A Heuristic-based Communication-aware Hardware Optimization Approach in Heterogeneous Multicore Systems

Cuong Pham-Quoc, Zaid Al-Ars, Koen Bertels
Computer Engineering Lab, Delft University of Technology
Email: {P.PhamQuocCuong, Z.Al-Ars, K.L.M.Bertels}@tudelft.nl

***Abstract*—Multicore processing, especially heterogeneous multicore, is being increasingly used for data intensive processing in embedded systems. An important challenge in multicore processing is, efficiently, to get the data to the computing core that needs it. In order to have an efficient interconnect design for multicore architectures, a detailed profiling of data communication patterns is necessary. In this work, we propose a heuristic-based approach to design an application-specific custom interconnect using quantitative data communication profiling information. The ultimate goal is, automatically, to have the most optimized custom interconnect design taking runtime communication pattern into account. Experimental results show that the hardware accelerators speed-up achieved in comparison with software is up to $7.8\times$, which is $2.98\times$ in comparison with the system without using our interconnect approach.**

***Keywords*-heuristic-based custom interconnect, hardware accelerator, data communication bottleneck, quantitative profiling.**

I. Introduction

In recent years, multicore architectures, especially heterogeneous multicores, are emerging as promising architectures for data intensive processing in embedded systems such as multimedia computing, HD digital TVs, etc. Single-core microprocessors have reached the end of their scaling capabilities. Meanwhile, Field Programmable Gate Arrays (FPGA) have shown their flexibility with their fine grained architecture, but they are limited by overhead in timing, area and power consumption. Software design in traditional microprocessors is straightforward while FPGA designs have benefits in performance. Therefore, hardware/software co-design (also called hardware accelerator systems) is one of the important approaches for multicore design.

In hardware accelerator systems, a processor has a main memory, while hardware accelerators usually have their local memory to improve the parallelism. The communication and the synchronization between computing cores (microprocessor and hardware accelerators) are normally done through an interconnect network such as buses, Networks on Chip (NoC), etc. In data intensive applications, a large amount of data needs to be transferred from core to core. Therefore, data communication is usually a primary anticipated bottleneck for system performance [1], [2], [3], [4]. One important method to improve the speed-up of such systems is reducing data communication overhead.

Reducing data communication overhead can be done by increasing communication throughput or decreasing the amount of data movement from one memory to another. Examples of the former are [1], [2], [3], [5], [6] and of the latter are [4], [7]. However, all the aforementioned works are based on static information of the application such as task graphs. The actual amount of data transferred between cores (which is responsible for data communication overhead) is not taken into consideration.

The above highlights are important challenges in multicore processing, namely to efficiently get the data to the computing core that needs it. The goal is, of course, to hide the communication delay such that a performance improvement can still be observed. The resource allocation decision requires detailed and accurate information on the amount of data that is needed as input and what will be produced as output. Evidently, there are dependencies between computations since data produced by one core will be needed by another. To have an efficient allocation scheme where the communication delays can be hidden as much as possible, a detailed profile on the data communication patterns is necessary for which the most appropriate interconnect infrastructure can be generated. Such communication patterns can be specific for each application (domain) and could, therefore, lead to different types of interconnect. The work presented in this paper is a first step towards a custom designed interconnect for an application. The ultimate goal is to change at runtime the interconnect infrastructure.

The main contributions of this paper can be summarized as follows: 1) the introduction of a heuristic-based and detailed profile-driven interconnect design with an emphasis on runtime management; 2) the presentation of experimental results with seven different applications on a real FPGA platform; and 3) identification of the most suitable interconnect for each application domain in our experiment.

The rest of the paper is organized as follows. Section II briefly describes the research context of the work presented in this paper and related works. Section III presents in detail our approach to reducing data communication overhead and our proposed heuristic-based algorithm for a specific application using profiling information. Section IV introduces the experiments of this work. Finally, Section V concludes the paper.

II. Related Work

In this section, we discuss the different interconnect techniques available in Section II-A, followed by the way these techniques are used at the system level in Section II-B.

A. Interconnect techniques

Point-to-point interconnect is considered as the simplest interconnect solution for a system-on-chip (SoC). In a *point-*

to-point interconnect architecture, the producer processing element (PE) is directly connected to the consumer PE. However, the biggest drawback of this architecture is the large number of wires required. This leads to difficulty in routing. Designs using this architecture are reported in [8], [9].

The bus architecture is a low cost interconnect for SoCs. The two standard and well-know bus architectures are AMBA developed by ARM [10] and CoreConnect developed by IBM [11]. Only CoreConnect has been adopted in Xilinx Virtex FPGA families. The main disadvantage of the bus architecture is the competition among to access the bus introducing arbitrary latencies. This competition potentially degrades the performance of the system.

The crossbar is a well-known architecture for providing a high-performance and minimum latency interconnect. The main drawback of a crossbar is its cost. An $n \times n$ crossbar can quickly become prohibitively expensive as its cost increases by n^2. To reduce the cost, many studies focusing on application-specific crossbars have been reported such as in [12], [13].

In recent years, many Network-on-Chip architectures for FPGA have been reported such as DyNoC [14], FLUX [15] and CuNoC [16]. For low-latency applications-specific NoCs, driven by task graph, ReNoC [5] and Skip-links [6] are used. Scalability is the main advantage of NoC. Moreover, NoCs are emerging as a high level interconnect solution ensuring parallelism and high performance. However, there are still several issues that need to be addressed such as latency, power consumption and especially high area cost.

B. System-level interconnect solutions

The *Molen* architecture [17], is a heterogeneous, shared memory multicore system for software/hardware co-design. The Molen architecture consists of two types of processing elements (PE): one *General Purpose Processor* (GPP) and one or more *Reconfigurable Processor(s)*, also so-called Custom Computing Unit(s) (CCUs). GPP has the main memory to contain application data while each CCU has each local memory (CCUMem) to contain its local data. The CCU exchanges parameters with GPP by exchange registers (CCUXreg) through an on-chip standard bus. While the GPP can access the main memory and the accelerator local memories, the accelerators can access only its local memory. The GPP and the accelerator local memories are also connected through an on-chip bus. When accelerator functions are needed, the GPP transfers data from the main memory to the local memory of the accelerator and copies the result back to the main memory.

The *MORPHEUS* architecture [18] has an ARM9 embedded RISC processor taking care for the control flow and synchronization, and three *heterogeneous reconfigurable engines* (HREs) for accelerating application kernels. The control infrastructure is done via an AMBA AHB bus which connects HREs and the ARM9 processor. The control flow is also performed via exchange registers, similar to the Molen architecture. A NoC is used to transfer data among HREs, main memory, off-chip memory. The data transfers via the NoC may be triggered by a Direct Network Access (DNA) hardware module. The MORPHEUS platform is implemented using STMicroelectronics CMOS090 technology. Although the platform shows very good simulation results, the NoC takes a huge resource toll up to 944Kgate.

A *Warp processor* [19] consists of a main general purpose processor, an *efficient on chip profiler*, *an on-chip computer aid design module* (CAD) and *a warp-oriented FPGA* (w-FPGA). The main processor executes the software part of an application while the critical software regions are synthesized and mapped onto the w-FPGA. The selection, synthesis and mapping the critical software kernels are done automatically by the profiler and the CAD module. The w-FPGA and the processor share the main data cache by using a mutually exclusive execution model. The main process, CAD module and the w-FPGA are connected together through an on-chip standard bus to configure the w-FGPA as well as to provide a mechanism for communication and synchronization between the main processor and the w-FPGA.

LegUp [20] is an open source high-level synthesis tool for FPGA-based processor/accelerators systems. The target system contains a processor connecting with custom hardware accelerators through a standard on-chip bus interface. The current version is implemented on the Altera Cyclone II FPGA with an Altera Avalon Bus as the interface for processor and accelerators communication. In this version, a shared memory architecture is used for exchanging variables between the processor and the accelerators. The shared memory uses an on-FPGA data cache and off-chip memory. The authors indicate that limitations of the bus system need to be further investigated.

III. Custom Interconnect and System Design

In this section, we introduce a heuristic-based algorithm to design an optimized application-specific custom interconnect. The heuristic-based algorithm uses data communication profiling information as a parameter to choose the most optimized interconnect solution.

A. Assumptions & definitions

In hardware accelerator systems such as Molen, MORPHEUS and LegUp, besides the main memory (on-chip memory or off-chip memory), each hardware accelerator has its own local memory to improve the parallelism. In our discussion, we assume that the memory hierarchy is as follows: 1) GPP can access the main memory as well as the local memories of hardware accelerators through a standard on-chip bus; and 2) Hardware accelerators can access their local memory only.

Before presenting the proposed custom interconnect design using quantitative data communication profiling, we need to define some equations used to estimate the quality of the solutions. The following vocabulary is used:

- **Hardware accelerator function**: A hardware accelerator function is defined by *Function*(H, D_i, D_o); where H is the execution time of the hardware accelerator only (without data communication overhead), D_i and D_o are the total amount of data input and output in bytes, respectively.
- **Data communication**: A communication between two functions is defined by $C_{ij}(F_i, F_j, D_{ij})$; where F_i and F_j are the producer and the consumer function, respectively, and D_{ij} is the total amount of data in bytes transferred from F_i to F_j. The functions F_i and F_j can be accelerated on hardware as well as run on the GPP.

- **The average time** taken by the GPP for transferring 1 byte from the main memory to a hardware accelerator local memory or vice versa is t_g, and the average time for transferring 1 byte from a hardware accelerator local memory to another one on the bus using direct memory access (DMA) is t_d. These values are platform dependent, however $t_d < t_g$.

The execution time of the hardware accelerator can be estimated using simulation tools such as ModelSim while the amount of data input (D_i) and output (D_o) as well as the amount of data communication (D_{ij}) can be generated using a data communication tool such as the QUAD tool [21].

To estimate the quality of solutions, we compare the solutions with a base system. In this base system, whenever a hardware accelerator is needed, the GPP copies all required data from the main memory to the local memory of the hardware accelerator and transfers the output result from the local memory to the main memory when the hardware accelerator is done. While the hardware accelerators are executed, the GPP is set into a waiting state.

B. Different interconnects

1) Crossbar-based shared local memory: The first interconnect we will use is a crossbar-based shared local memory in the case where two CCUs need to exchanges data. In this work, we use a crossbar for only two CCUs which communicate together to improve the parallelism of the CCUs as well as to reduce the area overhead. Figure 1a illustrates a simple system with the two hardware accelerators HW_1 and HW_2 sharing their local memories using a crossbar based on the Molen architecture. Figure 1b depicts the detailed structure of the crossbar for the Molen hardware accelerator functions.

The QUAD tool identifies functions communicating together and how much data is transferred between them exactly. Based on this information, we can choose which functions should share their local memories via a crossbar. The execution time can be computed as follows.

Consider two hardware accelerators $HW_1(H_1, D_{1i}, D_{1o})$ and $HW_2(H_2, D_{2i}, D_{2o})$ which communicate together with the data communication $C_{12}(HW_1, HW_2, D_{12})$. Following the base system model presented above, the total execution time of the two hardware accelerators is as follows:

$$T = H_1 + H_2 + (D_{1i} + D_{1o} + D_{2i} + D_{2o})t_g \qquad (1)$$

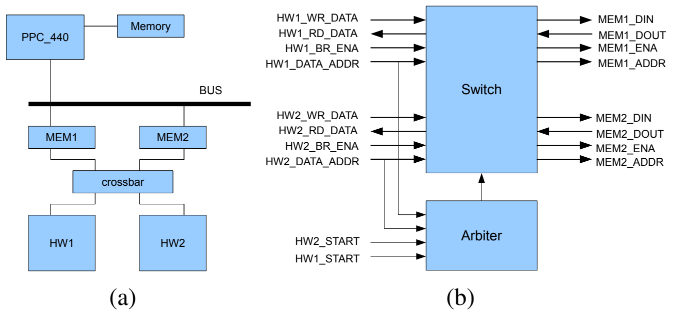

Fig. 1. (a) HW_1 and HW_2 share their memories using a crossbar; (b) Structure of the crossbar

Through the crossbar, each hardware accelerator, HW_1 or HW_2, can access not only its own local memory but also the local memory of the another one. Therefore, HW_1 can write part of its result (D_{12} bytes) which is used by HW_2 to the local memory of HW_2. Assume that the output of HW_1 is used by two different consumers: D_{12} bytes by HW_2, and at least $D_{1o} - D_{12}$ bytes by other software functions. Therefore, the GPP needs to transfer $D_{1o} - D_{12}$ bytes from the local memory of HW_1 to the main memory. However, rather than waiting for the execution of HW_2 to complete, the GPP can transfer this amount of data in parallel with the execution of HW_2. This reduces a lot of execution time in comparison with the base system model. Similarly, the GPP can move $D_{2i} - D_{12}$ bytes from the main memory to the local memory of HW_2 in parallel with the execution of HW_1. In this model, the total execution time of the two hardware accelerators is as follows.

$$T_{crossbar} = H_1 + H_2 + (D_{1i} + D_{2o})t_g \qquad (2)$$

With the crossbar, the total reduction time in comparison with the base system is $\Delta_c = T - T_{crossbar} = (D_{1o} + D_{2i})t_g$.

2) DMA support for parallel processing: For some applications (multimedia) data can be processed as streaming input. Using this concept, we can reduce the data communication time by segmenting the input data and running the hardware accelerator on each data segment independently. When the data input is segmented, hardware accelerators can be executed in parallel. However, when two accelerators communicate together through the crossbar, they cannot process in parallel due to the fact that a local memory port conflict may arise. One solution is to use DMA to transfer data directly from one local memory of a given hardware accelerator to another one via a bus.

Consider again the two hardware accelerators and the same communication that can execute in parallel on different segments, S_1 and S_2, of the input data. The processing flow following the pseudo code in Algorithm 1 can be applied to parallelize the data transfer from the main memory the the hardware accelerator local memory and the processing of the hardware accelerators.

Algorithm 1 Pipelining data communication

```
1: GPP copies $S_1$ from the main memory to $HW_1$ local memory;
2: $HW_1$ processes $S_1$ while GPP copies $S_2$ from the main memory to $HW_1$ local memory in parallel;
3: DMA transfers result of $S_1$ from $HW_1$ to $HW_2$ local memory;
4: $HW_1$ processes $S_2$ while $HW_2$ processes the first segment in parallel;
5: DMA transfers result of $S_2$ from $HW_1$ to $HW_2$ local memory;
6: $HW_2$ processes the second segment while GPP copies final result of the first segment from $HW_2$ local memory to the main memory in parallel;
7: GPP copies final result of the second segment from $HW_2$ local memory to the main memory;
```

With this processing model, the total execution time of the two hardware accelerators is as follows.

$$T_p = \frac{H_1}{2} + max(\frac{H_1}{2}, \frac{H_2}{2}) + \frac{H_2}{2} + (\frac{D_{1i}}{2} + \frac{D_{2o}}{2})t_g + D_{12}t_d + O \quad (3)$$

where O is the overhead for processing streaming input. The total reduction time in comparison with the base system is $\Delta_p = T - T_p = min(\frac{H_1}{2}, \frac{H_2}{2}) + (\frac{D_{1i}}{2} + \frac{D_{2o}}{2})t_g - D_{12}t_d - O$.

3) Local buffer: Consider the two hardware accelerators HW_1 and HW_2 as in the base system. They communicate together with $C_{12}(HW_1, HW_2, D_{12})$ as depicted in Figure 2. Assume that HW_1 is executed only one time while HW_2 is accelerated on hardware and iterated n ($n > 1$) times. HW_2 also communicates with itself with a communication $C_{22}(HW_2, HW_2, D_{22})$. Due to the iteration of HW_2 using the same data, the part of data input for this hardware accelerator produced by HW_1 (D_{12} bytes) should be kept locally, which eliminates the need to transfer data from the main memory $n-1$ times (we need to transfer for the first time). Therefore, only $D_{2i} - D_{12}$ bytes need to be transferred from the main memory to local memory of HW_2 in each iteration. The total time of the two hardware accelerators is as follows.

$$T_{dl} = H_1 + H_2 + (D_{2i} - (n-1)D_{12} + D_{2o} + D_{1i} + D_{1o})t_g \quad (4)$$

The total reduction time in comparison with the based system is $\Delta_{dl} = T - T_{dl} = D_{12}t_g(n-1)$.

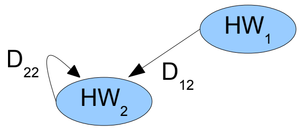

Fig. 2. Local buffer at HW_2

4) Hardware accelerator duplication: In Section III-B2, we introduced a way to parallelize the execution of the hardware accelerators. In case the data processed by a hardware accelerated function can be segmented and processed independently in parallel, we can duplicate the hardware accelerator twice to further reduce the execution time. When the hardware accelerator is duplicated twice, the data input of this accelerator is divided into two segments and each core processes each segment in parallel. A DMA is used to transfer the result of these hardware accelerators to others.

Consider a hardware accelerator $HW_1(H_1, D_{1i}, D_{1o})$, the execution time of the hardware accelerator in the non-duplication case and in the duplication case with DMA is as follows.

$$T = H_1 + (D_{1i} + D_{1o})t_g \quad (5)$$

$$T_{dp} = \frac{H_1}{2} + (D_{1i} + D_{1o} - D_{dma})t_g + D_{dma}t_d + O \quad (6)$$

where D_{dma} is the total amount of data transfer from the duplicated hardware accelerators to others using DMA and O is the overhead for parallel processing. The total reduction time in comparison with the base system is $\Delta_{dp} = T - T_{dp} = \frac{H_1}{2} + D_{dma}(t_g - t_D) - O$.

C. Heuristic-based algorithm

In the previous sections, we introduced different solutions to design a custom interconnect as well as a system using the quantitative data communication profiling. This section proposes a heuristic based algorithm to select the best and most suitable solution for each application. The pseudo code for the proposed algorithm is shown in Algorithm 2.

In this algorithm, at most five of the most computationally intensive functions suitable to implement on hardware are selected to accelerate on hardware. Currently, we choose only five functions as candidates for accelerating because our platform used to do experiments can support up to five hardware accelerators. Only the most computationally intensive hardware accelerator is considered for duplication. The QUAD profiling tool is used to identify the communication among the hardware accelerated functions. Based on this information, the algorithm examines the local buffer characteristic of the functions first. Then the interconnect solutions presented above are considered for each data communication between two hardware accelerators.

Algorithm 2 Data communication profiling-driven design

```
Input: Application source code
Output: The most optimized interconnect
1:  $L_{hw} \leftarrow$ List of at most five of the most computationally intensive functions suitable to implement on hardware;
2:  $G \leftarrow$ Quantitative data communication graph for functions in $L_{hw}$;
3:  $G \leftarrow$ Sort G in decreasing amount of data transfer order;
4:  Calculate $\Delta_{dp}$ as described in Section III-B4 for the most computationally intensive hardware accelerator;
5:  if $\Delta_{dp} > 0$ then
6:      Apply the solution in Section III-B4
7:  end if
8:  for each function in $L_{hw}$ do
9:      Check for iteration (Figure 2) and apply the Local buffer solution;
10: end for
11: for each data communication in $G$ do
12:     if the producer and the consumer can be executed in parallel then
13:         Calculate $\Delta_c$ and $\Delta_p$ as described in Section III-B1 and III-B2;
14:         if ($\Delta_p > \Delta_c$) then
15:             Apply the solution in Section III-B2;
16:         else
17:             Apply the solution in Section III-B1;
18:         end if
19:     else if The crossbar solution is applied to the producer or the consumer then
20:         Use DMA to transfer data from the producer to the consumer
21:     else
22:         Apply solution in Section III-B1;
23:     end if
24: end for
25: return A hardware accelerator system with the most optimized interconnect
```

IV. Experiments

A. Experimental setup

In this work, we use the Molen architecture as the experimental platform. The Molen system is implemented on the Xilinx ML510 [22] board which contain a xc5vfx130t-ff1738 FPGA device. The PowerPC is used as the GPP and the hardware accelerators are mapped onto the reconfigurable area. The main memory is the off-chip SDRAM connected to the PowerPC through a high performance IP core from Xilinx. The local memories of hardware accelerators are on-chip BRAMs. The PowerPC and the hardware accelerators are running at 400MHz and 100MHz, respectively. In our experiment, the Molen architecture can support up to five different hardware accelerators due to limited FPGA resource.

We use the *gprof* profiling tool [23] to identify which part of the application takes most of the execution time. Functions with high computational-intensity are good targets for acceleration. The *QUAD* toolset [21], which provides a comprehensive overview of the data communication behavior of an application, is used to generate the amount of data transfer between the functions of the application. The output of QUAD is a Quantitative Data Usage (QDU) graph in which the amount of data transfer among functions is shown.

B. Experimental results

We did experiments with 7 different well-known applications. Those are the Canny edge detection [24], the Susan edge detector [25] (with an implementation version of Oxford University), KLT feature tracker [26], Fluid simulation [27], the Blowfish application (a symmetric block cipher) from the CHStone benchmark [28], AES [29] and Bloom Filter [30]. The DWARV tool [31] is used to synthesize the VHDL code for the hardware accelerators from the original C code. As an example, Figure 3 and Figure 4 present the QDU graph and the final hardware accelerator system for Canny based on the Molen architecture and using our algorithm and solutions, respectively.

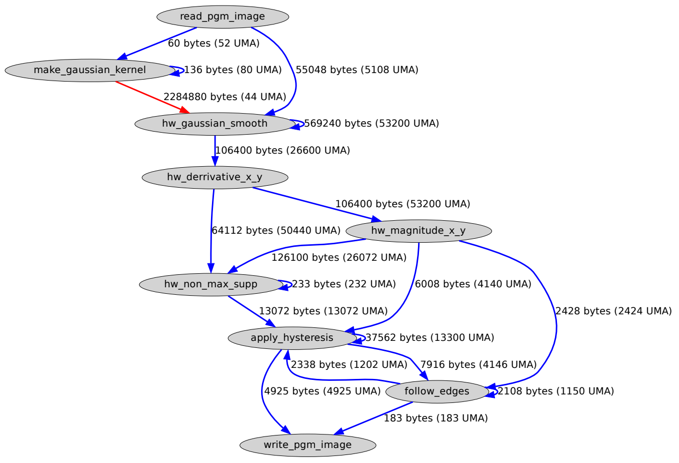

Fig. 3. QUAD graph for the Canny edge detection application

Table I shows our experimental results for the seven different applications. Column 2 in this table shows the custom interconnect techniques applied to each application. The crossbar technique is used for all applications in both the multimedia domain (the first four applications) and the cryptography domain (the last three applications). In the multimedia processing domain, the crossbar and the DMA techniques are frequently used. In the cryptography domain, only the crossbar and the local buffer techniques are used.

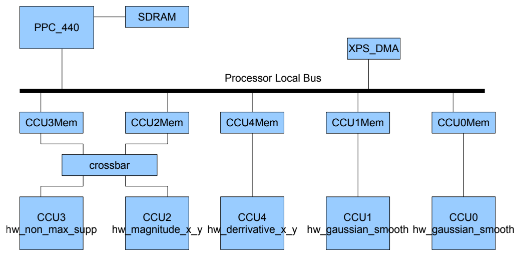

Fig. 4. Final system for Canny based on the Molen architecture and proposed solutions

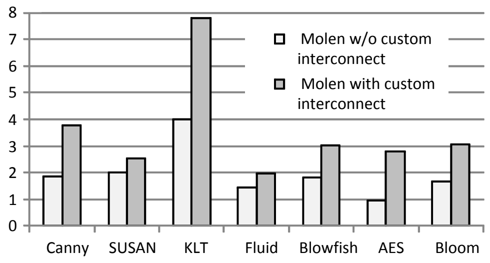

Fig. 5. Speed-up (w.r.t software) of hardware accelerators using Molen platform with and without using custom interconnect

Column 4 in this table shows the number of hardware accelerators used for each application. This column indicates that the multimedia processing applications have a tendency to use more hardware accelerator units, making them more suitable to accelerate on FPGAs compared to the cryptography applications. Column 5 shows the overall application speed-up of Molen architecture with custom interconnect in comparison with software. Column 6 and column 7 show the speed-up of hardware accelerators (with respect to software) with and without custom interconnect design.

As shown in Table I, the speed-up of hardware accelerators and the overall application go up to $7.8\times$ and $3.05\times$, respectively. Figure 5 shows the comparison of the speed-up of the Molen system with and without using our algorithm to choose the most optimized interconnect solutions. As shown in this figure, hardware accelerators which apply the communication profiling-driven acceleration solutions provide up to $2.98\times$ execution time improvement in comparison with the accelerators that do not apply these acceleration solutions.

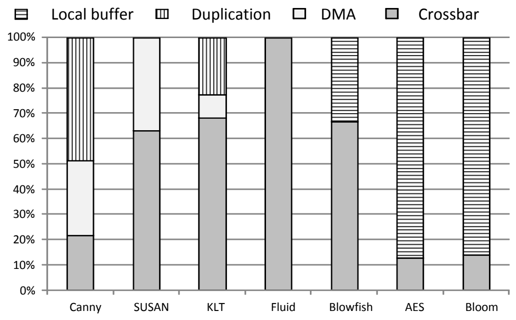

Fig. 6. The contribution of each solution to the speed-up

Figure 6 shows the contribution of each solution in the speed-up of each application in comparison with the standard Molen architecture. From the figure, the crossbar-based shared local memory always contributes to the speed-up of applica-

TABLE I
SPEED-UP, INTERCONNECT TECHNIQUES AND RESOURCE USAGE OF APPLICATIONS

Application	**HW techniques**	**# of LUTs**	**# of HW accelerators**	**Application Speed-up**	**accelerators speed-up w.r.t. SW**	
					with custom interconnect	w/o custom interconnect
Canny	Crossbar, Duplication, DMA	12026	5	3.05×	3.79×	1.85×
SUSAN	Crossbar, DMA	21504	3	2.51×	2.55×	2.02×
KLT Feature tracker	Crossbar, Duplication, DMA	6553	3	2.24×	7.8×	4.01×
Fluid simulation	Crossbar	12569	2	1.50×	1.95×	1.45×
Blowfish	Crossbar, Local buffer	16444	2	2.86×	3.02×	1.83×
AES	Crossbar, Local buffer	19544	2	1.41×	2.81×	0.94×
Bloom filter	Crossbar, Local buffer	1242	2	2.29×	3.05×	1.65×

tions. In the first four applications (Canny, Susan, KLT and Fluid), which belong to the multimedia processing domain, the different interconnects contribute in a different way to the overall speedup. The crossbar has a highest contribution in SUSAN, KLT and Fluid while the duplication gives the best performance in Canny. The DMA has a higher contribution to the speed-up when compared to the crossbar in Canny and when compared to the duplication in SUSAN. In the last three applications (which belong to the cryptography domain) the local buffer gives the best performance. The crossbar has a higher contribution while compared to the DMA and the duplication. Because the cryptography domain is very computation intensive on few data, it does not need a lot of communication. Therefore, a crossbar-based shared local memory accessible by all hardware accelerators will suffice.

V. CONCLUSIONS

In this paper, we presented a heuristic-based approach using a detailed data communication profiling to optimize an application-specific heterogeneous multicore system. The proposed approach mainly focuses on custom interconnect design. A heuristic-based algorithm is proposed to choose the most optimized interconnect solutions for each application. The algorithm uses data communication profiling information rather than task graphs as a guidance parameter because this information allows the designer to make better founded decisions regarding the most appropriate interconnect. Our experimental results show that we can gain speed-up of hardware accelerators up to 7.8× in comparison with software and to 2.98× in comparison with a base system without using our approach. We also considered the contribution of each interconnect solution to each application as well as to the application domain. Future research will investigate the possibility of tuning the interconnect at runtime. This runtime reconfigurability can be exploited evidently between applications but also within the execution of one application.

ACKNOWLEDGMENT

This research has been funded by the projects Smecy 100230, iFEST 100203, REFLECT 248976 and Vietnam Ministry of Education and Training.

REFERENCES

[1] B. Donchev, G. Kuzmanov, and G. N. Gaydadjiev, "External memory controller for Virtex II Pro," in *SoCs*, 2006, pp. 1–4.

[2] J. Becker, M. Hubner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka, "Dynamic and partial FPGA exploitation," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 438–452, 2007.

[3] Altera, "Accelerating Nios II systems with the C2H compiler tutorial," 2008.

[4] S. G. Kavadias, M. G. Katevenis, M. Zampetakis, and D. S. Nikolopoulos, "On-chip communication and synchronization mechanisms with cache-integrated network interfaces," in *Computing frontiers*, 2010, pp. 217–226.

[5] M. B. Stensgaard and J. Sparso, "ReNoC: A network-on-chip architecture with reconfigurable topology," in *NoCs*, 2008, pp. 55–64.

[6] C. Jackson and S. J. Hollis, "Skip-links: A dynamically reconfiguring topology for energy-efficient NoCs," in *SoC*, 2010, pp. 49–54.

[7] V. Papaefstathiou, D. Pnevmatikatos, M. Marazakis, G. Kalokairinos, A. Ioannou, M. Papamichael, S. Kavadias, G. Mihelogiannakis, and M. Katevenis, "Prototyping efficient interprocessor communication mechanisms," in *IC-SAMOS*, 2007, pp. 26–33.

[8] C. Dick, "Computing the discrete fourier transform on FPGA based systolic arrays," in *FPGA*, 1996.

[9] ARM Limited, "Multi-layer AHB overview," 2001.

[10] ——, "AMBA specification (rev 2.0)," 1999.

[11] IBM, "Coreconect bus architecture," 1999.

[12] J. Y. Hur, T. Stefanov, S. Wong, and S. Vassiliadis, "Systematic customization of on-chip crossbar interconnects," in *ARC*, 2007.

[13] S. Murali, L. Benini, and G. De Micheli, "An application-specific design methodology for on-chip crossbar generation," *Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 7, pp. 1283 –1296, 2007.

[14] C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. Fekete, and J. van der Veen, "DyNoC: A dynamic infrastructure for communication in dynamically reconfugurable devices," in *FPL*, 2005, pp. 153–158.

[15] S. Vassiliadis and I. Sourdis, "FLUX interconnection networks on demand," *J. Syst. Archit.*, vol. 53, no. 10, pp. 777–793, Oct. 2007.

[16] S. Jovanovic, C. Tanougast, S. Weber, and C. Bobda, "CuNoC: A scalable dynamic NoC for dynamically reconfigurable FPGAs," in *FPL*, 2007, pp. 753–756.

[17] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, 2004.

[18] M. Kuhnle, M. Hubner, J. Becker, A. Coppola, L. Pieralisi, R. Locatelli, G. Maruccia, T. DeMarco, F. Campi, A. Deledda, C. Mucci, and F. Ries, "An interconnect strategy for a heterogeneous, reconfigurable SoC," *Design Test of Computers, IEEE*, vol. 25, no. 5, pp. 442 –451, 2008.

[19] R. Lysecky and F. Vahid, "Design and implementation of a microblaze-based warp processor," *Embed. Comput. Syst.*, vol. 8, pp. 1–22, 2009.

[20] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *FPGA*, 2011, pp. 33–36.

[21] S. A. Ostadzadeh, M. Corina, C. Galuzzi, and K. Bertels, "tQUAD - memory bandwidth usage analysis," in *International Conference on Parallel Processing*, 2010, pp. 217–226.

[22] Xilinx, "Ml510 reference design," June 23 2009.

[23] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, 1982.

[24] J. Canny, "A computational approach to edge detection," *Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679 –698, 1986.

[25] S. M. Smith, "Susan low level image processing," 1992.

[26] J. Shi and C. Tomasi, "Good Features to Track," in *CVPR*, 1994.

[27] J. Stam, "Real-time fluid dynamics for games," in *Game Developer Conference*, 2003.

[28] S. H. Y. Hara, H. Tomiyama and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-Based high-level synthesis," *JIP*, vol. 17, pp. 242–254, Oct. 2009.

[29] P. SSL, "Polarssl library," 2012. [Online]. Available: http://polarssl.org/

[30] K. Christensen, "Christensen tool page - department of computer science and engineering - university of south florida," 2012. [Online]. Available: http://www.csee.usf.edu/ christen/tools/toolpage.html#bloom

[31] R. Nane, V. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels, "DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler," in *FPL*, 2012.