of data that have to be transferred. Note that the generated accelerator can only access them: all the other data of the application cannot be accessed. After that all the input parameters have been written, the driver starts the execution of the hardware accelerator by writing (`start()`) a memory mapped control register. Next, it continuously checks (`poll()`) for the value stored in the memory mapped control register until the hardware accelerator ends its computation. Finally, it performs a set of memory readings (`read_out(...)`) aimed at retrieving the output of the hardware accelerator computation, and then returns these data to the function which performs the call to the hardware module. Also in this case, the information collected by the proposed analysis technique is mandatory to correctly perform the data transfer. Note that a different driver is automatically generated for each application since they depend on the particular signatures of the functions mapped on hardware. The drivers introduce an overhead in the overall execution time of the computation which is mainly due to the data transfers at the beginning and at the end of the execution of the accelerated functions and which depends on the amount of data to be exchanged. The more precise the results of the PDS analysis are, the smaller is the introduced overhead.

The limitation in the support of other FPGA devices is mainly related to the generation of software drivers. Adding the support to other FPGA devices connected through PCI bus can be easily achieved. On the contrary, adding the support to new devices connected through different buses (e.g., Spicewire) requires the implementation of the necessary software driver generator.

4.2 Proposed Design Flow

This section presents the design flow for the automatic generation of hardware accelerators in the TASTE framework exploiting the information generated by the PDS analysis and High Level Synthesis methodologies. These are implemented in *Bambu*, an open source tool, part of the PandA framework [30], developed at Politecnico di Milano and aimed at assisting the designer during the High Level Synthesis of complex applications. *Bambu* is written in C++, it can be freely downloaded under GPL license and it has been tested with different Linux distributions (e.g., Centos, Debian, Ubuntu). *Bambu* supports the generation of hardware accelerators for different devices of different vendors (i.e., Xilinx, Intel, Lattice).

The selection of the functions that have to be synthesized in hardware is demanded to the designer of the application: the integration of automatic techniques for performing HW/SW partitioning [31] is out of the scope of this work. Since *Bambu* accepts only C source code files as input, only functions written in this language can be synthesized as hardware accelerators. The support to the automatic generation of hardware accelerators starting from SDL has been indirectly added by modifying *OpenGEODE* [32]. The tool has been extended by adding a backend for the generation of C source code starting from SDL descriptions. This C source code can then be used as input of *Bambu*, so that, by combining *OpenGEODE* and *Bambu*, it is possible to generate hardware modules starting from SDL descriptions. All the other languages supported by the TASTE framework are instead not supported by the tool.

In the proposed design flow, *Bambu* performs a preprocessing step to include the HDL of hardware accelerated

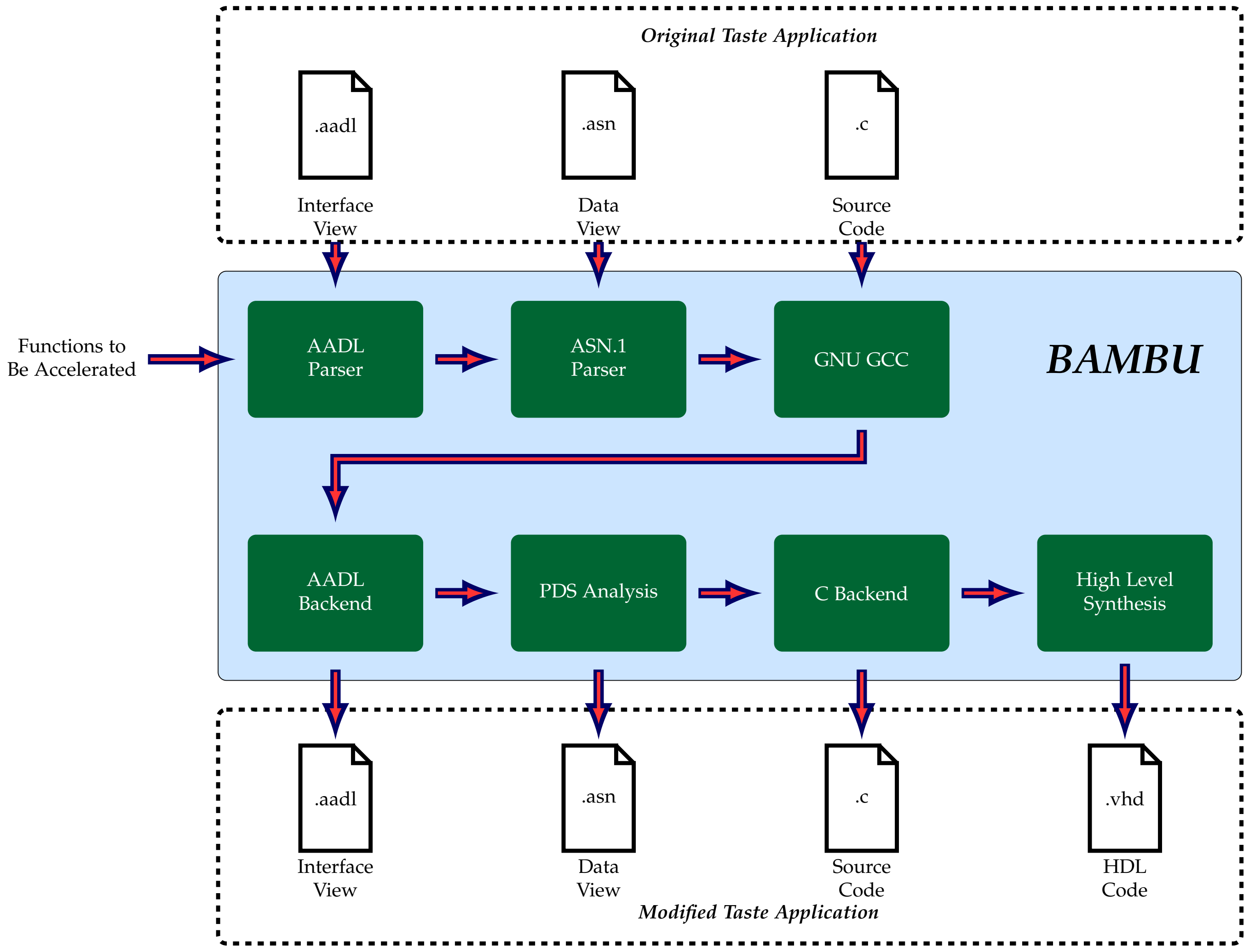

Fig. 3. The High Level Synthesis flow targeting the TASTE FPGA architecture.

functions in the TASTE application description which is then used as input of the rest of the design flow. Figure 3 shows the design flow implemented in *Bambu* for this aim. Its inputs are:

- *Interface View*, i.e., the file specifying the different functions composing the application. Note that the functions to be accelerated in hardware have not to be specified in this file.
- *Data View ASN.1 files*, i.e., the ASN.1 descriptions of the types used in the interfaces of the functions. Information about parameters of hardware accelerated functions can be missing.
- *C Source files*, i.e., the files containing the implementation of the functions of the application.
- *Hardware Accelerated Functions*, i.e., the list of the functions that have to be accelerated by executing them on FPGA.

At the end of its internal design flow *Bambu* produces:

- *Interface View*, i.e., the file specifying the different functions composing the application enriched with information about hardware accelerated functions.
- *Data View ASN.1 files*, i.e., the ASN.1 descriptions of the types used in the interfaces of the functions enriched with information about parameters of hardware accelerated functions obtained with PDS analysis.
- *C Source files*, i.e., the files containing the implementation of the functions of the application modified by replacing calls of hardware accelerated functions with calls to the corresponding drivers.
- *HDL description of the FPGA architecture*, i.e., HDL files containing the VHDL description of the accelerators and of the architecture connecting them with the rest of the system.

In the following, the *Bambu* intermediate steps will be detailed.

The High Level Synthesis flow starts from the analysis of the *Interface View* file, which is the only one that has actually to be explicitly provided. All the other files previously listed are automatically identified recursively analysing the extracted information. From the *Interface View* file two types of information can be extracted:

- The source files containing the C implementation of functions.
- The list of the *Data View aadl* files to be analysed; from these aadl files, the tool extracts the list of *Data View ASN.1* files that have to be modified.

Bambu reads by means of a GCC plugin [33] the C implementation of the functions to be synthesized in hardware and produces an intermediate representation based on GIMPLE [34]. After that all the necessary information has been collected, the rest of the *Bambu* design flow can be applied.

The next step (*AADL Backend*) is the generation of the modified *Interface View*: the original version is updated including the information about the hardware accelerated functions. This information includes their list, which files contain their HDL description and which are the other functions calling them.

Next, the *PDS analysis* presented in Section 3 is performed. By analyzing the GIMPLE intermediate representation of the functions that have to be implemented on the FPGA, the PDSs of all the pointer parameters are computed. Applying the analysis directly to this type of intermediate representation instead of C source code presents several advantages:

- with respect to the starting C source code, the GIMPLE intermediate representation has a more regular structure so that the number of different patterns that have to be considered is smaller.
- since the intermediate representation is in Static Single Assignment form [24], a different pointed data size can be computed for each version of a pointer.
- the results of the alias analysis and of the optimizations already performed by the compiler can be exploited to improve the accuracy of the analysis.

The produced information will be used by High Level Synthesis methodology, but it is also embedded in the updated version of the *Data View ASN.1* file to be used during the generation of the coupled software drivers controlling the data transfers to and from the FPGA memory. If during the analysis a non-supported pattern is found, the tool produces an incomplete version of the *Data View ASN.1* file which has to be completed by the designer. The results presented in Section 5 will show how, even considering generic embedded system code without the restrictions required in critical systems, these patterns are actually rare.

For each accelerated function parameter type the information to be included is:

- *Its structure*: the padding data that the hardware accelerator expects to find in input aggregate parameters (i.e., structures and arrays) and the padding data added by accelerators in output aggregate parameters; this information guarantees the portability of exchanged data between software and hardware and it will be exploited to generate the adding and the removal of the padding data in the software driver.
- *Its size*: ASN.1 requires to specify the size of all the exchanged data. In case of pointer parameters, the size depends on the number of pointed elements. The analysis presented in Section 3 is exploited to compute these values.
- *Its endianness*: the hardware accelerators generated by *Bambu* internally adopt little endianness.

Next step (*C Backend*) is the generation of the modified version of the C source code of the application. With respect to the source code of the starting version of the application, the only applied change is the replacement of all the calls of the hardware accelerated functions with the calls to the corresponding software drivers (described in Section 4.1). Note that *Bambu* just introduces the call to the drivers, but does not generate their implementation which will be produced by the rest of the TASTE framework using the information contained in the modified *Interface View* and in the modified *Data View*.

Last step of the design flow executed by *Bambu* is the actual *High Level Synthesis*. In the current version of the tool the TASTE FPGA architecture is generated targeting the GR-CPCI-XC4V board [29], but *Bambu* supports 15 FPGA devices of 3 different vendors. Moreover the support of new devices can be easily integrated in *Bambu* by means of XML files. The outcome of the High Level Synthesis design flow targeting the TASTE FPGA architecture is:

- a VHDL file containing the description of the top architecture to be implemented on the FPGA, i.e., the architecture connecting accelerators with the rest of the system.
- a VHDL file containing the structural descriptions of all the synthesized C functions.

The TASTE framework automatically generates the software drivers for interfacing the accelerators implemented on the FPGA. The software drivers hide most of the implementation details of the interaction between the general purpose processor and the FPGA, but they are not sufficient to make software functions and hardware modules communicating. Indeed, as in the software part of the application it is necessary to use software drivers to fill the gap between the low level bus and the application, in a similar way on the FPGA device it is necessary to instantiate some components to connect the PCI bus with the accelerators. The architecture which *Bambu* generates to make hardware accelerators accessible through the PCI bus is presented in Figure 4. The hardware accelerators are not directly connected with the PCI bus, but are directly connected to an internal communication infrastructure based on the *ARM Advanced Microcontroller Bus Architecture (AMBA)* [35].

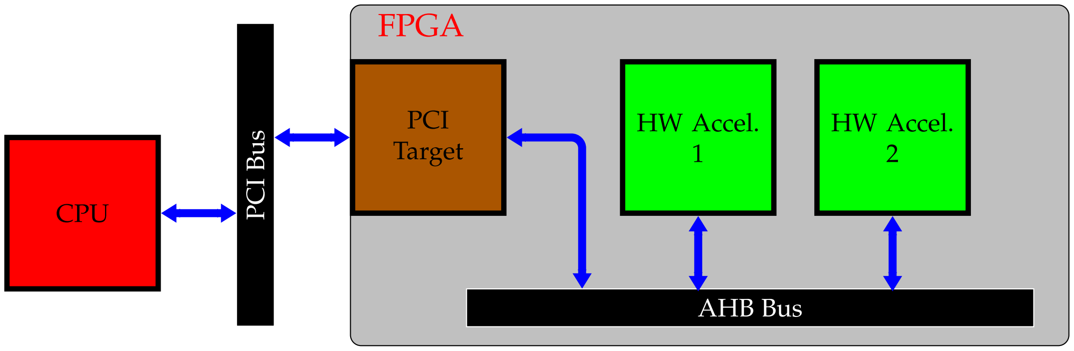

Fig. 4. The TASTE FPGA Architecture.

AMBA bus is an open standard on-chip interconnect specification, originally developed by ARM, aimed at facilitating the interconnection on System-on-Chip of components developed by different designers. The components used in the TASTE FPGA architecture to implement such type of communication infrastructure are taken from the GRLIB IP Library [36], a set of reusable IP components designed for FPGA released under GNU GPL License. The advantages of using such type of communication infrastructure are:

- more than one hardware accelerator can be integrated in the architecture;
- each hardware accelerator can be designed independently from the rest of the system;
- the implementation details of the selected FPGA board are hidden since ad-hoc implementations of all the components for all the supported boards are provided;
- the porting of the TASTE FPGA architecture to different boards is facilitated.

More in details the exploited components are:

- the *PCI target interface* component which implements a PCI slave interface towards the external PCI bus and a AHB master interface towards the internal communication infrastructure. This component acts as a bridge connecting the PCI bus and the AMBA bus.
- the AHB bus. The hardware accelerators are connected directly to the AHB bus, so they have to implement a slave interface for this type of bus.

The FPGA communication infrastructure originally adopted in the TASTE FPGA architecture included a AHB bus and a APB bus. The updating of the FPGA communication infrastructure was possible since this is completely transparent to the generated software driver and since *Bambu* generates not only the HDL of the hardware accelerators but also of the whole FPGA architecture.

At the end of the execution of *Bambu*, the TASTE design flow can be applied to its outcome. From this point on, the only significant differences with respect to a TASTE design flow targeting a system not using FPGA are the generation of the FPGA software drivers and the generation of the bitstream to be loaded on the FPGA, but these steps are completely transparent to the user.

5 Experimental Results of PDS Analysis

In this section the accuracy, the correctness, and the completeness of the PDS analysis are evaluated by applying it to the benchmarks of the following suites: DSP Stone [37], NAS Parallel [38], OmpSCR [39], Powerstone [40], and Splash 2 [41]. Table 3 reports their characteristics: these benchmarks contain a significant number of pointer parameters, despite being designed for embedded systems. Moreover, even if they have not been explicitly developed for space systems, they implement algorithms which are used in this scenario (e.g., image processing, data elaboration, graph analysis).

As described in Section 2, since to the best of our knowledge there is not any other technique explicitly aimed at computing the size of transferred data, the results obtained with the proposed analysis technique cannot be compared with the results of other techniques.

The real PDS of each pointer is collected by profiling the memory allocations and accesses with an ad-hoc instrumented version of each analyzed benchmark run on a LEON2 processor [42]. This is a 32-bit microprocessor compliant with the SPARC V8 ISA, it is based on the LEON architecture originally designed by the European Space Agency, and it is currently developed by Cobham Gaisler AB and licensed under GNU GPL.

Table 3 reports how many functions cannot be analyzed by PDS analysis because of the presence of one of the unsupported patterns described in Section 3.1. All the benchmarks from DSP Stone and Powerstone suites can be fully examined since they do not include any of the unsupported patterns. The NAS Parallel benchmarks can be almost fully analyzed: a function of one of these benchmarks cannot be analyzed because of the presence of a global pointer which is initialized in a called function. On the contrary, the number of functions of benchmarks from OmpSCR and Splash 2 suites where the analysis fails is larger (16 and 17 respectively). The patterns which cause the failures are mainly the initialization of global pointers in called functions and the functions returning pointers to dynamically allocated memory.

The real PDSs of the pointers of all the other functions have been compared with the results of the analysis to verify

TABLE 3
Characteristics of the benchmarks of each suite.

Suite	Number of benchmarks	Avg. number of functions	Avg. number of non-analyzable functions	Avg. number of call sites	Avg. number of parameters	Avg. number of pointers included in parameters
DSP Stone	35	3.23	0.00	3.95	0.79	0.67
NAS Parallel	8	17.25	0.13	96.38	3.79	1.86
OmpSCR	17	14.94	0.94	76.00	2.33	0.83
Powerstone	10	6.00	0.00	15.10	1.13	0.42
Splash 2	12	26.62	1.41	204.50	2.47	0.94

TABLE 4
Results of application of PDS analysis on all the functions

Suite	Actual Pointer Parameters	%Overestimations	Average overestimation	
			Relative	Absolute (bytes)
DSP Stone	93	0%	0%	0
NAS Parallel	1,434	0%	0%	0
OmpSCR	1,072	1.56%	0.01%	0.25
Powerstone	64	1.97%	0.01%	0.98
Splash 2	2,307	3.89%	0.01%	1.33

Actual Pointer Parameters is the overall number of parameters of the functions of the benchmarks of the suite which are pointers or which include at least a pointer; *Overestimations* is the percentage of parameters for which the PDS analysis computes a PDS larger than the real; *Average overestimation* is the average size of the overestimations of PDS.

TABLE 5
Results of application of PDS analysis only on misestimated functions.

Suite	Average overestimation		Maximum Overestimation	
	Relative	Absolute (bytes)	Relative	Absolute (bytes)
DSP Stone	0%	0	0%	0
NAS Parallel	0%	0	0%	0
OmpSCR	37.12%	16.22	86.97%	80
Powerstone	55.5%	5	55.54%	5
Splash 2	37.5%	34.21	80.00%	640

Average overestimation is the average size of the overestimations of PDS; *Maximum overestimation* is the largest computed overestimation.

its correctness (i.e., a computed PDS should be always not smaller than the real PDS) and accuracy (i.e., how much large is the difference between the computed PDS and the real PDS). For the sake of brevity, the results of the benchmarks belonging to the same suite have been aggregated and reported in Table 4 and Table 5. Note that, all the results, originally computed in terms of pointed element size (i.e., the number of elements of pointed type), have been converted in bytes to allow the aggregation of the data. The computation of the average and of the maximum of the absolute overestimations in bytes can produce different results for different architectures because of the different data sizes: the data reported in the tables use the same data sizes considered by GNU GCC when targeting LEON2 processor. On the contrary, the average and the maximum of the relative overestimations do not depend on the particular target architecture.

Table 4 reports the average results obtained by considering only the functions with at least one pointer parameter. The results show how the number of function parameters whose PDS is misestimated is very small (less than 4% in the worst case). The proposed methodology computes accurately the PDSs of all the pointer parameters of benchmarks from DSP Stone and NAS Parallel suites since the memory allocations and accesses are characterized by code patterns with limited complexity. The benchmarks from Powerstone suite have a more complex structure, but they can still be accurately analyzed. The analysis has to make a conservative approximation introducing an overestimation of its PDS only for one parameter of a function. The accuracy obtained in the analysis of the functions of Splash 2 benchmarks is instead slightly worse. These benchmarks are characterized by the presence of more complex code patterns, like multiple assignments of a same pointer, which introduce approximations in the computed PDSs. These approximations however mainly concern some small auxiliary data structures of the application: the PDSs of the main data, which are the object of the most computational intensive parts of the applications, are correctly computed.

Table 5 reports the average and the maximum overestimations obtained by considering only misestimated functions: both the functions without pointer parameters and the functions correctly analyzed are not included in the aggregated data of this table. These data show how, even if the average and the maximum relative overestimations introduced during the few inaccurate computations are quite large (37.5% and 80.0%), their absolute sizes are very small (the average is 34.21 bytes while the maximum is 640 bytes). For this reason, the further overhead in terms of data transfer delay and memory usage caused by the inaccurate computation of PDSs is not relevant. Finally, significant overestimations are also obtained for a limited number of parameters of benchmarks from OmpSCR suite. Even for the benchmarks of this suite, the overestimations mainly concern some auxiliary data, which are exploited in internal generation of execution reports, while the PDSs of the main data are correctly computed: the absolute maximum overestimation is indeed of only 80 bytes.

The experimental results obtained by analyzing a set of embedded system benchmarks show how the proposed analysis technique is effectively able to accurately compute the PDSs of pointers for most of the functions. The few overestimations indeed are small enough, in absolute value, to not introduce any extra significant penalty in the data transfer time nor in the FPGA memory resource usage.

6 Conclusions

In this paper an analysis technique to accurately identify the pointed data sizes in legacy C applications has been proposed. This technique has been integrated in a new design flow for the automatic generation of hardware accelerators integrated in the TASTE framework. The use of explicit data transfers and of ASN.1 in TASTE indeed requires to correctly compute the sizes of transferred data. The experimental results show that the proposed analysis technique is effectively able to accurately identify the pointed data sizes in legacy C applications, allowing its integration in the presented design flow.

References

[1] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE TCAD*, vol. 30, no. 4, pp. 473–491, April 2011.

[2] P. Vogel, A. Marongiu, and L. Benini, "Lightweight virtual memory support for many-core accelerators in heterogeneous embedded socs," in *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2015, pp. 45–54.

[3] M. Lattuada, F. Ferrandi, and M. Perrotin, "Computer assisted design and integration of fpga accelerators in aerospace systems," in *2016 IEEE Aerospace Conference*, March 2016, pp. 1–11.

[4] European Space Agency, "TASTE," http://taste.tuxfamily.org/.

[5] D. Ludtke, K. Westerdorff, K. Stohlmann, A. Borner, O. Maibaum, T. Peng, B. Weps, G. Fey, and A. Gerndt, "OBC-NG: Towards a reconfigurable on-board computing architecture for spacecraft," in *Aerospace Conference, 2014 IEEE*, March 2014, pp. 1–13.

[6] N. Wulf, A. George, and A. Gordon-Ross, "A framework to analyze, compare, and optimize high-performance, on-board processing systems," in *Aerospace Conference, 2012 IEEE*, March 2012, pp. 1–14.

[7] ——, "Memory-aware optimization of FPGA-based space systems," in *Aerospace Conference, 2015 IEEE*, March 2015, pp. 1–13.

[8] J. Greco, G. Cieslewski, A. Jacobs, I. Troxel, and A. George, "Hardware/software interface for high-performance space computing with FPGA coprocessors," in *Aerospace Conference, 2006 IEEE*, 2006, pp. 10 pp.–.

[9] M. Deshmukh, B. Weps, P. Isidro, and A. Gerndt, "Model driven language framework to automate command and data handling code generation," in *Aerospace Conference, 2015 IEEE*, March 2015, pp. 1–9.

[10] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct 2016.

[11] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013. [Online]. Available: http://doi.acm.org/10.1145/2514740

[12] Xilinx, "Vivado Design Suitei - High-Level Synthesis," http://www.xilinx.com, 2016.

[13] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in c with roccc 2.0," in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2010, pp. 127–134.

[14] O. Dubuisson and P. Fouquart, *ASN.1: Communication Between Heterogeneous Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[15] OpenMP, "Application Program Interface, version 4.0," July 2013. [Online]. Available: http://www.openmp.org

[16] J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. de Supinski, "OpenMP for accelerators," ser. IWOMP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 108–121.

[17] R. Ferrer, J. Planas, P. Bellens, A. Duran, M. Gonzalez, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Optimizing the exploitation of multicore processors and GPUs with OpenMP and OpenCL," ser. LCPC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 215–229.

[18] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, May 2010.

[19] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with fpgas," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 35:1–35:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014951

[20] OpenACC, "Application Program Interface, version 1.0," November 2011. [Online]. Available: http://www.openacc.org

[21] V. T. Chakaravarthy, "New results on the computability and complexity of points–to analysis," *SIGPLAN Not.*, vol. 38, no. 1, pp. 115–125, Jan. 2003.

[22] M. Hind, "Pointer analysis: haven't we solved this problem yet?" ser. PASTE '01. New York, NY, USA: ACM, 2001, pp. 54–61.

[23] S. Yong and S. Horwitz, "Pointer-range analysis," in *Static Analysis*, ser. Lecture Notes in Computer Science, R. Giacobazzi, Ed. Springer Berlin Heidelberg, 2004, vol. 3148, pp. 133–148.

[24] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct 1991.

[25] W. W. Cheng, B.and Hwu, "Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation," *SIGPLAN Not.*, vol. 35, no. 5, pp. 57–69, May 2000. [Online]. Available: http://doi.acm.org/10.1145/358438.349311

[26] M. Gonzalez Harbour, J. Gutierrez Garcia, J. Palencia Gutierrez, and J. Drake Moyano, "MAST: Modeling and analysis suite for real time applications," in *Real-Time Systems, 13th Euromicro Conference on, 2001.*, 2001, pp. 125–134.

[27] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: A Flexible Real Time Scheduling Framework," *Ada Lett.*, vol. XXIV, no. 4, pp. 1–8, Nov. 2004. [Online]. Available: http://doi.acm.org/10.1145/1046191.1032298

[28] P. Feiler, D. Gluch, and J. Hudak, "The Architecture Analysis & Design Language (AADL): An Introduction," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2006-TN-011, 2006. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7879

[29] Cobham Gaisler AB, "GR-CPCI-XC4V Leon Compact-PCI Deveopment board," http://www.gaisler.com.

[30] Politecnico di Milano, "Panda framework," http://panda.dei.polimi.it.

[31] G. De Micheli, R. Ernst, and W. Wolf, Eds., *Readings in Hardware/Software Co-design*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[32] "OpenGEODE," http://opengeode.net.

[33] S. Callanan, D. J. Dean, and E. Zadok, "Extending gcc with modular gimple optimizations," in *Proceedings of the 2007 GCC Developers' Summit*, Ottawa, Canada, July 2007, pp. 31–37.

[34] L. J. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan, "Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations," in *5th International Workshop on Languages and Compilers for Parallel Computing*, 1993, pp. 406–420.

[35] “ARM Advanced Microcontroller Bus Architecture (AMBA),” http://www.arm.com/products/system-ip/amba-specifications.php.

[36] Cobham Gaisler AB, “GRLIB Ip Library,” http://www.gaisler.com.

[37] V. živojnović, J. M. Velarde, C. Schläger, and H. Meyr, “DSP-STONE: A DSP-oriented benchmarking methodology,” in *ICSPAT '94*, 1994.

[38] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, “The NAS Parallel Benchmarks 2.0,” Tech. Rep.

[39] A. J. Dorta, C. Rodriguez, F. de Sande, and A. Gonzalez-Escribano, “The openmp source code repository,” in *PDP*, 2005, pp. 244–250.

[40] A. Malik, B. Moyer, and D. Cermak, “A low power unified cache architecture providing power and performance flexibility (poster session),” in *ISLPED '00*. New York, NY, USA: ACM, 2000, pp. 241–243.

[41] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 programs: characterization and methodological considerations,” in *ISCA*, 1995, pp. 24–36.

[42] Cobham Gaisler AB, “Leon 2 Processor.” [Online]. Available: http://www.gaisler.com

Marco Lattuada received the Master and the PhD degrees in Computer Engineering from Politecnico di Milano, Italy, in 2006 and 2010 respectively. In 2012 and in 2013 he was visiting researcher at European Space Agency. Since 2010, he has been temporary researcher and lecturer at Dipartimento di Elettronica, Informazione e Bioingegneria of Politecnico di Milano. His research interests include methodologies for embedded system design and in particular High Level Synthesis, performance estimation and automatic generation of code for multiprocessor heterogeneous architectures.

Fabrizio Ferrandi received his Laurea (cum laude) in Electronic Engineering in 1992 and the Ph.D. degree in Information and Automation Engineering (Computer Engineering) from the Politecnico di Milano, Italy, in 1997. He has been an Assistant Professor at the Politecnico di Milano, until 2002. Currently, he is an Associate Professor at the Dipartimento di Elettronica, Informazione e Bioingegneria of the Politecnico di Milano. His research interests include synthesis, verification simulation and testing of digital circuits and systems. Fabrizio Ferrandi is a Member of IEEE, of the IEEE Computer Society and of the Test Technology Technical Committee.

Maxime Perrotin received his Diploma in Engineering from the Conservatoire National des Arts et Mtiers in 2001 (Diplme d'Ingnieur). After working in the space industry he joined the European Space Agency as technical officer. He took the lead of the R&D activities related to formal methods, modelling and code generation, and is providing engineering support to satellite development, with a special focus on in-orbit demonstration missions, such as formation flying systems.