A Design Methodology to Implement Memory Accesses in High-Level Synthesis

Christian Pilato
Politecnico di Milano
Dipartimento di Elettronica ed Informazione
Milano, 20133, Italy
pilato@elet.polimi.it

Fabrizio Ferrandi
Politecnico di Milano
Dipartimento di Elettronica ed Informazione
Milano, 20133, Italy
ferrandi@elet.polimi.it

Donatella Sciuto
Politecnico di Milano
Dipartimento di Elettronica ed Informazione
Milano, 20133, Italy
sciuto@elet.polimi.it

ABSTRACT

Nowadays, the memory synthesis is becoming the main bottleneck for the generation of efficient hardware accelerators. This paper presents a design methodology to efficiently and automatically implement memory accesses in High-Level Synthesis. In particular, the approach starts from a behavioral specification (in pure C language) and a set of design constraints, such as the memory addresses where some of the data are stored.

The methodology classifies which variables can be internally or externally allocated to the different modules to generate the proper architecture, fully supporting a wide range of C constructs, such as pointer arithmetic, function calls and array accesses. Moreover it allows to parallelize the accesses when the memory address is known at compile time, resulting in an efficient execution of the specification.

Categories and Subject Descriptors

B.5 [**RTL Implementation**]: Design Aids

General Terms

Design, Algorithms, Experimentation

Keywords

Memory Optimization, High-Level Synthesis

1. INTRODUCTION

The advancement of RTL synthesis tools and the standardization of hardware description languages, like VHDL or Verilog, shifted the focus of the research towards an efficient implementation of algorithmic descriptions into hardware, both in terms of area and performance.

There is a growing consensus among VLSI designers that one of the most effective methods to handle the complexity of today's system-on-chip (SoC) designs is to use techniques, such as *High-Level Synthesis* (HLS), that start with an abstract behavioral or algorithmic description of a circuit and automatically synthesize a structural description of a digital circuit that realizes the behavior. In fact, the designer of the application is usually able to program in a High-Level Language (HLL), such as the C, but he/she has often a limited experience in hardware design. For these reasons, we strongly believe that the HLS tools should support specification provided in pure C language, where, in case, the designer can specify additional information with custom pragmas. In such a way, the migration from software to hardware requires a limited intervention performed by hand. However, this introduces additional issues, in particular when the application contains different constructs which imply memory operations (e.g., pointer arithmetic or function calls with parameters passed by reference), where part of the data have to be allocated out of the accelerator.

The synthesis of memories is one of the most important aspects in the design of efficient embedded systems [14, 15]. For example, at design time, the designer can analyze or profile the memory accesses to obtain different information about the data, and can also perform different optimizations to resolve pointers and avoid unnecessary memory accesses [16]. On the other hand, modern FPGA devices allow the implementation of large on-chip memories, exploiting block RAMs (BRAMs) or distributed RAMs. As a result, this feature can be exploited to improve the memory architecture by creating internal memories for certain variables and by parallelizing the accesses when the operations do not overlap their addressing spaces. However, this necessarily requires an efficient design methodology to synthesize the accelerator, ranging from the analysis at compile time to the generation of the architecture, in order to support a wide range of C constructs and evaluate the effects of the different transformations or design decisions, in terms of area and performance.

This paper proposes a semi-automatic framework to assist the designer during HLS, focusing on memory aspects. It adopts a compile-time analysis to determine the data (e.g., scalar variables, arrays, ...) to be allocated in memories. Then, it proposes a simple allocation policy, combined with the decisions performed by the designer (specified by user `#pragma`) about the physical allocation of the data, to determine where the data are stored. For example, it is possible to specify constraints on the space available for internal allocation, as well as the physical addresses of the variables that the designer decides to allocate in the external memories. Finally, the methodology produces the proper architecture to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES+ISSS'11, October 9–14, 2011, Taipei, Taiwan.
Copyright 2011 ACM 978-1-4503-0715-4/11/10 ...$10.00.

implement the resulting decisions: internal variables are allocated on heterogeneous and distributed memories, whose addresses are determined at compile time. On the other hand, for the variables allocated on external memories, the methodology is able to follow the decisions suggested by the designer.

The main contributions of this work can be thus summarized as follows:

- it proposes a comprehensive compile-time analysis to automatically determine the variables which need a memory allocation and the corresponding scope;
- it describes a preliminary technique to automatically determine the localization on internal memories or to take into account previous designer's decisions;
- it demonstrates how the resulting allocations can be efficiently implemented with a proper architecture which allows both static and dynamic resolution of the accesses, supporting a wide range of C constructs.

It is worth noting that, when addresses can be statically resolved and data are allocated into internal memories, the methodology adopts direct connections to such elements, allowing parallel reads and writes when addresses do not overlap. On the other hand, when this is not possible, it is able to dynamically resolve addresses and correctly perform the computation. Finally, the methodology has been validated on different real-life test cases, demonstrating how it is possible to effective implement efficient solutions for memory allocation, with a very limited interaction with the designer.

The rest of this paper continues as follows. Section 2 overviews existing approaches for the synthesis of memory accesses. In Section 3, we introduce and motivate the proposed memory architecture which is the basis for our work. Section 4 details the proposed methodology for the allocation of variables in either internal or external memories, as well as the synthesis of the global architecture for the entire accelerator. The methodology is then evaluated in Section 5, while Section 6 concludes the paper, also outlining future directions of work.

2. RELATED WORK

In the last years, several approaches have been proposed for the synthesis of memory accesses for embedded systems. However, a comprehensive review of related work is beyond the scope of this paper. We restrict our attention to interactions between compile-time analyses and synthesis of memory architectures in HLS.

The first complete modeling of memory accesses in HLS has been introduced in [13], where the authors present different techniques to introduce such concepts in the common HLS phases, such as the scheduling. Our work is complementary to this one, since it is more oriented to determine which variables require the access to a memory location, to perform the allocation and then to generate the proper architecture in order to support the accesses. Any existing technique for the scheduling can be then applied to derive an efficient scheduling of the accesses.

Semeria and De Micheli [16] analyze the implementation of pointers and propose different optimizations to statically resolve the addresses and eliminate some of the memory accesses, based on the SUIF compiler. On the other hand, this work does not support the sharing of function calls since it does not support the resolution of pointers inside a function. Similarly, the techniques proposed in [12] can be adopted to efficiently resolve the addresses, also performing dynamic computations, but it is not described how to implement an efficient architecture able to support function calls. Instead, the proposed approach is based on GCC, taking advantages of the different analyses and optimizations that are performed by this compiler. GCC represents the accesses as the classical *base+offset* representation, where both of them can be constant or variables and thus they can be statically (e.g., at compile-time) or dynamically resolved. This allows to support resolution of addresses inside the functions and thus the sharing of function calls, even if all optimizations proposed in [16] can be integrated as well.

In [10], the authors proposed a distributed memory architecture. They partition the design and the corresponding data to reduce the overall connections into the chip and improve the performance. Conceptually, we target a similar architecture, where the variables allocated inside the core are assigned to local internal memories, distributed with respect to a classical monolithic memory. On the other hand, we mainly focus on supporting both local and external allocation, addressing the issues related to static/dynamic resolution of addresses.

In [11], the authors propose an architecture to improve the memory accesses, exploiting local caches and speculative reads. On the other hand, the authors do not describe how to derive the allocation of operations to units and how to manage allocation on internal memories as well as memory accesses for nested functions.

Recently, [5] proposes LegUp, a HLS tool targeting Altera devices, where all memory elements are explicitly identified by memory tags. Then, a unique memory controller unit is introduced in the top architecture to decode the different requests, in terms of tags, and actually perform the memory operations. However, they do not support accesses to external memories and having a unique memory controller forces to perform only one memory operation for each clock cycles, even when the code accesses independent memory locations (e.g., it reads from different arrays). The proposed memory architecture is able to overcome most of these limitations. Indeed, the variables allocated on internal memories are distributed across the different functions and operations which addresses are known at compile-time. The data-path can thus directly access the corresponding units and it also support external accesses by integrating the decision performed by the designer, if any.

3. PROPOSED ARCHITECTURE

This section describes the general architecture of the accelerators produced by the proposed methodology, motivating the different design choices.

For sake of clarity, let us assume that the designer wants to implement the piece of code shown in Figure 1 as a hardware accelerator. Note that the array `arr` can be allocated internally to the function `foo` or in an external memory. This decision can be performed by the designer by means of `#pragma`'s or automatically determined by an allocation methodology, such as the proposed one that simply allocates the variables in the minimal scope for each of them in order to minimize the storage resources.

The proposed architecture of each function module is based

```
int arr[2] = {1,2};

void bar(int* a, int b, int *c)
{
  int d;
  *c = 0;
  for (d = 0; d < b; d++)
    if (*(a+d) > *c) *c = a[d];
}

void foo(int a, int* e)
{
  int max = 0;
  bar(arr, 2, &max);
  *e = a + max;
}
```

Figure 1: A simple example of C code to be implemented in hardware.

on the classical controller/data-path paradigm (i.e., FSMD [17]). Existing HLS tools, such as SPARK [7], implement pointer parameters as IN/OUT ports. However, considering the `bar` function, the parameter `a` is the base of an array and it cannot be implemented in such a way. For this reason, following the C semantic, these ports are implemented as IN ports, where the memory addresses of the variables have to be specified. Moreover, the same function `bar` accesses to an array that is allocated and initialized outside the function itself. The function `bar` thus needs a memory interface which has to be integrated with the architecture at the higher level of the hierarchy. This element has to be included into the data-path in order to be connected with the functional units or the storage elements which provide the values (e.g., address to be accessed or data to be written in memory) or with the elements where the read data will be eventually used.

Note that, in function `bar`, there are different memory accesses that can be avoided. For example, `*(a+d)` (or the equivalent `a[d]`) is repeated twice in the same iteration, one to perform the test and one, if needed, to update the value pointed by `c`, as shown in Figure 2. Note that, in this figure, each access to array has been translated into the form `*(a+d)`, which represents the situation *"access to the data contained into the location pointed by the address (base+offset)"*. In this case, to optimize such memory accesses, the designer could rewrite the code as shown in Figure 3. Considering two iterations over the array, the version of the `bar` code in Figure 2 requires 9 memory accesses to complete the execution, while the one shown in Figure 3 requires only 3 accesses. Such transformations could be automatically performed, but this is out of the scope of this work, where we assume to implement the specification as it is. Such transformations have thus to be performed *before* providing the code to our methodology and the proposed methodology can be also adopted to evaluate their effects. Moreover, since the designer can also decide to allocate the variables in different memories, the methodology also allows to evaluate the cost of taking such decision, both in terms of area (e.g., the internal memory allocation of the array elements requires a dedicated memory of a certain size, as described in the Section 3.2) and performance.

Considering the function `foo`, it has the same structure, with a memory interface for the variable `e` since it is a pointer parameter that potentially accesses the memory. Moreover, it contains two variables which require a memory allocation,

```
int arr[2] = {1,2};

void bar(int* a, int b, int *c)
{
  int d;
  *c = 0;
  for (d = 0; d < b; d++)
  {
    int tmp_1 = *(a+d);
    int tmp_2 = *c;
    if (tmp_1 > tmp_2)
    {
      int tmp_3 = *(a+d);
      *c = tmp_3;
    }
  }
}

void foo(int a, int* e)
{
  int max, max_2, max_1 = 0;
  max = max_1;
  bar(arr, 2, &max);
  max_2 = max;
  *e = a + max_2;
}
```

Figure 2: Restructuring of the code performed by the complier to explicit the memory accesses.

that are `arr` and `max`. In fact, the former is an array, which base address is given to the function `bar` to perform the scan of all the elements, while the latter is a location where the function `bar` will write the result. Thus the memory interface of the called function has to be necessarily connected to these memory elements in order to correctly access the data.

Furthermore, function calls and accesses to external memories are usually operations with unknown latency, especially when complex memory hierarchies are adopted outside the hardware accelerator, ranging from local on-chip buffers to different levels of caches or shared RAMs. For this reason, we need to take this information into account during the synthesis of the controller in order to ensure a correct execution. We thus adopt the classical request/acknowledge paradigm [6], represented by the *start* and *done* signals, respectively. The controller can be then implemented with a classical FSM, which is modified to wait for operation termination, as shown in Figure 4. Moreover, since each function call can have in turn different memory accesses and the current architecture allows only one memory operation per cycle (only one address bus), we constrain the scheduling and the controller to have no more than one operation with unknown latency per clock cycle.

```
void bar(int* a, int b, int *c)
{
  int d, max = 0;
  for (d = 0; d < b; d++)
  {
     int v = *(a+d);
     if (v > max) max = v;
  }
  *c = max;
}
```

Figure 3: Code transformation on the function `bar` to reduce memory accesses.

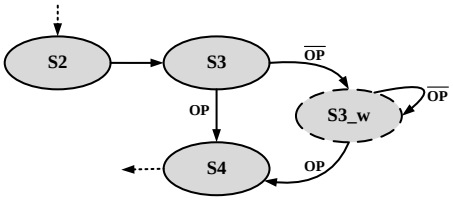

Figure 4: Example of FSM with an unbounded operation in S3. The input OP represents that the *done* signal has been received, while $\bar{OP}$ means that the controller is still waiting for the same signal. S3_w is the waiting state associated with S3.

3.1 Memory Interface

This interface is generally adopted when the memory operation needs to access a variable that is not allocated into the same function module. In particular, it can be allocated either in one of the outermost functions or in an external memory. Such interface is shown in Figure 5 and it is connected to the data-path with the following ports:

- **LOAD/STORE** are the requests directly performed by the data-path, i.e., an operation requests a memory access;
- **done** notifies the termination of the execution to the controller (it represents the *done* signal);
- **data_r/data_w** represent the values from/to the memory, respectively;
- **addr/offset** represent the address to be effectively accessed, in terms of base address and the corresponding offset, respectively;
- **size** represents the number of bytes to be effectively read/written.

Note that the **size** signal has been introduced to perform operations on data with different sizes (e.g., 8-bit, 16-bit, 32-bit and 64-bit variables).

Then, since only one memory operation can be performed at each time, the memory interface of the current function can form a chain with all interfaces of the functions which are directly called by the function itself in order to propagate the request until the correct addressed memory is found. In particular, the request can be propagated from inside to outside a function, up to the external memory, in case. The module is thus augmented with the following ports:

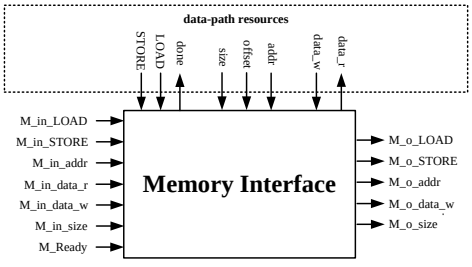

Figure 5: Structure of the interface to the external memory.

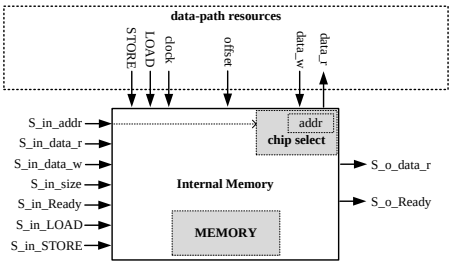

Figure 6: Structure of the interface with an internal memory.

- **M_o_LOAD/M_o_STORE** are the requests generated by this module or the previous ones in the chain, represented by the corresponding **M_in_LOAD** and **M_in_STORE** signals;
- **M_in_addr/M_o_addr** represent the address to be effectively accessed;
- **M_in_data_w/M_o_data_w** represent the value to be written in memory that is generated by this module or the previous ones in the chain;
- **M_in_data_r** represents the value eventually read from the slave memories and returned to this module if it is waiting for the request;
- **M_in_size/M_o_size** represent the number of bytes to be effectively read/written.
- **M_Ready** is the signal that notifies to this interface that the LOAD request has been effectively completed, if any, in order to consider valid the current value on **M_in_data_r**.

Note that, only one of **LOAD**, **STORE**, **M_in_LOAD** and **M_in_STORE** signals can be active at the same time. If one of **LOAD/STORE** signals is active, the operation inside the data-path generates the request and the controller waits until the **M_Ready** notifies the end of execution. In particular, for **LOAD**, the **addr/offset** is converted into the proper **M_o_addr** to be accessed, as well as the **size** is propagated to **M_o_size**. Then, the module will receive the correct data on the **M_in_data_r**, after the reading has been performed (i.e., the **M_Ready** raises), returning to the data-path through the port **data_r**. On the other hand, for **STORE**, the value **data_w** is put also on the port **M_o_data_w** to be effectively written to the proper location. When the operation is not generated by the current function, both **LOAD** and **STORE** signals are not active, the **M_in_data_w**, **M_in_addr** and **M_in_size** signals are just propagated to the corresponding output ports, as well as the command signals.

3.2 Internal Memory

A dedicated memory is introduced for each variable that is directly allocated into the function, as shown in Figure 6. Since this memory is defined at compile time, an absolute and unique address (**addr**) can be assigned within the core. The module has to support both direct access and dynamic resolution of the address. The former case occurs when the base address can be statically resolved (e.g., `a[2]` or `a[i]`,

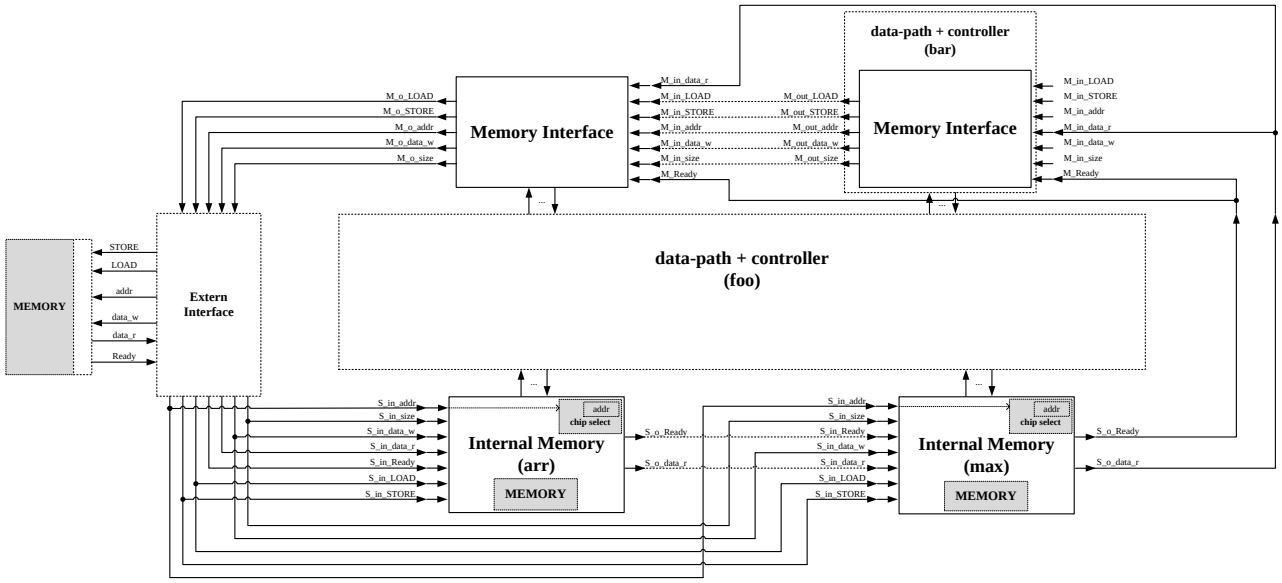

Figure 7: Resulting architecture for the example shown in Figure 1.

where `a` is the variable allocated to the memory), while the latter one when the actual address has to be resolved dynamically (e.g., `*b++`, where `b` could be initialized to any of the memory addresses). In this case, the module has to verify that the request is effectively addressing its memory.

For this reason, to support direct accesses, the module has the following ports:

- **LOAD/STORE** are the requests directly performed by the data-path, i.e., an operation requests an access to the variable allocated on this memory;
- **offset** represents the offset which the request is performed with respect to the base address **addr** (in case of scalar variables, it will be forced to 0);
- **data_r/data_w** represent the values from/to the memory, respectively.

On the other hand, to support dynamic resolution, the following ports have been provided:

- **S_in_LOAD/S_in_STORE** are the potential requests to the module;
- **S_in_addr** represents the address to be effectively accessed;
- **S_in_data_w** potentially contains the value to be written in the memory;
- **S_in_size** represent the number of bytes to be effectively read/written.
- **S_in_Ready** notifies the current value on **S_in_data** represents the data generated by another module and thus it has to be propagated;
- **S_o_data_r** potentially contains the value read from the memory, otherwise it propagates the incoming signal **S_in_data_r**;
- **S_o_Ready** notifies that the request has been effectively completed and thus the value on **S_o_data** can be considered as valid;

Also in this case, only one memory operation can be performed at each time. For this reason, only one of **LOAD**, **STORE**, **S_in_LOAD** and **S_in_STORE** signals can be active at the same time. If one of **LOAD/STORE** signals is active, it means that an operation inside the data-path generated the request. In particular, for **LOAD**, **offset** is added to the base address and the corresponding location into the memory is accessed, directly returning the value on **data_r**, without any access to the bus. On the other hand, for **STORE**, the value **data_w** is also put on the port **M_o_data** to be effectively written to the proper location.

When the request is not generated by the current function, none of the signals **LOAD** or **STORE** signals are active, one of the **S_in_LOAD** or **S_in_STORE** signals is active and the module has a chip select (CS) to determine if the incoming address **S_in_addr** is effectively addressing this memory. In fact, given a request at the input (either **S_in_LOAD** or **S_in_STORE**), the CS determines if **S_in_addr** refers to a variable within the space of the memory by verifying the following condition:

$$addr \leq S\_in\_addr < addr + memory\_size \qquad (1)$$

where *memory_size* represents the size of the memory, expressed in bytes. If this condition is satisfied, it means that the bus is effectively addressing this memory and the request can be satisfied. In particular, the data value **S_in_data_w** is written at the proper location (in case of **S_in_STORE**) or the stored value is returned on **S_o_data_r** (enabling also the **S_o_Ready** signal) when a load (**S_in_LOAD**) is required. On the contrary, when the CS is not activated, it means that the request refers to a location outside this memory and thus it simply propagates the incoming value **S_in_data_r** (potentially written by the actual addressed memory) to **S_o_data_r**, as well as **S_in_Ready** to **S_o_Ready**.

3.3 Global Interconnections

After all modules have been generated, we need to effectively connect all of them in order to correctly perform the memory operations. Considering the example in Figure 1, the innermost function `bar` has no internal memories. For this reason, its interface contains only master ports to perform operations on external memories. Since the global array `arr` can be allocated either in the memory external to the core or in one of the internal memories to the `foo` function, the resulting architecture should support any of these situations. In the following, let us assume to implement `arr` in an internal memory (e.g., by declaring it as `static int arr[2] = {1,2}`).

The proposed global architecture is shown in Figure 7, adopting separated read/write chains. Such chains are effectively required on modern FPGA devices (e.g., Xilinx Virtex-5 FPGA), which do not support three-state elements and, thus, require to physically separate reading and writing chains to avoid such kind of problems, as well as avoiding to generate combinational loop paths.

Considering the synthesis of the top function `foo`, we create a master chain collecting all memory requests (one at each time), starting from the function `bar` contained into the specification and going through all the remaining functions, if any, as well as the memory interface of the function itself. The master chain terminates on the interface of the external memory. In fact, since the internal allocation is statically performed, starting from the address $0x0$, this element is able to identify if the address refers to a variable allocated internally to the module (i.e., **M_o_addr** $<$ *allocated*, where *allocated* represents the amount of allocated space). In this case, the address and the requests are forwarded in broadcast to all internal memories, otherwise the access refers to the external memory and the request is accordingly routed. Then, only one of the memories (either external or internal) will effectively provide the data. For this reason, only one memory will provide the correct data to the corresponding signal **S_o_data_r** and activate the **S_o_Ready** one, while the memories that are not activated (i.e., which CS is not enabled) will simply propagate the data from their input **S_in_data_r** to the corresponding **S_o_data_r** port. At the end, the slave chain is reconnected to the master one, in order to effectively provide the data to the module which performed the request and which is waiting for the corresponding **M_Ready** signal.

Note that the architecture behaves in the same way for STORE operations, where the data to be written is forwarded in broadcast to all the slave modules in order to find the proper memory. This memory will be the only one which effectively performs the writing, while the other ones (which CS is not enabled) simply discard the data.

4. PROPOSED DESIGN METHODOLOGY

An overview of the proposed approach is shown in Figure 8. It receives as input the C code to be implemented in hardware, annotated with pragmas to specify additional information for the synthesis (phase **1** in Figure 8). As output, it produces the structural RTL description, in Verilog, of all modules, implementing the decisions which have been performed during the synthesis (phase **5**). Internal memories are also included in such description, along with the proper initialization values. The proposed methodology interfaces

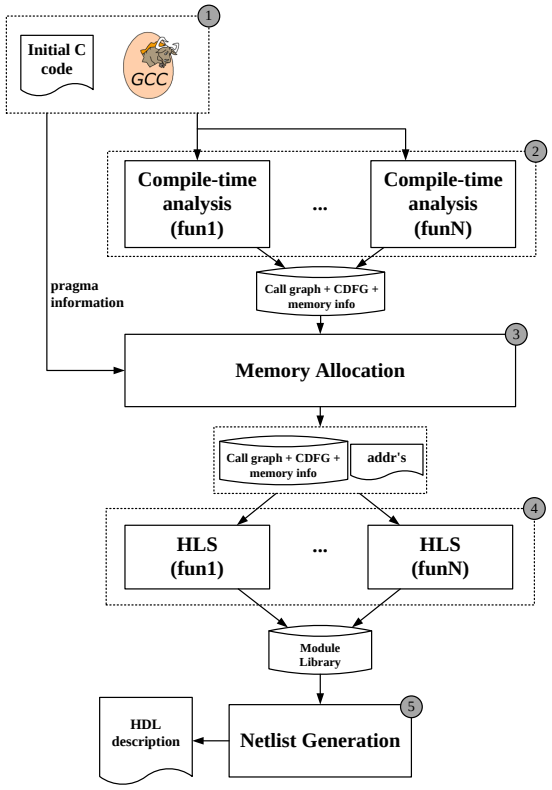

Figure 8: Overview of the proposed methodology.

the GNU C Compiler (GCC) [1] to derive the call graph, as well as the intermediate representation of the entire specification. Then, the phases to complete the synthesis are the following:

1. the analysis of variables and operations (phase **2**),
2. the memory allocation (phase **3**) and
3. the actual HLS of the modules (phase **4**).

Note that the phase **2** and the phase **4** are performed for each function separately, while the phase **3** is globally performed since it has to take into account the interactions among the functions to determine the scope of the variables and thus the proper location. In the following, we will effectively detail each of these steps.

4.1 Compile-Time Analysis

This phase works on the intermediate representation obtained from the GCC compiler and it is in turn divided in two sub-phases:

1. the identification of variables which are involved in memory operations;
2. the identification of LOAD/STORE operations, as well as operations referring to memory.

In the first step, we analyze each operation of all the functions to retrieve information about the operator and the corresponding operands. The GCC adopts the GIMPLE intermediate representation [9], where each of the elements (e.g., operations, variables, etc.) is represented by a unique

```
 @1: int arr[2] = {1,2};

void bar(int* a, int b, int *c)  //@2 (@3, @4, @5)
{
 @6: int d;
 @7: *c = 0;
 @8: for (d = 0; d < b; d++)
 @9: {
@10:   int tmp_1 = *(a+d);
@11:   int tmp_2 = *c;
@12:   if (tmp_1 > tmp_2)
@13:   {
@14:     int tmp_3 = *(a+d);
@15:     *c = tmp_3;
@16:   }
@17: }
}

void foo(int a, int* e) //@18 (@19, @20)
{
@21: int max, max_2, max_1 = 0;
@22: max = max_1;
@23: bar(arr, 2, &max);
@24: max_2 = max;
@25: *e = a + max_2;
}
```

Figure 9: Identifiers associated with each operation of the example in Figure 2.

node, with additional information, such as, for example, `extern/static` attributes associated with the gloval variables. In the following, we will assume the notation shown in Figure 9, where, for example, the identifier `@2` represents the definition of the function `bar`, while the identifiers `@3`, `@4` and `@5` represent the related parameters.

There are several elements which refer to memory by definition, such as arrays or pointer variables. On the other hand, considering, for example, the scalar variable `max` in function `foo`, it is required to have a memory address (expressed by the operator `&`) to be given to the function `bar`. In fact, the called function will write into this location through a pointer operation. Thus, considering the example in Figure 9, Table 1 recaps all the memory variables accessed by each operation.

Note that, in this table, the identifiers `@1` and `@22` represent variables to be stored in memory, that are `arr` and `max`, respectively. On the other hand, `@2` and `@5` represent accesses to memory locations through pointers. Then, since each operation can be associated with the corresponding function, we can project these sets of variables onto the call graph, to represent all the variables accessed by each function. The resulting annotated call graph is shown in Figure 10. In this graph, each node represents a function, while the

Table 1: Memory variables for each operation.

Function	Operation	Variables
bar	@7	@5
	@10	@2
	@11	@5
	@14	@2
	@15	@5
foo	@22	@21
	@23	@1, @21
	@24	@21
	@25	@19

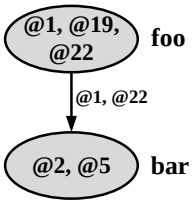

Figure 10: Call graph of the example shown in Figure 2, annotated with memory variables.

arcs represent function calls, with the exchanged data. Note that this graph can be also adopted to perform additional memory analysis, such as pointer optimization, but this is out of the scope of this work.

After performing such analysis, each operation can be then classified as LOAD/STORE if 1) it is an assignment between two values and 2) the operation refers to a memory variable in the left/right part of the assignment. In this way, besides classical array operations (e.g., `@10` and `@14`) and pointer ones (e.g., `@7`, `@11`, ...), also the operations `@22` and `@24` can be considered as LOAD/STORE since the value contained into `max_1` has to be stored in a memory location before starting the function `bar` and, then, the value resulting after the function execution has to be retrieved to continue with the computation. These are the operations that will generate the requests to the memory interfaces or the internal memories. Finally, note that the operation `@23` is a function call. In this case, there is not any direct memory operation, but the corresponding variable addresses are just given to the module implementing the function `bar` to access the proper memory locations.

4.2 Memory Allocation

In this phase, we propose a very simple algorithm to determine where the variables have to be allocated. In particular, we analyze the call graph resulting from the previous step in order to determine the minimal scope of all the variables and, then, the actual memory allocation for each of them.

We thus adopt a simple procedure, that starts from innermost functions, up to the outermost one. For example, considering the simple call graph in Figure 11, the functions are analyzed in the following order: `f3`, `f2` and `f1`. Then, for each function, we analyze the corresponding memory variables, tagging as `true`, i.e., to be declared, a variable when it is referred into the function, and as `false` otherwise. In the example, `@40` is marked as `true` for both `f2` and `f3`. Then, we propagate this information to outermost ones and, when there are at least two called functions referring to the same

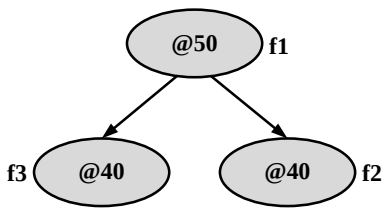

Figure 11: Simple call graph to show the scope identification.

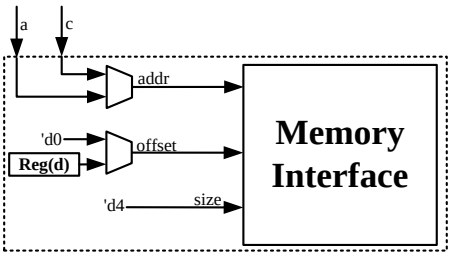

Figure 12: Resulting connections to the memory interface for the function `bar`. $Reg(d)$ represents the out port of the register where the variable d is stored.

variable (e.g., `@40` by `f2` and `f3`), this is marked as `true` also in the current function (e.g., `f1`) and as `false` otherwise.

At the end of this analysis, the functions contain the variables effectively in their scope. For the example in Figure 11, `f1` contains `@40` and `@50`, while `f2` contains only `@40`, as well as `f3`. In such a way, if the variable has to be internally allocated, performing a top-down analysis, the first *declaring* tag (i.e., where it is marked as `true`) for variable `@40` is in `f1`. In this way, the variable will be allocated in this function and it is effectively visible, through the slave chains, to the other two functions.

After performing this analysis, the variables assigned by the designer to external memories are directly assigned to these addresses. For example, if the variable `arr` in Figure 1 has been declared as:

```
#pragma MEM_address arr 0x0f000
int arr[2] = {1,2};
```

it would be assigned to the external memory, with address `0x0f000`. On the contrary, for all remaining variables, we perform a progressive local allocation (similarly to the tagging performed by LegUp [5]), starting from the base address `0x0` in order to statically determine the addresses of all the variables. Considering the same example, with a local allocation of both the variables `arr` and `max`, we can assign the addresses `0x0` and `0x8`, respectively. In fact, considering 32-bit integers, the former variable requires 8 bytes, while the latter one requires 4 bytes.

4.3 Memory-Aware High-Level Synthesis

In this last phase, we finally perform the actual high-level synthesis of the different functions, taking into account the memory decisions performed in the previous steps. Note that any of the existing algorithms can be adopted to perform the different sub-steps, such as scheduling, resource binding and interconnection optimization.

Considering memory aspects, given a LOAD/STORE operation, if the corresponding variable has been allocated to the same function, the request is directly performed on the corresponding local memory, which is allocated in the same data-path. On the other hand, if the variable has been allocated in an outermost function, the request is then performed though the memory interface to access the read/write chains and get the value back. Thus, the scheduling can take into account such information (i.e., the unit where the operation is executed) and then produce an efficient solution, as well as the finite state machine to correctly manage the execution. Moreover, when performing such operations, we need to specify the base address (apart from internal memories, which is directly encoded into the memory) and the corresponding offset. Both of them can be statically precomputed (e.g., they are known at compile time) or stored into a register or given as input (e.g., pointer parameter). In the first case, the constant value is directly connected with the *addr* or the *offsets* ports. In the remaining cases, the output of the corresponding register or input port is connected with the corresponding port of the memory unit. An example of the resulting interfacing is shown in Figure 12, where the input addresses are connected with the input port of the unit (for the dynamic resolution of the addresses), as well as the offset. Note that this value is null for the parameter `*c` since we are accessing a scalar variable. All variables are 32-bit integers and thus each LOAD/STORE operates on 4 bytes. For this reason, the port **size** is set to the constant value 4. Note that, in this way, we are able to perform the sharing of function calls, where different executions of the same module will simply have different input addresses to work on, similarly to the corresponding software counterpart.

Finally, at the end of the synthesis of each function, the master/slave chains are composed taking into account the modules of the called functions, as well as the internal memories of the function itself. The resulting architecture is thus similar to the one obtained for the example shown Figure 1 and shown in Figure 7.

Table 2: Source-level characteristics of the benchmarks.

	Repr. data type	Lines code	Func.	Variables Scalar	Variables Array
MIPS	32-bit scalar	232	1	32	5
GSM	16-bit array	393	12	150	10
ADPCM	32-bit array	541	15	269	26

5. EXPERIMENTAL RESULTS

The proposed methodology has been implemented in a C++ prototype, interfacing the GNU C Compiler (GCC) ver. 4.5 [1], and it has been applied to some real-life benchmarks from CHStone [8], a suite of benchmark programs for C-based high-level synthesis tools, which details are shown in Table 2. Of our benchmarks, MIPS describes instruction-level behaviors of a simplified MIPS processor which has 30 types of instructions, along with a simple program as test vector. GSM is a program for Linear Prediction Coding (LPC) analysis of Global System for Mobile Communications (GSM), which is a communication protocol for mobile phones. ADPCM implements the CCITT G.722 ADPCM algorithm for voice compression. It includes both encoding and decoding functions.

We performed the HLS with state-of-the-art algorithms for the different steps. In particular, we adopted a classic list-based scheduling, based on dynamic mobility values, followed by a resource binding targeting multiplexer minimization. We targeted a Xilinx Virtex-5 XC5VLX50 FPGA device [4], where three-states are not supported and, thus, the separation of read/write chains is effectively required. The target frequency has been set to 100MHz for all the experiments. Internal memories have been implemented with BRAMs, which require 2 cycles (in pipelining) for reading and 1 cycle for writing the data. The resulting modules have been then interfaced with proper test-benches to simu-

Table 3: Performance results obtained varying the latency of the external memory.

Experiment		2 cycles		5 cycles		10 cycles	
Benchmark	Case	Latency	Diff. (%)	Latency	Diff. (%)	Latency	Diff. (%)
MIPS	M.1	12,950	-	17,159	-	24,174	-
	M.2	12,950	0.00%	15,326	-10.68%	19,286	-20.22%
GSM	G.1	20,828	-	26,090	-	34,860	-
	G.2	20,681	-0.71%	21,647	-17.03%	23,257	-33.28%
	G.3	20,529	-1.44%	20,559	-21.20%	20,609	-40.88%
ADPCM	A.1	95,570	-	157,217	-	259,962	-
	A.2	95,458	-0.11%	156,355	-0.55%	257,850	-0.81%
	A.3	107,819	+12.82%	136,508	-13.17%	184,323	-29.10%

late different delays for the external memories, ranging from 2 to 10 cycles in order to potentially model different external memory architecture such as external buffers, caches, etc... The resulting systems have been then simulated with Mentor ModelSim SE 6.6d [2] to obtain the number of clock cycles required to complete the execution. Then, the accelerators have been also synthesized with Xilinx ISE ver. 12.3 [3] to obtain an estimation of resource requirements.

For the MIPS, we would like to verify the impact of moving the test program from the external memory to the internal one. Conceptually, in the former case, namely $M.1$, the resulting MIPS implementation is able to execute any application, provided that it is loaded in the corresponding external memory, starting from the same location. In the latter one, namely $M.2$, the MIPS is able to execute only the specified program, stored into the internal memory, but it does not require to access the external one, with benefits from the performance point of view. We performed three experiments for the GSM. In the first one ($G.1$), we allocated all the data arrays on external memories. In the second one ($G.2$), we allocated into internal memories only the arrays containing the final results, while the input data are stored into the external memory. In the last experiment ($G.3$), we evaluated the opposite situation, where the input data have been stored into internal memories, while output data have been allocated into the external memory. Considering the ADPCM, we performed three different experiments also in this case. In the first one ($A.1$), all memory variables are allocated into the external memory. On the contrary, in the second one ($A.2$), all the data referring to input and output are allocated into internal memories, while, in the last experiment ($A.3$), we left these data in the external memory, but we allocated internally the value referring to the constant tables used for the computation. It is worth noting that all these experiments have been performed just by putting pragmas on the variables, in order to move them from/to the external memory.

The results of the different experiments are reported in Table 3 and Table 4, in terms of clock cycles and requirements of resources, respectively. Note that, in these tables, we reported the percentage difference of each value with respect to the corresponding $x.1$, considered as reference. Table 3 shows that, as expected, limiting the number of variables allocated in external memories improves the performance, especially when the cost of accessing them becomes significant. For example, in the last experiment of the GSM benchmark, we are able to reduce the number of cycles by more than 40%. On the other hand, when the cost is comparable with the internal memories, an efficient scheduling of the operations becomes crucial and, in some cases (e.g., $A.3$), moving data to internal memories leads to decrease the performance. On the other hand, increasing the latency of the external memory, significant performance improvements can be still obtained. It is also interesting to note the results of $G.3$. In this case, moving the input data to local memories allows to dramatically reduce the number of memory accesses. As a result, the execution remains almost constant also varying the latency of the external memory, while, in the corresponding reference solution, that is $G.1$, the execution time almost doubles.

Table 4: Estimation of resource requirements to implement the different solutions.

Experiment		LUT/FF pairs		Allocated bytes	
Benchmark	Case	(#)	Diff. (%)	Int.	Ext
MIPS	M.1	5,859	-	0	624
	M.2	5,722	-2.24%	176	448
GSM	G.1	18,158	-	390	948
	G.2	18,655	+2.74%	442	896
	G.3	18,500	+1.88%	1,066	272
ADPCM	A.1	21,145	-	0	3,660
	A.2	21,471	+1.54%	1,200	2,460
	A.3	25,065	+18.54%	1,704	1,956

Table 4 shows the resources required to implement the resulting architectures on the target device, in terms of LUT Flip Flop pairs and bytes allocated in the memories. The results show that the area overhead due to internal memories is negligible. In fact, these memories are implemented as BRAMs and, thus, we only require the logic to correctly drive the signals, but it is limited with respect to the requirements for the data-path resources. Concerning the case $A.3$, we need to consider that it corresponds to the scheduling solution which leaded to obtain a performance degradation. For this reason, we can expect that this area overhead can correspond, in the same way, to a suboptimal allocation of data-path resources (e.g., functional units, registers and interconnection elements).

On the other hand, the requirements in terms of BRAMs (less then 2Kbytes in the worst case, that is $A.3$) can be effectively satisfied by modern FPGA devices. In fact, for example, the Xilinx Virtex-5 devices contain 4-Kbytes of BRAMs [4].

We can thus conclude that the proposed methodology can be effectively adopted to evaluate the effects of moving data from external to internal memories by adopting appropriate annotations on the source code. Moreover, the resulting architectures is effectively able to implement the resulting decisions and can be efficiently implemented on modern FPGA devices, where three-states are not currently supported.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed an architecture and a design methodology to efficiently implement memory accesses in HLS, starting from pure C code. The proposed approach is able to identify actual memory elements and to integrate designer's decisions, provided by source code annotations. The methodology has been validated on different real-life test cases, showing how it is possible to evaluate the effects of allocating the data either in internal or external memories, supporting a wide range of C constructs.

Currently, we are going to extend this methodology with more efficient solutions for the allocation of variables, able to automatically re-organize the memory allocation in order to further improve the resulting architectures and to analyze the effects of interfacing such cores with microprocessors in complex systems-on-chip. Moreover, we are also interested to propose effective scheduling solutions to take into account the allocation of variables and thus further improve the performance.

7. REFERENCES

[1] GCC - GNU Compiler Collection. http://gcc.gnu.org.

[2] Mentor ModelSim SE 6.6d User Guide. http://www.mentor.com.

[3] Xilinx ISE Design Suite 12.3 User Guide. http://www.xilinx.com.

[4] Xilinx Virtex-5 FPGA. http://www.xilinx.com/products/virtex5/.

[5] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM.

[6] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[7] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: a high-level synthesis framework for applying parallelizing compiler transformations. *Proceedings of the 16th International Conference on VLSI Design*, pages 461–466, 4-8 Jan. 2003.

[8] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[9] 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*, pages 406–420, 1993.

[10] C. Huang, S. Ravi, A. Raghunathan, and N. Jha. Generation of distributed logic-memory architectures through high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(11):1694 – 1711, Nov. 2005.

[11] H. Lange, T. Wink, and A. Koch. Marc ii: A parametrized speculative multi-ported memory subsystem for reconfigurable computers. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '11, 2011.

[12] M. Miranda, F. Catthoor, M. Janssen, and H. De Man. High-level address optimization and synthesis techniques for data-transfer-intensive applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(4):677 –686, Dec 1998.

[13] P. Panda, N. Dutt, and A. Nicolau. Exploiting off-chip memory access modes in high-level synthesis. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 333 –340, November 1997.

[14] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 6:149–206, April 2001.

[15] P. R. Panda, N. D. Dutt, and A. Nicolau. Architectural exploration and optimization of local memory in embedded systems. In *Proceedings of the 10th international symposium on System synthesis*, ISSS '97, pages 90–97, Washington, DC, USA, 1997. IEEE Computer Society.

[16] L. Semeria and G. De Micheli. Resolution, optimization, and encoding of pointer variables for the behavioral synthesis from C. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):213 –233, February 2001.

[17] J. Zhu and D. D. Gajski. A unified formal model of ISA and FSMD. In *Proceedings of the seventh international workshop on Hardware/software codesign*, CODES '99, pages 121–125, New York, NY, USA, 1999. ACM.