

Interface Synthesis using Memory Mapping for an FPGA Platform *

Manev Luthra[†] Sumit Gupta[†] Nikil Dutt[†] Rajesh Gupta[§] Alex Nicolau[†]

[†]Center for Embedded Computer Systems
University of California at Irvine
{mluthra, sumitg, dutt, nicolau}@cecs.uci.edu

[§]Dept. of Computer Science and Engineering
University of California at San Diego
gupta@cs.ucsd.edu

Abstract

Several system-on-chip (SoC) platforms have recently emerged that use reconfigurable logic (FPGAs) as a programmable co-processor to reduce the computational load on the main processor core. In this paper, we present an interface synthesis approach that forms part of our hardware-software co-design methodology for such an FPGA-based platform. The approach is based on a novel memory mapping algorithm that maps data used by both the hardware and the software to shared memories on the reconfigurable fabric. The memory mapping algorithm couples with a high-level synthesis tool and uses scheduling information to map variables, arrays and complex data structures to the shared memories in a way that minimizes the number of registers and multiplexers used in the hardware interface. We also present three software schemes that enable the application software to communicate with this hardware interface. We demonstrate the utility of our approach and study the trade-offs involved using a case study of the co-design of a computationally expensive portion of the MPEG-1 multimedia application on to the Altera Nios platform.

1 Introduction

Platform based designs provide promising solutions for handling the growing complexity of chip designs [1]. FPGAs often play an important role in platforms by providing flexible, reconfigurable circuit fabrics to build and optimize target applications.

Our focus is on platform designs for multimedia and image processing applications. Our target architecture models many emerging platforms that contain a general purpose processor assisted by dedicated hardware for computationally intensive tasks. The hardware assists are implemented on-board FPGA blocks and thus, can be programmed for different applications. Binding the application functionality to software and hardware requires automated methods to specify, generate and optimize the interface between them. The interface should be fast, transparent and require minimal hardware and software resources. This paper presents a methodology to generate these interfaces.

Multimedia and image processing applications typically operate on a large data set. Consequently, when these applications are partitioned and mapped onto an FPGA-based platform, this data has to be communicated and shared between the hardware and software components using large

memories. Configurable logic blocks in FPGAs are typically inefficient for use as memories: if we store each data element in a register and provide an independent access mechanism for each one, then the resulting memory implementation occupies a large portion of the FPGA fabric. Instead, an efficient way to implement these large memories is to cluster the data elements into RAMs or register banks. In this paper, we present our interface synthesis approach that efficiently utilizes embedded RAMs in FPGAs to implement the memory. Our approach is based on a novel memory mapping algorithm that generates and optimizes a hardware interface used for integrating the computationally expensive application kernels (hardware assists) with the rest of the platform.

Our memory mapping algorithm makes use of scheduling information on per cycle data access patterns (available from the high-level synthesis tool) in order to map registers to memories. The unique feature of this algorithm is its ability to efficiently handle designs in which data access patterns are unknown during scheduling - for example, an array being indexed by variable indices which become known only at run-time. This feature proves to be extremely useful when dealing with designs involving control flow.

To validate our co-design methodology, we present a case study of the co-design of a computationally expensive portion of the MPEG-1 multimedia application. We find that without using our memory mapping algorithm, the portion mapped to the FPGA is too big to fit inside it. We also compare results for various hardware-software interfacing schemes used with this design.

The rest of this paper is organized as follows: in the next section, we discuss related work. In Section 3, we describe our methodology and the relation between memory mapping and hardware interface synthesis. We formulate the memory mapping problem and present an algorithm for solving it in the next two sections. In Sections 6 and 7, we describe the hardware and software interface architectures. We then present our MPEG-1 case study and finally, conclude the paper with a discussion.

2 Related Work

Hardware-software partitioning [2] and high level synthesis [3, 4] have received significant attention over the past decade. Interface synthesis techniques have focused on various issues like optimizing the use of external IO pins of micro-controllers and minimizing glue logic [5, 10]. However, the use of memory mapping for interface synthesis has not been considered.

*This work is partially supported by NSF grants CCR0203813 and CCR0205712

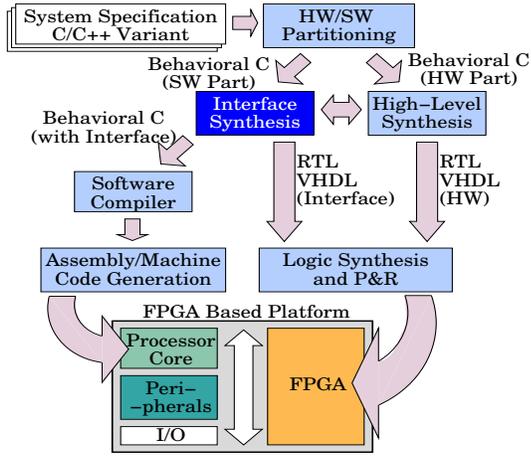


Figure 1. *Role of interface synthesis in a co-design methodology*

Most previous work on memory mapping and allocation of multi-port memories has focused on purely data flow designs (no control constructs) [4, 6, 8]. They do not deal with unknown data access patterns because no control flow is involved. Memory mapping and register binding algorithms in the data path synthesis domain are based on variable lifetime analysis and register allocation heuristics [4, 7].

Earlier work on memory mapping for FPGAs has not utilized scheduling information [7]. Several groups have looked at packing uneven sized data structures into fixed aspect ratio memories available on FPGAs [9].

3 Role of Interface Synthesis and Memory Mapping in a Co-Design Methodology

Interface synthesis is an important aspect of our hardware-software co-design methodology, as shown in Figure 1. In our approach, we rely on a C/C++ based description for the system model. After hardware-software partitioning, the hardware part is scheduled using a high-level synthesis tool [3] and the scheduling information is passed to the interface synthesizer.

This interface synthesizer – described in detail in the rest of the paper – generates the hardware interface and re-instruments the software component of the application to make appropriate calls to the hardware component via this interface. It also passes the addresses of all registers that have been mapped to memories in the hardware interface to the high-level synthesis tool.

The RTL code generated by the high-level synthesis tool and the interface synthesizer are then downloaded to the FPGA. Similarly, the software component is compiled and downloaded into the instruction memory of the processor.

3.1 Memory Mapping

Multimedia and image processing applications process large amounts of data. After partitioning, the hardware component has to operate on the same data that the software operates on. Thus, the hardware component needs to store this data on the FPGA (see Section 6 for how this is achieved). The way this data is mapped to a memory has tremendous impact on the complexity of the multiplexers and the generated control logic. Ideally, we would store

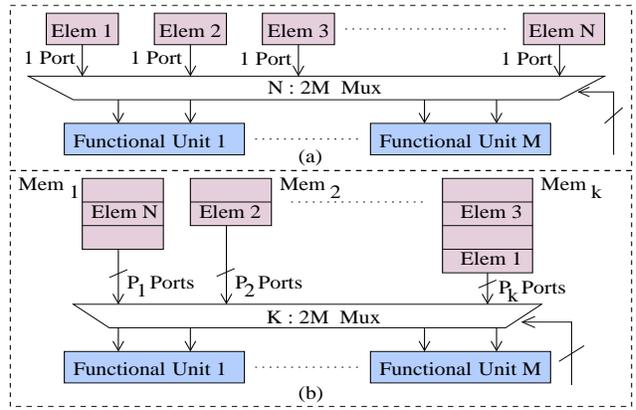


Figure 2. (a) *Unmapped Design: Registers for each data element* (b) *Mapped Design: Data elements mapped to memories, $K \ll N$, where K and N are the number of memory banks and variables respectively.*

all data in a single large memory. However, such a memory would require as many ports as the maximum number of simultaneous memory accesses in any cycle [8]. This is impractical for programmable FPGA platforms, since they provide memories with only a limited number of ports [12, 13]. Consequently, memories with a larger number of ports have to be implemented using individual registers. This requires a large number of registers and complex, large multiplexers as shown in Figure 2(a).

In our memory mapping approach, we utilize scheduling information – available from the high-level synthesis tool – about data accesses and the cycles that they occur in. We can then map the data elements to memory banks, given constraints on the maximum number of ports each memory in the target FPGA can have. This approach eliminates the use of registers for storage, thus, saving a large amount of area. This way, we can also use much smaller and faster multiplexers in the data-path as illustrated in Figure 2(b).

Arrays and data structures are mapped to memories after being broken down into their basic constituents (variables). These can then be mapped in a way identical to regular variables. Consequently, these basic constituents might get mapped to non-contiguous memory addresses. In Section 7 we show how this drawback can typically be overcome by making a few changes to the application software.

4 Problem Formulation

We are given a set of n variables, $V = \{v_i; i = 1, 2, \dots, n\}$ that are accessed (read and written) by all the kernels of the application. In our current model, only one kernel executes at any given time. This implies that contention for variable accesses between two kernels can never occur. Note that, each element in an array or data structure is considered as a distinct variable v_i in V ; so for example, an array of size n will have n entries in V . We are also given a set of memory resource types, $M_{type} = \{m_j; j \in \mathbb{Z}^+\}$ where the subscript j indicates the maximum number of ports available. The number of read ports of memory type m_j are given by $Ports_{read}(m_j)$ and write ports by $Ports_{write}(m_j)$.

```

Algorithm 1: MapVariablesToMemories( $V$ )
Output: Memory instances used in the design  $M$ ,
Mapping between memory instances and variables  $\phi$ 
1 : Initialize  $M \leftarrow \emptyset$ 
2 : foreach ( $v_i \in V$ ) do
3 :    $\mathcal{L} \leftarrow \text{GetListOfCandMems}(M, v_i)$ 
4 :   if ( $\mathcal{L} = \emptyset$ ) then /* Create a new memory instance */
       /* with a minimal number of ports to satisfy  $v_i$  */
5 :     Add new instance ( $m_p, n$ ) of memory type  $m_p$  to  $M$ 
6 :      $\phi(m_p, n) \leftarrow v_i$  /* map  $v_i$  on  $n$ th instance of  $m_p$ 
7 :   else /*  $\mathcal{L}$  is not empty */
8 :     Pick  $((m_j, k), m_p) \in \mathcal{L}$  with lowest cost
9 :     if ( $m_p \neq m_j$ ) then
       /* Add new  $q$ th instance of mem type  $m_p$  to  $M$  */
10 :       $M \leftarrow M \cup (m_p, q)$ 
11 :       $\phi(m_p, q) \leftarrow \phi(m_j, k)$  /* Move variables to  $(m_p, q)$  */
12 :       $M \leftarrow M - (m_j, k)$  /* Discard  $(m_j, k)$  */
13 :       $\phi(m_p, q) \leftarrow \phi(m_p, q) \cup v_c$ 
14 :     else /* map  $v_c$  to  $(m_j, k)$  */
15 :        $\phi(m_j, k) \leftarrow \phi(m_j, k) \cup v_c$ 
16 :     endif
17 :   endif
18 : endforeach

```

Figure 3. *The memory mapping algorithm*

Definition 4.1 *The memory mapping problem is to find a memory allocation $\phi : (M_{types} \times Z^+) \rightarrow V$ that is a mapping of memory instances to variables assigned to them in the design. This mapping also gives the list M of all the memory instances allocated to the design. $\phi(m_j, n)$ represents the list of variables mapped to the n -th instance of memory type m_j . The optimization problem is to minimize the total number of memory instances, given by $size(M)$, with the constraint that for each memory instance (m_j, n) used in the design, the number of simultaneous accesses during any cycle should not exceed the number of memory ports available on m_j .*

5 Mapping Algorithm

The problem defined above is an extension of the memory mapping and value grouping problem for datapath synthesis, which is known to be NP-complete [14]. We adopt a heuristic approach to solving it; our memory mapping algorithm is listed in Figure 3. The algorithm processes variables in the order that they are declared in the application. For each variable to be mapped to a memory instance, the algorithm calls *GetListOfCandMems* to get a list of candidate memory instances (\mathcal{L}) onto which the current variable v_c can potentially be mapped (line 3 in Figure 3).

If this list is empty, a new memory instance with just enough ports for v_i is created, and v_i is mapped to it (lines 4 to 6). If the list is non-empty, we pick the memory instance with the lowest cost. If the number of ports available on this memory instance are sufficient to map v_i to it, then v_i is added to the list of variables $\phi(m_j, k)$ mapped to this instance; otherwise, a new memory instance (m_p, q) with enough ports is created. The old memory instance (m_j, k) is discarded after all variables mapped to it have been remapped to (m_p, q) . Finally, v_i is mapped to (m_p, q) (lines 9 to 13 in the algorithm).

The algorithm for the function *GetListOfCandMems* is listed in Figure 4. This algorithm considers each memory

```

Algorithm 2: GetListOfCandMems( $M, v_c$ )
Return: Available Memories List  $\mathcal{L}$ 
1 : Initialize List  $\mathcal{L} \leftarrow \emptyset$ 
2 : foreach (memory instance  $(m_j, k) \in M$ ) do
3 :   if ( $v_c$  does not conflict with  $\phi(m_j, k)$  in any cycle)
       or ( $(m_j, k)$  has enough ports to map  $v_c$ ) then
4 :      $\mathcal{L} \leftarrow \mathcal{L} \cup ((m_j, k), m_j)$ 
5 :      $Cost(m_j, k) \leftarrow Ports_{read}(m_j) + Ports_{write}(m_j)$ 
6 :   else /*either conflict or insufficient ports in  $(m_j, k)$ */
7 :     if (there exists  $m_p \in M_{type}$  with enough ports
8 :       to map all variables from  $(m_j, k)$  and  $v_c$ ) then
9 :        $\mathcal{L} \leftarrow \mathcal{L} \cup ((m_j, k), m_p)$ 
10 :       $Cost(m_j, k) \leftarrow Ports_{read}(m_p) + Ports_{write}(m_p)$ 
11 :     endif
12 :   endif
13 : endforeach

```

Figure 4. *Determining the list of available memories*

instance (m_j, k) in M already allocated to the design, and adds this instance to the list \mathcal{L} of candidate memory instances if the variable v_c can be mapped to (m_j, k) . A variable v_c can be mapped to (m_j, k) when, v_c does not conflict in terms of reads or writes with any other variable mapped to (m_j, k) , or (m_j, k) has enough ports for accessing variable v_c besides all the variables already mapped to it (line 3 in Figure 4). Note that these two constraints are identical, i.e., if one is true, it implies that the other is also true.

If (m_j, k) does not have enough ports to map variable v_c , then we try to find a memory of type m_p , such that, an instance of m_p will satisfy the port constraints when variables v_c and $\phi(m_j, k)$ (variables already mapped to (m_j, k)) are mapped to it. If such a memory type exists, it marks memory instance (m_j, k) for an upgrade to an instance of memory type m_p ($p > j$) and adds it to \mathcal{L} (lines 7 to 9).

The algorithm in Figure 4 also calculates a cost for mapping v_c to each memory instance in \mathcal{L} . This cost equals the total number of read and write ports of the memory instance.

Assume that A is the total number of hardware kernels, s is the length of the longest schedule among these kernels, while z is the maximum number of memory accesses occurring in a single cycle by any one variable. Then, lines 2 and 3 in Figure 4 individually contribute n and Asz to the time complexity respectively. So the *GetListOfCandMems* algorithm has a worst case time complexity of $O(nAsz)$. The loop in line 2 of the *MapVariablesToMemories* algorithm in Figure 3 causes the *GetListOfCandMems* algorithm to execute n times. Thus, the worst case time complexity of the *MapVariablesToMemories* algorithm is $O(n^2Asz)$.

5.1 Construction of Conflict Graphs

The *GetListOfCandMems* algorithm determines if variable v_c can be mapped to memory instance (m_j, k) by checking for potential conflicts with the variables $\phi(m_j, k)$ that have already been mapped to (m_j, k) . This is done for every cycle. Thus, in every cycle, we create *conflict graphs* where nodes represent variables and edges denote a conflict between variables (both variables are accessed in that cycle).

To understand how we use these conflict graphs, consider a design with three variables v_1, v_2 and v_3 . Assume that v_1 and v_2 are accessed during cycle 1, while v_2 and v_3 are accessed during cycle 2. The corresponding conflict graphs

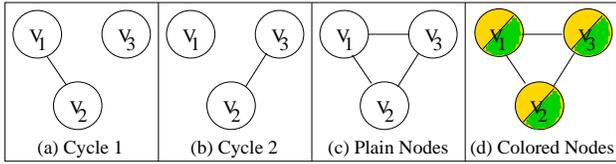


Figure 5. (a) *Conflict graph for cycle 1* (b) *Conflict graph for cycle 2* (c) *Conflict graph with colorless nodes* (d) *Conflict graph with multi-colored nodes*

for the two cycles are given in Figures 5(a) and 5(b). If we have only one memory resource type, namely, a dual ported memory m_2 , then, each of the three variables can be mapped to the same instance of the dual ported memory without violating the port constraints. This is because only two of the three variables conflict in any cycle. If we had represented this using a single conflict graph for all cycles, variable v_2 would not have been mapped to memory because two conflict edges would have been associated with it, even though the accesses occur in different cycles.

Let us explore this further with another example. Consider an array arr consisting of three elements, $arr[1]$, $arr[2]$ and $arr[3]$. The corresponding variables in V are v_1 , v_2 and v_3 . Also, assume that dual ported memories are the only memory types available. In any given cycle, if there are multiple accesses to arr using variable indices i and j (for example $arr[i]$ and $arr[j]$), then we cannot determine which elements of the array actually conflict until runtime. Hence, we create conflict edges between each pair of elements in arr in the conflict graph corresponding to that cycle. This results in the fully connected conflict graph shown in Figure 5(c). We can conclude from this conflict graph that none of the three variables can be mapped to the same memory instance since the memory has only two ports.

But, this is an incorrect conclusion because only two of the three variables will be accessed in any cycle. This implies that the three variables can be mapped to a dual ported memory. Thus, we find that per cycle conflict graphs are, by themselves, not powerful enough to capture all the information necessary to perform effective memory mapping.

To address such issues, we introduce the notion of *accumulating colors* in the nodes of the conflict graphs of each cycle. From among a group of variables V_G , if access of any one in a cycle rules out access of the rest in that cycle, then the nodes corresponding to each variable in the group V_G are marked with the same color c_G . This color is unique from that of all other groups. A node can accumulate colors by being a member of more than one such group.

Applying this to our example, we get the conflict graph depicted in Figure 5(d). v_1 , v_2 and v_3 form a group corresponding to access by $arr[i]$. Each of these three variables are marked with one color since accessing any one of them rules out accessing the other two. Similarly, v_1 , v_2 and v_3 form another group corresponding to access by $arr[j]$ and are marked with a second color. Thus, each of the three nodes/variables end up accumulating two colors.

The number of ports needed to map a set of variables to a single memory instance, is equal to the maximum number

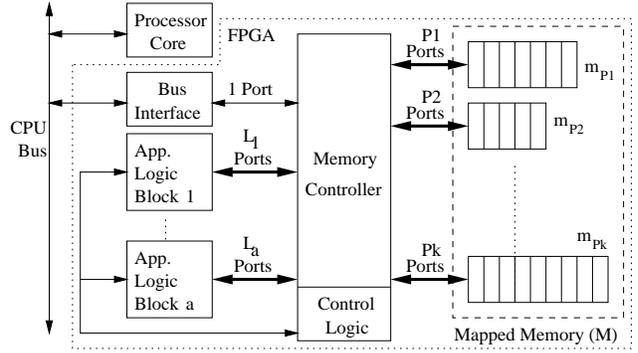


Figure 6. *The hardware interface architecture*

of colors in any cycle of all the variables being mapped to that memory instance. In our example, the number of colors accumulated by v_1 , v_2 and v_3 is two. Thus, we can safely map these three variables to a dual ported memory. We use this coloring technique while creating the per cycle conflict graphs used by our memory mapping algorithm. Note that although we need to store each per cycle conflict graph, we found that for large applications such as MPEG, the storage sizes are easily manageable.

6 The Hardware Interface Architecture

An overview of the architecture obtained after hardware interface synthesis is shown in Figure 6. The bus interface generated is specific to the bus protocol used. The control logic contains memory mapped registers that can be used to reset or start/stop execution of any of the application logic blocks through software. It also contains registers reflecting the execution status and logic for interrupt generation, masking, et cetera. The memory controller services all memory access requests for data residing in the mapped memory M . It is designed to give a higher priority to access requests by the application logic blocks. An access request by software is serviced only if a free port is currently available on the memory instance.

7 The Software Interface

In the hardware interface shown in Figure 6, the memory M uses a contiguous address space. Hence, data declarations in the software code have to be reordered so that they conform to the order in which they were mapped to this address space. The example in Figure 7 illustrates this. Note that, when multiple arrays or data structures get sliced apart due to memory mapping, it is possible to perform address translations in the memory controller in order to abstract the memory architecture details from software.

The software can interface and share data with the hardware mapped to the FPGA by either transferring all the data to the hardware, or they can use a shared memory [11], as explained in the next two sections.

7.1 Data Transfer Based Scheme

In a *data transfer* scheme, all the shared data is copied from the processor (or main) memory to the mapped memory. Then, the hardware executes and the results are moved back from the mapped memory to the main memory.

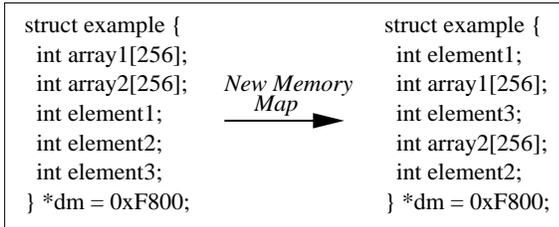


Figure 7. *Modified address offsets in the software interface after memory mapping*

The advantage of using this scheme is that the execution speed of the software portion of the application is independent of the memory used by the hardware. The disadvantages are: (a) the large communication costs of copying data from software to hardware and back, and (b) the creation of heavy bursts of traffic on the processor bus, which can potentially starve other devices that want to use it. Thus, to amortize the communication cost, the hardware-software partitioning has to be done in such a manner that communication between hardware and software is minimized.

7.2 Shared Memory Based Schemes

The other way hardware and software can interface is through *shared memory* (i.e. M in Figure 6 is shared). This can be done by using shared memory *with no local storage* or shared memory *with local storage*. In the scheme with no local storage, variables and data structures in the shared memory are declared such that the compiler does not apply any memory optimizations and uses processor registers minimally (for example, by declaring them as *volatile* in the C code). Other forms of local storage like processor caches are also bypassed when accessing the shared memory. Data to be processed by software is always read from the shared memory (in hardware) and the results are immediately written back (no caching). Due to its simplicity, this scheme can be used with any processor.

In contrast, the shared memory with local storage scheme can only be used when the processor supports explicit instructions for flushing all local storage to memory.

The clear advantage of both these schemes is the zero data transfer (communication) cost between software and hardware. However, the shared memory with no local storage scheme has the additional advantage that it maintains data coherency since the data is not buffered inside the processor’s local memory. But, this causes extra traffic on the processor bus whenever the software portion of the application is running. A disadvantage of both schemes is that a larger access time for mapped memory can degrade performance significantly. Thus, the performance with these schemes depends critically on the speed of the processor bus, the mapped memory and associated access logic.

8 Experimental Setup and Results

We used Altera’s Nios development platform [12] for implementing the system. This platform consists of a soft core of the Nios embedded processor (no caches) to which various peripherals can be connected for customization. Once the system has been customized, it can be synthesized and mapped onto an FPGA based board provided by Altera. The

HW Module	Area	Max Freq	RAM Bits
Nios System	2800 LCs	47 MHz	26496
Unmapped User-defined	9750 LCs	27 MHz	0
Mapped User-defined	2450 LCs	32 MHz	2048

Table 1. *Logic synthesis results for an Altera APEX20KE EP20K200EFC484-2X FPGA target*

user can use dedicated RAM block bits inside the FPGA to implement memories having two or fewer ports. The entire system, consisting of the Nios processor and its standard peripherals, the main memory, the CPU bus and the user-defined module operate at a frequency of 33.33 MHz. The user-defined module is connected to the CPU bus and contains the hardware interface and all application logic.

We synthesized application kernels using a parallelizing high level synthesis framework called *SPARK* [3]. This framework takes a behavioral description in C as input and generates synthesizable register-transfer level VHDL. The VHDL was synthesized using the logic synthesis tool, Leonardo Spectrum. The resultant netlist was mapped to the FPGA using the Altera Quartus tool. The portions of the application that were not mapped to hardware, were compiled and mapped onto the processor on the FPGA platform using the compiler suite from the Nios development toolkit.

8.1 Case Study: MPEG-1 Prediction Block

In this section, we present a case study using a multimedia application to evaluate the effectiveness of our mapping algorithm and to study the trade-offs between the three interfacing and communication schemes described in Section 7. We used the *prediction block* from the MPEG-1 multimedia application.

To begin with, three computationally intensive kernels in the application were identified using profiling information and these were mapped to hardware without using our interface synthesis approach. The globals declared by this application were 53 integer variables and two integer arrays of 64 entries each; making a total of 181 integer data elements. The results of conventional logic synthesis of the user-defined module are listed in Table 1. Note that, the maximum frequencies shown in the table are only the estimates by the logic synthesis tool and were found to be slightly conservative. We could not integrate this design with the Nios embedded system because it was too big to fit inside the FPGA, given that the capacity of the FPGA was 8320 Logic Cells (LCs).

Next, we applied the memory mapping algorithm to the 181 data elements in the design and came up with a new memory configuration. Then, we made a new memory controller based on this new memory map and reperformed conventional logic synthesis on the user-defined module. The results obtained are shown in Table 1. The area, speed and the number of RAM block bits used by the rest of the Nios embedded system (without the user-defined module) have also been shown for reference. The observed 75% reduction in area, along with the 19% increase in operating frequency, clearly shows the benefits of our memory mapping technique. As a result, we were also able to fit the hardware inside the FPGA easily.

Scheme	Entity	SW (ms)	HW (ms)	Speedup
Data Transfer Based	Kernel 1	110	12(17)	9.2
	Kernel 2	110	12(17)	9.2
	Kernel 3	108	12(17)	9.0
	Misc	42	96	-
	<i>Total</i>	370	132	2.8
Shared Memory with no Local Storage	Kernel 1	178	12	14.8
	Kernel 2	178	12	14.8
	Kernel 3	183	12	14.8
	Misc	42	60	-
	<i>Total</i>	581	96	6.1
Shared Memory with Local Storage	Kernel 1	116	12	9.7
	Kernel 2	116	12	9.7
	Kernel 3	109	12	9.1
	Misc	42	59	-
	<i>Total</i>	383	95	4.0

Table 2. Execution times for 500 iterations of various kernels of the application

For studying the three interfacing and communication schemes described in Section 7, the memory mapped version of the user-defined module was integrated with the Nios embedded system. The variables and data structures in the application were re-arranged as per the new memory map and appropriate function calls were inserted to invoke the hardware. Execution times were recorded using a timer peripheral that had been integrated with the Nios system.

Table 2 shows the breakup of execution times of various parts of the application based on a run of 500 iterations of the application. For the data transfer based scheme, the value given in parentheses for each kernel (under the HW column) indicates the cost of copying data from main memory to mapped memory and then vice-versa after execution. *Miscellaneous cost* represents the cost of executing everything other than the three kernels. Under the HW column, it represents the cost of running that portion of the application in software, which did not get mapped to hardware. It also includes all software overheads of using the hardware.

Figure 8 shows the variation in execution time for each of the three schemes as the application shifts execution from software to hardware. This graph shows that the shared memory with local storage scheme performed the best from among the three, for various hardware software partitions. The data transfer based scheme was the fastest software only scheme; so taking it as the reference, the shared memory with local storage scheme gave a speedup of 3.9.

9 Conclusions and Future Work

We presented an interface synthesis approach that forms part of a hardware-software co-design methodology targeted at FPGA-based platforms. This approach is based on a novel memory mapping algorithm that maps the data elements operated on by both, software and hardware, to shared memories. These shared memories, thus, form a hardware interface between the hardware and software components of the application. We also presented three software schemes for bridging between the hardware interface and the application software. Results for experiments performed on an MPEG-1 application demonstrated an improvement in performance by a factor of 3.9 when three application kernels were accelerated in hardware using our

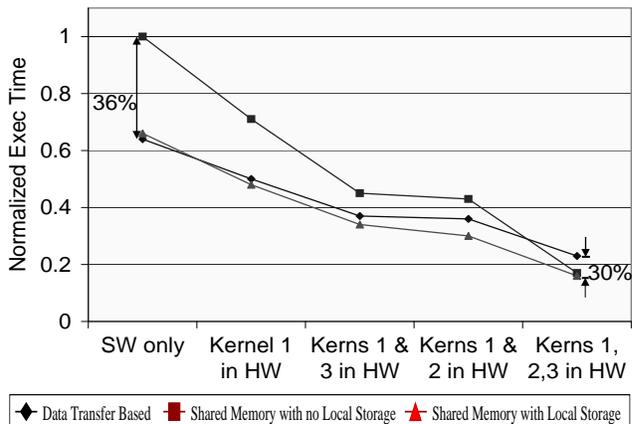


Figure 8. Execution times for 500 iterations of the application with different hardware software partitions

interface synthesis approach. In future work, we want to validate our methodology for other FPGA platforms and develop a generalized hardware-software synthesis framework that includes task analysis and partitioning. We also want to investigate improvements to the heuristic used in the memory mapping algorithm.

References

- [1] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE D&T*, Dec. 2001.
- [2] R.K. Gupta and G. De Micheli. Hardware-software cosynthesis for digital systems. *IEEE D&T*, Sept. 1993.
- [3] S. Gupta et al. Spark : A high-level synthesis framework for applying parallelizing compiler transformations. In *International Conference on VLSI Design*, 2003.
- [4] D. Gajski, N. Dutt, A. Wu, S. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic, 1992.
- [5] P. Chou, R. Ortega, G. Borriello. Interface co-synthesis techniques for embedded systems. *ICCAD*, 1995.
- [6] M. Balakrishnan et al. Allocation of Multiport Memories in Data Path Synthesis. *IEEE TCAD*, April 1988.
- [7] I. Ouass and R. Vemuri. Hierarchical memory mapping during synthesis in FPGA-based reconfigurable computers. In *DATE*, 2001.
- [8] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.
- [9] D. Karchmer and J. Rose. Definition and solution of the memory packing problem for field-programmable systems. In *ICCAD*, 1994.
- [10] J. Daveau et al. Protocol selection and interface generation for HW-SW Codesign. *IEEE TVLSI*, March 1997.
- [11] Z. Chen et al. Pilot - a Platform-based HW/SW synthesis system for FPSoC. In *Workshop on Software Support for Reconfigurable Systems*, 2003.
- [12] The Altera Website. <http://www.altera.com>.
- [13] The Xilinx Website. <http://www.xilinx.com>.
- [14] E.G. Woutersen and S.H. Gerez. Some complexity results in memory mapping. *HCM BELSIGN Workshop*, 1996.