Advanced Profiling of Applications for Heterogeneous Multi-Core Platforms

Koen Bertels, S. Arash Ostadzadeh, Roel Meeuws
Computer Engineering Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, Delft, the Netherlands
Email: {K.L.M.Bertels,S.A.Ostadzadeh,R.J.Meeuws}@tudelft.nl

***Abstract*—The increased complexity of programming on multi-processors platforms requires more insight into program behavior, for which programmers need increasingly sophisticated methods for profiling, instrumentation, measurement, analysis, and modeling of applications. Particularly, tools to thoroughly investigate the memory access behavior of applications have become crucial due to the widening gap between the memory bandwidth/latency compared to the processing performance. To address these challenges, we developed the Q^2 profiling framework in the context of the Delft Workbench (DWB), which is a semi automatic tool platform for integrated hardware/software co-design, targeting heterogeneous computing systems containing reconfigurable components. The profiling framework consists of two parts, a static part which extracts information related to memory accesses during the execution of an application. We present an advanced memory access profiling toolset that provides a detailed overview of the runtime behavior of the memory access pattern of an application. This information can be used for partitioning and mapping purposes. The second part involves a statistical model that allows to make predictions early in the design phase regarding memory and hardware usage based on software metrics. We examine in detail a real application from the image processing domain to validate and specify all the potentials of the Q^2 profiling framework.**

I. Introduction

As computer manufacturers move increasingly beyond the traditional single-core platforms, system developers are confronted with an increasing number of complex architectures. There has already been great proliferation of multi-core architectures and reconfigurable devices. Currently, the trend in research is to utilize an increasing number of cores (many-core) and to mix different types of processing elements. These elements include GPPs, reconfigurable devices, GPUs, and ASICs, just to name a few. Such architectures not only require new tool-chains, libraries, and interconnects, among others, but also demand more insight into program behavior in order to take advantage of the characteristics of these heterogeneous systems. As a result, it is increasingly important to support developers in their analysis and understanding of different platforms and application behaviors. Particularly, tools to thoroughly investigate the memory access behavior of applications become crucial due to the widening gap between the memory bandwidth/latency and the processing performance. Furthermore, these heterogeneous architectures involve development of hardware blocks that can take much time to develop. Even in the case of automatic generation of the hardware, the actual synthesis of the design can take a long time. As a consequence, there is a clear need to have early and fast predictions of the hardware cost of the different parts of an application.

These two challenges are the main concern of the Q^2 profiling framework developed in the context of the Delft Workbench (DWB) [1]. DWB is a semi-automatic tool platform for integrated hardware/software co-design, targeting heterogeneous computing systems containing reconfigurable components. The profiling framework consists of two parts: a static part, which can provide estimates of some critical metrics in a small amount of time, and a dynamic part, which provides accurate measurements of some of those metrics.

The aim of this paper is to demonstrate the need for and the usage of such profiling tools. This will be done using a case study of a well known image processing application. The main contributions of this paper are the following:

- the introduction of the Q^2 Profiling Framework,
- a case study of the Q^2 framework on the Canny Edge Detection application.

The paper is structured as follows. First, Section II evaluates the related research and establishes the niche for our framework. Then, in Section III the direct research context of Q^2 is presented. The main exposition of the Q^2 framework can be found in Section IV. Subsequently, we present the case study of the Canny Edge Detection application in Section VII. Finally, Section VIII concludes the paper.

II. Related Work

As heterogeneous computing systems become increasingly more complex and demanding, utility tools turn out to be essential in assisting application developers to analyze and optimize the performance of applications. Developers require these tools to improve existing applications or drive the development of new applications for heterogeneous reconfigurable systems. Profiling tools focus on finding critical code regions and provide hints for optimizations. Critical code regions are conventionally associated with computational hot-spots or communication bottlenecks of an application. Potential optimizations for such code regions include: code revisions (memory usage reduction, FPGA area optimization, etc.), functional merging/splitting, or HW/SW partitioning; which all first need the identification of the critical region(s).

Dynamic profilers, in contrast with *static profilers*, examine an application during execution to provide information about the

run-time behavior of the application. Many dynamic profiling tools have been developed so far. General profilers, such as *gprof* [2], normally provide profiling information at function-level granularity. Furthermore, they do not distinguish between computation and communication times. On the other hand, there are some profilers that focus on more detailed levels, such as, statement-level, block-level, or loop-level [3]. Some profilers, such as Intel's *vTune* [4] and AMD's *CodeAnalyst* [5], integrate hardware event monitoring in addition to software-based sampling or instrumentation techniques. There also exist profiling tools, which are specific for particular architectures. As an example, SpixTools [6] is a collection of programs that allow profiling of applications at different levels of granularity for the Sun Microsystems SPARC architectures. *QPT* [7], is dynamic profiler and tracing system that rewrites the executable file of an application by inserting extra code to record the execution frequency or sequence of each basic block. The execution cost of functions in the program can be extracted from this information. Unlike *gprof*, *QPT* records the exact execution frequency, not a statistical sample. When tracing a program, *QPT* produces a trace regeneration program that reads the highly compressed trace file and regenerates a full program trace.

Apart from general profilers, which mostly aim to provide execution time related information, there is a different class of profilers that focus on resources, such as the memory system, rather than on computation. MemSpy [8] instruments applications with Tango [9], an execution-driven simulator, by putting calls to the memory simulator for each memory reference associated with dynamically allocated memory or explicitly-identified address ranges. The CPROF [10] system is a cache performance profiler that annotates source code to identify the source lines and data structures that cause frequent cache misses. The Memory Trace Visualizer (MTV) [11] is a tool that provides interactive visualization and analysis of the sequence of memory operations performed by a program as it runs. It uses visual representations of abstract data structures, a simulated cache, and animating memory operations. *ProfileMe* [12] takes samples of instructions as they move through an out-of-order issue pipeline and provide some statistical reports, such as, cache miss rates. *Cacheprof* [13] is a memory simulator that annotates each memory access instruction and links a cache simulator into the resulting executable. During the execution, all data references are intercepted and sent to the simulator. It is able to report on the number of memory references and the number of misses for each line of the source code.

When mapping candidate regions onto reconfigurable components, detailed evaluation of the required resources, power consumption, and speed-up are also essential. Over the years, many hardware performance estimation schemes have become available. Part of these schemes drive low-level design processes [14], [15]; for example, in the placement and route phases [16]. However, in the early stages of design, normally, there are no low level description available and other more high-level estimation schemes are required. Therefore, others, including our estimation scheme, operate on a HLL description such as C [17], [18], [19]. These approaches allow for early estimation of resources, and as such can help designers to tailor their code for heterogeneous platforms at an early stage. However, most of the high-level methods focus either on a specific application domain [18] or on a specific kind of design [19], [20], while other high-level methods are only applicable to certain types of platforms [17], [14]. For example, [18] is tailored for the SA-C language, which is a restricted C-dialect for the Multimedia domain. Furthermore, this approach was embedded in the SA-C compiler and, as such, cannot be simply assumed to be valid for other compilers or platforms. These issues reduce the applicability of this approach for more general use.

Most works base their claims on the quality of their models in terms of error on validation sets containing between four [18] to twelve [19] kernels. *However, if the target is to report errors that are not biased to a small data set, a larger set of validation data, such as the one used in this paper, becomes vital.* Let's suppose, for example, one validates a model using a set of only 12 kernels. It is very unlikely that these kernels can represent the whole spectrum of possible kernels. For one, with such a limited set of kernels the possibility of cherry-picking your validation set increases. Secondly, an anomalous kernel that is not well modeled by a certain model can adversely affect the validation error when one uses a small set of kernels.

III. The Research Context

The Q^2 profiling framework is developed in the context of the Delft Workbench (DWB) [1]. The DWB is a semi-automatic tool platform for integrated hardware/software co-design, targeting heterogeneous computing systems containing reconfigurable components. It targets the Molen machine organization [21], an architectural template for heterogeneous reconfigurable platforms developed at the Delft University of Technology. This architectural template adheres to the Molen programming paradigm [22]. This paradigm is a sequential consistency paradigm for programming reconfigurable machines.

The DWB addresses the entire design cycle from profiling and partitioning to synthesis and compilation of an application and it focuses on four main steps within the entire heterogeneous system design, namely:

- *the code profiling and the cost modeling* [23], [24], [25];
- *the graph transformations and optimizations* [26], [27], [28];
- *the retargetable compiler* [29]; and
- *the VHDL generation*[30].

For a given application, *code profiling and cost modeling* identify which parts of the application are good candidates for hardware implementation. This decision takes into consideration the available hardware resources and the speed-up provided by the hardware implementation of the application, or parts of it, versus a software implementation. *Graph transformations and optimizations* analyze the candidate parts of the application for hardware implementation to find out if the code segments can be clustered/partitioned according to various targets as, for example, hardware resource sharing. Next, an *optimization* phase is performed to spot parallelization opportunities.

After making the decision of which parts of the code segments to implement in hardware, the code is annotated. Subsequently, the *retargetable compiler* generates the new object code, which contains the call to the reconfigurable hardware for the selected segments of code. Finally, the identified instructions (code segments) pass through a *VHDL generation* phase, which generates hardware description of the instructions.

IV. The Q^2 Profiling Framework

The focus of this work is on code profiling. Figure 1 depicts in detail the Q^2 profiling framework, which is part of the DWB platform. We distinguish between static and dynamic profiling paths. Static profiling can provide estimates in a small amount of time, whereas dynamic profiling provides accurate measurements of several aspects, such as, execution time estimates and memory access intensity of individual functions in an application. The static profiling path first extracts code characteristics from the application source code. These characteristics, or software complexity metrics, are then used by the *Quipu* model to make fast and early predictions of components with specific implementation details like FPGA area or speed-up. Based on the code characteristics and *Quipu* estimates, a conjecture is made on which kernels are interesting for further analysis regarding hardware implementation. The dynamic profiling path focuses on run-time behavior of an application and, therefore, is not as fast as the static profiling. Furthermore, the dynamic profiling requires the application binaries and representative input data in order to provide relevant measurements. This, in itself, can raise some concerns, if the application acts quite distinctively in different circumstances. However, that rarely happens in practice. The common *gprof* [2] profiler is utilized to identify application hot-spots in terms of execution cost and to recognize frequently executed functions. *gprof* also provides execution timing estimates, which can only be useful for applications running for relatively large period of time. On the other hand, *MAIP* provides accurate measurements for the contribution percentage of individual functions with respect to the whole execution cost of the application. It also distinguishes between memory access related and computation related operations. The *QUAD* [23] core module provides a comprehensive quantitative analysis of memory access behavior of an application with the primary goal of detecting actual data dependencies at the function-level. The tracing modules implemented in the *QUAD* core is utilized in the *cQUAD* tool to reveal the data communication pattern of a pair of cooperating functions in an application. The *xQUAD* [31] tool provides detailed, fine-grained memory access information for each function in the application. The information includes runtime memory usage statistics regarding individual data objects defined in an application source code. The *tQUAD* [24] tool reveals the memory bandwidth usage of each function in terms of relative execution timings.

V. The *Quipu* Modeling Approach

The static profiling part in this paper consists of the *Quipu* modeling approach, which we have developed and presented in [32], [33], and [34]. The approach is generic and not limited to any particular platform or tool-chain by allowing the generated models to be recalibrated for different tools and platforms, contrary to the majority of the existing techniques. Furthermore, a major strength of *Quipu* models is their linear nature. Although, the statistical techniques to create the models may be very time-consuming, the resulting prediction model requires only a number of multiplications in addition to parsing of the source code. This allows for integration of *Quipu* models in highly iterative design processes where estimates are recalculated many times in a short time period. Additionally, as *Quipu* models are based on measurements from C code, very early predictions become possible, allowing designers to take important decisions on hardware mapping at an earlier stage.

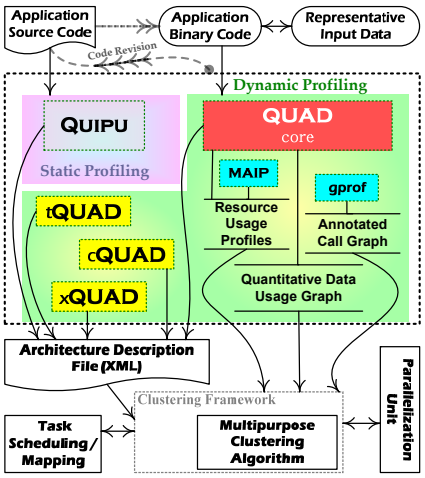

Fig. 1. The Q^2 Profiling framework within the DWB tool platform.

The *Quipu* models that we utilize in this paper are generated for a combination of the DWARV C-to-VHDL compiler [35] and the Xilinx ISE Synthesizer for the Virtex 5 FX 130T FPGA. Notwithstanding, *Quipu* can calibrate models for other combinations as well, such as, an Altera Stratix IV FPGA combined with the C-to-Verilog compiler from the Haifa University [36], [37]. The output data of a combination of tools and platform is used to calibrate a specific model instance. Because the set of calibration data can vary depending on certain tool options, the user may want to consider generating models for each option value. For example, area measurements will be much lower when optimizing for area compared to when optimizing for speed.

In the following, we first define the models and the criteria. Subsequently, we discuss the key components of the *Quipu* approach. Last but not least, the newly developed techniques that constitute our approach are discussed in detail.

A. The Models and the Criteria

It is essential to quantify the characteristic aspects of the software description at hand, when we consider the modeling of hardware from software descriptions. In [33] and [34], we have introduced Software Complexity Metrics (SCMs) as a way to address this. SCMs are indicators of specific characteristics of the software code. Examples of SCMs are the number of operators,

the number of loops, or the cyclomatic complexity, but also more complex metrics involving data-flow analysis and the attribution of operations to functional units. Currently, we use a set of 58 SCMs as a base for our model. We refer the reader to our previous work, for more information on many of the implemented SCMs. [33], [32]

To characterize hardware performance in terms of area, interconnect, power, and other parameters, we measure and predict 49 different hardware performance indicators. For instance, the number of slices, the number of wires, the number of LUTs, and the number of clock wires. Most of these parameters are related to interconnect and provide specific information on wires, nets, route-throughs, or switch-boxes, further subdivided for logic, power, or clock resources.

Given these criteria, we look for a model that describes the relation between hardware and software:

$$y_{HW} = g(\bar{\mathbf{x}}_{\mathbf{SCM}}) + \epsilon. \tag{1}$$

This is the theoretical optimal model relating some hardware metric y_{HW} to a vector of SCMs $\bar{\mathbf{x}}_{\mathbf{SCM}}$ with the ideal relation $g(\cdot)$ and some error ϵ inherent to the problem at hand. In practice, an ideal model cannot be found. Instead, any modeling scheme is an approximation to some level. Therefore, we model the relation $g(\cdot)$ with an approximated relation $\hat{g}(\cdot)$. This results in the introduction of some estimation error $\hat{\epsilon}$ inherent to our approximation scheme:

$$\hat{y}_{HW} = \hat{g}(\bar{\mathbf{x}}_{\mathbf{SCM}}) + \hat{\epsilon}. \tag{2}$$

The approximation $\hat{g}(\cdot)$ can be, for example, an ad-hoc model, a Linear Regression Model (LRM), or an Artificial Neural Network (ANN). In case of LR techniques, $\hat{g}(\cdot)$ is a linear equation:

$$\hat{y}_{HW} = \hat{\mathbf{a}}\bar{\mathbf{x}}_{\mathbf{SCM}} + \hat{b} + \hat{\epsilon}, \tag{3}$$

where $\hat{a}$ is a vector of estimated coefficients $\hat{a}_i$ corresponding to element x_i of the vector of SCMs $\bar{\mathbf{x}}_{\mathbf{SCM}}$ obtained for some kernel that corresponds to the estimated hardware metric $\hat{y}$, and $\hat{b}$ is the offset of the linear model. Note, that these variables are stochastic variables. *This means that reporting a simple percentage error is not enough. As a result, the characterization of the error distribution must be addressed as well.*

B. The Tools and the Kernel Library

Quipu consists of a set of tools and a kernel library, as depicted in Fig. 2. In the modeling flow, *Quipu* extracts SCMs and hardware performance indicators from a kernel library. *This is a library of 178 kernels from a wide variety of application domains, contrary to many existing techniques, which use libraries of tens of kernels at most.* This allows us to build models that are generally applicable. It is also possible to build domain-specific models by using, for example, only the 57 Cryptography-related kernels out of the 178 kernels considered. An overview of the kernels in this library is provided in Table I.

Regarding the tools in the *Quipu* approach, we have:

Domain	Kernels	Size[a]	Bit-Based	Streaming	Control
Compression	12	47.6 (14-95)	x		x
Cryptography	57	192.2 (15-1107)	x	x	some
DSP	12	32.3 (10-110)	x	x	some
ECC	13	74.8 (10-496)	x	x	x
Mathematics	29	33.9 (5-100)			
Multimedia	42	81.8 (6-1107)	some	x	x
General	13	72.9 (22-163)			x
Total	178	102.6 (5-1107)			

[a]avg. size in number of statements (range).

TABLE I
OVERVIEW OF THE KERNEL LIBRARY, THE NUMBER OF KERNELS AND THEIR MAIN ALGORITHMIC CHARACTERISTICS IN EACH APPLICATION DOMAIN.

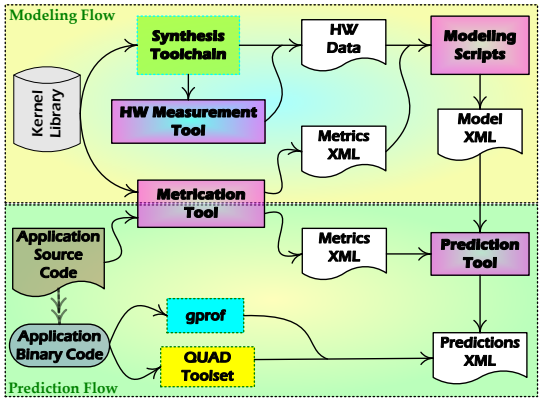

Fig. 2. An overview of the *Quipu* modeling approach, its tools (the boxes with thick border), and the accompanying tools (the boxes with dashed borders).

1) The Metrication tool: The 58 different SCMs can be extracted using the *Quipu Metrication tool*, which produces an XML file containing SCM measurements for each kernel. This tool is written as an engine in the CoSy compiler system [38] developed by ACE Associated Compiler Experts. This comprehensive compiler contains a large set of optimizations and is easily extensible by writing engines that can be plugged into the framework. Using the existing engines of the CoSy framework, different optimizations can be performed before measuring the SCMs. This flexibility allows to choose specifically a set of optimizations that fit a target tool-chain, which in turn adds to the generic character of the *Quipu* modeling approach. Apart from being an essential part of the *Quipu* modeling framework, the metrication tool can also be useful within an Software Measurement framework that helps drive management of software development processes.

2) The Hardware Measurement tool: The *Quipu hardware Measurement tool* measures 49 different hardware parameters, such as area and interconnect, from synthesized hardware targeted at Xilinx FPGAs. The tool keeps track of all nets and components in the design by processing the XDL file generated by the Xilinx synthesis tool-chain. It provides detailed interconnect measurements, such as the number of clock wires or the number of power nets. For Altera FPGAs, such a tool is not needed, as the

report files generate detailed numbers on the interconnect usage automatically.

3) The Modeling Scripts and the Prediction tool: The gathered SCMs and hardware measurements are run through a set of modeling scripts that automatically evaluate different modeling techniques. The output model XML file can be used in the prediction flow, where, based on SCM inputs, the *Quipu prediction tool* provides estimates of any required hardware aspects. All intermediate files in the prediction flow are saved in XML format for an easy integration. For example, the results of execution and memory profiling tools might be integrated as depicted in Fig. 2.

C. Utilization of Quipu models

The *Quipu* models can be used in several contexts. In the first place, developers can use the predictions for function kernels they are working on in order to find problems with potential placement on hardware. If a kernel is predicted to have a large amount of slices on a particular platform, the developer might move to split the function in several parts, or address the root cause for the large area (e.g. a large local array). Secondly, a tailored *Quipu* model may drive the optimization pass in a hardware compiler by predicting the hardware size of the contained basic blocks at different unroll factors. In this way, the hardware compiler can automatically choose a beneficial unroll factor. Thirdly, a high-level HW-SW partitioning algorithm could use a *Quipu* model to evaluate many different partitionings at an early stage of the development.

VI. QUAD Dynamic Memory Profiling Toolset

Nowadays, Dynamic Binary Instrumentation (DBI) is gaining popularity among the available methods for intercepting memory accesses. DBI can be used to develop dynamic profilers. The dynamic part of the Q^2 profiling framework falls under this category of profilers. The tools are implemented as Dynamic Binary Analysis (DBA) tools using the Pin [39] DBI platform. In this section, we first present a brief overview of the Pin DBI platform and then describe the individual tools developed in the dynamic profiling framework of the DWB and discuss their potentials.

A. Pin Dynamic Binary Instrumentation

Run-time instrumentation is a technique for injecting extra user defined code into an application during its execution to study the behavior of the application. The primary advantage of this kind of instrumentation is that, in order to profile an application, we only need the binary executable code of the application. Furthermore, Pin adopts the dynamic compilation technique that uses a Just-In-Time (JIT) compiler to (re)compile and instrument the application code on the fly. This capability provides the benefits of portability, transparency and efficiency to the end user. In summary, Pin supplies a fast instrumentation system, which is able to work with unmodified executables in addition to the preservation of the application's original uninstrumented behavior.

Pin provides a DBI platform for building a variety of DBA tools for multiple architectures, namely, the IA32, 64bit x86, Itanium®, and ARM architectures. A primary key feature of the Pin DBI platform is instrumentation transparency. It means that Pin preserves the original behavior of the application. As a result, the application uses the same addresses (both instruction and data) and the same values (both register and memory) as it would do in an uninstrumented execution. Furthermore, Pin does not modify the application stack, as some applications may deliberately reference memory addresses beyond the top of the stack.

Generally, two types of routines are defined in Pin-based tools, namely, *instrumentation* routines and *analysis* routines. Instrumentation routines determine where, in the application code, to place calls to analysis routines. Analysis routines are customizable by the user and they are called while the program executes. The arguments to analysis routines can be, for example, the instruction pointer, the effective memory address of the instruction, the memory or stack value, the address of branch instruction, the system calls values, and others. The actual instrumentation is performed by the JIT compiler. Pin intercepts the very first instruction of the application and recompiles the executable generating *basic blocks* code starting at that instruction, and instrumenting the code according to the specified instrumentation type. The generated code sequence is almost identical to the original one, except that it runs under the control of Pin. When a branch exits a basic block, Pin generates more basic blocks code for the branch target and it continues the execution. The JIT generated code and its instrumentation are saved in a code cache for future execution of the same sequence of instructions to improve performance.

Instrumentation with Pin can be done at different levels of granularity. The finest level is instrumentation at the *instruction level*, i.e. instrumenting the application one instruction at a time. It is also possible to instrument code at the *trace level*[1], at the *procedure level*, and at the entire *image level*.

The execution of an instrumented application usually shows a considerable slowdown. This depends on the nature of the instrumented application, as well as on the overhead caused by the analysis routines in the tool. It appears that most of the slowdown is caused by the execution of the code, rather than on-the-fly code compilation (which includes the insertion of the instrumentation code). In Pin, some performance improvements are done during the compilation phase of the application. Improvements are on register reallocation, inlining, liveness analysis, and instruction scheduling. This results in an instrumented code, which run very fast compared to other DBI platforms.

B. The QUAD Tools

The *QUAD* toolset consists of several related tools developed to demonstrate a comprehensive overview of the memory access behavior of an application, as well as, to provide fine-grained detailed memory access related statistics. The *QUAD* core module has been designed and implemented as the primary component to provide useful quantitative information about

[1]A trace is defined as a straight-line sequence of instructions executed sequentially. Pin guarantees that traces only enter at the top, but may have multiple exits.

the data dependence between any pair of cooperating functions in an application. Data dependence is estimated in the sense of producer/consumer bindings. More precisely, *QUAD* core reports which function is consuming the data produced by another function. The exact amount of data communication and the number of Unique Memory Addresses (UnMA) used in the communication process are also calculated. The *QUAD* core contains a fast and efficient Memory Access Tracing (MAT) module, which detects and traces all the memory references made during an application execution. It acts as a kind of shadow memory for the whole memory access space. Considering the fact that *MAT* structures the shadow memory in such a way to make tracing as fast as possible, the size of the space overhead can be very huge. The tracing mechanism is also utilized by *cQUAD* tool to monitor in detail each and every data transfer event occurring between a pair of communicating functions. This data communication can be viewed as a dedicated virtual channel for transferring data items from the producer side to the consumer end. The Data Communication Channel Pattern Detector (DCCPD) module thoroughly analyzes the extracted raw profile data to compute several critical metrics that can classify and describe the pattern of the communication between the two functions. These metrics include the interleaving balance factor, spatial/temporal localities and data communication pattern complexity. In general, these information is required to reveal the coupling intensity and regularity between the communicating functions in an application. It can be very useful in understanding the behavior of the functions with regard to their data dependencies and requirements, as well as, in mapping and scheduling potential parts of the application onto reconfigurable devices.

The *xQUAD* extension to the *QUAD* toolset augments the memory access analysis of the application by providing a very detailed, fine-grained intra-function information. The information is provided based on the application source code data object granularity. In this respect, the programmers can utilize the information to fine-tune the application behavior regarding it memory access references. The main motivation for this extension is that a coarse view of the intra-function data access makes it difficult for the application developers to attribute the extracted information to particular user-defined data objects in an application at the source code level. Hence, fine-grained code revision/optimization becomes a burdensome task. The Pin DBI framework does not provide API functions for retrieving data objects information. Therefore, source-level information about data objects should be extracted directly from the binary file(s). The *DWARF* module is utilized to extract low-level source code representations, such as, information about source code types, function and object names, and line numbers. *tQUAD* is another component in the *QUAD* toolset. It extracts the timing information of the functions, as well as, their memory bandwidth usages during the application execution. The information extracted by *tQUAD* can lead to the recognition of the main *execution phases* within the application, which in the end can be used to identify the related functions in each phase. Each execution phase ordinarily corresponds to a specific and well-defined (sub)task within the application process, which is based on the high-level algorithm of the application source code. The logical steps of the algorithm are clear for the programmers. However, the information provided by *tQUAD* can help users to have a more solid vision of what are the steps the application actually takes to perform its task. In addition, this information can also be regarded as valuable hints for the application developers to understand the (re)usability scope of the functions defined in the application, and probably serve as further optimization directives. As an example, a general function can be used in different phases of the application, which makes it a good candidate for replacement with several specific-purpose tailored functions.

Figure 3 illustrates the architectural overview of the QUAD toolset along with the Pin DBI components. At the highest level in the Pin software layer, there is a Virtual Machine (VM), a code cache, and an instrumentation API. The VM consists of a JIT compiler, an emulator, and a dispatcher. After Pin gains control of the application, the VM coordinates its components to execute the application. The JIT compiles and instruments the application code, which is then launched by the dispatcher. The compiled code is stored in the code cache. Entering (leaving) the VM from (to) the code cache involves saving and restoring the application register state. The emulator interprets instructions that cannot be executed directly. It is used for system calls that require special handling from the VM. Since Pin does not reside in the kernel of the operating system, it can only capture user-level code.

The main component inside the *QUAD* core is the MAT module, which is responsible for building and maintaining the dynamic trie [40] data structure to provide relevant memory access tracing information. As Figure 3 shows, three binary objects are present when an instrumented application is running: the to-be-profiled application, Pin, and the *QUAD* toolset. Pin is the engine that instruments the application. The *QUAD* toolset contains instrumentation and analysis routines and it is linked with a library that allows *QUAD* to communicate with Pin.

To the best of our knowledge, *QUAD* is the first profiling toolset that focuses on examining the run-time memory access behavior of an application and on providing some valuable quantitative information. Even though *QUAD* toolset can be employed to spot coarse-grained parallelism opportunities in an application, it practically provides a more general-purpose facility that can be utilized in various heterogeneous reconfigurable systems optimizations by estimating effective memory access related parameters.

MAIP. The Memory Access Intensity Profiler (*MAIP*) is developed as a standalone Pin-based DBA profiler. *MAIP* aims to provide some unprocessed basic data for each function in an application with the main objective of revealing the intensity of memory access operations. Furthermore, *MAIP* can act as an enhanced alternative to the traditional *gprof* profiler, allowing accurate measurements for applications with a small running time. The problem with *gprof* is that the derived run-time figures are based on a sampling process. As a result, besides being subject to statistical inaccuracy, if a function runs for only a

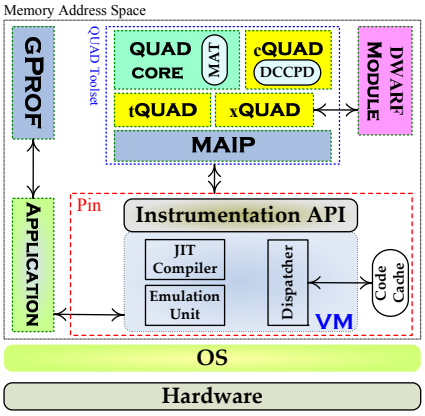

Fig. 3. Architectural overview of the dynamic part of the Q^2 profiling framework.

small amount of time, there is a pretty good chance that the profiler actually overlooks that function in action. Only if the total run-time of an application is large enough, a small run-time value for a function discloses that the function uses an insignificant fraction of the application's whole execution time. Otherwise, no valuable information can be gained from the *gprof* analysis at all. The extracted raw data by *MAIP* can be further processed to get some valuable information, such as, the ratio of memory access instructions to the total executed instructions in a program.

The basic data that are measured by the profiler are classified into three main classes: instructions, operands, and bytes. Each class is subsequently divided into Read/Write and Stack/Non-stack sub categories. A variety of parameters can be computed to measure the intensity of the memory access operations within an application by using a precise memory access tracking module, which intercepts and inspects every memory access instruction for detailed information. Specifically, *MAIP* extracts the following data for each function:

- *The total number of instructions executed within each function call.*
  If there is more than one call instance for a particular function, the aggregate value of this parameter is used in subsequent processing to have a more accurate estimation.
- *The total number of memory access instructions executed within each function call.*
  As before, if there is more than one call instance for a particular function, the aggregate value of this parameter is used in subsequent processing.
- *Memory Access Ratio (MAR)*
  This is the percentual ratio of the total number of memory access instructions to the total number of instructions.
- *NLOC-MAR*
  This is similar to the previous parameter. However, here we only consider memory access instructions within the heap and global data regions. This parameter tends to provide a more accurate estimation of the MAR, when the cost of local memory access is considered to be low compared to the expensive external memory access.
- *Memory Operand Ratio (MOR)*
  This is the percentage ratio of the total number of memory access operands to the total number of operands.
- *NLOC MOR*
  This is similar to the previous parameter. However, here we only consider the operands of the memory access instructions referencing the heap and the global data regions.
- *Stk Ratio*
  This is the percentage ratio of the total number of memory access instructions referencing the stack region to the total number of memory access instructions.
- *Flow Ratio*
  This is the total number of bytes read minus the total number of bytes written divided by the total number of bytes accessed. An extreme value of -1 means a write-only function, +1 represents the counterpart read-only function, and 0 represents a balanced R/W inert function.
- *NLOC-Flow Ratio*
  This is similar to the previous parameter. However, here we only consider memory accesses referencing the heap and the global data regions.
- *Byte-wise-Stk ratio*
  This is the percentage ratio of the total number of bytes accessed by the memory access instructions referencing the stack region to the total number of bytes accessed.
- *Bytes/Acc. ratio*
  This indicates the average value of the number of bytes accessed within each memory access instruction.

In the interpretation of these parameters, it should be generally noted that in some architectures, a single memory operand can be both read and written, for instance incl (%eax) on IA-32. In this case, we instrument it once for read and once for write. The same holds for combined R/W instructions.

VII. Case Study

In this section, we present a detailed analysis of an image processing application, i.e. Canny Edge Detection [41], to demonstrate the potentials of the advanced profiling techniques developed within the DWB platform. It serves as the direct approach to validate the usefulness, efficiency and applicability of the tools presented in this research work. *The main objective of the case study is to have an early yet comprehensive and thorough understanding of the application behavior, in particular of its memory access behavior and requirements.* Although the focus of the analysis is on the run-time attributes, we also examine the application source code to extract some valuable information. The result of this detailed profiling is primarily utilized to spot bottlenecks and deficiencies, particularly related to the application memory usage. It will provide hints for application developers to revise/optimize the application source code in order to gain better performance when running the application on a particular heterogeneous reconfigurable architecture. Throughout the experimental analysis, we used the same strategy and conducted several phases of source code optimizations as a means of verification of the profiling data. The extracted information can be further utilized in critical decisions during the design space

exploration in heterogeneous multiprocessor systems, such as, HW/SW task partitioning, mapping and scheduling.

In this case study, we specifically aim to demonstrate the following qualities of the Q^2 framework:

- the visualization of the data communication between different kernels in the application,
- the prediction of hardware resource consumption for these kernels,
- the value of the generated data in identifying candidates for migration to reconfigurable components.

In the following, we will first discuss the Canny Edge Detection method in Section VII-A. Then, in Section VII-B, we will present our experimental setup. Finally, the results of the different steps in our case study are discussed in Section VII-C.

A. Canny Edge Detection

Canny [41] is a well-known edge detection algorithm. The method aims to achieve three main goals:

- *good detection* - this translates to the detection of as many of the real edges as possible, while also not falsely detecting non-existing edges as much as possible.
- *good localization* - this denotes the fact that the detected edges are as close as possible to the actual edges, i.e. the distance between the edge pixels as found by the detector and the actual edge is to be minimal.
- *unique detection of edges* - this means that real edges should be detected only once. This aspect of the detection method was added because the previous criteria do not imply that edges are only identified once.

Based on these criteria, the Canny edge detector first smoothes the image to eliminate any noise. It then finds the image gradient to highlight regions with high spatial derivatives. The algorithm then tracks along these regions and suppresses any pixel that is not at the maximum (non-maximal suppression). The gradient array is then further reduced by hysteresis, which tracks along the remaining pixels that have not been suppressed so far. Hysteresis uses two thresholds to accomplish this task. If the magnitude is below the first threshold, it is set to zero (made a non-edge). If the magnitude is above the high threshold, it is made an edge. In case the magnitude is between the two thresholds, then it is set to zero, unless there is a path from this pixel to a pixel with a gradient above the second threshold.

For our experiments, we have used the implementation provided by the Computer Vision Laboratory at the University of South Florida [42]. The application has the following steps:

- **Step 1**. *Filtering out any noise in the original image.* The Gaussian filter is used exclusively due to its simplicity. Once a suitable mask has been calculated, the Gaussian smoothing can be performed using standard convolution methods. A convolution mask is usually much smaller than the actual image. As a result, the mask is slid over the image, manipulating a square of pixels at a time. lower is the detector's sensitivity to noise. The localization error in the detected edges also increases slightly as the Gaussian width is increased. The width of the Gaussian mask used in the implementation is determined based on the standard deviation of the Gaussian smoothing filter that should be input by the user.
- **Step 2**. *Finding the edge strength.* This is done by taking the gradient of the image. The Sobel operator performs a 2D spatial gradient measurement on an image. Then, the approximate absolute gradient magnitude (edge strength) at each point is found. The Sobel operator uses a pair of 3×3 convolution masks, one estimating the gradient in the x-direction and the other estimating the gradient in the y-direction.
- **Step 3**. *Applying non-maximal suppression.* After finding the edge directions using the gradient values, non-maximal suppression is used to trace along the edges and suppress any pixel value that is not considered to be an edge. This will result in a thin line in the output image.
- **Step 4**. *Performing hysteresis.* Hysteresis is used to eliminate the breaking up of an edge contour caused by the edge pixels fluctuating above and below a threshold. Thresholding with hysteresis requires a low and a high threshold. Making the assumption that important edges should be along continuous curves in the image allows to follow a faint section of a given line and to discard a few noisy pixels that do not constitute a line but have produced large gradients. The low and high threshold values for hysteresis should also be specified as input parameters by a user.

B. Experimental Setup

All the experiments were performed on two different platforms. The general profiling of the CED application with *gprof* was done on an Intel 32-bit Core2 Duo E8500 @3.16 GHz with 4GB of RAM, running Linux kernel v2.6.34.7-0.7-pae. The application source code was compiled with *gcc* v4.5.0 and without any compiler optimization. We also utilized *gprof* on the embedded PowerPC 440 @400 MHz with 512 MB DRAM, which is integrated in a Xilinx ML510, Virtex 5 FX 130T with 1.3 MB BRAM FPGA board. In order to profile the application using the *QUAD* toolset, Pin DBI framework is needed which does not support PowerPC architecture. As a result, the *QUAD* profiling information on Intel x86 can demonstrate some level of inaccuracy for the architecture-specific data when targeting a different architecture. However, the overall image of the application should stay similar. The *Quipu* predictions target the Virtex 5 platform. For other FPGA devices, conversion formulas should be used based on the authentic published data-sheets.

C. Experimental Analysis

We have investigated the Canny Edge Detector (CED) application in several phases and report here the results of the different analysis steps that were taken. First, we present some conventional profiling analysis. Afterwards, we present the results of using the Q^2 profiling framework. Finally, we make some conjectures on how to adjust the program based on these data.

TABLE II
GPROF FLAT PROFILE FOR THE *CED* APPLICATION ON THE INTEL X86 ARCHITECTURE.

Kernel	%time	self seconds	calls	self ms/call	total ms/call
gaussian_smooth	70.11	0.0985	1	98.50	98.50
non_max_supp	12.46	0.0175	1	17.50	17.50
magnitude_x_y	6.41	0.0090	1	9.00	9.00
apply_hysteresis	4.63	0.0065	1	6.50	11.50
follow_edges	3.56	0.0050	1433	0.00	0.00
derrivative_x_y	2.85	0.0040	1	4.00	4.00
canny	0.00	0.0000	1	0.00	140.50
make_gaussian_kernel	0.00	0.0000	1	0.00	0.00
read_pgm_image	0.00	0.0000	1	0.00	0.00
write_pgm_image	0.00	0.0000	1	0.00	0.00

% time is the percentage of the total execution time of the program used by the function; ***self seconds*** is the number of seconds accounted for by the function alone; ***calls*** is the number of times a function is invoked; ***self ms/call*** is the average number of milliseconds spent in the function per call; ***total ms/call*** is the average number of milliseconds spent in the function and its descendants per call.

TABLE III
GPROF FLAT PROFILE FOR THE *CED* APPLICATION ON THE EMBEDDED PPC.

Kernel	%time	self seconds	calls	self s/call	total s/call
gaussian_smooth	69.09	5.79	1	5.79	5.79
non_max_supp	19.81	1.66	1	1.66	1.66
magnitude_x_y	5.61	0.47	1	0.47	0.47
derrivative_x_y	3.70	0.31	1	0.31	0.31
apply_hysteresis	0.95	0.08	1	0.08	0.14
follow_edges	0.72	0.06	1433	0.00	0.00
canny	0.00	0.00	1	0.00	8.37
make_gaussian_kernel	0.00	0.00	1	0.00	0.00
read_pgm_image	0.00	0.00	1	0.00	0.00
write_pgm_image	0.00	0.00	1	0.00	0.00

Conventional profiling. We utilized *gprof* in order to have an approximate general understanding of the original CED application. The CED implementation consists of three source files with, in total, 15 functions. As mentioned earlier, three input parameters are required for the application to run. For all the experiments, we used a sample gray-scale image with a resolution of 800×600 and 8 bits per pixel in the *PGM* format. The input parameter for the standard deviation of the Gaussian blur kernel is set to 2.0. The values of the low and the high thresholds for performing hysteresis are both set to 0.5.

We used *gprof* for a typical run of the CED application without any compiler optimizations. The reported numbers demonstrate a considerable amount of error and do not reveal much valuable information. This is due to the fact that the application runs for a quite short time (approximately 150 milliseconds). Considering the sampling period which is 10 milliseconds, functions do not get much chances of being examined by the profiler. As mentioned previously, this is the main problem with *gprof*. To alleviate the problem, we run the application 20 times and recorded the average values. Table II summarizes these results. As seen in Table II, all the functions are called once with the exception of *follow_edges*, which is a recursive function. The number of times *follow_edges* is called depends on the image itself and on the values of input parameters. The primary share of the total execution time is attributed to *gaussian_smooth*. It is also interesting to note that there is no self contribution for *canny*, while the total contribution of this function and its descendants is around 140 milliseconds, which is equal to the sum of contributions from *gaussian_smooth*, *non_max_supp*, *magnitude_x_y*, *apply_hysteresis*, and *derrivative_x_y*. It indirectly implies that *canny* is the main function doing all the processing by just calling individual functions to carry out different phases in the edge detection algorithm described before. The time taken for reading the input image file and writing the output data is negligible.

Table III presents the *gprof* profiling results on the embedded PowerPC. As seen in the table, the total execution time of the application is increased by approximately 60x due to the decrease in the processing speed and simulated floating point arithmetic. Hence, the length of the application execution is large enough to get somehow accurate data with the sampling technique. Some changes are evident in the contribution percentages and the ordering of the functions. *follow_edges* and *apply_hysteresis* each now contributes less than 1 percent of the whole execution time.

A brief inspection of the application reveals the links between the main conceptual steps described in the CED algorithm and the corresponding functions defined in the source code. The top kernel, *gaussian_smooth*, utilizes the Gaussian filter to blur the input image. The filter itself is created in *make_gaussian_kernel*, which only allocates and fills a one-dimensional floating array. The size of the array is dependent on the input parameter *sigma* (standard deviation of the filter). However, for a predefined sigma value, there is a possibility of hard-wiring the individual calculated values. It is clear that the first step of the algorithm is the most time consuming task. The blurring procedure is performed on each pixel in the input image. *derrivative_x_y* computes the first derivatives (gradient) of the image along both the x and y directions. Subsequently, *magnitude_x_y* calculates the magnitude of the gradient. These functions together relate to the second step of the CED algorithm. Step 3 of the algorithm is implemented by *non_max_supp* which applies non-maximal suppression to the magnitude of the gradient image. Finally, *apply_hysteresis* implements the last step in the CED algorithm. Basically, the function finds edges that are above a high threshold or connected to a high pixel by a path of pixels greater than the low threshold. This is done by first initializing the edge map with all the possible edges that the non-maximal suppression suggested, except for the borders. Then, when a pixel is located above the high threshold, the function calls the recursive function *follow_edges* to continue tracking the edge along all paths.

In order to have accurate contribution estimates and also an overview of some memory access related statistics, we used *MAIP* to profile the application. Table IV presents a summary of the results. The most accurate values for the execution contribution of each function is calculated with *MAIP*, as it accounts for each single instruction within a function, contrary

TABLE IV
MAIP FLAT PROFILE FOR THE *CED* APPLICATION.

Kernel	% time	MAR	NLOC-MAR	MOR	NLOC-MOR	Stk Ratio	Flow Ratio	NLOC-Flow Ratio	Bytes/Acc.
gaussian_smooth	69.89	29.05	11.75	9.56	3.74	61.54	0.7376	0.9213	3.3090
non_max_supp	14.22	51.89	10.36	20.58	3.84	89.79	0.3635	0.8712	3.4318
magnitude_x_y	5.62	50.00	9.37	17.98	3.37	87.51	0.4168	0.3333	3.0007
apply_hysteresis	4.97	23.21	21.71	8.26	7.54	27.10	0.3288	0.4026	1.2998
derrivative_x_y	2.99	64.74	35.22	26.70	13.32	66.79	0.5557	0.3334	3.0028
follow_edges	2.24	48.33	7.79	18.25	4.41	94.85	-0.0030	0.8225	3.4357
read_pgm_image	0.03	46.81	22.56	18.78	9.39	59.02	0.5881	0.7547	3.4548
write_pgm_image	0.00	60.61	28.72	24.32	15.41	43.57	0.1258	0.1652	3.8105
make_gaussian_kernel	0.00	45.14	10.72	15.34	5.20	80.06	0.2529	0.8504	4.7549
canny	0.00	52.26	12.22	20.23	7.34	75.84	0.1889	0.6508	3.9658

MAR is the percentage ratio of the memory access operations to the total instructions executed in the application; ***NLOC-MAR*** is the same as ***MAR*** except that only references to the non-local region are considered; ***MOR*** is the percentage ratio of the memory access operands to the total number of operands; ***NLOC-MOR*** is the same as ***MOR*** except that only references to the non-local region are considered; ***Stk Ratio*** is the percentage ratio of the memory access instructions within the local region to the total memory access instructions; ***Flow Ratio*** is an indication of a function being more memory reader or writer. -1 means that the function only writes to the memory and +1 means that the function only read from memory; ***NLOC-Flow Ratio*** is the same as ***Flow Ratio*** except that only references to the non-local region are considered;

to the sampling technique used in *gprof*. As seen in Table IV, the values more or less conform to the numbers that were calculated previously. *gaussian_smooth* is not only the top contributor for the CED application, but also it demonstrates a relatively low percentage of memory access related workload compared to the computational burden. The *NLOC-MAR* and *Stk ratio* columns in Table IV can reveal valuable hints regarding the tendency of the function to go for local or non-local memory accesses. This is an important attribute if we opt to maintain the dynamically allocated memory on external sources (off-chip memory) in reconfigurable devices. For example, we would rather map *gaussian_smooth* or *non_max_supp* on FPGA than *apply_hysteresis* as a higher portion of memory accesses are intended for external memory regarding the latter function. In the case of the Molen platform, however, this is not relevant, as Molen does not currently support off-chip memory. As such, all the memory space required for the execution of the application should be allocated and managed on-chip. In this respect, a memory management module particularly takes care of all the dynamic memory allocations in the application.

Quipu Profiling. In the continuing work with the CED application, we want to be able to map different kernels to hardware. An important caveat for mapping kernels to hardware is that within the DWB some restrictions apply to the kernels that will be mapped to hardware. For this reason, we have modified the CED application where necessary, so that the intensive kernels could be mapped to hardware. There were two main issues that were solved:

- **Memory allocation**
  The DWARV compiler currently does not support allocating memory blocks at runtime from hardware. Therefore, all memory allocations were moved from the kernels into function stubs that allocate the required memory blocks first and then call the real kernels.
- **(Recursive) function calls**
  Furthermore, at this time function calls are not supported in DWARV. For most cases, manually inlining the code solved the problem. However, there was one instance of recursion in the *follow_edges* function, which inhibited simple inlining in the *apply_hysteresis* function. Therefore, the recursive function call was moved to an appropriate function stub.

Kernel	Area[a]			MAIP %time	Speed-up[b]	
	Slices	% of area	cum.[c] % of area		single kernel	cum.
hw_gaussian_smooth	2265	11.1%	11.1%	70.59%	3.40×	3.40×
hw_derrivative_x_y	2720	13.3%	24.3%	2.49%	1.03×	3.71×
hw_magnitude_x_y	1143	5.6%	29.9%	5.14%	1.05×	4.59×
non_max_supp	5599	27.3%	57.3%	14.36%	1.17×	13.48×
hw_apply_hysteresis	36617	178.8%	236.1%	2.68%	1.03×	21.10×

[a]Area predicted by a *Quipu* prediction model for the Virtex 5 FX 130T.
[b]Theoretical application speed-up, assuming 0s execution time for each kernel.
[c]cumulative

TABLE V
AREA PREDICTIONS AND THEORETICAL SPEED-UPS FOR THE KERNELS IN THE CED APPLICATION.

Of course, these changes required new profiling results. In Table V, the top 5 kernels are listed with their associated new time contributions, as reported by MAIP.

Evidently, in order to partition the application over hardware and software components, the evaluation of the computational and memory hot-spots is not sufficient. As there is only a limited amount of reconfigurable hardware area available, an investigation of the size of potential hardware designs is warranted. For this purpose, we used a *Quipu* prediction model for the Virtex 5 FX 130T FPGA. The results of the area prediction of the top 5 contributing kernels are presented in Table V. The table lists the actual number of slices, as well as the percentual area with respect to the target FPGA. The kernels in the table are in the order of execution in the CED application. As such, we can evaluate subsequent merging options by providing cumulative area figures as well. Note, that the *apply_hysteresis* kernel is exceedingly resource-intensive, taking up 178.8% of the target FPGA area. This large area requirement can be traced back to

a local array of 32K 32-bit integers. The used *Quipu* model targets the DWARV C2VHDL compiler, which converts such local arrays to registers, resulting in a large number of slices. When considering to merge several kernels together, the predictions suggest the first four kernels will easily fit together on the target FPGA.

In order to find a candidate set of functions to be moved to the FPGA hardware, we need an idea of the speed-up that might be achieved. Therefore, in addition to the area predictions, the theoretical application speed-ups for each kernel are also reported in Table V. These speed-ups are calculated using Amdahl's law assuming an unlimited speed-up for the kernel(s) in question, as follows:

$$\lim_{p\to\infty} \frac{p}{1-f(p-1)} = \frac{1}{f} = \frac{1}{1-s} \tag{4}$$

where p stands for the speed-up factor that is achieved in the accelerated part of the application, f stands for the percentual contribution of the remaining sequential part of the application, and s stands for the percentual contribution of the accelerated part of the application before acceleration. Table V lists both the speed-up when one kernel is accelerated and the cumulative speed-up, where each subsequent kernel is accelerated together with the previously accelerated kernels. Observe that as large parts of the application are accelerated, the contribution of the remaining kernels becomes more significant. For example, *apply_hysteresis* has a contribution of 2.68%, but with much of the application already accelerated, the difference in theoretical speed-up is 56.8%.

As we have mentioned before, merging the first four kernels in Table V would yield a hardware block that would fit in the target FPGA. The maximum speed-up of the application using that block would be 13.48×. Of course, the efficiency of accelerating this block will never be 100%. And the actual speed-up will be lower. The designer would have to decide whether to go for the larger kernel or to instantiate several blocks of smaller size in parallel. This last option can especially be interesting when edge detection is performed on a video stream.

QUAD profiling. In order to have a clear insight into the CED application behavior regarding the data communication between different functions, we utilized the *QUAD* core to profile the application. The Quantitative Data Usage (QDU) graph of the CED is depicted in Figure 4. The graph makes it possible to visually follow the journey of data objects through the sequences of function calls. In this way, we can trace what is actually happening to the input image as it moves along different phases of the CED algorithm. The detailed information presented in the graph also helps to understand what are the memory requirements of each function to accomplish its task. Furthermore, it plainly identifies the actual data dependencies between functions. The critical data path is highlighted in the graph to distinguish it from all the subordinate data communications between functions. *QUAD* starts instrumenting the application as soon as the main function gains control. In other words, the data communication associated with the operating system calls and libraries are somehow overlooked. Although this normally does not contain much valuable information about the critical data path of the application, it may cause missing starting or ending points for communication edges, i.e. the producer or consumer of the data. Currently, for every byte with an unidentified producer a manual investigation has to be conducted to verify the source of data. In the CED application, the main part of the input image file (800×600 bytes) has been tracked down to be part of the *unknown producer* entity. We revised the graph to replace this with a dummy *Image* node. A small part of the input image (4864 bytes) is recognized to be attributed to *read_pgm_image*. Since the file reading process is considered to be I/O, the operating system will take care of it, and how the process is implemented is completely platform dependent. Hence, slight inconsistencies appear in numbers reported by *QUAD* for such cases. Nevertheless, this is not affecting the overall picture of the data communication in the CED application.

make_gaussian_kernel is responsible for creating the Gaussian filter array. Based on the standard deviation used in our case study, an array of eleven elements is allocated. This is verified by the 44 UnMA reported on the edge from *make_gaussian_kernel* to the *hw_gaussian_smooth*. It should be noted that the array of eleven floating point values is accessed repeatedly throughout the smoothing process. As a result, huge data communication is reported between the two functions (about 40MB). *hw_gaussian_smooth* produces a temporary image data object filled with calculated intermediate floating point values. This is clearly reported by a self edge of 800×600×4 bytes UnMA in the graph. *hw_derrivative_x_y* uses the smoothed image data object as an input. The smoothed image used in *hw_derrivative_x_y* is of the short integer type, which is appearing as a corresponding edge of 960000 UnMA between the *hw_gaussian_smooth* and *hw_derrivative_x_y*. It can also be derived that the smoothed image data object is accessed four times in total (3840000 bytes), two times for calculating the derivative in the X direction and two times for the Y direction.

The calculated derivatives are stored separately in two arrays of the short integer type. They serve as inputs to *hw_magnitude_x_y*, which will compute edge strength values based on the gradient of the image. The result will be saved in an array called *magnitude*. It is also clear from the graph that the input arrays are scanned only once to compute the magnitudes. The third step of the CED algorithm needs the computed *magnitude* array and also the derivative arrays. This can be verified by the graph edges connecting *hw_derrivative_x_y* and *hw_magnitude_x_y* to the *non_max_supp*. For each pixel, the values of neighboring pixels in some directions are also examined. This results in a large number reported for memory accesses (approximately 4.5 MB). The resulting binary image, also known as *thin edges*, is output into an array with the same size of the input image. Regarding the final step, *hw_apply_hysteresis* (excluding the last recursive part to trace along the identified edges) uses the *magnitude* array and the output array from non-maximal suppression. As shown in the extracted data, the usage of *magnitude* is conditioned to only those pixels which indicate a possible edge. This is dependent on

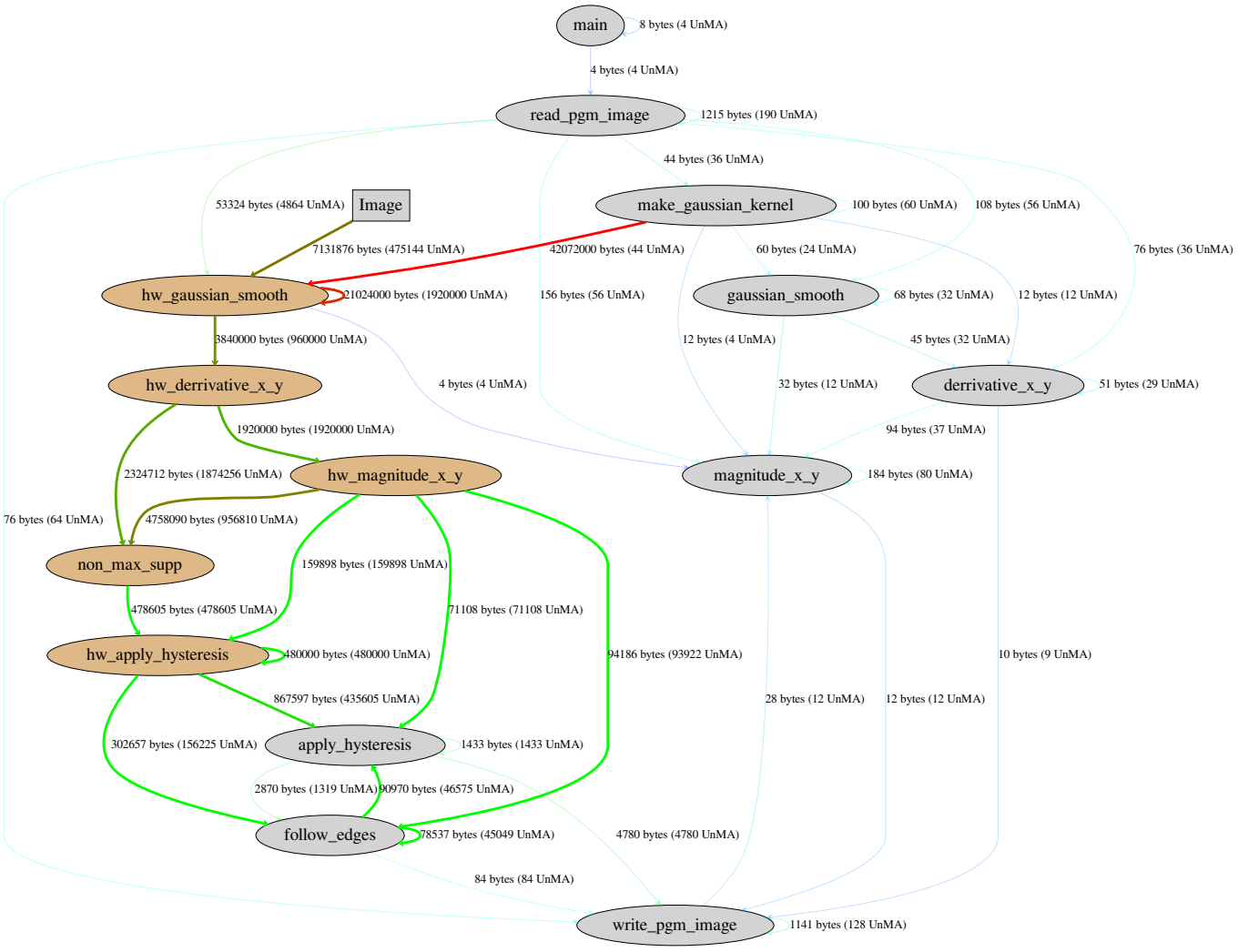

Fig. 4. Quantitative Data Usage graph for the hardware version of the Canny application.

the input image. In our case, about one sixth of the total pixels are identified as possible edge pixels. The self edge of 480000 bytes in *hw_apply_hysteresis* is attributed to the initialization of the *edge map* array. The array is subsequently processed in *hw_apply_hysteresis* for the computation of the histogram of magnitude values. *apply_hysteresis* finalizes the *edge map* by using the output of *hw_apply_hysteresis* and the *magnitude* array. To accomplish this task, it in turn utilizes *follow_edges*. All the corresponding data communications to perform the hysteresis process is identified and presented in Figure 4. *write_pgm_image* is responsible for writing the *edge map* array to the output file. As explained before, no major consumer of the array is identified by *QUAD* due to the I/O process.

VIII. Conclusions

The primary obstacle for improving the overall performance of computing systems arises from the communication bottleneck between processing elements and memory subsystem. This bottleneck is even more evident with the introduction of heterogeneous architectures containing reconfigurable fabrics; where, in addition to the memory bottlenecks, there are resource constraints that have to be taken into account. We have demonstrated in this paper that advanced profiling tools such as present in our Q^2 profiling framework can alleviate this problem. We plan to investigate the utilization of this framework more in-depth in the near future.

Acknowledgment

This research is partially supported by Artemisia iFEST project (grant 100203), Artemisia SMECY (grant 100230), and FP7 Reflect (grant 248976).

References

[1] K. Bertels and et al., "Developing applications for polymorphic processors: The delft workbench," Delft University of Technology, Tech. Rep., January 2006.

[2] 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.

[3] D. C. Suresh, W. A. Najjar, F. Vahid, J. R. Villarreal, and G. Stitt, "Profiling tools for hardware/software partitioning of embedded applications," in *LCTES*, 2003, pp. 189–198.

[4] "Intel's vTune," http://software.intel.com/en-us/intel-vtune.

[5] "AMD CodeAnalyst," http://developer.amd.com/cpu/codeanalyst.

[6] R. F. Cmelik, "Spixtools: Introduction and user's manual," Sun Microsystems Inc., Mountain View, CA, USA, Tech. Rep., 1993.

[7] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Trans. Program. Lang. Syst.*, vol. 16, pp. 1319–1360, July 1994.

[8] M. Martonosi, A. Gupta, and T. Anderson, "Memspy: analyzing memory system bottlenecks in programs," *SIGMETRICS Perform. Eval. Rev.*, vol. 20, no. 1, pp. 1–12, 1992.

[9] H. Davis and S. R. Goldschmidt, "Tango: A multiprocessor simulation and tracing system," Stanford University, Stanford, CA, USA, Tech. Rep., 1990.

[10] A. R. Lebeck and D. A. Wood, "Cache profiling and the spec benchmarks: A case study," *IEEE Computer*, vol. 27, pp. 15–26, 1994.

[11] A. I. Choudhury, K. C. Potter, and S. G. Parker, "Interactive visualization for memory reference traces," *Computer Graphics Forum*, vol. 27, no. 3, pp. 815–822, May 2008.

[12] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "Profileme: hardware support for instruction-level profiling on out-of-order processors," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 30, 1997, pp. 292–302.

[13] "Cacheprof," http://www.cacheprof.org.

[14] P. Schumacher and P. Jha, "Fast and accurate resource estimation of rtl-based designs targeting fpgas," in *FPL '08: Proceedings of 18th International Conference on*, sep. 2008, pp. 59 –64.

[15] C. Brandolese, W. Fornaciari, and F. Salice, "An area estimation methodology for fpga based designs at systemc-level," in *DAC '04: Proceedings of the 41st annual*, New York, NY, USA, 2004, pp. 129–132. [Online]. Available: http://doi.acm.org/10.1145/996566.996606

[16] P. Kannan and D. Bhatia, "Interconnect estimation for FPGAs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 8, pp. 1523–1534, Aug. 2006.

[17] L. M. Chuong, S.-K. Lam, and T. Srikanthan, "Area-Time Estimation of Controller for Porting C-Based Functions onto FPGA," in *RSP '09: Proceedings of the 2009 IEEE/IFIP International Symposium on Rapid System Prototyping*, 2009, pp. 145–151.

[18] D. Kulkarni, W. A. Najjar, R. Rinker, and F. J. Kurdahi, "Compile-time area estimation for LUT-based FPGAs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, no. 1, pp. 104–122, 2006.

[19] T. Degryse, H. Devos, and D. Stroobandt, "FPGA Resource Estimation for Loop Controllers," in *ODES'08: Proceedings of the 6th Workshop on Optimizations for DSP and Embedded Systems*, Boston, MA, USA, 2008, pp. 9–15.

[20] L. Deng, K. Sobti, and C. Chakrabarti, "Accurate models for estimating area and power of fpga implementations," in *ICASSP 2008: IEEE International Conference on*, 31 2008-april 4 2008, pp. 1417 –1420.

[21] S. Vassiliadis and et al., "The molen polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, 2004.

[22] S. Vassiliadis, G. Gaydadjiev, K. Bertels, and E. Moscu Panainte, "The molen programming paradigm," in *Computer Systems: Architectures, Modeling, and Simulation*, ser. Lecture Notes in Computer Science, A. Pimentel and S. Vassiliadis, Eds., vol. 3133. Springer Berlin / Heidelberg, 2004, pp. 1–10.

[23] S. A. Ostadzadeh, R. Meeuws, C. Galuzzi, and K. Bertels, "QUAD - a memory access pattern analyser," in *ARC 2010*, 2010, pp. 269–281.

[24] S. A. Ostadzadeh, M. Corina, C. Galuzzi, and K. Bertels, "tQUAD - memory bandwidth usage analysis," in *ICPP 2010*, September 2010, pp. 217–226.

[25] R. J. Meeuws, K. Sigdel, Y. D. Yankova, and K. Bertels, "High level quantitative interconnect estimation for early design space exploration," in *ICFPT '08*, December 2008, pp. 317–320.

[26] S. A. Ostadzadeh, R. J. Meeuws, K. Sigdel, and K. Bertels, "A clustering framework for task partitioning based on function-level data usage analysis," in *FPGA '09*, 2009, pp. 279–279.

[27] C. Galuzzi, "Automatically fused instructions - algorithms for the customization of the instruction-set of a reconfigurable architecture," Ph.D. dissertation, TU Delft, May 2009.

[28] S. A. Ostadzadeh, R. J. Meeuws, K. Sigdel, and K. Bertels, "A multipurpose clustering algorithm for task partitioning in multicore reconfigurable systems," in *CISIS '09*, 2009, pp. 663–668.

[29] E. M. Panainte, K. Bertels, and S. Vassiliadis, "The molen compiler for reconfigurable processors," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 1, 2007.

[30] Y. D. Yankova, G. Kuzmanov, K. Bertels, G. N. Gaydadjiev, Y. Lu, and S. Vassiliadis, "Dwarv: Delftworkbench automated reconfigurable VHDL generator," in *Proc. of FPL07*, 2007, pp. 697–701.

[31] S. A. Ostadzadeh, M. Corina, C. Galuzzi, and K. Bertels, "Runtime extraction of memory access information from the application source code," in *to appear in the proceedings of the International Conference on High Performance Computing & Simulation (HPCS)*, July 2011.

[32] R. J. Meeuws, "A quantitative model for hardware/software partitioning," Master's thesis, Delft University of Technology, Delft, Netherlands, Delft, Netherlands, May 2007.

[33] R. J. Meeuws, Y. D. Yankova, K. Bertels, G. N. Gaydadjiev, and S. Vassiliadis, "A quantitative prediction model for hardware/software partitioning," in *FPL '07: Proceedings of 17th International Conference on*, August 2007, pp. 317–320.

[34] R. Meeuws, K. Sigdel, Y. Yankova, and K. Bertels, "High level quantitative interconnect estimation for early design space exploration," in *FPT'08. International Conference on Field Programmable Technology*, dec 2008, pp. 317 –320.

[35] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, "Dwarv: Delftworkbench automated reconfigurable vhdl generator," in *FPL '07: Proceedings of 17th International Conference on*, 27-29 2007, pp. 697 –701.

[36] Y. Ben-Asher and N. Rotem, "Synthesis for variable pipelined function units," in *System-on-Chip, 2008. SOC 2008. International Symposium on*, nov. 2008, pp. 1 –4.

[37] ——, "Automatic memory partitioning: increasing memory parallelism via data structure partitioning," in *CODES: IEEE/ACM/IFIP international conference on*, ser. CODES/ISSS '10, New York, NY, USA, 2010, pp. 155–162. [Online]. Available: http://doi.acm.org/10.1145/1878961.1878989

[38] A. A. C. Experts. Cosy: Compiler system. [Online]. Available: http://www.ace.nl/

[39] 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.

[40] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, pp. 490–499, September 1960.

[41] J. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 8, pp. 679–698, November 1986.

[42] "Canny Edge Detector, Image Analysis Research Lab., USF," http://marathon.csee.usf.edu/edge/edge_detection.html.