Memory and Communication Profiling for Accelerator-based Platforms

Imran Ashraf, *Member, IEEE,* Nader Khammassi, *Member, IEEE,* Mottaqiallah Taouil, *Member, IEEE,* and Koen Bertels, *Member, IEEE,*

***Abstract*—The growing demand of processing power is being satisfied mainly by an increase in the number of homogeneous and heterogeneous computing cores in a system. Efficient utilization of these architectures demands analysis of memory-access behaviour of applications and perform data-communication aware mapping of applications on these architectures. Appropriate tools are required to highlight memory-access patterns and provide detailed intra- application data-communication information to assist developers in porting existing sequential applications efficiently to these architectures. In this work, we present the design of an open-source tool which provides such a detailed profile for C/C++ applications. In contrast to prior work, our tool not only reports detailed information, but also generates this information with manageable overheads for realistic workloads. Comparison with the state- of-the-art shows that the proposed profiler has, on the average, an order of magnitude less overhead as compared to the state-of-the-art data-communication profilers for a wide range of benchmarks. The experimental results show that our proposed tool generated profiling information for image processing applications which assisted in achieving a speed-up of $6.14\times$ and $2.75\times$ for heterogeneous multi-core platforms containing an FPGA and a GPU as accelerators, respectively.**

***Index Terms*—Memory Profiling, Data-communication Profiling, Architecture-independent Profiling, Accelerator-based Computing, Communication-aware Mapping, Shadow Memory.**

I. Introduction

Although transistor scaling yields to more transistors per chip, inherent physical limits prevent further cost-effective down scaling due to multiple challenges such as increased power consumption and complex fabrication process [1]. As a result, designers have shifted the computational paradigm by integrating more and more homogeneous and heterogeneous processing cores in the architectures. A well known form of heterogeneous computing utilizes accelerators to gain performance. Example of such architectures exist in general purpose [2], [3], embedded [4]–[6], and high-performance computing platforms [7]–[10].

To exploit an accelerator in the architecture, applications must be partitioned, where compute intensive parts are offloaded to the accelerator. This is a critical task as an improper application partitioning may diminish the anticipated performance improvements. The main identifiable reason for such performance degradation is the huge data-communication overhead [11] between CPU and accelerator. Moreover, accelerators generally have a deep memory hierarchy, thus, improper mapping of data structures to the available memory levels constitutes the second main reason for poor utilization of accelerators. The performance degradation exacerbates due to growing memory wall [12]. Hence, it is considered as the major design challenge for multi-core architectures [13]. In addition, it is a major source of energy consumption [14]. These problems could be alleviated by profiling the data-communication and analysing memory-access patterns of the data-structures to perform communication-aware application partitioning and proper mapping of data-structures to memory levels. With the growing application complexity, driven by an increasing demand of processing, it is time-consuming, tedious and error-prone to manually analyse these complex applications. Hence, application analysis tools are required to identify the hot-spots and/or bottlenecks pertaining to the target platform.

Several existing tools provide data-communication information for *parallel* applications where communication is either implicit (such as Posix threads) [15]–[17] or explicit (such as MPI) [18], [19]. However, these tools cannot profile the intra-application data-communication inside *sequential* applications. Other tools focus on *architecture-dependent* memory and communication profiling [20]–[22]. However, these tools are only useful in highlighting performance issues in applications which are already ported to the target architectures. Hence, all the above tools cannot be utilized for *architecture-independent* profiling of a large base of *sequential* legacy code that has not been parallelized and ported to accelerator-based platforms.

Architecture-independent data-communication profilers for *sequential* applications based on static-analysis [23] can only be used for regularly structured applications. Additionally, pointer analysis and the dynamic nature of applications make tracking the data-communication statically very hard. Hence, dynamic methods are required. Few data-communication profilers [24]–[27] based on dynamic analysis have been reported. These profilers have a high execution-time and memory-usage overheads as compared to the static ones. One of the reasons for these high overheads stems from the fact that the dynamic generation of the application's representative profile requires the use of the realistic workloads, which results in an increase in overheads. Another reason for these high overheads is an improper design of the shadow memory scheme in these tools, which is critical to the performance. Apart from high overheads, another problem with these profilers is the unclear correlation between the generated information and the application's source-code. This information, though very useful for developers, makes the design of such tools challenging and

I. Ashraf, N. Khammassi, M. Taouil and K. Bertels are with the Department of Quantum & Computer Engineering, Delft University of Technology, 2628 CD Delft, The Netherlands. E-mail: {i.ashraf, m.taouil, n.khammassi, k.l.m.bertels}@tudelft.nl

further increases their overheads.

In this work, we present an open-source, architecture-independent memory-access and data-communication profiler based on dynamic analysis which addresses these issues. To demonstrate its applicability, we present two case-studies where we analyse memory-access patterns and data-communication bottlenecks of image processing applications. The main contributions of this work can be summarized as follows:

- The design of a hybrid shadow memory scheme, which is the most important data-structure as it has a major effect on the space and time overheads caused by such tools.
- The design and implementation of an open-source memory-access and data-communication profiler which utilizes the proposed hybrid scheme.
- Methodology based on the proposed profiler to port existing sequential applications systematically onto an accelerator based platform.
- The analysis of memory-accesses and data-communication behaviour of two image processing applications as case-studies demonstrating the utilization of the methodology steps. The generated information is utilized to port these applications onto systems using the two widely used accelerators, namely, GPU and FPGA.
- An empirical comparison of our tool with the state-of-the-art focusing on the two criteria: execution-time and memory-usage overhead.

Experimental results show that in the case of FPGA, a speedup of up-to $6.14\times$ was achieved, compared to the baseline FPGA implementation where these optimizations were not applied. In the case of GPU, a speedup of up-to $2.75\times$ was achieved, compared to the baseline GPU implementation where this information was not utilized. Furthermore, comparison with state-of-the-art shows that the proposed tool has, on the average, an order of magnitude less execution-time and memory-usage overheads.

The remainder of this paper is structured as follows. We provide the necessary background and related work in Section II. The design of the proposed profiler is discussed in detail in Section III. Section IV enumerates the steps involved in mapping existing applications on accelerator based platforms. The practical use of this methodology is presented in Section V and Section VI, where the use of information generated by the proposed profiler for image processing applications are discussed as case-studies to efficiently map these applications on accelerator-based platforms using FPGA and GPU as accelerators. An empirical comparison of the overheads is presented in Section VII, followed by conclusions in Section VIII.

II. Background

In this section, we define the basic terminology, provide some design considerations and discuss the related work to give reader the necessary background to understand the work presented in the rest of this paper.

A. Terminology

Static-analysis refers to the analysis performed at compile-time, whereas **dynamic-analysis** refers to the analysis performed at runtime. Commonly, dynamic-analysis relies on registration of callback routines (**analysis routines**). These routines are executed when an event of interest happens. The process of routine registration is known as **instrumentation**, which when performed at runtime, in the binary of the application, is referred as **dynamic binary instrumentation**.

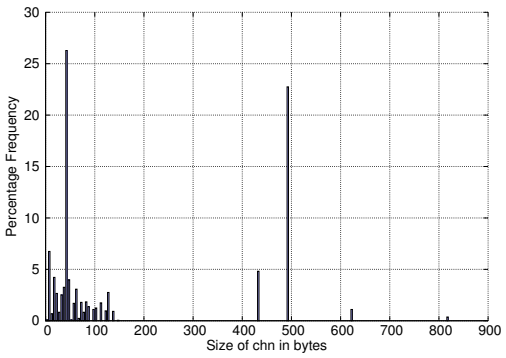


Fig. 1: Percentage Frequency Distribution of the Size of *chn* object in a bwa-mem [29] Application.

Memory profilers utilize instrumentation to monitor memory reads and writes to provide information about memory-access, such as, memory-access pattern of functions. Cache profilers are a subset of memory profilers and provide information about cache utilization. This information is also utilized by memory debuggers to detect memory-related bugs.

Memory-access information is also utilized by data-communication profilers to record data-communication in an application. **Data-communication** occurs when a part of an application writes data into the memory which is later read by another (or same) part of the application. At the fine-granularity level, this data-communication can be reported at instruction or basic block level. At the coarser-granularity, data-communication can be reported between functions in a sequential application or between threads in a parallel application. Profilers that provide this data-communication information are termed as data-communication profilers.

Data-communication profilers typically use some **shadow memory** scheme to record meta-information about the actual memory used by the application [28]. For instance, a bit in the shadow memory may indicate weather a particular byte used by an application is initialized or not.

B. Static vs Dynamic Analysis

Memory-access and data-communication profiling based on static-analysis can only be used for regularly structured applications [23]. Following are some of the reasons which make static analysis infeasible to track memory-accesses and intra-application data-communication:

1) **Dynamic nature:** Most of the real-world applications and variety of benchmarks rely on information which is not available statically at compile-time. For instance, runtime allocation of an image based on an input argument.
2) **Irregular behavior:** Apart from being dynamic in nature, applications also exhibit irregular behavior, making it

impossible for tools to statically perform analysis. Figure 1 demonstrates one example of this irregularity, where the variation of allocation size of *chn* object in a bio-informatics application (bwa-mem [29]) is depicted as a percentage frequency plot.

3) **Pointer arithmetic:** Considerable amount of applications are written in C/C++ which heavily utilize pointers. These applications are hard to track statically, because it is not always possible to de-reference pointers and pointers to pointers statically. Furthermore, statically knowing if a pointer even points to a valid pre-allocated or properly initialized memory, is not always possible.
4) **Global variables:** The use of global variables makes it hard to statically analyse memory reads and writes.
5) **Shared library calls:** Applications relying on shared library calls cannot be statically analysed without re-compiling the library.

C. Practical Design Considerations

In order to record the memory-access behaviour of an application and generate an inter-function data-communication profile, an important requirement is to know the producer-consumer relationship among functions. The consumer of a memory location is trivial to determine, as it is the currently executing function. On the contrary, efficiently obtaining the information about the producer is not trivial, as this requires recording the producer of each memory location at a certain granularity. The main reason for this difficulty is the huge user space. For example, on a 64-bit system with 48-bit virtual addressing, memory addresses can be anywhere in the 128 TB memory-map in the user space. Hence, the problem boils down to efficiently recording the producer of a memory location, such that the profiling approach has a balanced trade-off between execution-time and memory-usage overhead. Furthermore, this mechanism should be flexible and portable, thereby making almost no assumptions about the memory map or other Operating System (OS) specific functionalities.

D. Related Work

Various open-source [30]–[32] and propriety [33], [34] existing tools perform memory profiling. However, these tools only provide the information about the cache misses and do not report data-communication information in an application. The reader is referred to [35] for a detailed survey of such tools.

Though static-analysis tools [23] can be used to track memory reads and writes and record data-communication, a large number of tools utilize dynamic-analysis to collect accurate information at runtime. Architecture simulation is one of the dynamic analysis technique which has been used to track the data-communication among threads in parallel applications [24]; for example, by using a cycle-accurate architecture simulator. However, such simulations are generally computationally intensive which limits the use to small data inputs. Furthermore, it requires the design and development of a cycle-accurate simulator of these architectures.

A well-known dynamic-analysis technique used by large number of tools is instrumentation, which is performed either at compile-time [36] or at run-time [28], [37]. Various tools based on this technique are used for finding memory-management [37] and threading bugs [36], [37]. However, few tools exist [25]–[27], which perform detailed data-communication characterization, especially for sequential applications.

Redux [25], a Valgrind-based tool, draws the detailed Dynamic Data-Flow Graphs (DDFGs) of applications at the instruction-level. This tool has very high overhead as it generates fine-grained DDFGs. Hence, it can only be used for very small applications or parts of applications, as mentioned by authors. In addition, the target of the tool is to represent the computational history of an application and not to report its communication behaviour, as explained by authors.

PINCOMM [26] reports the data-communication using Intel Pin Dynamic Binary Instrumentation (DBI) framework [38]. PINCOMM uses a hash-map to record the producer of a memory location. Due to this map, the tool has a high memory overhead. Due to this overhead, PINCOMM stores the intermediate information to the disk and reads it later by a script to generate the communication graph. This disk writing incurs a high execution-time overhead. Furthermore, the authors have mentioned the use of markers in the source-code to reduce the overhead and manage the output complexity. However, in complex applications inserting these markers manually is time-consuming. In addition, this marking requires knowledge of the application in order to understand what the important parts of the application are, which is not trivial.

QUAD (Quantitative Usage Analysis of Data) [27], also based on Pin [38], provides data-communication information between functions by tracking memory access at byte-granularity. The Trie data-structure is used to store producer-consumer relationships and it does so with a low memory overhead, as memory in the Trie is allocated on demand at the byte-granularity. However, this approach has a high execution-time overhead, mainly because of the access-time of the Trie and the frequent memory allocations. Furthermore, the cumulative information is reported at the application-level, which makes it difficult to utilize. In addition, the information generated is not really useful when the application has a different memory access behaviour per call. Moreover, the provided information has a limited relationship to the application source-code, which makes its use tedious for developers.

The work presented in [39] removes the TLB miss overhead for big-memory workloads. Though the focus of this work is not on memory profiling and data-communication analysis, however, the proposed scheme utilizes a combination of page-tables and direct mapping, quite analogous to our work. The authors propose the necessary hardware support as well as the OS modifications to get benefit of this scheme. Our hybrid scheme neither requires any architectural support nor any OS modifications.

Summarizing, existing approaches generally have high execution-time and memory-usage overhead, which limits their use for realistic workloads. This may affect the quality of the generated profile. Furthermore, the provided information lacks

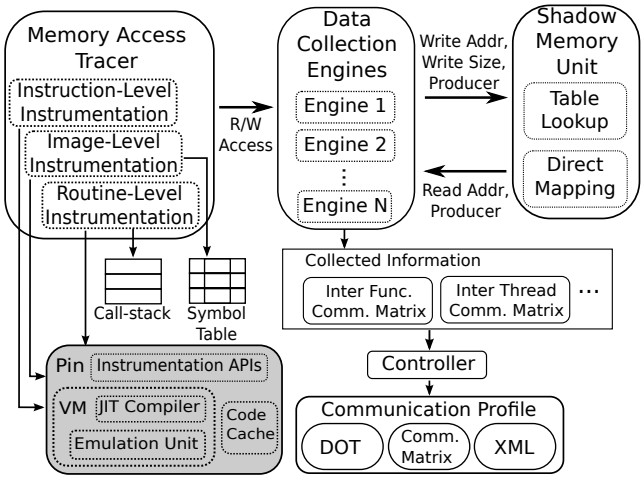


Fig. 2: Internal organization of MCPROF.

necessary dynamic details and has limited correlation to the source-code, making it hard to utilize this information.

III. MCPROF: MEMORY AND COMMUNICATION PROFILER

In this section, we present the design of MCPROF [1] which can conceptually be divided into the main blocks depicted in Figure 2. Application (binary) to be profiled is given as input to obtain data-communication profile in various formats at the output. Shaded block in this figure is the Intel's Pin DBI framework. Rest of the parts are discussed in detail below.

A. Memory Access Tracer

The memory access tracer uses Intel's Pin [38] DBI framework to trace memory reads and writes performed by the application. Pin provides instrumentation APIs to instrument at various granularity levels, such as instructions, basic-blocks, routines and image level. The instrumentation APIs allow the user to register callback routines, known as analysis routines, which are called when a certain event happen. For instance, a registered instruction-level analysis routine will be triggered on each executed instruction.

We utilize instruction-level instrumentation to track memory reads and writes by each instruction. Furthermore, routine-level instrumentation is utilized to keep track of the currently executing function. These are tracked by maintaining a call-stack of the functions executing in the application. Static symbols are obtained by reading Executable and Linkable Format (ELF) [40] header. To track the dynamic allocations, image-level instrumentation is utilized to selectively instrument library images for memory (re)allocation/free routines.

An important point worth mentioning here is that we not only maintain call-stack, but also the call-site stack to record source-code location of each call in the call-stack. Source-code information can be obtained by using `PIN_GetSourceLocation` Application Programming Interface (API) call available in Pin framework. Using this API call in the analysis routine results in high execution-time overhead of the tool because of two reasons: 1) First, because, analysis routine is executed large number of times as compared to an instrumentation routine. 2) Secondly, according to Pin documentation [38], when calling this API from analysis routine, client lock must be obtained before the call to this function and released afterwords.

[1] https://bitbucket.org/imranashraf/mcprof/downloads

```
1  void B() { /* memory allocation by malloc/new */ }
2  void A(){
3    B();
4    B();
5  }
6  int main(){ A(); }
```

Fig. 3: Code snippet (file.c) showing some function calls.

In MCPROF, we call `PIN_GetSourceLocation` API in the instrumentation routine to generate a unique location-ID corresponding to the source-code location. This location-ID is passed to the analysis-routine responsible for function call and return, which only pushes and pops this integer number onto the call-site stack. In this way, we can relate the generated runtime information with the application source-code, without requiring the need to call `PIN_GetSourceLocation` API in the analysis routine, resulting in reduced execution-time overhead of tool.

It is also important to highlight the technique we use to assign a unique object-ID to each allocated object. Instead of using the current function at the top of call-stack, or even the whole call-stack, we utilize the call-site stack to generate a unique ID to identify allocated objects. We would like to motivate the importance of our technique by using the code-snippet listed in Figure 3 where memory allocation happens in function B. Top of stack (B) will not be unique, if used to generate IDs for these allocations. Moreover, the call-stack to the two allocations ($main \rightarrow A \rightarrow B$) is also not unique. However, the call-site stack to the two allocations is unique ($file.c : 6 \rightarrow file.c : 3 \rightarrow file.c : 1 \neq file.c : 6 \rightarrow file.c : 4 \rightarrow file.c : 1$). This pattern of allocations is quite common, as applications usually have utility functions to allocate objects. A limitation of this approach can be seen in the situation when multiple calls to B() will be on the same source-code line, however, this is not a common practice.

A final comment we would like to make is that although in the current implementation we have used the Pin framework to trace memory accesses, in the future, if desired, with minor modifications, it is possible to use any other DBI framework or any other technique to trace memory accesses.

B. Data Collection Engines

On each memory access traced by the Memory Access Tracer, a specific callback function is triggered based on the selected engine. In the case of a write, the producer of the memory address is recorded in the shadow memory. On a read access, the producer is retrieved from the shadow memory, while the consumer of the memory access is the function at the top of the call-stack. Furthermore, based on the information required by each engine, extra information is also

recorded such as the source-code line and file-names of the allocated blocks as well as the allocation size, which is stored in the symbol-table. The complexity of these engines varies with the variation in the amount of details collected in the profile, hence, command-line switches are provided to select the desired engine. Switches are also provided to control the granularity at which memory reads and writes are recorded. This can be at the lower granularity of bytes or higher granularity of 4-bytes to reduce further the execution-time and memory-overhead of the tool. Finally, the modular design of the tool helps in easily adding new engines by modifying the existing engines to generate the desired information or produce the output in the desired format. Currently we have implemented the following three engines in MCPROF.

Engine-1: This engine reports the memory-intensive functions and objects in an application. This information, combined with the execution profile of an application, can be automatically utilized, if desired, to reduce the overhead by performing selective instrumentation to reduce the complexity of generated profile. Another important output generated by this engine is the callgraph in JSON format which is converted to Graphviz DOT format using a python script [2]. We would like to mention here that this callgraph can also be generated by existing tools like gprof [41] at the coarser granularity of functions. However, MCPROF provides this information not only at the granularity of functions but also at the loop-nest level. Furthermore, users can also mark arbitrary regions of interest in an application which are then detected by MCPROF and results are reported against these marked regions.

Engine-2: This engine records inter-function/inter-thread data-communication at the application level. The data-communication information is stored in a data-communication matrix, where indices of the matrix are the producer and consumer function/threads. When object-tracking is enabled, MCPROF reports data-communication to/from the objects in the source-code.

Engine-3: This engine generates per-call data-communication information, which is especially important for applications with irregular memory access behaviour per-call. Each call is also given a unique sequence number which also helps in identifying the temporal information of each call.

An important metric that is reported by this engine is the measure of spatial locality. This quantification of locality is based on the work in [42], [43], where authors assign a score (between 0 and 1) to the locality of access, based on the reciprocal of the stride of the memory access. For example, stride-1 access will get a score of 1 (1/1 is 1), stride-2 access will get a score of 0.5 (1/0.5 is 2), stride-4 access will get a score of 0.25 (1/0.25 is 4) and so on.

On each read/write memory-access, we calculate its stride to maintain a histogram to record frequency of various strides. Before a subroutine return, we calculate strides using this histogram to assign locality score. Thanks to the careful design of the shadow memory scheme (coming up in the following sub-section), we can efficiently obtain the object-ID based on the read/write address. In this way, we can report detailed results as we can relate the calculated locality score with each object read/written for each individual call.

[2]https://github.com/jrfonseca/gprof2dot

TABLE I: Memory Access Frequency in Bottom (Mem0), Top (Mem1) and Middle Memory Regions.

Application	Memory Access Frequency (%)		
	Bottom	Top	Middle
canny	68.3	22.2	9.5
KLT	33.7	37.1	29.2
ocean-NC	13.1	86.9	0
fmm	32.9	67.1	0
raytrace	11.9	88.1	0
bwa-mem	11.7	87.8	0.5
Mean	**28.6**	**64.9**	**6.5**

C. Shadow Memory

This block is responsible for recording the producer of each byte. On each write access, the selected engine sends the address, size, thread-ID and the function at the top of the stack, which is the writer (producer) of this byte to the shadow memory unit. When a function reads a byte, the reader (consumer) is the function currently running, while the the producer is retrieved from the shadow memory unit. These reads and writes can happen anywhere in the $128\,\mathrm{TB}$ user address space, so keeping track of the producer efficiently, is not trivial. Hence, the design of this shadow memory block has a great impact on the execution-time and memory-usage overheads of a profiler. Hence, we have combined the following two techniques in the design of the shadow memory unit.

- **Direct Mapping** in which an application's address is translated to a shadow memory address by using a *Scale* and *Offset*. Given an address *Addr*, its shadow address will be $(Addr \times Scale) + offset$. Although this address translation is fast, it assumes a particular OS memory layout and requires the reservation of a huge amount of virtual memory at fixed addresses.
- **Table-lookup** in which multi-level tables are used to map addresses in an application to their shadow addresses. This is similar to the page look-up tables utilized in OSes. This approach is more flexible as it neither requires a fixed memory layout, nor an initial reservation of a huge memory. This is because tables are allocated on demand. The downside of this approach is that the multi-level table look-up is slower than the address translation in direct mapping.

The key motivation behind the design of our shadow memory scheme is based on the our analysis of the memory-access frequency of various regions in the memory map. Table I depicts the results of this analysis for applications from various domains, namely; image-processing (canny [44], klt [45]) domain, SPLASH-2 benchmarks [46] (ocean-NC, fmm, raytrace) and a bio-informatics application (bwa-mem [47]). These results show that the most frequently accessed memory regions are the bottom (Mem0) and the top (Mem1) in the memory map. From the empirical results we have found out that for most of the applications, the size of these bottom and top regions can be $2\,\mathrm{GB}$ each. The memory accesses in

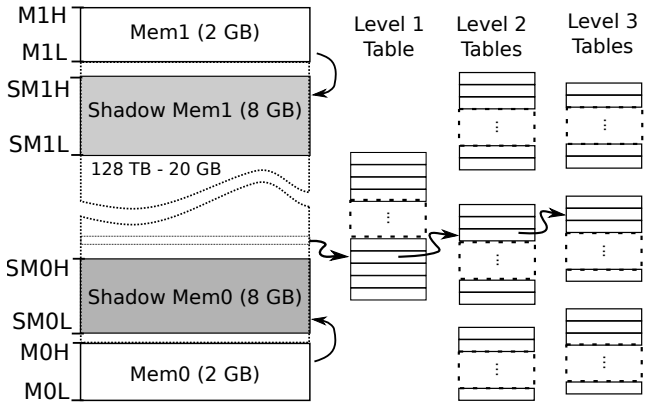


Fig. 4: Hybrid Shadow Memory Scheme Utilized by MCPROF. Most Frequently Accessed Regions of Memory (Mem0 and Mem1) use Direct Mapping. Rest of the Memory Map is Shadowed by 3-Level Table Lookup.

these regions correspond to heap and stack memory accesses, respectively. This is because in the Windows/Linux OS, heap grows from lower to higher addresses, while stack grows from higher toward lower addresses. The shared libraries are mapped by the OS somewhere between these bounds.

To make a well-informed trade-off between flexibility, execution-time and memory-usage overheads, we utilized a hybrid design of the shadow memory unit as shown in Figure 4. To make accesses to the Mem0 and Mem1 regions faster, we reserve[3] in advance two shadow memories corresponding to these two memory regions, shown as *Shadow Mem0* and *Shadow Mem1*, respectively. This results in a simpler mapping of addresses in these regions to the shadow addresses by Equation (1), without requiring any lookup.

$$\begin{aligned} Addr_{sh} = &((Addr \& M0H) << log_2(SCALE)) + \\ &(Addr \& (SM1L + SM0L)) + SM0L \end{aligned} \quad (1)$$

where, $Addr$ is the address of the original byte, $Addr_{sh}$ is the address of the corresponding shadow bytes, $SCALE$ is 4, and $M0H$, $SM1L$ and $SM0L$ are constants as shown in Figure 4.

A point worth mentioning here is that a simpler and faster approach could have been to restrict the address mapping of an application to the lower half and use the upper half for the shadow memory. However, to the best of our knowledge, there is currently no (portable) method to restrict the address allocation in this manner. So, for the middle ($128\,\mathrm{TB} - 20\,\mathrm{GB}$) less frequently used region, we utilize a 3-level table-lookup scheme as shown in Figure 4.

Initially, the level-1 table is created and all its entries are marked as *UNACCESSED*. Tables in the remaining two levels are created on demand when the address in that range is touched for the first time. The address of the memory accessed in this region is used to index these tables to reach level-3 where 4 shadow bytes are written for each byte memory accessed in the original application. One byte for the function-ID, one byte for thread-ID and 2 bytes for the-ID of the object this address belongs to. Therefore, we currently restrict the number of function and thread-IDs to 256. In the future we will investigate more applications, and if required, increase the number of bytes to store the IDs, as it is simply a parameter in the tool. Furthermore, the output of Engine-1 can also be utilized to automatically focus on the important functions/objects in the application, to reduce overheads and to get a clear view of the hot-spots in the applications, as will be discussed later in the case-studies.

[3]These regions are only reserved in the memory map, actual memory-usage is $4\,\mathrm{B}$ for each byte of memory used by the application.

IV. Application Mapping Methodology for Accelerator Based Platforms

In this section, we describe the mapping methodology based on MCPROF to efficiently map an existing sequential application onto an accelerator based platform. This methodology consists of the following four steps:

1) **Hotspot analysis:** locate the parts of an application causing high execution-time and memory access, known as hot-spots. Typical profilers provide information about execution-time contribution at the granularity of functions in the application. MCPROF provides this information not only at function-level granularity but also at lower granularity levels, such as loop-nest granularity. Moreover, MCPROF also provides the information the data-structures responsible for most of the memory-access in the application.
2) **Granularity adjustment:** analyse data and control dependencies to to extract more parallelism by decomposing functions to tasks at lower granularity, such as loop-nest granularity.
3) **Task mapping:** map the tasks to the suitable execution units available in the architecture, which can be Central Processing Unit (CPU) and accelerator. This is an important step to optimize the data-communication by completely avoiding or reducing data transfers between CPU and accelerator. Optimization of the data-communication requires understanding of the data-flow in the application. Understanding this data-communication by analysing the source-code manually, is not trivial, especially for complex applications Furthermore, pointer arithmetic exacerbates this problem making it hard to determine the actual producers and consumers of the data to establish intra-application data-communication.
4) **Memory mapping:** map the data-structures to the suitable memory spaces available in the architecture based on the access pattern and sizes of the data-structures in the application.

V. Case-study 1: Communication driven mapping of Canny on FPGA-based platform

The first case-study is on the FPGA-based acceleration of an image processing application. The target architecture used in this case-study is Zynq [5] System on Chip (SoC) on Zedboard [48]. Zynq SoC has a dual core ARM processor tightly coupled with a Field-Programmable Gate Array (FPGA). The

TABLE II: MCPROF flat profile showing the top functions responsible for most of the execution in Canny application.

Function Name	%Contribution
gaussian_smooth	79.01
non_max_supp	8.65
apply_hysteresis	4.3
magnitude_x_y	3.17
derrivative_x_y	2.81

TABLE III: MCPROF flat profile showing the top functions responsible for most of memory-access in Canny application.

Function	Reads	Writes	Reads/Writes	%Total
gaussian_smooth	6.7e+07	4.7e+06	14.2	61.9
derrivative_x_y	1.9e+07	3.1e+06	5.9	18.9
non_max_supp	1.2e+07	7.8e5	14.8	10.6
apply_hysteresis	3.6e+06	1.7e+06	2.2	4.5
magnitude_x_y	3.1e+06	1.6e+06	2.0	4.1

FPGA can be used as a reconfigurable accelerator to efficiently implement some functionality in hardware. The ARM side of the SoC is known as Processing System (PS), whereas, the FPGA side is termed as Programmable Logic (PL). When an application is ported to Zynq SoC, it can run as a pure software implementation on PS. In order to gain performance, compute-bound parts of an application can be mapped as hardware accelerators on PL.

Xilinx Software Defined System-On-a-Chip (SDSoC) is one way of porting applications to Zynq-based SoC. Pragmas are used to mark the functions which need to be synthesized for PL. Use of SDSoC greatly reduces the development time and effort as compared to describing hardware in Hardware Description Language (HDL). However, this does not always result in the generation of optimized hardware, and therefore, may not always result in performance improvement. This is mainly because of the improper inference of communication infrastructure between PS and PL by SDSoC as well as the poor selection of memory by SDSoC, resulting in reduced anticipated performance improvement. These challenges can be tackled by carefully analysing the application in hand to guide SDSoC in the mapping process by virtue of SDSoC pragmas. By utilizing these pragmas, developers have control over the selection of communication infrastructure to be used to pass data between PS and PL as well as the control of where the allocated objects should be mapped in PL.

The image processing application used in this case-study is based on Canny edge detection algorithm [44]. Canny is a well-known edge detection algorithm, which outperforms other edge detection methods. The algorithm first eliminates any noise from the image. It then finds the image gradient to highlight regions with high spatial derivatives. The next step is to track along these regions and suppress any pixel that is not at the maximum. The gradient array is further reduced by hysteresis. We have used the implementation provided by the CVL at the University of South Florida [49].

A. Hotspot analysis

In this step, the application is profiled using MCPROF to highlight the computational hotspots, also known as kernels in the application. Table II shows the flat profile generated by MCPROF for the Canny application listing the functions responsible for the bulk of execution in this application. Unlike conventional tools, MCPROF also provides information regarding the most memory intensive functions and objects in the application. Table III depicts the top memory intensive functions in the application and lists the read and write accesses performed by these functions. It can be seen from these tables that *gaussian_smooth* function is responsible for most of the execution-time and memory-accesses in Canny application.

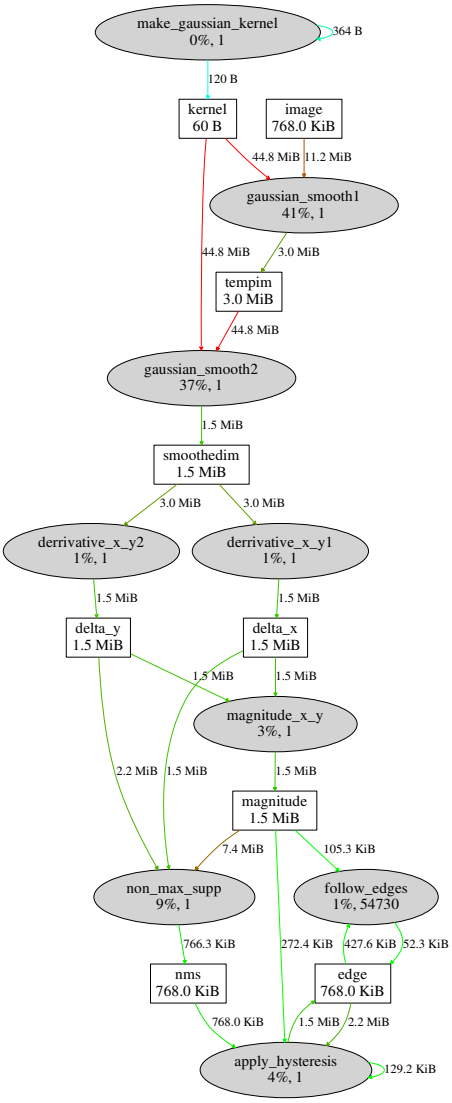


Fig. 5: Data-flow graph of Canny application generated by MCPROF.

B. Granularity Adjustment

As shown in Table II, *gaussian_smooth* contributes about 80% to the application execution. In order to extract parallelism inside this function, we use the MCPROF feature of splitting functions at loop level granularity. Figure 5 shows

TABLE IV: MCPROF flat profile showing the memory intensive objects in Canny application.

Object	Reads	Writes	Reads/Writes	Size	Alloc. Path
kernel	6.7e+07	120	559241	60 B	canny_edge.c:527
tempim	4.7e+07	3.1e+06	15	3.0 MiB	canny_edge.c:448
image	1.2e+07	0	∞^*	768.0 KiB	pgm_io.c:72
smoothedim	6.3e+06	1.6e+06	4	1.5 MiB	canny_edge.c:454

* this is the input image which is read from a file. As it is not produced (written) by a function inside this application, the Writes to this image are *zero* resulting in ∞ Reads/Writes ratio.

the simplified output graph generated by MCPROF where the loops in *gaussian_smooth* are split as separate nodes represented by *gaussian_smooth1* and *gaussian_smooth2*. A similar split of *derivative_x_y* is also shown as *derivative_x_y1* and *derivative_x_y2*. The nodes in this figure represent function, loop-nest or any other region of interest marked by the user. Each node contains its name, the percentage of dynamically executed instructions by this function with respect to the whole application, as well as the total number of calls to this function. For instance, *gaussian_smooth1* is executed once and it covers 41% of the instructions executed by the whole application. The communication is shown by edges, where the intensity of communication is quantitatively shown by the number of bytes on each edge. This intensity is also illustrated by the colour of the edges from red (highest) to green (lowest). In this way, programmer is able to visualize the computation and communication intensive parts of an application, in a single graph, without manual source-code inspection.

As depicted in the Figure 5, the *gaussian_smooth* function exposes dependencies between its two loop nests, thus they cannot be executed concurrently. However, each of these loops can be parallelized individually since they operate independently on columns and rows of the image. On the other hand, the two parts of the *derivative_x_y* do not expose any dependency and therefore can be executed in parallel, since these loops are using a common read-only input while producing separate outputs. A similar analysis is performed on the other functions.

C. Task mapping

In this step the data flow in the application is analysed to perform function chaining. The idea is to map the functions communicating heavily with the kernels in the application to the accelerator to reduce the data communication between CPU and accelerator. MCPROF highlights the data communication in the application which can guide this step. Figure 5 shows the communication graph generated by MCPROF for the complete Canny application.

As can be seen from Figure 5, *gaussian_smooth1* takes the *image* and *kernel* as input to generate *tempim*. As *gaussian_smooth1* and *gaussian_smooth2* are communicating heavily through *tempim* object, so it is obvious that they should be chained together in the accelerator. Another important point is that `make_gaussian_kernel` is also communicating heavily with *gaussian_smooth1* and *gaussian_smooth2*. Though, *make_gaussian_kernel* is not computationally intensive, still it should be chained together with *gaussian_smooth1* and *gaussian_smooth2* on the accelerator.

On the other hand *derivative_x_y1* and *derivative_x_y2* should not be mapped to accelerator with *gaussian_smooth1*. This is because, these functions are not computationally intensive and mapping them to accelerator will double the communication between CPU and accelerator as *delta_x* and *delta_y* will need to be communicated back to CPU instead of *smoothedim*.

D. Memory mapping

As discussed earlier, MCPROF detects statically and dynamically allocated objects involved within flows. It provides information about access to these object as flat profile (Table IV) and communication graph (rectangles in Figure 5). It can be seen from first row in Table IV that kernel has very high Reads/Writes ratio indicating that these are some constant values. As size of kernel is only 60 B, hence it should be allocated as local memory on PL and split it to individual elements. This corresponds to implementing an array as a collection of registers rather than as a memory. This can be achieved by the following pragma:

```
#pragma HLS array_partition variable=kernel \
dim=0 complete
```

Size of *tempim* is big and cannot fit in local Block RAM (BRAM). Moreover, it can be seen from Figure 5 that it is used internally to pass output of *gaussian_smooth1* to *gaussian_smooth2*. Hence, it should be used as shared memory without caching. This is important as it will instruct SDSoC to not cache it on PS side and save considerable time in flushing the cache. This can be achieved by the following pragmas:

```
#pragma SDS data mem_attribute(tempim:NON_CACHEABLE)
#pragma SDS data zero_copy(tempim)
```

Another observation is that *tempim* is being reused large number of times. It will incur a huge performance hit if accessed directly from shared memory. Hence, caching a part of *tempim* into a BRAM will provide two main performance benefits:

1) *tempim* will be loaded once to BRAM and successive access will be from BRAM rather than DDR.
2) DDR will be accessed in burst, resulting in higher performance as compared to when accessed one element at a time when required.

image is used as input to the *gaussian_smooth1* hence no writes. Size of *image* is big (768.0 KiB) hence cannot fit completely into BRAM. There is also a reuse of *image* values resulting in 1.18e+07 total reads. Hence, it will be beneficial to cache a part of *image* in to a local BRAM. Secondly, as *image* is used as input, it will be possible to stream *image* into PL

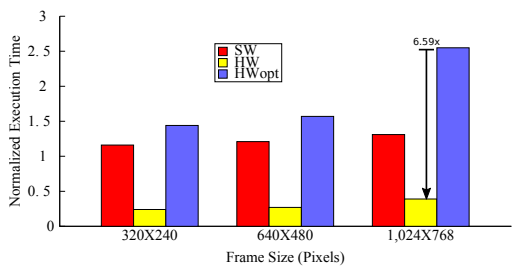


Fig. 6: Speedup results for the Canny application on Zedboard

if it is accessed sequentially. Output of Engine-3 reports that the locality score of accessing *image* by *gaussian_smooth1* is 1, clearly reporting a spatial locality with a stride of 1. This suggests that a single row can be buffered into BRAM using streaming PL interface for *image*. This can be achieved by the following pragma:

```
#pragma SDS data access_pattern(image:SEQUENTIAL)
```

smoothedim is used as output and it cannot fit into BRAM. Furthermore, the accesses to *smoothedim* are not sequential as reported by Engine-3, hence *zero_copy* data mover should be used for it by the following pragma:

```
#pragma SDS data zero_copy(smoothedim)
```

E. Experimental Results

Figure 6 provides performance comparison of various versions we implemented for this case-study. These results are the normalized execution-times compared to the orignal software implementation (y-axis) and are provided for various image sizes (x-axis). *SW* represents the optimized software implementation. *HW* represents an implementation with top contributing functions (kernels) implemented in hardware without performing the memory and communication optimizations mentioned in the case- study. *HWopt* represents an implementation with kernels implemented as optimized hardware based on the memory and communication optimizations mentioned earlier in the case-study.

It can be seen from Figure 6 that the *HW* implementation without memory-accesses and data-communication optimizations actually results in performance degradation. Furthermore, the *HWopt* version is about $6\times$ faster than the one where these optimizations are not applied.

VI. Case-study 2: Communication driven mapping of KLT on GPU-based platform

The focus of this case study is on the utilization of data-communication information provided by MCPROF to map an application onto the GPU, without performing algorithmic modifications. The use case involves Kanade-Lucas-Tomasi Feature Tracker (KLT) application [45]. This application detects interesting features in a frame and tracks them in the subsequent frames. We have used version 1.3.4, which is the latest version of KLT [50]. This *C* implementation has 102 functions in 17 source-files making up 5033 lines of code.

TABLE V: *gprof* flat profile for the *KLT* application.

Function Name	%Time
KLTSelectGoodFeatures	54.07
convolveImageVert	19.65
convolveImageHoriz	10.17
trackfeature	7.81
%Total Contribution	**91.7**

For the experiments performed in this case study, we used 64 bit, 2.5 GHz Intel(R) Xeon(R) CPU with 32 GB RAM. Nvidia GeForce GT 640 GPU, with 2 GB memory, is used as an accelerator which is connected to the PCIe slot of the CPU.

A. Hotspot analysis

In order to efficiently map an application onto an accelerator based platform, functions responsible for most of the execution-time in an application, known as kernels, are offloaded to the accelerator. We used *gprof* [41] to identify the kernels in the application as shown in Table V. For this run, 30 frames have been used with frame size chosen as 1024×768, to accumulate enough number of samples to generate representative profile of the application. The total percentage contribution of these kernels 91.7%. Based on Amdahl's law, this gives a theoretical maximum application speed-up of $12.04\times$.

As a first step in the mapping process, we mapped these kernels to the GPU. Table VI provides the timing results of the first mapping step. For these experiments, 1024 features were tracked from frames of size 1024×768. Column 1 contains the names of the kernel. Column 2 lists the execution-time of these kernels on CPU (t_{cpu}) in seconds. $t_{gpu_{comp}}$ is the time spent in performing the computation on GPU which is shown in Column 3. The communication time $t_{gpu_{comm}}$ is listed in Column 4 which is the time spent in transferring data to GPU before computation and reading the results back, after the computation is complete. The execution-time speed-up is the ratio of the execution-time on CPU and GPU. We have calculated various speed-ups to analyse the performance gain/loss in terms of ratios of various execution-times. Total kernel speed-up ($S_{K_{total}}$) is reported in Column 5, which is calculated as $\frac{t_{cpu}}{t_{gpu_{comp}}+t_{gpu_{comm}}}$. In order to highlight the effect of data-communication, Column 6 lists the kernel speed-up ($S_{K_{comp}}$) for only the computation, calculated as $\frac{t_{cpu}}{t_{gpu_{comp}}}$.

MCPROF provides the production-consumption information in the form of a data-communication graph in various formats. An overview of this information is shown as communication matrix in top right corner of Figure 7 representing inter-function communication intensity. MCPROF also generates the detailed quantitative data-communication information in the form of a directed graph. As there are large number of functions in the KLT application, the complete graph is too large to present here. Secondly, such large graphs are hard to be utilized by the developers. Typically, the functions contributing to most of the execution-time are selected for analysis. MCPROF detects memory-intensive objects and the functions communicating with these memory-intensive ob-

TABLE VI: Execution Time (sec) and Speedup results for the initial KLT implementation.

Kernel	t_{cpu}	$t_{gpu_{comp}}$	$t_{gpu_{comm}}$	$S_{K_{total}}$	$S_{K_{comp}}$
KLTSelectGoodFeatures	13.53	1.17	0.36	8.8×	11.52×
convolveImageVert	3.93	0.14	0.76	4.35×	28.08×
convolveImageHoriz	1.77	0.18	0.76	1.87×	9.89×
trackFeature	1.96	1.49	0.52	0.96×	1.31×

TABLE VII: Memory Intensive Objects in *KLT* reported by MCPROF.

Objects	Reads	Writes	Reads/Writes	Total	%Total
tmpimgCS	3.8e8	5.1e7	7.4	4.3e8	26.6
pointlist	1.3e8	1.3e8	1	2.6e8	16.3
pyramidImg	1.3e8	3.5e7	3.8	1.7e8	10.3
grady	1.34e8	3.1e6	42.7	1.3e8	8.3
gradx	1.34e8	3.1e6	42.7	1.3e8	8.3
tmpimgTF	6.7e7	9.4e6	7.1	7.6e7	4.6
guassderiv_kernel	6.7e7	4.7e3	14063.1	6.7e7	4.1
guass_kernel	6.7e7	4.6e3	14500.6	6.7e7	4.1
%Total Contribution					**82.6**

jects. Table VII lists the memory-intensive objects of the KLT application reported by MCPROF. Apart from mentioning the reads and writes accesses, the percentage accesses are also reported in the last column. The last row of the table shows that memory accesses through these 9 objects correspond to 82.6% of the total application memory accesses.

B. Granularity Adjustment

From Column 5 in Table VI, it can be seen that speed-up has been obtained for all the kernels except for `trackFeature` kernel. Hence, this kernel should not be mapped to GPU. This is because this kernel, in its current form, is not the type of computation which can benefit from the GPU architecture. It involves a large number of decision statements and variable number of data-dependent loop iterations, which are not efficiently handled by GPU Single Instruction Multiple Thread (SIMT) architecture. In order to achieve performance gain for this kernel, algorithmic level changes are required which require the understanding of the application domain. Another important result that can be deduced by comparing Column 5 to Column 6 is that the communication has significantly reduced the achieved speed-up. In the next sub-section, we will perform the optimization of this data-communication by utilizing MCPROF.

C. Task mapping

Figure 7 shows the data-communication graph of KLT application generated by MCPROF while tracking 256 features in 3 frames of size 1024×768. The ovals represent the functions in the application whereas the objects are represented by rectangles. The numbers inside the rectangles are the allocation sizes of these objects. The kernels in the application are shown as Grey ovals. The amount of communication in bytes is represented by directed edges, where the colour of the edges represent the intensity of the communication. To simplify the discussion, the dotted lines are used to mark the functions in the main stages of applications. Due to the detection of objects and associating communication with these objects, the flow of data between functions in various stages can be clearly visualized.

`tmpimgSF` and `tmpimgTF` are generated by `KLTToFloatImage` on CPU and transferred to GPU as an input to `convolveImageHoriz`. `KLTToFloatImage`, though not compute-intensive, still mapping it to GPU will be better as it will make `tmpimgSF` and `tmpimgTF` internal to GPU and reduce CPU-GPU communication.

`guassderiv_kernel` and `guass_kernel` are generated by `computeKernels` on the CPU and consumed by `convolveImageHoriz` and `convolveImageVert` on GPU. However, mapping `computeKernels` to GPU is not required as `guassderiv_kernel` and `guass_kernel` are consumed heavily but produced very infrequently.

Based on the preliminary results in Table VI, it was concluded that `trackFeature` should not be mapped to GPU because of slow-down. Even if this kernel is not so efficient on the GPU, we should still port it to the GPU to avoid the bulk of data-communication regarding the transfer of `pyramidImg` between GPU and CPU. This is clearly shown by the communication edges to the `computeIntensityDifference` and `computeGradientSum` function in the Feature Tracking stage.

D. Memory mapping

`guassderiv_kernel` and `guass_kernel` objects are consumed heavily but produced very infrequently as evident from the very high Reads/Write ratio (Table VII) implying very less production and a lot of consumption of data from these objects. Furthermore, these objects are very small in size (284 Bytes), hence can be easily mapped to GPU's constant memory.

`gradx` and `grady` objects are generated in the Convolution Stage which are consumed in the Feature Selection stage by `KLTSelectGoodFeatures`. Hence, these objects can be kept on the GPU for utilization in these stages. Furthermore, these objects should be mapped to GPU's shared memory. This is because of high Reads/Writes ratio depicted in Table VII, suggesting high re-use of these objects. This will result in performance improvement as shared memory has higher bandwidth as compared to global memory. In this way, data read from the global memory will be stored in to the shared memory. Therefore, multiple reads of this data will be from the faster shared memory, rather than the slower global memory, resulting in performance improvement.

On the contrary, `pointlist` has Reads/Writes ratio of 1 depicted in Table VII, which suggests no reuse, hence it should be kept in the global memory. Mapping `pointlist` to shared memory will only increase the overhead of the data

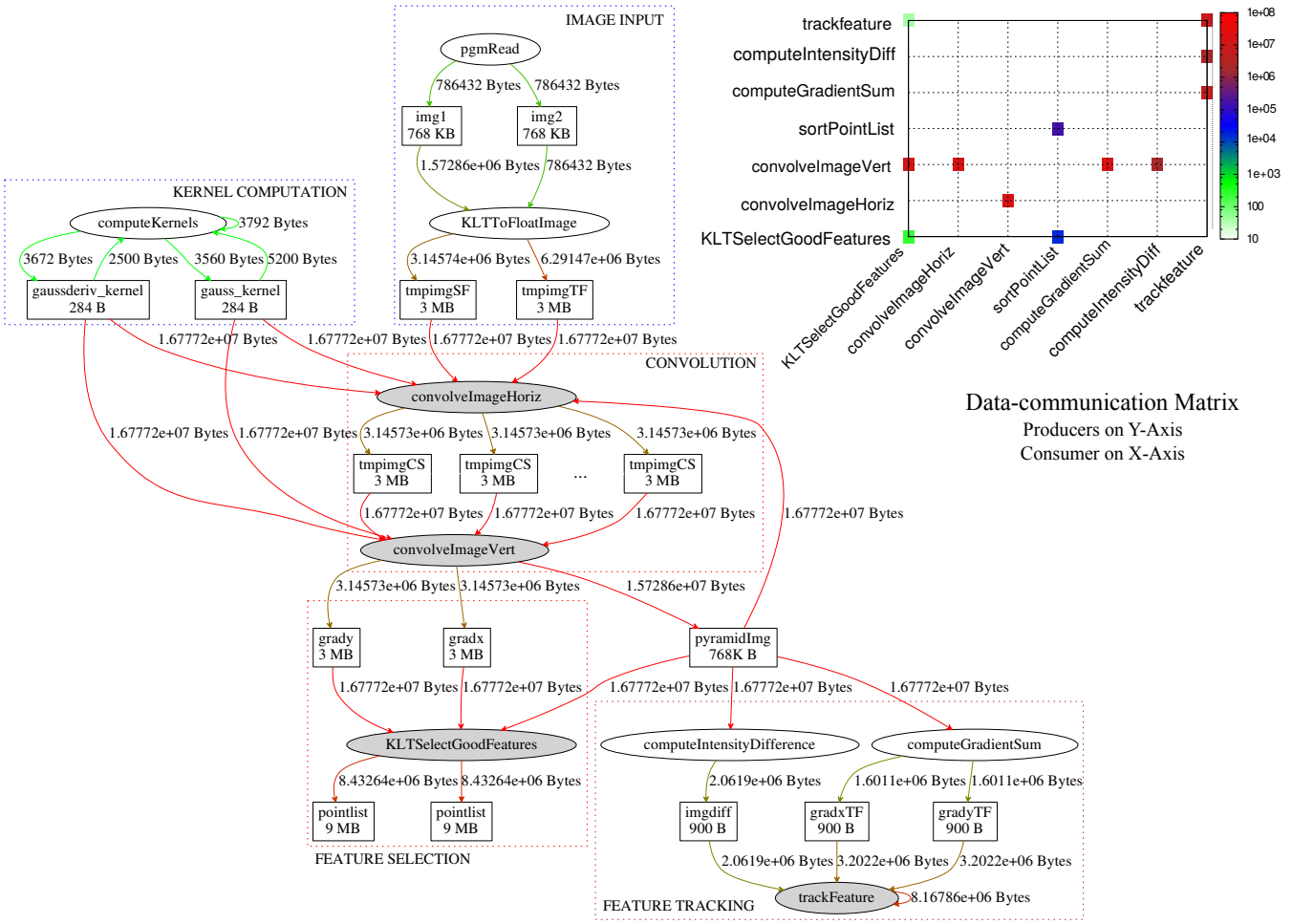


Fig. 7: KLT Communication Matrix (top right) and communication graph generated by MCPROF. Functions (ovals), compute-intensive functions (Grey ovals) and the objects(rectangles) involved in the communication are also shown.

transfer between global memory and shared memory without being reused.

Another optimization which can be performed in the convolution stage is the allocation and de-allocation of large number of `tmpimgCS` objects for each frame. Allocating a single object in the start and re-using it in the subsequent frames instead of re-allocating it will reduce the execution-time. Similar optimization can be performed for `pointlist` in the Feature Selection stage.

E. Experimental Results

In this section, we provide the performance results for the implementations discussed in the case study. The frame sizes have been selected corresponding to the frame dimensions used in various video standards [51]. The affect of varying the number of tracked features on the achieved application speed-up is also discussed.

After applying these optimizations to the initial GPU implementation (gpu), we obtained a data-communication optimized version of the GPU implementation (gpu_{opt}). Figure 8 shows the normalized Frames Per Seconds (fps) achieved by both the implementations for various frame sizes ranging from

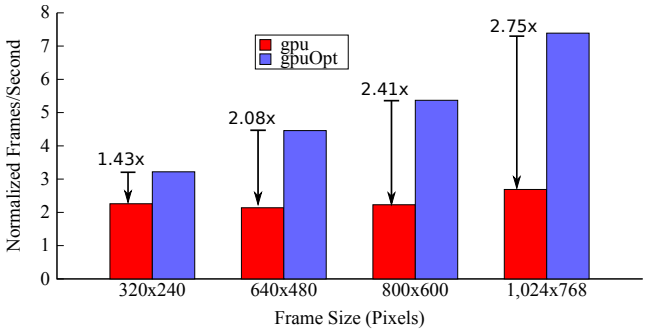


Fig. 8: Normalized Frames per seconds achieved by the GPU and data-communication optimized GPU implementation.

320×240 to 1024×768 while the number of tracked features is set to 1024. Increasing the frame size, results in an increase in the amount of computation performed on the GPU. Increased computation results in better utilization of the available resources of the GPU, resulting in higher speed-up as can be observed from Figure 8. On the other hand, increasing the frame size also increases the amount of frame data transferred to the GPU for processing and getting

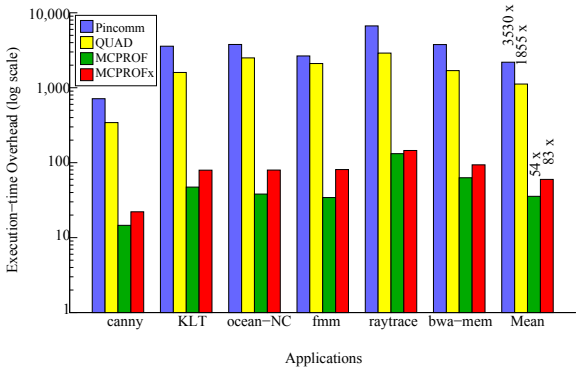


Fig. 9: Comparison of execution-time overhead.

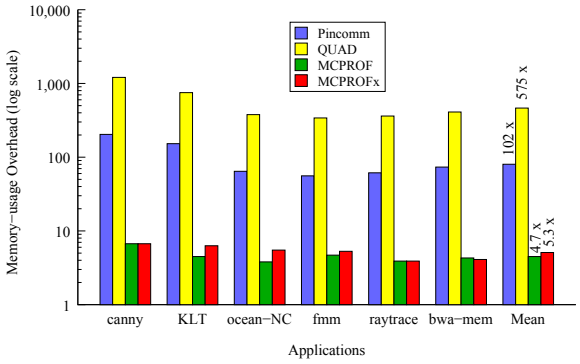


Fig. 10: Comparison of Memory-usage overhead.

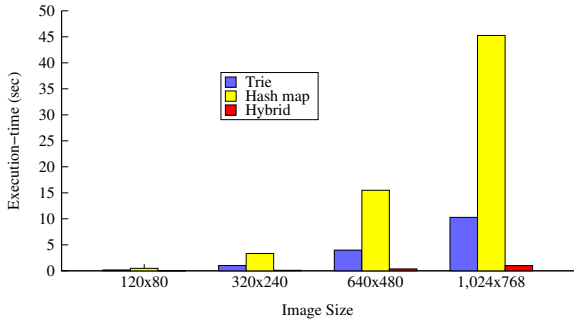


Fig. 11: Execution-time Comparison of Data-structure access only.

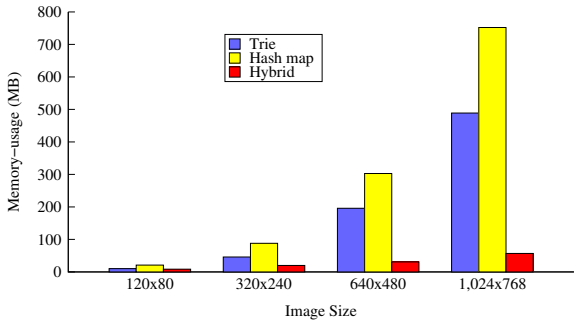


Fig. 12: Memory-usage Comparison of Data-structure access only.

results back. This data-communication has been optimized in the case of gpu_{opt} based on the information provided by MCPROF. Hence, gpu_{opt} implementation achieves up-to $2.75\times$ higher speed-up as compared to gpu implementation where this communication is not optimized.

VII. Overhead Comparison with Existing Profilers

In this section, we present the overhead comparison of MCPROF with other state-of-the-art data-communication profilers QUAD and PINCOMM as these are particularly designed to report data-communication. To make a fair comparison, these tools are configured in such a manner that all the profilers generate as much similar information as possible while running on the same platform. For these experiments, we used Pin v2.13 on the machine used in case-study. Figure 9 depicts the execution-time and memory-usage overhead of PINCOMM, QUAD and MCPROF for applications from various domains, namely; image-processing (canny [44], klt [45]) domain, SPLASH-2 benchmarks [46] (ocean-NC, fmm, raytrace) and a bio-informatics application (bwa-mem [47]). Each bar represents the ratios of the application execution-time with and without profiling for each profiler. Similarly, Figure 10 reports the ratios of application memory-usage with and without profiling.

We have reported MCPROF results with two different settings depicted as *MCPROF* and *MCPROFx* in these figures. Results with *MCPROF* legend are overheads while providing the common basic information which PINCOMM and QUAD can also generate. Whereas, *MCPROFx* report overheads of complex engine while generating the detailed data-communication information with stack recording and object detection. Mean overhead results are also depicted in these figures. These results show that MCPROF has, on the average, an order of magnitude less execution-time and memory-usage overheads. Main reason for this reduction in overhead is the well-thought-out design of the shadow memory scheme utilized by MCPROF. Due to the design, we were able to shift most of the processing from analysis-phase to instrumentation-phase. Moreover, the access-time and memory-usage overhead of the hybrid shadow memory scheme is significantly less as compared to Trie or Hash map utilized by QUAD and PINCOMM, respectively. In order to clearly illustrate this, we have plotted the execution-time and memory-usage of accessing only the data-structures of the three tools in Figure 11 and Figure 12, for the canny application for various image sizes.

Another important source of overhead in such kind of tools is the amount of work done at analysis-time as compared to instrumentation-time. This is important because the functionality in the instrumentation routine will only be executed once at the time of instrumentation. Whereas, the analysis-routine will be executed on each execution of the target instruction/routine. By careful design of the tool, we have managed to shift as much work as possible from analysis-time to instrumentation-time to reduce execution-time overhead.

A specific example in this regard is the way source-code information is obtained and related to the generated run-time information. PINCOMM uses this API call in the analysis routine, resulting in high execution-time overhead of the

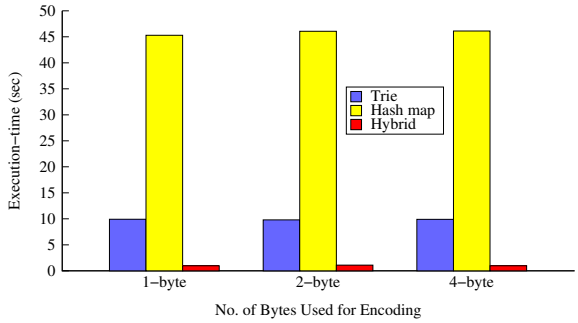


Fig. 13: Execution-time Comparison of three Data-structures with Varying Number of Encoding Bytes.

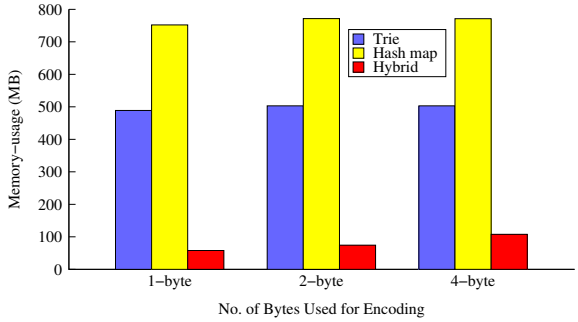


Fig. 14: Memory-usage Comparison of three Data-structures with Varying Number of Encoding Bytes.

tool, as detailed earlier in Section III-A. In MCPROF, we call `PIN_GetSourceLocation` API in the instrumentation routine, without requiring the need to call this API in the analysis routine.

Another important observation from Figure 9 is that MCPROF has an average memory-usage overhead of $4.7 - 5.3\times$, which is mainly because 4 shadow bytes are allocated for each byte used in the original application, plus some additional memory for storing extra information.

Finally, to study the effect of varying number of bytes used for encoding functions, objects and threads, we performed experiments to measure execution-time and memory-usage overheads with 1, 2 and 4-bytes encoding for the three data-structures used by MCPROF, PINCOMM and QUAD as depicted in Figure 13 and Figure 14, respectively. These results are for canny application with input image of size 1024×768. It can be seen from Figure 13 that there is no significant impact on execution-time but there is a slight increase in memory-overhead with increasing number of bytes used for encoding as shown by Figure 14.

VIII. CONCLUSION

Both the memory wall and the growing trend of accelerator-based computing demand detailed memory-access and data-communication profiling of an application. In this work, we presented MCPROF, a memory-access and data- communication profiler. We presented a methodology based on MCPROF to port existing applications to accelerator based platforms. We demonstrated the applicability of this methodology by accelerating image processing applications on platforms with FPGA and Graphical Processing Unit (GPU) as accelerators; we obtained a speed-up of $6.14\times$ and $2.75\times$, respectively, over the implementations where these optimizations are not applied. The unique design of the proposed profiler has resulted in significantly reduced execution-time and memory-usage overheads as compared to the state-of-the-art, making the profiler useful for applications with realistic workloads. Moreover, the reduced overheads allowed us to generate detailed memory-access and data- communication information which is not provided by existing tools.

Utilization of Pin to track memory-accesses makes our tool micro-architecture independent but not ISA-independent. Therefore, it is interesting to utilize LLVM-Tracer [52] to generate the required information for MCPROF and hence, obtain an ISA-independent memory-access and data-communication profile of the application.

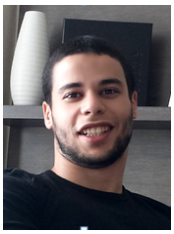
Our future work also involves relating the data-communication information generated by MCPROF with performance estimates generated by the profiling tools provided by Xilinx and Nvidia. This will assist programmers reason about the data-movement in comparison with the performance tradeoffs in a systematic way. Another interesting work revolves around the utilization of the currently generated information by MCPROF to automatically generate SDSoC and OpenACC [53] pragmas for FPGA- and GPU-based accelerators, respectively.

REFERENCES

[1] M. Horowitz and W. Dally, "How Scaling Will change Processor Architecture," in *ISSCC*, vol. 1, 2004, pp. 132–133.
[2] AMD, "AMD A10-7850K APU," http://www.amd.com/us/products/desktop/processors/a-series/Pages.
[3] D. Pham *et al.*, "The Design and Implementation of a First-generation CELL Processor," in *ISSCC*, Feb 2005, pp. 184–592 Vol. 1.
[4] Texas Instruments, "OMAP3530 Application Processors," http://www.ti.com/product/omap3530.
[5] Xilinx, "Zynq-7000 All Programmable SoC," http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.
[6] S. Vassiliadis and et al., "The MOLEN Polymorphic Processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, 2004.
[7] "Hybrid Core Computer by Micron," http://www.conveycomputer.com, 2012.
[8] J. Stuecheli, "Next Generation POWER Microprocessor," in *HotChips 2013*, August 2013.
[9] Nvidia, "GeForce GT 640 Specifications," http://www.geforce.com/hardware/desktop-gpus/geforce-gt640/specifications, April 2012.
[10] O. Pell and O. Mencer, "Surviving the End of Frequency Scaling with Reconfigurable Dataflow Computing," *SIGARCH Comput. Archit. News*, vol. 39, no. 4, pp. 60–65, Dec. 2011.
[11] J. Duato, "Beyond the Power and Memory Walls: The Role of HyperTransport in Future System Architectures," in *WHTRA*, February 2009.
[12] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.
[13] G. Martin, "Overview of the MPSoC design challenge," in *43rd ACM/IEEE DAC*, 2006, pp. 274–279.
[14] S. Borkar and A. A. Chien, "The Future of Microprocessors," *Commun. ACM*, vol. 54, no. 5, pp. 67–77, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1941487.1941507
[15] A. Knpfer *et al.*, "The Vampir Performance Analysis Tool-Set," in *Tools for High Performance Computing*, 2008, pp. 139–155.
[16] S. H. Hung *et al.*, "Trace-based performance analysis framework for heterogeneous multicore systems," ser. ASPDAC '10, pp. 19–24.
[17] M. Bach *et al.*, "Analyzing Parallel Programs with Pin," *Computer*, vol. 43, pp. 34 –41, Mar. 2010.

[18] I.-H. Chung, R. Walkup, H.-F. Wen, and H. Yu, "MPI Performance Analysis Tools on Blue Gene/L," in *SC*, Nov 2006.
[19] H. Brunst and B. Mohr, "Performance Analysis of Large-Scale OpenMP and Hybrid MPI/OpenMP Applications with Vampir NG," in *OpenMP Shared Memory Parallel Programming*, 2008, vol. 4315.
[20] Nvidia, "NVPROF and NVVP, Nvidia Command-line and Visual Profilers," http://docs.nvidia.com/cuda/profiler-users-guide.
[21] "PGPROF by The Portland Group (PGI) ," https://www.pgroup.com/products/pgprof.htm.
[22] "CodeXL by AMD," http://developer.amd.com/tools-and-sdks/opencl-zone/codexl.
[23] M. D. Ernst, "Static and Dynamic Analysis: Synergy and Duality," in *WODA 2003: Workshop on Dynamic Analysis*, Portland, Oregon, May 9, 2003, pp. 24–27.
[24] N. Barrow-Williams, C. Fensch, and S. Moore, "A Communication Characterisation of Splash-2 and Parsec," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009, pp. 86–97.
[25] N. Nethercote and A. Mycroft, "Redux: A Dynamic Dataflow Tracer," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, pp. 149–170, Oct. 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066104810478
[26] W. Heirman *et al.*, "PinComm: Characterizing Intra-application Communication for the Many-Core Era," in *ICPADS 2010*, Dec. 2010.
[27] S. Ostadzadeh, "Quantitative Application Data Flow Characterization for Heterogeneous Multicore Architectures," Ph.D. dissertation, Delft University of Technology, Delft, Netherlands, December 2012.
[28] N. Nethercote and J. Seward, "How to Shadow Every Byte of Memory Used by a Program," in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, ser. VEE '07. New York, NY, USA: ACM, 2007, pp. 65–74. [Online]. Available: http://doi.acm.org/10.1145/1254810.1254820
[29] "An Implementation of BWA Sequence Alignment Algorithm." https://github.com/lh3/bwa.
[30] W. Cohen, "Multiple Architecture Characterization of the Build Process with OProfile," 2003. [Online]. Available: http://oprofile.sourceforge.net
[31] N. Nethercote, "Dynamic Binary Analysis and Instrumentation," Ph.D. dissertation, University of Cambridge, UK, Nov 2004.
[32] A. Pesterev *et al.*, "Locating Cache Performance Bottlenecks Using Data Profiling," ser. EuroSys, 2010, pp. 335–348.
[33] "vTune by Intel," http://software.intel.com/en-us/intel-vtune.
[34] "Purify by IBM," http://www-03.ibm.com/software/products/us/en/rational-purify-family.
[35] I. Ashraf *et al.*, "Memory profiling for intra-application data-communication quantification: A survey," in *IDT 2015*, Dec 2015.
[36] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data Race Detection in Practice," in *WBIA 2009*.
[37] J. Seward and N. Nethercote, "Using Valgrind to Detect Undefined Value Errors with Bit-precision," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247362
[38] C. Luk and et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI '05*. New York, NY, USA: ACM, 2005, pp. 190–200.
[39] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 237–248.
[40] Committee, T.I.S., "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2," May 1995.
[41] 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.
[42] R. B. Bunt and J. M. Murphy, "The measurement of locality and the behaviour of programs," *Comput. J.*, vol. 27, no. 3, pp. 238–253, Aug. 1984. [Online]. Available: http://dx.doi.org/10.1093/comjnl/27.3.238
[43] J. Weinberg *et al.*, "Quantifying locality in the memory access patterns of hpc applications," in *SC 2005*.
[44] J. Canny, "A Computational Approach to Edge Detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 8, pp. 679–698, November 1986.
[45] B. D. Lucas and T. Kanade, "An Iterative Image Registration Technique with an Application to Stereo Vision," 1981, pp. 674–679.
[46] J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-memory," *SIGARCH Comput. Archit. News*, vol. 20, no. 1, pp. 5–44, Mar. 1992. [Online]. Available: http://doi.acm.org/10.1145/130823.130824
[47] H. Li, "Aligning Sequence Reads, Clone Sequences and Assembly Contigs with BWA-MEM." http://arxiv.org/abs/1303.3997, 2013.
[48] Z. TM, "Zedboard," http://zedboard.org/product/zedboard.
[49] "Canny Edge Detector, Image Analysis Research Lab., USF," http://marathon.csee.usf.edu/edge/edge_detection.html.
[50] "KLT: An Implementation of the Kanade-Lucas-Tomasi Feature Tracker," http://www.ces.clemson.edu/~stb/klt/installation.html.
[51] R. L. Myers, *Display Interfaces : Fundamentals and Standards*. New York ; Chichester : Wiley, 2002.
[52] Y. S. Shao and D. Brooks, "ISA-independent workload characterization and its implications for specialized architectures," in *ISPASS*, April 2013.
[53] "OpenACC," https://www.openacc.org/.

Imran Ashraf is a Postdoctoral researcher at Delft University of Technology. He received his Ph.D. in Computer Engineering from Delft University of Technology, The Netherlands in 2016. The focus of his research was advanced profiling, code parallelization, communication driven mapping of applications on multicore platforms. Currently, Imran is working as Post-Doctoral Researcher at Quantum Computing Lab, QuTech, TU Delft. His recent research also focuses on compilation techniques for quantum computing.



Nader Khammassi did his PhD at the National Engineering School of Advanced Technology (ENSTA) in Brittany, the focus of his research was on High Performance Computing (HPC) and more specifically on parallel programming for multicore architectures. During his PhD, Nader worked as a research engineer in Thales Airborne Systems on several high performance signal processing applications in the Electronic Warfare domain and obtained his doctorate in 2014. In 2015, he joined the Quantum Computing Lab of QuTech at Delft University of Technology in the Netherlands. The current topic of Nader's research is the design of a scalable quantum computer architecture, in particular he is focusing on designing high performance simulation tools for quantum computers and the compilation of quantum algorithms for different quantum computing devices.

Mottaqiallah Taouil received the M.Sc. and Ph.D. degrees (both with Hons.) in computer engineering from the Delft University of Technology, Delft, the Netherlands. He is currently a Post-Doctoral Researcher with the Dependable Nano-Computing Group, Delft University of Technology. His current research interests include reconfigurable computing, embedded systems, very large scale integration design and test, built-in-self-test, and 3-D stacked integrated circuits, architectures, design for testability, yield analysis, and memory test structures.

Koen Bertels is Professor and Head of the Computer Engineering Laboratory at Delft University of Technology. His research focuses on heterogeneous multicore computing, investigating topics ranging from compiler technology, runtime support and architecture. He recently started working on quantum computing as a principal investigator in the Qutech research centre. He served as general and program chair for various conferences such as FPL, RAW, ARC. He co-authored more than 30 journal papers and more than 150 conference papers.