Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 685139, 16 pages
doi:10.1155/2008/685139

Research Article

A Platform for the Development and the Validation of HW IP Components Starting from Reference Software Specifications

Christophe Lucarz,[1] Marco Mattavelli,[1] and Julien Dubois[2]

[1] *Multimedia Architecture Research Group, Microelectronic Systems Laboratory, Ecole Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland*
[2] *Laboratoire Electronique, Informatique et Image (LE2I), Université de Bourgogne, 21000 Dijon, France*

Correspondence should be addressed to Christophe Lucarz, christophe.lucarz@epfl.ch

Received 6 March 2008; Revised 20 July 2008; Accepted 25 November 2008

Recommended by Guy Gogniat

Signal processing algorithms become more and more efficient as a result of the developments of new standards. It is particularly true in the field video compression. However, at each improvement in efficiency and functionality, the complexity of the algorithms is also increasing. Textual specifications, that in the past were the original form of specifications, have been substituted by reference software which became the starting point of any design flow leading to implementation. Therefore, designing an embedded application has become equivalent to port a generic software on a, possibly heterogeneous, embedded platform. Such operation is getting more and more difficult because of the increased algorithm complexity and the wide range of architectural solutions. This paper describes a new platform aiming at supporting a step-by-step mapping of reference software (i.e., generic and nonoptimized software) into software and hardware implementations. The platform provides a seamless interface between the software and hardware environments with profiling capabilities for the analysis of data transfers between hardware and software. Such profiling capabilities help the designer to achieve different implementations aiming at specific objectives such as the optimization of hardware processing resources, of the memory architectures, or the minimization of data transfers to reach low-power designs.

1. INTRODUCTION

Video and multimedia algorithms have changed a lot since their first achievements in the early nineties. If we take the example of ISO/IEC video coding standards (better known as MPEG), we can observe that each successive codec generation released by MPEG has been substantially more complex than the previous, typically yielding twice the compression efficiency of its predecessor. Due to the growing complexity, the textual specification of recent standards (since MPEG-4) [1] has lost its normative role, being replaced by the reference software implementation as the true normative specification. Any ambiguous interpretation is solved by referring to the software description. Therefore, the generic and nonoptimized sequential software description became the starting point for any implementation of a video standard. Unfortunately, working and reasoning on architectural solutions, such as SW/HW partitioning, on a few tenth of thousands of reference software lines of code is a very time- and resource-consuming task [2]. In the traditional way of designing hardware blocks, designers must find out a suitable architecture and isolate the candidate hardware components. Appropriate test vectors must be also generated to test each of these processing elements. Such sequence of tasks could result in more resource demanding than the pure hardware development itself. Realizing this fact, two initiatives have been taken within the MPEG standardization committee. The first was to develop a generic-optimized reference software version of the standard (MPEG-4 Part 7) [3]. The second was to derive mixed SW/HW descriptions from the reference software. The blocks described in a hardware description language (HDL) are now included in the standard (MPEG-4 Part 9 [4]).

Nowadays, reference software is the true starting point for the implementation of video codecs on embedded systems and usually the reference software is written in C/C++. The first step of such process is to identify from the reference software candidate IP components for implementation in

HW and separate them from what should remain SW and should be optimized for the target-embedded platform. The platform described in this paper supports the designer in the identification and development of the different IP components extracted from the reference software and in the validation and test under the real test vectors. The validation and test is not a simulation stage, but the real execution of the partitioned system on a generic SW/HW architecture. The result does not constitute the final algorithm implementation, but provides useful information for the exploration of the SW/HW design space.

The paper is organized as follows. Section 2 exposes the state-of-the-art of the existing platforms supporting rapid prototyping of video and multimedia designs for embedded implementations. Section 3 presents the concept of the virtual socket platform. Section 4 describes in detail the implementation of the virtual memory extension. Section 5 explains how useful profiling information can be extracted by executing the algorithm on the platform. Section 6 outlines a design methodology for the development of heterogeneous implementations according to different optimizations starting from a reference software by exploiting the features of the virtual socket platforms. Section 7 demonstrates the usage of the platform and associated tools in a real case study: the design and optimization of an MPEG-4 motion estimation module. Section 8 concludes the paper.

2. STATE-OF-THE-ART

Even if C/C++ language is not an appropriate language to specify and model signal processing algorithms in the early steps of the design flow of embedded systems, it is still widely used in several contexts. Consequently, the first problem embedded systems designers have to face is the conversion of the generic sequential C/C++ reference software into a form that begins to include architectural features of the selected embedded platform. For instance, this means to identify functions that can be processed by software or hardware coprocessor units if available and to express them in a form that exploits the available parallelism.

In the early stages of the design flow, designers should have a feeling about the architecture of the system, that is, decide which part of the algorithm must be in hardware or software. Unfortunately, the intuition of the designer becomes not reliable when dealing with complex systems. It may lead to wrong initial decision that affects all other stages of the design states. The possibility of quickly testing possible solutions is a clear advantage to try to find the best architecture. In platform-based design, the entire algorithm must be mapped on the development platform, mapping SW parts on embedded processors and HW parts on FPGA. Platforms are composed of processors, dedicated hardware, and reconfigurable hardware (FPGA). For example, the XUP Virtex-II Pro Development System [5] provides an advanced hardware platform that consists of a high-performance Virtex-II Pro Platform FPGA (with PowerPC 405 cores) surrounded by a comprehensive collection of peripheral components that can be used to create a complex system. Another example of such platform is the Celoxica RC1000-PP PCI board [6].

From such class of platforms, in which the entire algorithm is mapped on the platform itself, the sharing of data between a host PC and the platform is possible [7]. The main program still lies in the embedded processor and data on the host are easily available by means of virtual serial ports. However, the plugging of hardware modules inside the reference software running on the host remains the most difficult task.

The more advanced step is reached by the work of Martyn Edwards and Benjamin Fozard [8]. An FPGA-based algorithm (implemented on an external platform) is activated from the host PC, directly from the reference software. Such platform is based on the Celoxica RC1000-PP board [6] and communicates with the host by using the PCI bus. The main program is on the host processor, sends control information to the FPGA, and transfers data in a small shared memory which is part of the hardware platform. In such case, the designer must explicitly transfer the data necessary for the processing (on the platform) from the host to the local memory. Other works addressing coprocessors and relative coprocessors interfaces have been reported in literature. Some examples are given in [9, 10]. However, the problem of seamlessly plugging hardware modules is not yet solved and the specification of the data transfers remains to the charge of the designer.

The problem appears when dealing with large amount of data and irregular accesses, as it occurs in complex data-dominated video or multimedia systems. Explicitly specifying all the data transfers might be a very burdensome task. In some works on coprocessors, data transfers can be generated automatically by the host like, for instance, in the case reported in [11]. However, data are copied in the local memory at a predefined location. Thus, the HDL module must be aware of the physical addresses of the data in the local memory. Again, the management of the addresses can be a nontrivial and resource consuming task when dealing with complex algorithms.

The virtual socket concept has been presented in [12–14] and has been developed to support the mixed specification of MPEG-4 Part 2 and Part 10 (AVC/H.264) specifications in terms of reference SW including the plug-in of HDL modules [4]. The platform is constituted by a standard PC, where the SW is running and a PCMCIA card that contains an FPGA and a local memory. In this platform, HDL modules on the platform can be started directly from the reference software, but the data transfers between the host memory and the local memory on the platform must be explicitly specified by the designer/programer.

In conclusion, testing the behavior of the implementation in HW of parts of a reference software is not a trivial task. It is very resource-consuming in term of design efforts, but is very attractive as a design exploration methodology. Designers need a framework in which it is easy to make seamless call the hardware components directly from the reference software and test the performances of the HW modules without worrying too much about low-level implementations details. This is possible only if the hardware components are closely linked to the reference software environment. Some HW/SW codesign platforms can be used to support the algorithm-architecture adaptation methodology [15], but all of them lack a simple procedure capable of plugging seamlessly hardware modules described in HDL within a pure reference software. The problem of the current platforms is that either the memory management is a burdensome task or the call to the hardware modules is done by an embedded processor on the platform.

3. OVERVIEW OF THE VIRTUAL SOCKET PLATFORM

As a possible solution to the problem described in Section 2, this paper presents an extension of the virtual socket platform [12–14] capable of easily plugging and calling hardware modules directly from the reference software without worrying about the data transfers between the host and the hardware modules. Specifying explicitly the data transfers would not constitute a burdensome task when dealing with simple deterministic algorithms for which the data required by the HDL module are known exactly. Data transfers cannot be explicitly specified in advance by the designer in the case of complex designs, where design tradeoffs are much more convenient and viable than worst case designs. The idea is not to build a final embedded system because the underlying hardware used in the platform (wildcard and PC processor) is very different from the hardware components of the platform used for a real implementation. The idea is to provide a framework to transform seamlessly C/C++ reference software into a mixed HW/SW representation (C/C++ and VHDL). The step-by-step transformation method is especially interesting when dealing with large reference software packages composed of several tens of thousand lines of codes. The work described here allows the designer to transform step by step a large reference software package into a mixed HW/SW without worrying about data transfers between the software and hardware environments.

Given a reference software and a given partitioning between SW and HW, so as to test each HW candidate module separately, the designer executes some parts of the reference algorithm using the host processor and runs the HW module on the virtual socket platform which is the implementation support for the hardware. So as to easily handle such mixed HW/SW specifications, it is very convenient that the HDL module and the C/C++ functions have access to the same user memory space. This latter is part of the host software and contains all the data to be processed. Such host memory space is trivially available by the processor which executes the reference software, but it is much less evident for the virtual socket platform which is the support for the HW modules and lies on the FPGA.

The extension consists in adding virtual memory capabilities to enable automatic data transfers between the host, running the SW part and the platform, running the HW modules. Thus, a portion of software reference code can be easily replaced and executed by an HDL module without the need of specifying any data transfer explicitly. HDL modules are started directly from the reference software.

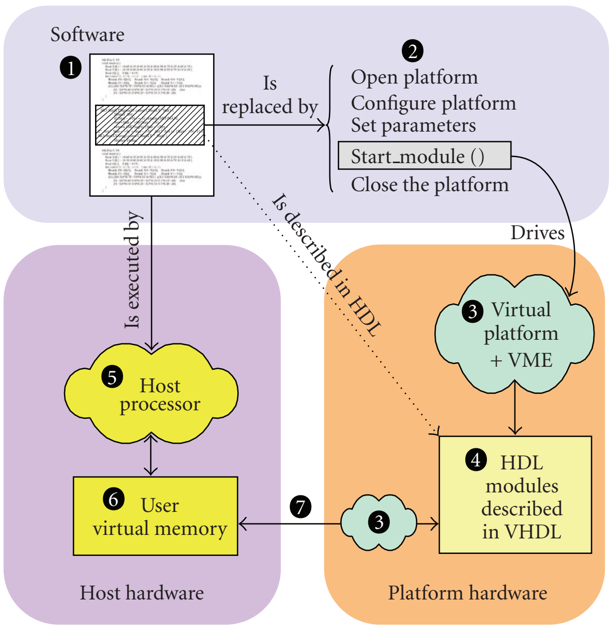

FIGURE 1: Relations between the unified virtual memory, the reference software algorithm, and the HDL module.

Having a shared memory-enforced by a cache-coherence protocol—between the host PC running, the SW sections, and the platform running HW modules—avoids the need of specifying explicitly all the data transfers. The clear advantage of this solution is that the data transfer needed to feed the HDL module can be profiled so as to explore different memory architecture solutions. Another advantage of such direct map is that conformance with the original SW specification is guaranteed at any stage and the generation of test vectors is naturally provided by the way the HDL module is plugged to the SW section. This platform can help in the design of low-power designs thanks to the profiling of data transfers [16] and enable algorithm-architecture codesign methodology to build complex multimedia systems [17].

3.1. *Principle*

Figure 1 illustrates the principle of the platform with its extension and shows the interactions between the unified virtual memory (6), the reference software (1), and an HDL module (4).

For a given algorithm described in a mixed C/C++ and HDL form (1), the software parts are executed by the host processor (5). The hardware parts are executed by the HDL modules described in VHDL (4) implemented on the FPGA. In the original version of the platform [12–14], hardware modules only have access to the local memory on the platform and all the data transfers from the host to the local memory have to be explicitly specified in advance by the designer. It is needless to underline how difficult this task can become when the designer deals with complex signal processing algorithms such as MPEG video codecs. Such operations can be especially error prone when the volume of data is large compared to the small size of the local memory. With the extension, the HDL modules (4) on the FPGA and the C/C++ functions (1) on the host processor (5) have access to the same user memory space (6). The part of C/C++ code of the reference software the designer intends to execute in hardware is replaced by the hardware call function (2). The `Start_module ( )` function drives the virtual socket platform and its extension (3) to trigger the execution of the HDL module (4). The platform (3) manages the data transfers between the main memory (6) and the local memory on the platform (3-4).

3.2. Architecture of the platform

The virtual socket platform is composed of a PCMCIA card and a set of software libraries in C to enable communications between the PC and the platform. The PCMCIA card contains an FPGA, memories, and an interface for the host computer (PC). The virtual socket platform handles the communications between the HDL modules implemented on the FPGA on the PCMCIA card and the host PC. Figure 2 shows the connections between the HDL modules, the host PC, and the virtual socket platform. The virtual memory extension comprises the window memory unit (WMU) and the virtual memory controller (VMC) hardware blocks and a software library (virtual manager window) not represented on the figure.

The extension is capable of automatically handling the data transfers and is implemented with three components. The window manager unit (WMU) hardware module translates virtual addresses into physical addresses. The virtual memory controller (VMC) hardware module is a kind of switch in order to manage two different modes: the original and virtual modes. These modes correspond to the type of access the HDL module asks for. If the HDL module sends a physical address, the explicit mode is active. This mode is the one implemented in the original platform. For further detail, the reader can refer to [12–14]. If the HDL module sends a virtual address to the virtual socket platform, the virtual mode is active. In the virtual mode, the cache-coherence protocol guarantees the coherency between the main and local memories. This protocol is implemented in the window manager unit (WMU) using a translation lookaside buffer (TLB) in cooperation with the virtual manager window (VMW) software library. The designer is free to choose one of the two modes for requesting data from the HDL module.

3.3. Integration of an HDL module into the platform

The platform supports the call of an HDL module from the reference software running on the PC. The first part explains how such a call is written inside the reference software. The second part explains how the VHDL code from the designer must be inserted in the platform.

3.3.1. Call of the HDL module from the reference software

the segment of C code of the reference software the designer wants to execute in hardware using a specific module is replaced by the hardware call function with optional parameters to configure the module. The hardware call function is composed of the following simple steps.

(i) *Open and configure the platform*: the designer configures the platform by using the `Platform_Init ( )` and `VMW_Init ( )` functions from the virtual socket API and VMW API.

(ii) *Parameters*: the designer sets a given number of parameters needed for the configuration of the HW module. Sixteen parameters are available for each HW module.

(iii) *Start of the HDL module*: the HDL module is started with the `Start_module ( )` function which sends first the parameters to the HDL module and then triggers the execution. This function is part of the VMW API. During the execution, the module sends requests to the platform to obtain data. These requests are treated by the virtual socket platform and the virtual memory extension to feed the HDL with the correct data coming from the unified virtual memory.

(iv) *Close the platform*: when the entire job is finished, the platform is closed.

The hardware call function is shown as in Algorithm 1.

3.3.2. The wrapper around the hardware module

the designer intends to test a hardware module corresponding to a piece of reference software. This hardware module has specific inputs and outputs. Around this hardware module, a wrapper provides the link between the hardware module and the virtual socket platform. Such wrapper is connected to the virtual socket platform with the standard interface and to the hardware module under test with its specific inputs and outputs. The wrapper appears as a finite state machine (FSM) and feeds the hardware module with data coming from the virtual socket platform through the standard interface. The wrapper is specific to each hardware module under test because only the designer knows how to feed the data into this hardware module. The FSM must respect the constraints of the hardware module in terms of data and timing. Consequently, there are several ways to integrate the hardware module into the platform.

(i) Designers can use FIFO in front of the module to test the true performances of the hardware module. FIFOs are necessary to avoid that the hardware module waits for data. Using the profiling feature of the platform, designers can test the performances of the module.

(ii) Designers can connect directly the interface of the platform to the inputs and outputs of the hardware module in order to test the functionality. Thus, the profiling tool indicates at which moment the data are accessed. In this case, the hardware module under test must be able to wait for data at their inputs.

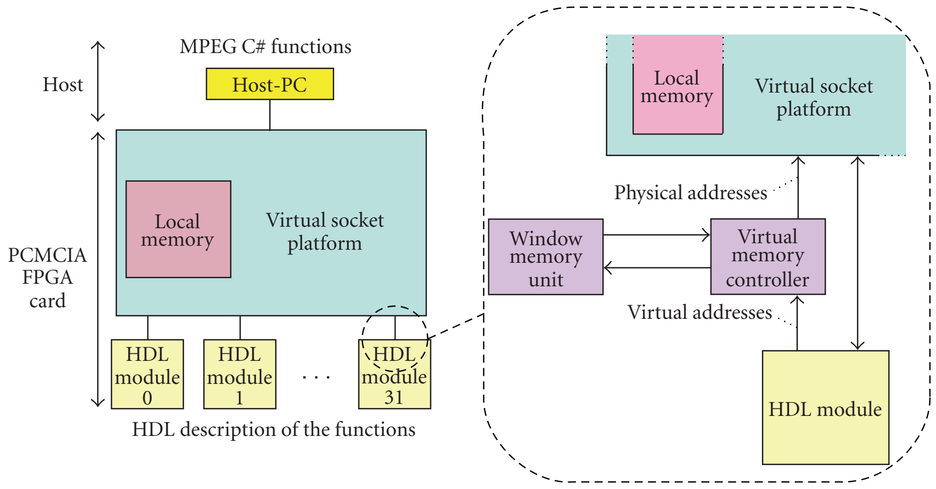

Figure 2: Overview of the virtual socket platform.

```
int main (int argc, char *argv []) {
/* [···] Reference Software Algorithm stops here */
/* Beginning of the HDL module calling procedure */
/******* OPEN AND CONFIGURE THE PLATFORM *******/
  Platform_Init ( ) ;     // Virtual Socket
  VMW_Init ( )  ;        // Virtual Memory Extension

/******* PARAMETERS *******/
  Module_Param.nb_param = 4 ;     // number of parameters
  Module_Param.Param [0] = A ;    // parameter 1
  Module_Param.Param [1] = B ;    // parameter 2
  Module_Param.Param [2] = C ;    // parameter 3
  Module_Param.Param [3] = D ;    // parameter 4

/******* HDL MODULE START *******/
  Start_module (1, &Module_Param)  ;

/******* CLOSE THE PLATFORM *******/
  VMW_Stop ( ) ;            // Virtual Memory Extension
  Platform_Stop ( ) ;     // Virtual Socket

/* End of the HDL module calling procedure */
/* [···] the Reference Software Algorithm continues */
}
```

Algorithm 1

Figure 3 illustrates the links between the wrapper, the virtual socket platform, and the hardware module under test. In order that a hardware module can be tested using this platform, minor constraints need to be considered.

(i) The HDL module must have a `start_process` signal as an input to trigger its execution.

(ii) The HDL module must have a `process_finished` signal as an output to indicate when the module finishes.

(iii) In the case where the designer tests the functionality of the module, this latter must accept latencies, the time for the data to come from the main memory through the virtual memory mechanism.

(iv) In the case in which the designer tests the performances of the module, the wrapper must contain FIFOs at the inputs and outputs so that the hardware module does not presents latencies.

4. IMPLEMENTATION OF THE VIRTUAL MEMORY EXTENSION

This section describes the implementation of the virtual memory extension (VME) based on the original virtual

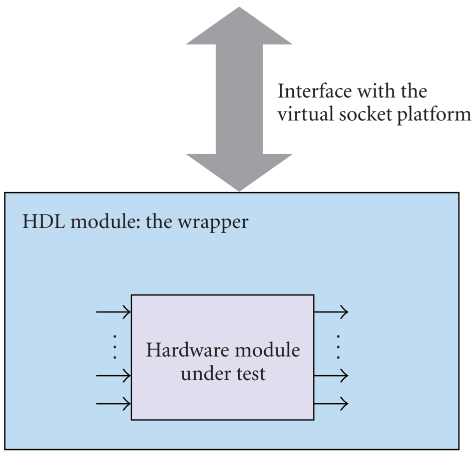
Interface with the virtual socket platform
HDL module: the wrapper
Hardware module under test

Figure 3: The HDL module comprises a wrapper and the hardware module under test.

socket platform. The first and second parts describe the window manager unit (WMU) and the virtual memory controller (VMC) hardware modules, respectively. A third part describes the virtual memory window (VMW) software library. Finally, the last part explains in detail how the virtual memory extension operates.

4.1. *The window manager unit (WMU) hardware module*

The WMU is a component designed by Vuletić et al. [18]. It translates a virtual address into a physical address and is the main component responsible for the cache-coherence protocol, maintaining memory coherence between the main memory of the host and the local memory on the platform. In our framework, the virtual addresses refer to the unified virtual memory space ((6) in Figure 1) and the physical addresses refer to the local memory on the platform (see left part of Figure 2). The WMU is composed of one major element: the translation lookaside buffer (TLB) which is built with a content addressable memory (CAM). It stores the status of the data of each page. When a virtual address inputs the WMU, this latter is translated into a physical address. This latter is composed of an offset and the page number (among the 32 pages available). The offset corresponds to the location of the data in a given page. If the CAM search yields a match between the page number of the physical address and an entry of the CAM, it means that the data are already present in the local memory. If no match exists, the VMW library will intervene in order to copy the data required and write the new entry in the page table in the TLB.

4.2. *The virtual memory controller (VMC) hardware module*

The VMC manages the two available modes: the virtual and explicit modes. In the virtual mode, the virtual addresses must be translated into physical addresses. Additional operations must be done compared to the original protocol (the address translation, e.g.). The VMC intercepts some signals in the standard interface between the HDL module and the platform and sends these signals to the WMU which executes the translation of the addresses. The signals intercepted by the VMC are the address, the count (amount of data requested by the HW module), and the strobe signals.

4.3. *The virtual memory window (VMW) software library*

The VMW library is built on top of the FPGA card driver (wildcard II API), the virtual socket API developed by Yifeng Qiu and Wael Badawy (based on [12, 13]), and the WIN32 API. The VMW library is in charge of transferring the data from the main memory of the host to the local memory on the platform. The WMU raises an interrupt when such a transfer is necessary. In this case, the VMW fills the pages of the local memory with the requested data coming from the virtual memory. The local memory is composed of 32 pages of 2 kB. When the data on the local memory is dirty and must be replaced by new data, the old data are copied back in the main memory.

4.4. *Mechanism*

The goal of the virtual memory extension is to support the direct access of HW modules to the software virtual memory space located on the host PC. The HDL module can ask for four types of access.

(i) Read data from the local memory on the platform.

(ii) Write data to the local memory on the platform.

(iii) Read data from the unified virtual memory space on the host.

(iv) Write data to the unified virtual memory space on the host.

The first two requests belong to the *explicit mode* in which the HDL module sends physical addresses (relative to the local memory). This mode is already implemented in the original version of the platform. It will not be detailed in this paper, the reader can refer to [12–14] for further information.

The last two requests belong to the *virtual mode* in which the HDL module sends virtual addresses (relative to the unified virtual memory). In the virtual mode, the addresses are the one of the data in the unified virtual memory space. In other words, it enables the HDL module to have a transparent access the host memory.

The following paragraphs explain in more details how data in the virtual memory of the host is accessed from the HDL without any intervention of the designer. The protocol to write and read in the unified memory space are very similar and for the sake of clarity, only the read protocol is described.

4.4.1. *Send the pointers*

the piece of reference code the designer wants to execute in hardware is replaced first by the hardware call function (see Section 3.3.1) in order to call the hardware module. This latter needs data to work and the designer must specify which data must be used. Pointers of the data are sent to the HDL module by using parameters. For example, if the HDL module needs two images as input, the designer specify the pointers to the images using the parameters in the hardware call function as in Algorithm 2.

When the execution of the hardware module is triggered with the `Start_module ( )` function (part of the hardware call function), the wrapper of the HDL module receives the parameters (pointers) because it is the wrapper which is connected to the virtual socket platform (see Figure 3). According to these pointers, the wrapper can generate the requests to the virtual socket platform to get the data.

4.4.2. *The requests of the module*

the wrapper can generate the requests (i.e., the addresses) from the pointers it just received. The way the requests are done depends on how the data must be inputted in the HDL module. In image processing, for example, there are different ways to read an image, raster or blocks. If the HDL module needs a raster format, the wrapper sends the necessary requests in order that the data received from the platform are in a raster format. The HDL module can send either an explicit or a virtual request, that is, physical addresses or virtual addresses. The use of the two modes is further detailed in Section 6. The wrapper uses the existing protocol of communication with the virtual socket to make the requests (explicit or virtual). A read request consists in asserting the following signals of the interface.

(i) `hw_mem_hwaccel`: number of the hardware module.

(ii) `hw_offset`: read address.

(iii) `hw_count`: number of data to read.

(iv) `memory_read`: strobe signal of the request.

A write request consists in asserting the following signals of the interface.

(i) `hw_mem_hwaccel1`: number of the hardware module.

(ii) `hw_offset1`: write address.

(iii) `hw_count1`: number of data to write.

(iv) `output_valid`: strobe signal of the request and the data.

4.4.3. *Mode management*

according to the type of request sent by the module, the VMC will send the address, count, and strobe signals to the WMU or not. If the request is explicit, the signals go directly to the virtual socket platform and this latter sends the data to the HDL module according to the initial protocol in the original platform (see [12–14]). But if the access is virtual, the requested address is virtual and it must be translated into a physical address so that the platform can send the corresponding data to the HDL module. The WMC catches some signals from the standard interface and sends them to the WMU which translates this virtual address into a physical one.

4.4.4. *The translation of the address*

the WMU receives a virtual address from a HDL module through the VMC. The WMU translates this virtual address into a physical address. The translation mechanism is illustrated in Figure 4.

The virtual address is 32 bits long. The page offset is 11 bits long because the size of a page in the local memory is 2 kb. Thus, the pattern to be translated is a word of 21 bits. The translation consists in simply replacing the pattern by a page number, knowing that the replacement strategy is FIFO. The first request of the HDL module is going to result in a miss because there is no data in the local memory yet. Thus, the VMW will fill the first page of the local memory with the requested data from the main memory. When the WMU detects a new missed request, the second page of the local memory is filled, and so on. When the 32 entries are complete, the first page is used again; it is an FIFO strategy. If the status of this page is dirty, the VMW will copy back the data to the main memory before replacing the old data by the new one.

Once the address is translated, the WMU checks if the data corresponding to the virtual address is already present in the local memory on the platform. If yes, the WMU sends the corresponding physical address to the virtual socket platform. If not, the WMU raises an interrupt and asks the VMW to copy the requested data in the local memory. The platform knows if the requested data is already present in the local memory thanks to the cache-coherence protocol which guarantees the coherence between the main memory and the local memory. The check is done by verifying the status of the data contained in the local memory thanks to the TLB/CAM (see Section 4.1).

4.4.5. *The automatic transfer of the data*

if the WMU raises an interrupt, the requested data in the main memory corresponding to the virtual address is copied in the local memory. The VMW library transfers all the data between the main memory (the unified virtual memory) and the local memory. The collaboration of the WMU and the VMW implements a cache-coherence protocol. This latter keeps the status of the data of the local memory on the platform.

(i) Dirty: data copied on the local memory and modified by the HDL module.

(ii) Valid: data copied on the local memory and not modified by the HDL module.

(iii) Invalid: no data copied at this address.

```
Module_Param.nb_param = 2 ;      // number of parameters
Module_Param.Param [0] =   &img1  ;  // pointer to image no. 1
Module_Param.Param [0] =   &img2  ;  // pointer to image no. 2
```

Algorithm 2

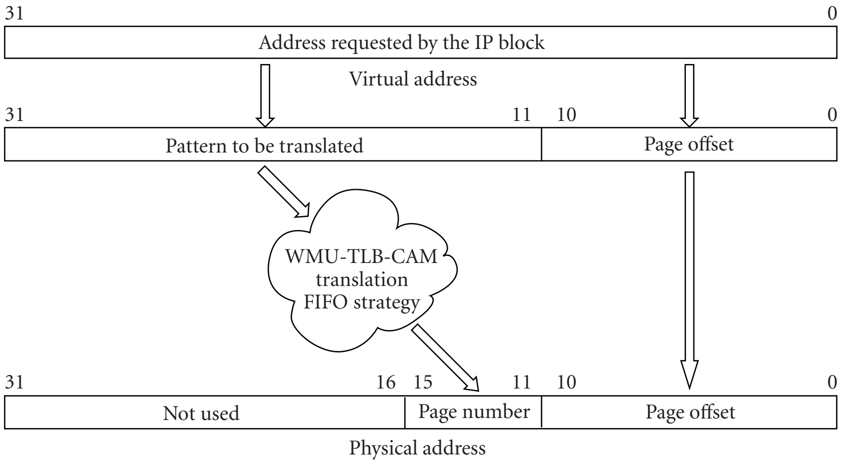

Figure 4: The translation of a virtual address into a physical address.

4.4.6. *Answer to the request of the module*

when the data has been transferred, the WMU sends the corresponding physical address to the virtual socket platform which sends the data (which are now in the local memory) to the HDL module. When the job of the HDL module is entirely finished, the data processed by the HDL module are copied back in the host memory in order that the reference software can continue running with updated data, created and/or modified by the execution of the HDL module. Thanks to the VME, the data are updated without any intervention of the designer.

Finally, from the designer point of view, using the virtual memory extension, the whole process of data transfers is completely transparent. The only issue the designer has to care for is to generate the virtual addresses accordingly to the data contained in the host memory space. The whole task of transferring data to local memory is done by the platform and its software support.

5. PROFILING FEATURES

Once the HDL module is correctly called by the reference software and executed on the platform, it would be interesting to get useful information on the execution of the hardware module. The profiling feature of the platform can provide such information by recording the data accesses between the HDL module and the virtual socket platform. Since in the field of signal processing, algorithms are essentially data driven, optimizing the data transfers between the different components is crucial for such signal processing systems. Data transfers provide also a relevant contribution to the overall power dissipation in embedded systems. Consequently, these transfers must be carefully optimized in order to be successful in low-power designs.

Table 1 is an example of profiling information given by the platform. The column **Rd/Wr** indicates if the request of the module is a read (r) or a write (w). The column **Addr** contains the address of the data requested by the module. The column **Count** indicates how much data is requested by the module from the address specified in the previous column (kind of a burst). The column **Clk** indicates at which clock cycle the request appends. The origin corresponds to the time at which the module begins its processing. The column **event** indicates the type of event happening. "v" corresponds to a virtual access, "o" corresponds to an explicit access, and "d" indicates the end of the job of the HDL module.

The example corresponding to Table 1 consists of copying 1000 dwords from one place in the main memory to another place in the same memory. The HDL module asks for reading 255 data from the address 0×366408 of the main memory. Once the HDL module received the data, it copies them to the new place in a main memory by asking for a writing request at address $0 \times$ 3673D8. The 1000 dwords are copied by sets of 255 dwords. Each virtual access requests a time overhead. In virtual mode, each virtual access requests 4 cycles. Therefore, between a read and write operations 1024 cycles are requested. The time (derived from the clock cycles information) recorded by the platform is the time at which the HDL module asks for data. When a request is sent by the hardware module to the platform, the time is "stopped" until the platform answers and data are inputted in the HDL module. Thus, the profiling tool does not take into account the time taken by the platform to feed the HDL module with

Table 1: Example of profiling information given by the platform.

Rd/Wr	Addr. (hex)	Count	Clk	Event
r	0x366408	255	0	v
w	0x3673d8	255	1024	v
r	0x366804	255	2048	v
w	0x3677d4	255	3072	v
r	0x366c00	255	4096	v
w	0x367bd0	255	5120	v
r	0x366ffc	239	6144	v
w	0x367fcc	239	7168	v
–	–	–	8192	d

the requested data. It is as if the data arrived immediately after the request. However, the time elapsed for the data transfer from the local memory to the module is taken under account. The profiling tool aims at studying how the HDL modules asks data and not how it communicates with the rest of the system because the architecture of the platform will never be implemented in a real embedded system.

The profiling feature is implemented in hardware (WMU) and in software (VMW). The WMU raises interruptions for each data transfers between the HDL module and the virtual socket platform. The VMW library handles these interruptions and records the profiling information in a simple text file. It catches the information presented in Table 1.

6. DESIGN METHODOLOGY

6.1. Objectives

The main goal of the virtual socket platform and its extension is to ease the communications between an SW environment (i.e., a PC) and an HW module (i.e., an FPGA) for the rapid evaluation and profiling of IP blocks derived from a reference software specification. These IP blocks are supposed to be used in the final implementation. In other words, this platform is a support in the process converting large C/C++ reference software packages into mixed HW/SW description/implementations. A step-by-step approach is the methodology adopted so as to reach the desired final implementation. By using the virtual memory extension, the data transfers between the SW environment and the HW modules are handled automatically. If the designer desires to convert entirely the reference software into a hardware implementation, he can replace step-by-step pieces of the C/C++ code with HW modules and proceed with this approach until the entire SW algorithm is converted and integrated in hardware. If the designer desires to target the conversion to a mixed representation of the reference software, he can stop at any intermediate step. Section 7 describes a methodology based on the profiling features of the virtual socket platform to reach optimized implementations.

6.2. Methodology

Figure 5 illustrates the different steps of the proposed methodology.

The *first step* (called "*Building the module*") consists in partitioning the C/C++ reference software into software and hardware partitions. The methodology that identifies these partitions is not the scope of this work and is not further discussed here. The second task consists in writing the hardware modules in HDL. This can be done manually or by using any other C to HDL tools and methodologies, according to the preferences of the designer. The third task consists in inserting the HW module by replacing the corresponding code by the hardware call function (see Section 3.3.1) with parameters if necessary (e.g., the virtual addresses of the data used by the module). The HDL module must be also wrapped in order to satisfy the communication protocol with the platform (see Section 3.3.2).

The *second step* (called "*Conformance tests*") consists in checking the conformance of the hardware module with respect to the reference software. The equivalency of the C/C++ and HW modules is checked relying on the virtual socket platform and the virtual memory extension in the virtual mode. The virtual memory feature simplifies the conformance tests procedures because the designer does not have to care about the data transfers between the main memory and the local memory because this stage is guaranteed by the platform. The conformance check is done by comparing the results generated by the reference software and the HW module. Such verification is done directly in the reference software environment. No profiling is needed at this step because the designer only checks the conformance of the module and not its performances.

The *third step* (called "*Optimization*") consists in optimizing the HW module by using the virtual mode and the profiling features of the platform. The designer can grasp an overview of the data transfers exchanged between the platform and the HW module. The way data are accessed can be the object of further optimization steps. Updates of the internal algorithm of the module can affect the way data are accessed, the amount of requested data and the timing at which the transfers are done. This phase helps the designer to refine the HDL code of the HW module. Using the support provided by the virtual memory extension, the designer can forget about the data transfers between the reference software and the HW module and has only to specify to the module the virtual addresses of the data used by the module. According to the profiling results, the designer faces different cases. The time elapsed for the data transfers and the processing time can be measured by the platform. According to these values, three cases are possible.

(i) Transfers < computations: the processing is too long compared to the data transfers. The optimization effort must focus on the processing in order to reduce the processing time.

(ii) Transfers = computations: the system is balanced. While the transfers occur, the processing is executed. When this latter finished, a new set of data is

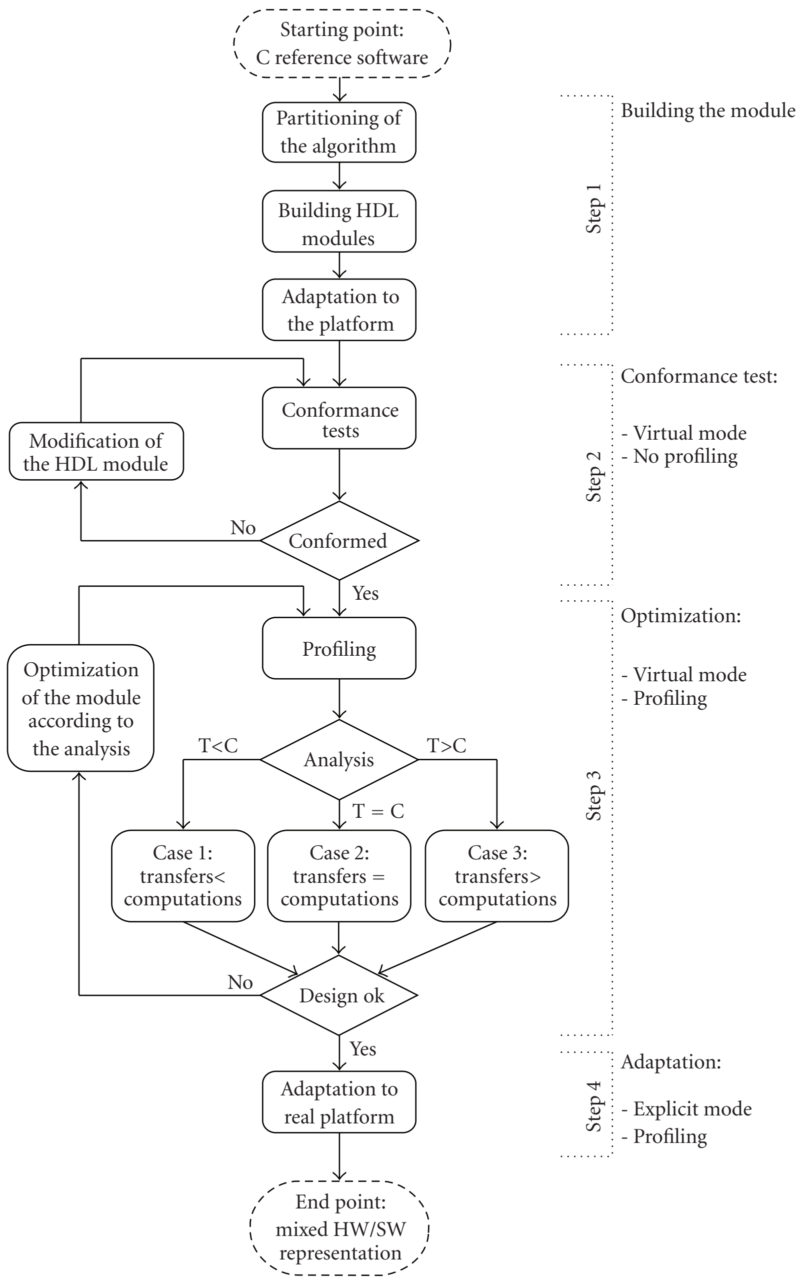
Starting point: C reference software
Partitioning of the algorithm
Building HDL modules
Adaptation to the platform
Conformance tests
Modification of the HDL module
Conformed
No
Yes
Profiling
Optimization of the module according to the analysis
Analysis
T<C
T>C
T = C
Case 1: transfers< computations
Case 2: transfers = computations
Case 3: transfers> computations
Design ok
No
Yes
Adaptation to real platform
End point: mixed HW/SW representation
Step 1
Building the module
Step 2
Conformance test:
- Virtual mode
- No profiling
Step 3
Optimization:
- Virtual mode
- Profiling
Step 4
Adaptation:
- Explicit mode
- Profiling

Figure 5: Flow chart representation of the design flow steps using the virtual socket platform.

immediately available at the same time. There is no loss of efficiency.

(iii) Transfers > computations: the module is waiting for data. The designer can choose either to reduce the amount of data used by the module or to implement an equivalent processing that is slower (by adding algorithmic complexity so as to get better results if possible and convenient) or by reducing resource usage or power dissipation whenever appropriate.

According to the overall design policy of the system, the designer can take appropriate design decisions by analyzing profiling results given by the platform. Either the design fills the requirements and the designer continues with the last step, or the design does not fill the requirements and he modifies the module until it is satisfactory (iteration of the third step).

Once the design is satisfactory, the designer can enter the *fourth step* (called "*Adaptation*"). This step consists in adapting the HW module to a real target platform. The explicit mode is used at the place of the virtual mode. At this step, the data exchanged by the HW module and the platform are well-defined. Data transfers are expressed explicitly using physical addresses. At the end of this step, the HW module sends physical addresses, relative to the final implementation design.

7. CASE STUDY: DESIGN OF AN MPEG-4 MOTION ESTIMATION MODULE

This design case intends to show how the virtual socket platform can be used to explore algorithmic/architecture tradeoffs so as to achieve the highest possible performances under bandwidth and processing constraints. In this context, an example of the implementation of a motion estimation with a reduced search strategy HW component is developed. The platform structure, and the processor possibilities, represents a real advantage for the effective implementation of a reduced search strategy algorithm that matches the HW platform capabilities. The performances of an HW block can be considerably increased and optimized using the described methodology and the profiling results obtained by the platform. The profiling information extracted by the virtual socket platform (specifically the timing performance) is investigated by using the virtual memory accesses mode.

7.1. Motion estimation in a video encoding context

The motion estimation is well known to be the most computation-intensive stage of video coding process and has been the subject of many research works (on both algorithmic and architecture sides) which aim at reducing the implementation complexity. For brevity, we cannot here provide an accurate review of the wide literature on the subject, we suggest referring, for instance, to [19, 20], and their references, for an updated review of the state-of-the-art. We just remind the main families of approaches developed so far.

The first family is the so-called "reduced search algorithms," which aims at reducing the complexity by limiting the measure of the matching criterion to only a (small) subset of candidate vectors in the search window. The key element here is the "intelligence" of the search algorithm that may completely change depending on the requirements of the video coding application (portable video telephony, HDTV for sport events, etc.).

A second family is constituted by the approaches in which the complexity reduction is achieved by using multiresolution searches on subsampled image search windows.

A third approach is to use simplified matching criteria in place of the classical maximum absolute difference (MAD) criterion.

A fourth family of approaches is based on various preprocessing stages that reduce the images to binary images for which simple XOR operations are used for the evaluation of the MAD.

7.2. On the choice of the design case study

For most of the algorithms, composed of one or more algorithmic elements from the different families sketched above, dedicated optimized implementations using systolic arrays and/or specific data flows handling are needed to achieve effective performances. While a lot of effort has been devoted to developing such reduced complexity solutions (search algorithm plus data flow implementation), much less effort has been devoted to study how to be able to scale the solutions versus the different parameters such as the size of the search window, the available memory bandwidth (that usually is an off-chip memory), and the processing power. Most of the solutions require a close coupling between data flow handling and the dedicated hardware. Nowadays, with the appearance of powerful processors with specialized instruction sets and new families of FPGA with embedded arithmetic for which the MAD evaluation is no longer a difficult burden, some of the reduced complexity solutions presented in the past have lost their interest. Modularity, flexibility to cover the different coding applications, and the possibility of upgrading using the more recent results and improvements are more desirable and possible implementation objectives.

For such reasons in this case study, we focus on architectures that present a wide degree of flexibility and modularity of the different elements versus the various performance and implementation parameters of motion estimation so as to be able to cover different applications ranging from high-quality HDTV up to mobile video. The family of approaches supported is the reduced search algorithms on a specific search window. The recent results, including the comparison with other approaches, have shown that using appropriate (for the video application) reduced search algorithms is possible to achieve optimal coding results [21]. The main idea of the architecture so as to achieve the desired level of flexibility and reprogramability for the search is thus to separate the data access and the matching criterion implementation stages from the intelligence of the algorithmic search. Therefore, the coprocessor architecture, depending on the support platform and matching criterion used, can be programed freely by the user at high level with his preferred intelligent search, exploiting the resource budget in terms of available candidate vectors for each search. The implementation objective of the architecture is not only an efficient hardware implementation not only to speed up matching operations, but also to provide the possibility of randomly matching any area in a search window without any other limitation on the access sequence. The random-block access in the search window is obtained with a specific data flow architecture and address generation strategy. This accelerator has been presented in [19]. The flexibility of the hardware accelerator supports all the different motion estimation modes which appear in the recent extension of the MPEG video standard family such as AVC. The search-window size is parametric so that it can be configured according to the search window size of the profile under consideration or reduced according to the desired application.

7.3. Design flow

This section describes the application of the proposed methodology supported by the virtual socket platform to the design of the motion estimation module that satisfies the objectives described above. The sections follows the design flow described in Figure 5.

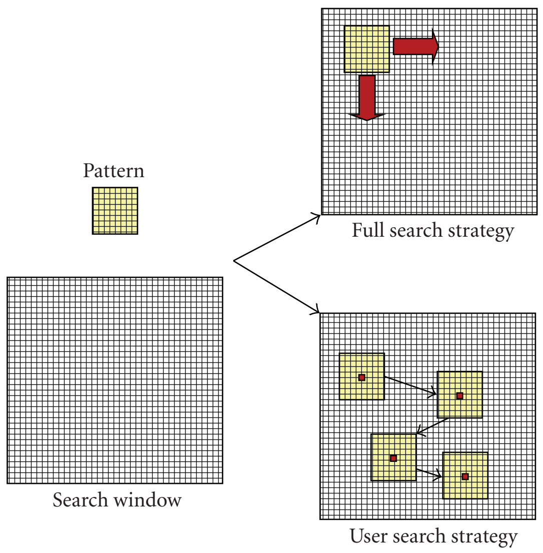

Figure 6: Full-search and reduced-search strategies.

7.3.1. Building the module

the conception and the integration of the HW accelerator represent the first step of the design flow. Therefore, the first step in the design flow must be the partitioning of the algorithm. The motion estimation takes place in the MPEG encoder, described in the reference software which is the starting point of the implementation. The piece of code relative to the motion estimation process is identified in the reference code, and then implemented. A full-search algorithm is implemented in a first time. The motion estimation process is regular, as illustrated in Figure 6, and all possible positions of the pattern in the search window are searched. In this case, the number of matching depends on the sizes of the pattern and of the search window (i.e., for 16 × 16 pattern and a 56 × 40 search window, 1025 matching are needed). Obviously, with such high number of operations (i.e., >1000 matching), the motion estimation represents then the most computation intensive stage of the video encoding. During a matching process, the architecture does not support any interruption. Therefore, the wrapper of the module requests an FIFO memory in front of the data interfaces (input and output). Finally, the HW module is integrated into the platform. The piece of code in the reference software relative to motion estimation is replaced by the hardware call function (see Section 3.3.1). The fundamental information, as the sizes the addresses of the pattern and the search window, is transferred to the HW module through the hardware call function and its parameters structure.

7.3.2. Conformance tests

since the HW module has been correctly designed and produces the requested results, the second step is completed. The conformance of the module is tested in the complete software environment by testing the results of the reference software using the results of the HW module. As far as the results of the HW module are not matching the reference software, the loop must be iterated by debugging and redesigning the module until it is correct. This step can be considered as an HW debugging step, and can complete the common simulation phase.

Table 2: Profiling results for the full-search algorithm.

Rd/Wr	Addr. (hex)	Count	Clk	Event
r	0x366408	64	0	v
r	0x366538	560	256	v
w	0x362f90	1	20950	v
–	–	–	20959	d

7.3.3. Optimization—first iteration

the third step consists in analyzing, profiling, and optimizing the HW module. The profiling results with a full-search methodology are obtained for the implementation setting on a search window with a size of 56 × 40 and 16 × 16 blocks. The frequency of the motion accelerator is 50 MHz. The study of the profiling highlights several aspects. Table 2 shows a fraction of the profiling results for this configuration. At the beginning of the processing, the FIFO receives the input data. The pattern is transferred and followed by the search window. The two first rows of Table 2 represent these two transfers. The data is stored on the hard disk space, therefore virtual accesses are required. The pointers of the pattern and the search window are used by the FSM of the wrapper to generate all the required addresses. As 32 bits (i.e., 4 pixels) are transferred at each access, the pattern and the search window are transferred, respectively, in 64 and 560 accesses. Due to the virtual mode overhead, each access is performed in 4 cycles. At the end of the processing, one write access enables the resulting motion vector to be stored into the output FIFO. The whole processing is completed in 20950 cycles. As the data transfers (pattern and search window) are done in 12480 nanoseconds (i.e., 624 accesses) and the whole matching processing (the 1025 possible matching) is finished in 369000 nanoseconds, the profiling confirms that a simple matching is processed at this frequency in 360 nanoseconds.

Moreover, the profiling results confirm that the processing represents the major part of the require time. With a higher resolution (for HDTV, e.g.), the size of the search window increases and the processing time even more. The processing stage must be improved in priority. Consequently, the search strategy is modified and the HW module modified.

7.3.4. Optimization—redesign

the architecture refinements are applied to implement a reduced search algorithm so as to decrease the processing time of the module.

Contrary to the full-search strategy, a reduced search aims at minimizing the number of matching, in this variant the objective is achieved by using previous matching information (i.e., the values of the vectors previously computed in time and space). As mentioned previously, using appropriate (for the video application) reduced search algorithms, it is possible, at the same time, to achieve optimal coding results [21] and reduce the processing time for each macroblock motion estimation. Figure 6 represents a reduced search (4 matching represented) strategy versus a full-search one.

The list of candidate matching to process is reduced according to the user's search algorithm. The motion estimation accelerator receives this list of candidates and processes only the listed positions. The possibility of the HW module of randomly matching any area in a search window without any other limitation on the access sequence is exploited in a first phase. In other terms, even with a random access, the performance per matching is not changed.

Therefore, the MAD matching implementation has not been modified. The implemented pipelined architecture enables to match a block and a portion of search window, row by row. The modification is done on the data structure to guarantee the data access. The data structure modification is detailed in a previous work [19]. Moreover, the address generation process is modified. The address generation is provided by a flexible unit to enable the translation of the user search strategy into the hardware setup. Contrary to previous work, this unit is not included into the HW accelerator (FPGA or embedded processor), the host processor is in charge of the generation to increase the system flexibility and to take advantage of the virtual memory mode in the different configurations to be tested (e.g., different sizes of the search window).

To integrate the HW module, the wrapper is modified by including an FIFO memory dedicated to matching addresses in front of the address interface. Moreover, a control register is added to start the accelerator or to define the number of matching. The resulting wrapper is presented in Figure 7.

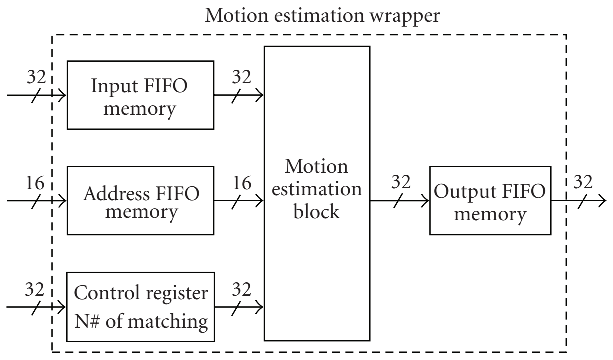

Figure 7: Motion estimation accelerator.

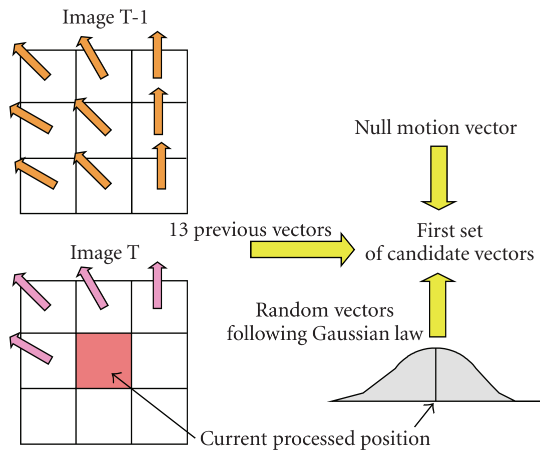
Image T-1
Null motion vector
13 previous vectors
First set of candidate vectors
Image T
Random vectors following Gaussian law
Current processed position

Figure 8: Determination of first set of candidate vectors.

7.3.5. *Optimization—second iteration*

the motion vector is usually highly correlated with the previous vectors. So as to exploit this feature, the user algorithm defines the different matchings to be processed. The proposed algorithm is presented below. The algorithm is split into two phases. As shown in Figure 8, initially nine motion vectors in a 3 × 3 corresponding neighborhood of the previous image (Image T-1) are listed as candidates. In the current image (Image T), the four motion vectors early processed are considered. The zero motion vector should also always be considered, therefore it is added as a candidate. In view of the profiling results in terms of data transfer and processing time, the user can then determine the number of matching (or motion vectors) that are still available as process budget. Therefore, depending on the amount of the budget left measured by the profiling results, the user can add some random vectors following, for instance, a Gaussian law centered on the current position.

These new candidate vectors are processed. In a second phase, the resulting score of each candidate vector enables them to be ranked (Figure 9). Vector additions of better ranked vectors are executed to define a second set of candidate vectors. Once again, the number of combinations to define new candidates depends on the profiled user budget [21]. Conformance with optimal coding results can be obtained with less than 60 candidate vectors in a 56 × 40 search window (phase 1 and phase 2 aggregate together) and can be directly validated by executing the HW module and the SW.

The scheduling of the tasks is described in Figure 10. The input data (pattern and search window) and the addresses of the matching to process are transferred to the HW module during task 1. Tasks 2 and 6 represent, respectively, the processing of the first and the second sets of candidate vectors. Tasks 3 and 7 correspond, respectively, to the transfer of the resulting vectors form the HW module to the host processor. Task 4 represents the computation of the second set of vectors. Finally, task 5 corresponds to the transfer from the PC to the HW module of the matching addresses relative to the second vector set. The data are still resident into the internal memory cache of the HW module, therefore are not required.

By using the proposed reduced search strategy, the number of matching has been reduced to 100 (50 candidate vectors for each phase). The profiling tests show that the first phase is achieved in 23 480 nanoseconds (random access and

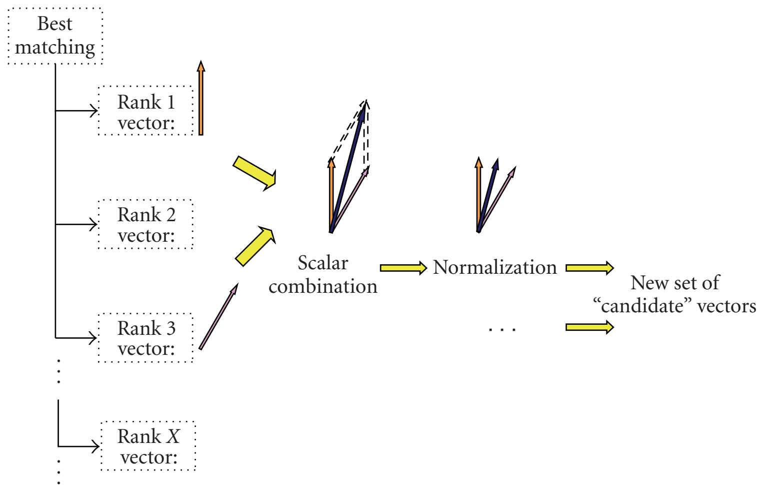

Figure 9: Determination of second set of candidate vectors.

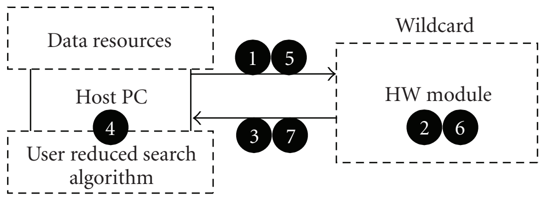

Figure 10: Scheduling of the different tasks.

data transfer included). The second phase requires only the new set of candidate vectors (matching address) as input data. This phase can be finished in 11 000 nanoseconds. Consequently, the whole process, excluding the determination of the second vector set, can be achieved in 34 000 nanoseconds. Depending on the complexity of the search algorithm, the acceleration of the processing, in comparison to the full-search strategy, can be very high (in this example a ratio equals to 10). Obviously, the required time for the determination of the second vector set has to be taken into account.

The data access flexibility of the VMW platform enables to easily implement and explore different search window sizes. With a 80 × 40 search window, the data transfer is measured to be equal to 17 280 nanoseconds and the processing time for a full search is 585 000 nanoseconds (for the 1625 possible matchings). With a reduced-search strategy of 200 candidate vectors, the processing time for the whole processing (excluding the determination of the second vector set) is then 61 280 nanoseconds. The processing time is much lower than for a full-search strategy (even with a 56 × 40 search window) and the acceleration achievable by the HW platform is even higher.

7.4. Discussion

Using the VMW platform capabilities and plugging the motion estimation accelerator are possible to easily explore a wide variety of optimization solutions without the need of detailing all necessary data transfers explicitly. The host processor remains in charge of the definition of the list of candidate matching. In summary, the system is highly flexible on three features and provides the following.

(i) A simple data-flow control (with virtual data access).

(ii) An easy integration of the HW accelerator implementation.

(iii) Profiling information on the data flow and the processing time for all tested implementation options.

Moreover, the system does not only provide flexibility of exploring and testing solutions, but also provides direct results of algorithmic-architectures tradeoffs in terms of overall coding performance increases. For instance, the user can scale the resource budget in terms of available candidate vectors for each search in view of the required performances and select the best configuration for the video encoding under test. Using the profiling information measured, the data transfer time and the processing time, the designer can

(i) adjust the number of vectors to obtain optimal coding results or in general select the tradeoff between the coding performances and desired processing time;

(ii) adjust/modify the search algorithm complexity (e.g., three vectors can be considered or different reduced search strategy investigated);

(iii) adjust/increase the size of the search window looking for optimal tradeoffs between coding performances and required memory bandwidths.

The major challenges of the architecture design of the co-processor for motion estimation are to guarantee the random access of any search strategy and to enable the setting of the size of the search window, hence providing sufficient processing resources for the different encoding applications. So as to guarantee the random access in any position of a search window, the search window pixels have to be accessible to the matching engine and need to be stored in the FPGA to reduce the number of accesses to the external memory, so as not to exceed the available bandwidth. A current investigation aims at minimizing the size of the internal cache in function of the kind of image sequence. The profiling statistics enable the number of virtual accesses to be measured, therefore the number of misses in the local cache. Different cache sizes can be quickly implemented thanks to the virtual access mode, therefore a local cache can be defined for the target sequences. Other parameters can be considered. Indeed, this definition represents a tradeoff between the optimum size and the latency tolerated which is correlated to the ratio between the processing and the data transfer times. Therefore, the cache memory size can be defined in view of the processing time, in other words, with consideration of the selected search algorithm parameters (algorithm, number of matching, or search window size).

8. CONCLUSION

This paper describes the implementation of a platform that enables SW and HW environments to share the same virtual memory space. The platform helps the designer in the different design steps aiming at developing implementations of heterogeneous embedded systems starting from a specification described by a reference software. The platform provides a seamless environment and a clear methodology to help designers to turn C/C++ reference software into HDL modules. The conformance tests become straightforward. The main advantage of the platform is that it provides a step-by-step approach for designing, evaluating, and integrating hardware modules into a heterogeneous environment. The profiling capabilities on the data transfers between the SW and HW components of the platform support the designer to explore different implementation and optimization options at different stages of the design process. Initially, design efforts can be focused on the module functionality without worrying about data transfers. Then, by using the profiled data transfer, designers can focus on appropriate memory architectures, on algorithm/architecture tradeoffs, or on any other design optimizations that match the specific desired criteria of the design that affects or are affected by data transfers between modules.

REFERENCES

[1] "Information technology—generic coding of audio-visual objects—part 2: visual," ISO/IEC 144962-2:2004, June 2004.

[2] K. Denolf, K. Vissers, P. Schumacher, et al., "A systematic design of an MPEG-4 video encoder and decoder for FPGAs," in *Proceedings of the Global Signal Processing Expo and Conference (GSPx '04)*, Santa Clara, Calif, USA, September 2004.

[3] "Information technology—generic coding of audio-visual objects—part 7: optimized reference software," ISO/IEC TR 14496-7:2004, October 2004.

[4] "Information technology—generic coding of audio-visual objects—part 9: reference hardware description," 2nd edition, ISO/IEC TR 14496-9:2007.

[5] The Xilinx XUP Virtex-II Pro Development System, http://www.xilinx.com/univ/xupv2p.html.

[6] Celoxica Handel-C and RC1000-PP PCI board Production Information, Celoxica Ltd., http://www.celoxica.com/.

[7] A. Koch, "A comprehensive prototyping-platform for hardware-software codesign," in *Proceedings of the 11th International Workshop on Rapid System Prototyping (RSP '00)*, pp. 78–82, Paris, France, June 2000.

[8] M. Edwards and B. Fozard, "Rapid prototyping of mixed hardware and software systems," in *Proceedings of the Euromicro Symposium on Digital System Design (DSD '02)*, pp. 118–125, Dortmund, Germany, September 2002.

[9] R. Pradeep, S. Vinay, S. Burman, and V. Kamakoti, "FPGA based agile algorithm-on-demand coprocessor," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '05)*, vol. 3, pp. 82–83, Munich, Germany, March 2005.

[10] C. Plessl and M. Platzner, "TKDM—a reconfigurable coprocessor in a PC's memory slot," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '03)*, pp. 252–259, Tokyo, Japan, December 2003.

[11] S. Sukhsawas, K. Benkrid, and D. Crookes, "A reconfigurable high level FPGA-based coprocessor," in *Proceedings of IEEE International Workshop on Computer Architectures for Machine Perception (CAMP '03)*, p. 4, New Orleans, La, USA, May 2003.

[12] T. S. Mohamed and W. Badawy, "Integrated hardware-software platform for image processing applications," in *Proceedings of the 4th IEEE International Workshop on System-on-Chip for Real-Time Applications (IWSOC '04)*, pp. 145–148, Banff, Canada, July 2004.

[13] I. Amer, C. A. Rahman, T. Mohamed, M. Sayed, and W. Badawy, "A hardware-accelerated framework with IP-blocks for application in MPEG-4," in *Proceedings of the 5th IEEE International Workshop on System-on-Chip for Real-Time Applications (IWSOC '05)*, pp. 211–214, Banff, Canada, July 2005.

[14] P. Schumacher, M. Mattavelli, A. Chirila-Rus, and R. Turney, "A virtual socket framework for rapid emulation of video and multimedia designs," in *Proceedings of IEEE International Conference on Multimedia and Expo (ICME '05)*, pp. 872–875, Amsterdam, The Netherlands, July 2005.

[15] L. Kaouane, M. Akil, T. Grandpierre, and Y. Sorel, "A methodology to implement real-time applications onto reconfigurable circuits," *The Journal of Supercomputing*, vol. 30, no. 3, pp. 283–301, 2004.

[16] C. Lucarz and M. Mattavelli, "A platform for mixed HW/SW algorithm specifications for the exploration of SW and HW partitioning," in *Proceedings of the 17th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS '07)*, pp. 485–494, Gothenburg, Sweden, September 2007.

[17] C. Lucarz, M. Mattavelli, and J. Dubois, "A HW/SW codesign platform for algorithm-architecture mapping," in *Proceedings of the Workshop on Design and Architectures for Signal and Image Processing (DASIP '07)*, Grenoble, France, November 2007.

[18] M. Vuletić, L. Pozzi, and P. Ienne, "Virtual memory window for application-specific reconfigurable coprocessors," in *Proceedings of the 41st Annual Conference on Design Automation (DAC '04)*, pp. 948–953, San Diego, Calif, USA, June 2004.

[19] J. Dubois, M. Mattavelli, L. Pierrefeu, and J. Miteran, "Configurable motion-estimation hardware accelerator module for the MPEG-4 reference hardware description platform,"

in *Proceedings of IEEE International Conference on Image Processing (ICIP '05)*, vol. 3, pp. 1040–1043, Genoa, Italy, September 2005.

[20] J.-H. Luo, C.-N. Wang, and T. Chiang, "A novel all-binary motion estimation (ABME) with optimized hardware architectures," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 12, no. 8, pp. 700–712, 2002.

[21] M. Mattavelli and G. Zoia, "Vector-tracing algorithms for motion estimation in large search windows," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 10, no. 8, pp. 1426–1437, 2000.