Communication-Aware HW/SW Co-design for Heterogeneous Multicore Platforms

Imran Ashraf, S. Arash Ostadzadeh, Roel Meeuws, Koen Bertels
Computer Engineering Lab, TU Delft, The Netherlands
{I.Ashraf,S.A.Ostadzadeh,R.J.Meeuws,K.L.M.Bertels}@TUDelft.nl

ABSTRACT

$QUAD$ is an open source profiling toolset, which is an integral part of the Q^2 profiling framework. In this paper, we extend $QUAD$ to introduce the concept of Unique Data Values regarding the data communication among functions. This feature is important to make a proper partitioning of the application. Mapping a well-known *feature tracker* application onto the multicore heterogeneous platform at hand is presented as a case study to substantiate the usefulness of the added feature. Experimental results show a speedup of 2.24× by utilizing the new $QUAD$ toolset.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Program analysis; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Performance, Measurement

Keywords

Multicore Heterogeneous Computing, Data Driven Application Mapping, Dynamic Analysis, HW/SW Co-design

1. INTRODUCTION

Heterogeneous multicore platforms are becoming the dominant architecture both for High Performance Computing as well as for Embedded Systems. Examples are the Convey supercomputer, Zynq by Xilinx and C66 by Texas Instruments. The heterogeneous components can be DSPs, FPGAs or GPUs. The main consequence of this trend is that developers can no longer develop their applications in a processor agnostic way and need to have appropriate tools highlighting the architectural constraints.

One well-known design constraint deals with the memory access and data transfer between the computing cores. Communication bottlenecks can kill the anticipated performance improvement. We extend the $QUAD$ [1] open source toolset that provides a detailed insight in how the data flows inside the application and what memory access patterns result out of it. The information generated by $QUAD$ can be used to support HW/SW co-design decisions.

In this paper, we focus particularly on a heterogeneous platform containing FPGA based kernels and demonstrate the usefulness of the information provided by $QUAD$ for making well founded design choices. The overall goal of $QUAD$ is to reduce the communcation overhead. We introduce the concept of Unique Data Values (UnDVs) to $QUAD$ that quantifies the uniqueness of data values and how often they are read/written. The main contributions of this paper can be summarized as follows:

- the introduction of a metric that quantifies unique data communication;
- the extension of the open source $QUAD$ toolset to compute this metric to provide the guidelines for HW/SW Co-design;
- the application of these guidelines on a real feature tracking application showing a speedup of 2.24× when mapped onto the reconfigurable platform at hand.

The paper is organized as follows. Section 2 provides the research context of our work. Section 3 introduces $QUAD$, with emphasis on the extension and the design choices which can be made based on this extension. Section 4 discusses the application of our work in a case study and presents the obtained experimental results. Section 5 provides the related work followed by Section 6 which concludes the paper.

2. RESEARCH CONTEXT

The work presented in this paper, although not restricted to any specific architecture, has been developed in the context of the *Molen* [2] polymorphic processor. The *Molen* architecture is based on the shared memory, processor, co-processor architectural paradigm [3]. It couples a General Purpose Processor (GPP) and one or more Custom Computing Units (CCUs). Each CCU has its own set of registers and local memory for processing. The GPP controls the execution and (re)configuration of the CCUs.

The Delft Workbench (DWB) [4] is a semi-automatic tool platform for integrated HW/SW co-design, targeting heterogeneous computing systems containing reconfigurable components. DWB addresses the entire design cycle from profiling [1, 5] and partitioning [6] to synthesis [7] and compilation [8] of an application. $QUAD$ is an integral part of Q^2

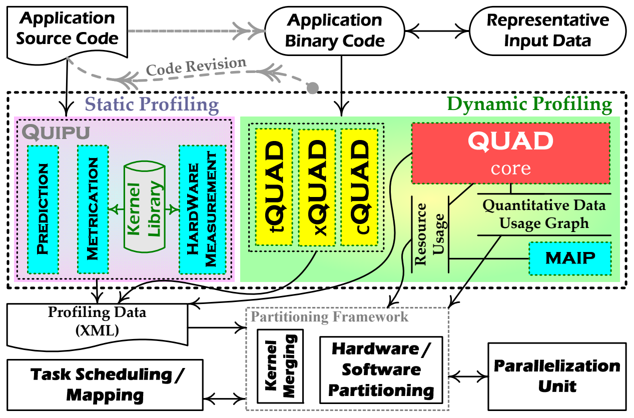


Figure 1: The Q^2 profiling framework within the DWB.

profiling framework (see Figure 1) which provides a comprehensive overview of the memory access behavior of an application. The *QUAD* toolset is based on PIN *D*ynamic *B*inary *I*nstrumentation (DBI) framework [9].

3. DATA COMMUNICATION QUANTIFICATION

QUAD traces each memory read and write access to record necessary information regarding the data communication among functions. The function writing to a memory location is called the producer and the one reading from it is known as the consumer of this data. The *QUAD* core tool reports the amount of data communication between a producer and a consumer in Bytes. Memory adresses are also analyzed to calculate the number of unique locations used in data communication. This information is reported at the end as Unique Memory Addresses (UnMAs).

3.1 Unique Data Values

In an application, there exist several production/consumption patterns among functions. As an example, consider an array of 100 bytes initialized by a function $f1$, which is then read 50 times by a function $f2$. The amount of data communication reported between $f1$ and $f2$ will be 5000 bytes. On the initial read, new values are read by $f2$, whereas successive reads involve the same old data. So, in this case, the unique data values communicated between $f1$ and $f2$ are 100.

In the context of HW/SW co-design, if $f1$ is mapped onto CCU1 and $f2$ is mapped onto CCU2, then the data communication between the two CCUs can be reduced by making a local copy of this array after its first transfer, reducing the communication from 5000 bytes to 100 bytes. In order to realize how much data communication is unique, the concept of a new metric known as UnDVs is introduced to *QUAD*.

3.2 Communication Classes & Design Choices

With the current extension, Bytes, UnMAs and UnDVs are the three metrics measured by *QUAD* for each data communication binding among the functions in an application. For the sake of simplicity, if we classify each quantity as having high and low values, there can be eight (2^3) possible combinations. For instance, high Bytes, low UnMAs and low UnDVs is one such combination.

However, the combinations where Bytes < UnMAs and Bytes < UnDVs are not possible. This is because the address of each byte recorded in the communication will be counted at most once in UnMAs, which implies that $Bytes \geq UnMAs$. Similarly, each byte will be counted at least once in UnDVs as it will be a unique value when accessed for the first time, hence, $Bytes \geq UnDVs$. This eliminates 5 combinations, leaving us with only three combinations of Bytes, UnMAs and UnDVs as described below.

- *Case 1- High Bytes, Low UnMAs and Low UnDVs:* In this case, there is intensive communication going on between the producer and the consumer on lower number of addresses. Low UnDVs in this case suggests that not most of the data being communicated is new and it is very much like reading some constant values or some initialized data by the producer, over and over again without data being modified. An example is an array of coefficients which is initialized once in the producer and same values of array are read by the consumer multiple times.
Design choices: A boundary condition of this can be when UnMAs = UnDVs, in which case the data on all UnMAs is same for multiple reads. This means that the data can be placed locally on the HW. In other cases, we need to place the producer and the consumer close to each other and if possible have them share some local memory. A small local buffer close to the consumer, containing the data is another design option.

- *Case 2- High Bytes, Low UnMAs and High UnDVs:* In this case of high communication between the producer and the consumer, less number of addresses are involved. High UnDVs point out that this data is written again and again by the producer and also the consumer is reading this data in an interleaved manner. This means that mostly unique data is read by the consumer resulting in high UnDVs. An example of this communication is an array which is modified, for instance, 10 times by the producer. After each modification, the consumer reads this array multiple times.
Design choices: A boundary condition in this case can be when Bytes = UnDVs, in case unique data is read each time by consumer. Mapping the consumer alone onto a CCU will cost high communication overhead, as the consumer has to get data from the producer to process. The design choice which can be made based on such an observation, is to merge the producer with the consumer and map this new merged function onto the HW, resulting in the reduction of communication overhead.

- *Case 3- High Bytes, High UnMAs and High UnDVs:* This case involves a large amount of data being communicated between the producer and the consumer. A large number of addresses are involved, for instance, a large image data which is first processed by the the producer and later processed by the consumer.
Design choices: The design decision which can be made based on this information is to merge the two functions together like the previous case. However,

this merging may not be feasible because of high memory requirements. Another solution can be to transfer the data in parallel to the computation, for instance, by the double buffering technique.

3.3 Implementation Details

In this section, we provide the implementation details of our extension to *QUAD*. In order to find out the unique data values, corresponding flags and counters are associated with each location. When a value is written to a location, flags associated with the location are set. Later, when the location is read, associated UnDV counter is incremented and the flag is cleared. Successive reads from the same location, without the values being re-written, notice the cleared flags and the UnDV counter is not incremented. A simplified logic of this implementation[1] is shown in Algorithm 1.

Algorithm 1 Pseudo-code for UnDVs in *QUAD*.

```
if location is seen for first time then
    $location \cdot Flags = newFlags()$
end if
if write then
    $location \cdot SetFlags()$
else
    if $location \cdot IsNew(consumer)$ then
        $location \cdot IncrementUnDVs()$
        $location \cdot ClearFlag(consumer)$
    end if
end if
```

We have implemented this feature as a dynamic array with an initial size picked as 5 elements to reduce the execution time overhead caused by the memory allocations for each element individually. The number was decided based on empirical results. The size of the array is increased by 5 in case more consumers are found for a location. Although, a penalty will be caused in this case, but, mostly there are no more than 5 consumers of any location. Furthermore, this fixed size may be passed as an argument by the user to the *QUAD*, which makes it customizable for each application.

4. CASE STUDY: KLT

In this section, we present a detailed discussion of a use case involving Kanade-Lucas-Tomasi Feature Tracker (KLT) application [10]. This application detects interesting features in a frame and tracks those features in the subsequent frames. We have used version 1.3.4, which is the latest version of KLT [11]. This *C* implementation has 102 functions in 17 source files. The focus of this case study is on the utilization of information provided by *QUAD* to map the application onto the *Molen* heterogeneous reconfigurable platform.

4.1 Experimental Setup

All the experiments were performed on two different platforms. The general profiling of the KLT application with *gprof* was done on an Intel 32-bit Core2 Duo E8500 @3.16GHz with 4GB of RAM, running the Linux kernel v2.6.34.10-0.6-pae. The application source code was compiled with *gcc* v4.5.0 with level two optimizations and without function inlining. The target platform is the *Molen* heterogeneous reconfigurable platform on Xilinx ML510, Virtex5 FX 130T with 2 MB BRAM FPGA board. A PowerPC 440 @400 MHz with 512 MB DRAM, is used as a GPP, and CCUs are implemented as HW modules on FPGA. 30 K slices are available for (re)configuration and there can be a maximum of 5 RUs on the FPGA, where each CCU has 64 KB of local Memory. A number of design choices can be made in mapping applications onto *Molen*, which are guided by the information provided by the Q^2profiling framework.

In order to profile the application using the *QUAD* toolset, the Pin Dynamic Binary Instrumentation (DBI) framework is needed which does not support the PowerPC architecture. As a result, the *QUAD* profiling information on Intel x86 can be biased. However, the overall behavior of the application regarding the data communication should stay similar. We have used the DWARV C-to-VHDL compiler [7] to generate the VHDL code for the reconfigurable part. Simulations were performed using *Modelsim 6.5f.* We have used Xilinx ISE 13.2 synthesis tools targeting the same Virtex5 FPGA containing a *Molen* machine implementation. The executable code for the PowerPC in the form of Executable Link Format (ELF) and the synthesized hardware modules in the form of *bitstream* files, were then used to run the application on the *Molen* platform.

Table 1: *gprof* flat profile for the *KLT* application on the Intel x86 architecture.

Kernel	%time	self sec	calls	self ms/call	total ms/call
_interpolate	48.5	0.97	26.26M	0.00	0.00
_convolveImageHoriz	16.0	0.32	183	1.75	1.75
_convolveImageVert	16.0	0.32	183	1.75	1.75
_KLTSelectGoodFeat.	6.0	0.12	1	120.0	141.14
_computeGradientSum	5.0	0.10	17249	0.01	0.04
_computeIntensityDiff.	2.5	0.05	23871	0.00	4.02

4.2 Mapping Steps

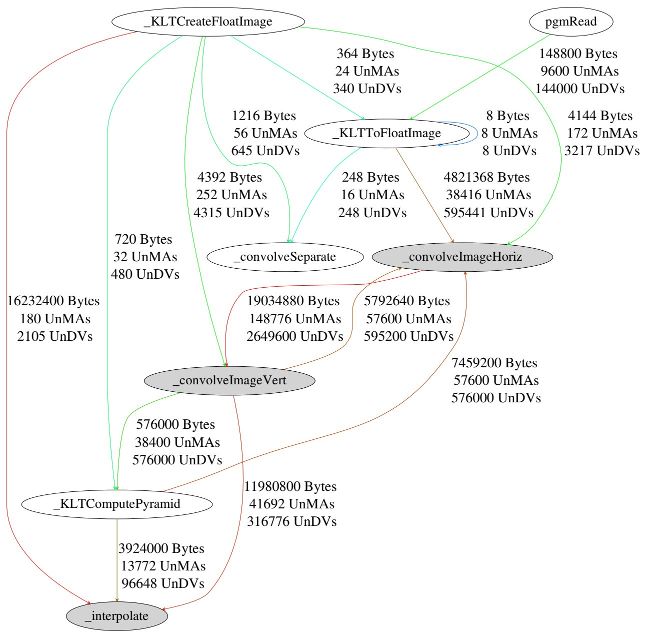
Table 1 shows the flat profile of the KLT application generated by *gprof* on the Intel x86 architecture. For this run, 30 frames have been used for feature tracking. The frame size has been chosen as 80 × 60 to be able to satisfy the memory requirement of the platform. It can be seen from this profile that we can map the top three kernels, namely *_interpolate*, *_convolveImageHoriz* and *_convolveImageVert* on each of CCUs in the platform. The combined execution time of these three kernels is 0.805 p.u (80.5%). Using Amdahl's law, the theoretical application speedup, assuming an unlimited speedup for the kernel(s) in question, can be calculated as follows:

$$\lim_{p\to\infty} \frac{p}{1 - f(p-1)} = \frac{1}{f} = \frac{1}{1-s} = \frac{1}{1-0.805} = 5.13, \quad (1)$$

where p is the speedup factor of the accelerated part, f is the percentage contribution of the sequential part, and s is the original percentual contribution of the accelerated part.

The mapping of the aforementioned kernels will result in performance improvements, but this performance can be improved by reducing the communication among all the computing elements. *gprof* and other traditional profilers do not provide information about the data communication in an application. In simple applications, it may be easy to analyze communication among various functions. However, in a complex application as in this case study (102 functions), it can be a tedious and time consuming task to manually un-

[1]Source on: http://sourceforge.net/projects/quadtoolset



_KLTCreateFloatImage
pgmRead
364 Bytes 24 UnMAs 340 UnDVs
148800 Bytes 9600 UnMAs 144000 UnDVs
1216 Bytes 56 UnMAs 645 UnDVs
_KLTToFloatImage
8 Bytes 8 UnMAs 8 UnDVs
4144 Bytes 172 UnMAs 3217 UnDVs
4392 Bytes 252 UnMAs 4315 UnDVs
248 Bytes 16 UnMAs 248 UnDVs
4821368 Bytes 38416 UnMAs 595441 UnDVs
720 Bytes 32 UnMAs 480 UnDVs
_convolveSeparate
_convolveImageHoriz
16232400 Bytes 180 UnMAs 2105 UnDVs
19034880 Bytes 148776 UnMAs 2649600 UnDVs
5792640 Bytes 57600 UnMAs 595200 UnDVs
7459200 Bytes 57600 UnMAs 576000 UnDVs
_convolveImageVert
576000 Bytes 38400 UnMAs 576000 UnDVs
_KLTComputePyramid
11980800 Bytes 41692 UnMAs 316776 UnDVs
3924000 Bytes 13772 UnMAs 96648 UnDVs
_interpolate

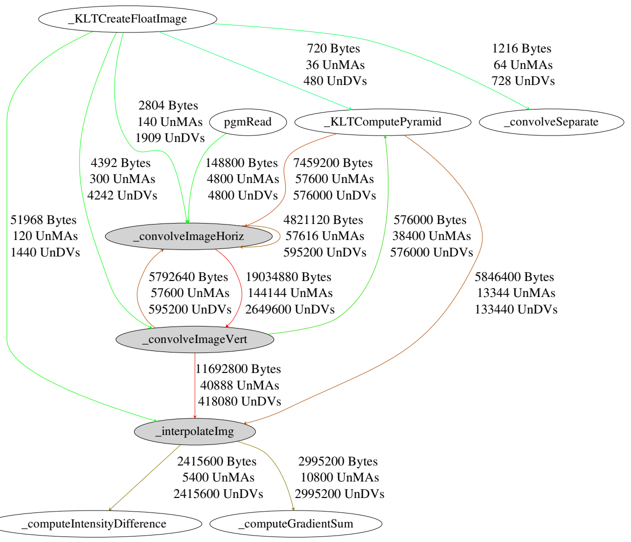
Figure 2: QDU Graph of the original KLT application.

derstand the intensity of the communication, the addresses, and the amount of unique data involved in each communication. In short, an automatic tool like *QUAD* can provide this information and guide us in the mapping process based on the communication of functions with the top contributing kernels. We describe the mapping process below.

Step 1: In the original application, *gprof* shows that the *self* contribution of *_interpolate* per call is quite low (0.00 in Table 1). Mapping *_interpolate* as it is onto a CCU will result in performance degradation, due to the large number of calls (about 24.2M) to this CCU. This is because the overhead of each call to CCU will be more than the execution time of this function. So, we modified *_interpolate* to process a complete frame per call, resulting in reduced number of total calls.

Step 2: The complete Quantitative Data Usage (QDU) graph [1] is quite complex due to the large number of functions in this application. So, a reduced QDU graph of these top contributing kernels (dark grey ovals) and the functions communicating with these kernels is shown in Figure 2. The amount of data communication is shown in bytes. Furthermore, the intensity of the communication is indicated by the colour of the links in the descending order of red, brown, dark green, green and blue.

Figure 2 depicts that the image is read by *pgmread* and this image data is fed to *_KLTToFloatImage*. The size of 30 frames equals 30x80x60 = 144000, which is accurately indicated by the number of UnDVs for this communication link. The number of bytes is 4800 bytes more than the number of UnDVs, which indicates that 11 images are read. When we look into the code, it becomes clear that the first image is read twice, initially for the detection of features and later for their tracking. UnMAs involved in this communication are 9600. This is easily verified as it relates to the size of 2 images (2x80x60). As an initial improvement, based on the information provided by *QUAD*, we modified the code to not read the first image twice.



_KLTCreateFloatImage
720 Bytes 36 UnMAs 480 UnDVs
1216 Bytes 64 UnMAs 728 UnDVs
2804 Bytes 140 UnMAs 1909 UnDVs
pgmRead
_KLTComputePyramid
_convolveSeparate
4392 Bytes 300 UnMAs 4242 UnDVs
148800 Bytes 4800 UnMAs 4800 UnDVs
7459200 Bytes 57600 UnMAs 576000 UnDVs
51968 Bytes 120 UnMAs 1440 UnDVs
_convolveImageHoriz
4821120 Bytes 57616 UnMAs 595200 UnDVs
576000 Bytes 38400 UnMAs 576000 UnDVs
5792640 Bytes 57600 UnMAs 595200 UnDVs
19034880 Bytes 144144 UnMAs 2649600 UnDVs
5846400 Bytes 13344 UnMAs 133440 UnDVs
_convolveImageVert
11692800 Bytes 40888 UnMAs 418080 UnDVs
_interpolateImg
2415600 Bytes 5400 UnMAs 2415600 UnDVs
2995200 Bytes 10800 UnMAs 2995200 UnDVs
_computeIntensityDifference
_computeGradientSum

Figure 3: QDU graph of modified KLT application.

An interesting point to note here is that, the bytes transferred by *_KLTToFloatImage* to *_convolveImageHoriz* are roughly 4 times higher than the image data input to *_KLTToFloatImage*, which shows some kind of expansion being performed here. If we look into the code, it becomes clear that the image data is converted from *char* to *float* data type. As *_convolveImageHoriz* is one of the top contributing kernels, to further reduce the communication, we can transfer the *char* data to *_convolveImageHoriz* and cast it inside this function to reduce the external communication. Figure 3 shows the QDU graph after this modification. It can be seen that the image data is now fed directly to *_convolveImageHoriz*, hence, reducing the communication from 4.82 MB to 1.44 MB. Another interesting observation is that *_KLTToFloatImage* completely disappeared after this modification, as this was the only place where it was being used.

Step 3: Figure 3 shows that *_convolveImageHoriz* and *_convolveImageVert* are also communicating heavily with each other, indicated by the high value of reported Bytes. High UnDVs indicate that a high amount of unique data is involved in this mutual communication, which is similar to Case 2, Section 3.2. If we merge these two functions together as a single *_convolveImage* function, this communication will be performed locally. Figure 4 shows the QDU graph with the merged *_convolveImage*, resulting in a reduction of about 24.8 MB of external communication.

Step 4: According to Figure 3, *_KLTComputePyramid* is communicating heavily with *_convolveImage*. In this case, we cannot merge *_KLTComputePyramid* together with *_convolveImage*. The resulting UnMAs indicate that the memory requirements for the merged kernel will be increased by 57600 (56.25 KB), whereas, the maximum memory available for a CCU is 64 KB. Secondly, *_interpolateImg* is also communicating heavily with *_computeGradientSum* as well as with *_computeIntensityDifference*. *_computeGradientSum* is consuming 2995200 bytes of data produced by *_interpolateImg*, and all of this data is unique as indicated by the

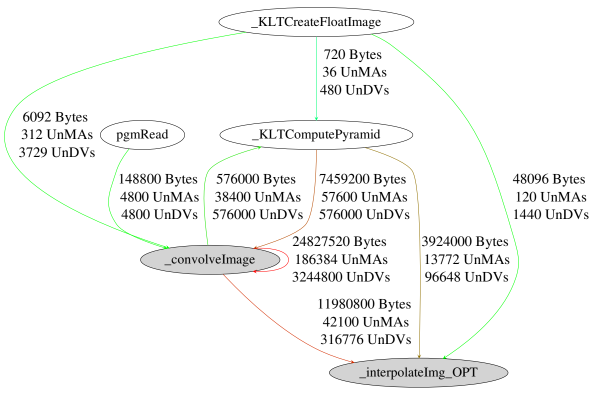


Figure 4: QDU Graph of KLT with optimized interpolate.

fact that it is equal to UnDVs. Similarly, *_computeIntensityDifference* is also consuming 2415600 bytes of data produced by *_interpolateImg*, and again equal number of UnDVs suggests that all of these bytes are unique as discussed in Case 3, Section 3.2. The UnMAs required for these two communications are lower (maximum 10 KB in case of *_computeIntensityDifference*) compared to the above situation. So these numbers suggest that we can merge *_computeIntensityDifference* and *_computeGradientSum* with *_interpolateImg* to reduce both of these communications. In fact, when we look at the code, we can see that at one time two complete interpolated images are produced by *_interpolateImg*. All the pixel values in these two interpolated images are then added or subtracted in *_computeGradientSum* and *_computeIntensityDifference*, respectively. Therefore, we can completely eliminate this intermediate communication to the memory, by calculating the interpolated pixel for each of the two images and adding/subtracting them on the fly and save only the last single output image instead of keeping multiple intermediate interpolated images.

Figure 4 shows the result of merging the *_computeIntensityDifference* and the *_computeGradientSum* functions into the *_interpolateImg* which results in the elimination of 2.99 MB and 2.41 MB of data communication, respectively. The *_interpolateImg* is also consuming 48096 bytes of data produced by *_KLTCreateFloatImage*, but the lower number of UnDVs suggests that this is some kind of constant data, as discussed in Case 1, Section 3.2. When we look into the code, we see that these correspond to the number of *rows* and *cols* of frames produced at the time of creation of the frame, and later they are read multiple times without being re-written. Finally, Figure 4 shows that we have two modified kernels namely *_convolveImage* and *_interpolateImg*, which can be mapped onto two CCUs.

4.3 Experimental Results

Table 2 shows the experimental results of the intermediate steps performed during the process of mapping the KLT application onto the *Molen* platform. The third column contains the name of kernels under discussion in the corresponding step, as shown in QDU graphs in Figures 2-4.

The first row is the original software implementation which is provided here for comparison. It does not involve a HW implementation, hence, mentioned Not Applicable (**NA**) in HW execution times. The second row is the HW implementation based on the *gprof* information, giving a total speedup of 1.71. The third row corresponds to Step 2 in Section 4, where *_KLTToFloatImage* was merged with the *_convolveImageHoriz* to reduce the data communication. It can be seen that we have achieved a speedup of 1.73 by this communcation reduction. Furthermore, compiler was also able to optimize the code efficiently when most of the functionality was placed in a single function.

The fourth entry corresponds to Step 3 where we achieved an overall speedup of 1.79, by using merged *_convolveImage* and reducing external expensive communication. Row 5 corresponds to Step 4 in Section 4. The overall speedup obtained is 2.24 which is considerably higher because of three factors. At first, communication was completely eliminated in this case, instead of making it local as in Step 2. Secondly, *_interpolateImg* has an overall contribution of 50% in the application as shown in Table 1. This contribution is further increased because we have moved the functionality of *_computeGradientSum* and *_computeIntensityDifference* to this function, resulting in about 56% of the total contribution. Therefore, improving this kernel resulted in higher speedup. Furthermore, the additional computations contained in *_interpolateImg_OPT* were mostly independent, resulting in higher kernel speedup, and hence, a higher overall speedup.

For this application, we have measured overhead increases of 1.6% and 26.4% for the execution time and memory usage respectively, when the UnDV concept was added to *QUAD*.

5. RELATED WORK

HW/SW partitioning has been an active field of research in the last decade. Many approaches have been proposed to address the problem in diverse ways. Generally, the process can be carried out at various granularity levels, ranging from fine-grained basic blocks or loops [12, 13] to coarse-grained functions [14, 15]. Apart from the traditional partitioning methods, different heuristic and evolutionary approaches have also been investigated to address this problem [16]. Work on compiler-directed method for program parallelization by exploiting fine-grain instruction level parallelism is discussed in [17]. However, this approach is not scalable as is also discussed by the authors in their work.

Our partitioning methodology is similar to the one presented in [15], which supports the partitioning of an application between several processing elements (SW/SW partitioning) at the function-level, as well as HW/SW partitioning utilizing some profiling information. However, in [15], as in most other approaches, partitioning is performed based on the call graph, whereas we utilize the QDU graph as the main reference. The quantitative information about the data communication between functions in an application is extracted automatically by our advanced profiling toolset. In this way, complex data-flows between functions can be made clear, enabling developers to find better partitions compared to the ones obtained using only the call graph and general execution time profiling data. In [18], *QUAD* has been used to map three applications onto molen platform, however, no report on UnDVs requires the user to perform extra manual analysis. In this work, we extended *QUAD* to automatically provide this information.

Table 2: Results of various intermediate implementation steps performed in mapping.

Entry	Implementation	Kernel	SW Time(μsec)		HWTime (μsec)		Speedup	
			Kernel	Application	Kernel	Application	Kernel	Application
1	Original SW	_interpolate	4.75	12154566	NA	NA	NA	NA
2	Original HW	_interpolateImg	1310	11786160	831	7110129	1.58	1.71
		_convolveImageHoriz	16132		4007		4.03	
		_convolveImageVert	16491		4016		4.11	
3	Modified HW 1	_convolveImageHoriz	16689	11654476	4013	7015287	4.16	1.73
4	Modified HW 2	ConvolveImg	32125	11154476	6683	6797151	4.81	1.79
5	Modified HW 3	intrpolate_OPT	2747	11025172	957	5429859	2.87	2.24

6. CONCLUSIONS

In this paper, we presented an important extension to the *QUAD* toolset which aims to quantify the unique data values in communication among functions of an application. We demonstrated how this information can be used to make better partitioning and mapping decisions while taking into account the hardware constraints. We used *Molen* as the the target heterogeneous platform which consists of a general purpose processor and one or more application specific hardware kernels. Using the *QUAD* extension, we are able to eliminate power- and time-consuming shared memory accesses and change the code such the that information which is produced locally can also be consumed locally. We managed to obtain a 2.24× speedup. In the future work, we are planning to develp a tool which can use the *QUAD* quantitative information to automatically perform this communication aware HW/SW partitioning.

7. ACKNOWLEDGMENTS

This research is partially supported by the Artemisia iFEST project (grant 100203), the Artemisia SMECY project (grant 100230), the FP7 Reflect project (grant 248976) and Higher Education Commission (HEC) Pakistan.

8. REFERENCES

[1] S. A. Ostadzadeh et al. QUAD - A Memory Access Pattern Analyser. In *ARC 2010*, pages 269–281, 2010.

[2] S. Vassiliadis et al. The MOLEN Polymorphic Processor. *IEEE Transactions on Computers*, 53(11):1363–1375, 2004.

[3] S. Vassiliadis et al. The Molen Programming Paradigm. In A. Pimentel and S. Vassiliadis, editors, *Computer Systems: Architectures, Modeling, and Simulation*, volume 3133 of *LNCS*, pages 1–10. Springer Berlin / Heidelberg, 2004.

[4] K. Bertels et al. Developing Applications for Polymorphic Processors: The Delft WorkBench. Technical report, Delft University of Technology, January 2006.

[5] R. J. Meeuws et al. High level quantitative interconnect estimation for early design space exploration. In *ICFPT '08*, pages 317–320, 2008.

[6] S. A. Ostadzadeh et al. A Clustering Framework for Task Partitioning Based on Function-level Data Usage Analysis. In *FPGA '09*, pages 279–279, 2009.

[7] Y. D. Yankova et al. DWARV: Delft Workbench Automated Reconfigurable VHDL Generator. In *In Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07)*, pages 697–701, August 2007.

[8] E. M. Panainte et al. The Molen Compiler for Reconfigurable Processors. *ACM Trans. Embed. Comput. Syst.*, 6(1), 2007.

[9] C. Luk et al. Pin: Building Customized Program Analysis Tools with Dynamic Instr. In *PLDI '05*, pages 190–200, New York, USA, 2005. ACM.

[10] B. D. Lucas and T. Kanade. An Iterative Image Registration Technique with an Application to Stereo Vision. pages 674–679, 1981.

[11] KLT: An Implementation of the Kanade-Lucas-Tomasi Feature Tracker. `http://www.ces.clemson.edu/~stb/klt/installation.html`.

[12] Y. Li et al. Hardware-software Co-design of Embedded Reconfigurable Architectures. DAC '00, pages 507–512, 2000.

[13] M. Baleani et al. HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform. CODES '02, pages 151–156, 2002.

[14] M. Santambrogio et al. A Novel SoC Design Methodology Combining Adaptive Software and Reconfigurable Hardware. In *ICCAD 2007*, pages 303–308, November 2007.

[15] D. Gohringer et al. A Design Methodology for Application Partitioning and Architecture Development of Reconfigurable Multiprocessor Systems-on-Chip. *FCCM'10*, pages 259–262, 2010.

[16] G. Wang, W. Gong, and R. Kastner. Application Partitioning on Programmable Platforms Using the Ant Colony Optimization. *Journal of Embedded Computing*, 2(1):119–136, 2006.

[17] M. Chu, R. Ravindran, and S. Mahlke. Data Access Partitioning for Fine-grain Parallelism on Multicore Architectures. In MICRO 40, pages 369–380, Washington, DC, USA, 2007. IEEE Computer Society.

[18] S. A. Ostadzadeh, R. J. Meeuws, I. Ashraf, C. Galuzzi, and K. Bertels. The Q^2 Profiling Framework: Driving Application Mapping for Heterogeneous Reconfigurable Platforms. In *Proceedings of the 8th International Symposium on Applied Reconfigurable Computing (ARC)*, pages 76–88, March 2012.