# Profile-Guided Application Partitioning for Heterogeneous Reconfigurable Platforms

S. Arash Ostadzadeh, Roel Meeuws, Imran Ashraf, Carlo Galuzzi, Koen Bertels
Computer Engineering Group
Department of Software and Computer Technology
Delft University of Technology
Delft, The Netherlands
Email: {S.A.Ostadzadeh,R.J.Meeuws,I.Ashraf,C.Galuzzi,K.L.M.Bertels}@TUDelft.nl

***Abstract*—The increased complexity of programming heterogeneous reconfigurable platforms requires a thorough understanding of application behavior, for which developers need sophisticated analysis tools. One particular problem, which severely limits the performance gain of running applications on these platforms, is the inappropriateness of the kernels mapped onto the reconfigurable fabrics. Efficient porting of legacy applications to these emerging heterogeneous platforms demands code tuning considering several critical points, such as, proper kernel size and small memory communication overhead. Detailed profiling information is thus vital for an efficient HW/SW co-design. To facilitate addressing these issues, we developed the $Q^2$ *profiling framework*. It consists of two parts: an advanced memory access profiling toolset that provides detailed information on the run-time memory access patterns of an application and a statistical modeling framework that makes predictions for resources, early in the design phase, based on software metrics. The code optimizations triggered by careful analysis of the profiling information is used to tailor existing applications for heterogeneous reconfigurable platforms. In this paper, we examine a real application in detail to show the potential of the proposed profiling framework. Experimental results show that a speedup of 1.3× is achieved by accelerating a merged kernel of four critical functions in the application.**

## I. Introduction

As computer manufacturers incline towards multi-processor platforms, application developers are confronted with an increasing number of complex architectures. The current research trend is to combine different types of processing elements, such as General Purpose Processors (GPPs), Field Programmable Gate Arrays (FPGAs), or Graphics Processing Units (GPUs), in a single platform. Developing applications for such platforms not only requires dedicated toolchains and programming paradigms, but also demands comprehensive insight into the behavior of legacy applications, in order to tune them for performance gain on these heterogeneous systems. In particular, tools to thoroughly investigate the memory access behavior of applications become crucial, due to the widening gap between memory bandwidth and processing performance. Furthermore, in the case of systems utilizing reconfigurable devices, such as FPGAs, HW/SW partitioning is one of the critical stages that has a tremendous impact on the overall performance gain of these systems. The problem arises from selecting function candidates, or *kernels*, to map onto the reconfigurable fabrics. On systems where, due to technical restrictions, such as data dependencies and synchronization, there is no possibility to have more than one main kernel running in hardware, we usually end up with rather small candidates which have no capacity to compensate for the extra overhead imposed to the system. As a result, speedup is hardly seen or, even worse, one may encounter system performance degradation. A straight forward solution for this problem can be obtained by merging kernels, when possible, to come up with larger kernel candidates, which have the potential to improve the overall performance when mapped onto the hardware. The merging process should be carried out with care. The process has to consider the data communication between the kernels, while respecting the hardware resource limitations of the reconfigurable devices.

Additionally, hardware area estimation involves building and synthesizing hardware blocks, which is quite time-consuming. As a consequence, there is a need for fast and early predictions of the hardware costs depending on the different merging policies employed. These challenges are the main concern of our $Q^2$ *profiling framework*. The framework consists of two parts: a static part, which provides fast and accurate estimates of kernels' hardware area requirements, and a dynamic part, which provides precise measurements of memory access related metrics. In this paper, we show the need for and the usage of the proposed profiling framework by analyzing and porting a well-known standard speech coding application to the Molen reconfigurable architecture [1].

The main contributions of this paper can be summarized as follows:

- the introduction of the concept of nontrivial merging of coupled functions based on detailed data communication information and hardware area prediction;
- the investigation of application partitioning based on a coarse granularity level beyond just a single function;
- the presentation of a detailed case study, which shows an application speedup of 1.3× when mapped onto the reconfigurable platform at hand.

The rest of this paper is structured as follows. Section II briefly describes the related work regarding the HW/SW partitioning approaches in systems utilizing reconfigurable units. In Section III, the research context of the profiling framework is presented. A description of the components in the $Q^2$ profiling framework is presented in Section IV. After that, in Section V, we present a case study. Finally, Section VI concludes the paper.

## II. Related Work

The widespread utilization of heterogeneous reconfigurable systems relies on the availability of appropriate tools to assist developers in mapping existing applications onto these systems by reducing the time effort and efficiently exploiting the provided flexibility. The focus of this work is on application partitioning to determine which part(s) of the code should be mapped onto reconfigurable fabrics (hardware tasks) and which part(s) should be retained for execution on the GPP (software tasks). Generally, the HW/SW partitioning process can be performed based on different levels of granularity, such as, loops or functions. Profiling, the process of monitoring an application to spotlight specific code regions that intensely use up resources, is an essential step within the design process for many software and hardware systems. HW/SW partitioning has been an active field of research in the last decades. Many different results have been proposed during these years. For this reason, in the following, we limit the analysis of the related work only to those that consider application partitioning based on some profiling data.

In [2], a HW/SW partitioning approach is proposed for dynamically reconfigurable architectures consisting of a GPP and an FPGA. The main focus of the presented approach is on the temporal aspect of an application not on the spatial one. The partitioning is carried out at a fine-grained level, i.e., at loop and basic block levels, with the goal of optimizing the total application execution time. In the compilation flow, C source code of an application is preprocessed to extract loops as hardware candidates. Furthermore, multiple optimized versions of the loops are created by compiler transformations. Each extracted loop candidate is profiled to estimate the total software time, execution frequency, memory bandwidth requirement and the trace behavior. A quick synthesis is also performed to estimate the delay and the area needed for the hardware implementations. The loop entry trace profiling is used to find out the exact runtime sequence of all hardware candidate loops. The details of the various profiling processes are not presented. In the end, the extracted information are fed to the HW/SW partitioner to decide which loops should go to the hardware, and which versions of the loops should be used.

A HW/SW partitioning and code generation flow for reconfigurable platforms is presented in [3]. Given the C code of a target application, the user has to manually identify and tag computation blocks in the source code to be extracted for implementation on FPGA. Since in their target architecture, the reconfigurable array is part of the GPP's data path, there is a severe restriction of candidates that can be moved to the hardware. Candidates can only contain small computation blocks with few inputs and outputs. After that, a simulator evaluates the code using the cycle counts specified by the user for the tagged blocks. A profiler returns the number of cycles used to execute each line of code. To estimate the cycle count of the FPGA code blocks, the authors use heuristics to have a quick performance evaluation. As a result, the estimation may not be accurate enough. The partitioning problem is addressed by exploring different HW/SW trade-offs based on the performance profiles, with the objective of maximizing the overall performance, while satisfying FPGA mapping size constraints. This objective is formulated as a boolean programming problem.

Santambrogio et al. [4] proposed a methodology based on the *adaptive programming* technique to evaluate and subsequently perform HW/SW partitioning for a SoC that employs dynamically reconfigurable hardware and software programmable cores. They developed quantitative evaluation metrics to evaluate the reconfigurable system performance and to represent the performance of software in a SoC from an application-specific, input-oriented point of view. A performance model is built with the associated evaluation metric to identify application-specific input behavior of software modules. This general performance model is then embedded along with hardware performance models to yield a flexible mean to evaluate the performance impact of different partitioning and allocation decisions. A profiler enhanced by implementing *adaptive metrics* is used to reveal the potential in functions for performance improvement as a result of transformation into the adaptive form.

Apart from the traditional partitioning methods, different heuristic and evolutionary methods are also investigated to solve this problem. In [5], a heuristic searching approach is presented based on the *Ant Colony Optimization* (ACO) algorithm. Both global and local heuristics are combined in a stochastic decision making process to effectively explore the search space. As authors state, profiling information can also be utilized in the decision strategies to assign tasks to different resources. However, no reported study exists for such a work.

A design methodology for the application partitioning of reconfigurable MPSoCs is presented in [6]. The methodology supports the partitioning of an application between several processing elements (SW/SW partitioning) at the function level, as well as, HW/SW partitioning. A combination of a dynamic profiler (AMD *CodeAnalyst*) and a developed tracing tool, is utilized together with manual code analysis to analyze data communication between functions. The parameters used for the partitioning decision are extracted from the results obtained in the code analysis step, such as, the execution contribution of each function, the call graph, and the communication graph. For the SW/SW partitioning, hierarchical clustering is utilized, which is based on heuristics and, thus, it is faster than ILP. The partitioning algorithm can consider some critical issues, such as, workload balancing and minimal inter-processor communication. A tool is also developed to analyze the output of detailed *CodeAnalyst* profile to automatically calculate block and loop nesting, the timing of loops and functions, the function affiliation of loops and the calculation of the threshold, which defines hotspots in code fragments. This information can be used for HW/SW partitioning in systems incorporating reconfigurable fabrics.

The approach taken in [6] is similar to our work with regard to addressing application partitioning on the coarse function level. However, in our approach we utilize a different set of input data. *Instead of the common call graph, utilized by [6], we employ the Quantitative Data Usage (QDU) graph [7] as the primary reference for partitioning the application*. The information about the data communication between functions, represented by this graph, is *automatically* determined by our advanced profiling toolset. In contrast, other prominent works perform this task manually. Although this may be feasible for small examples, more complex applications, such as the case study presented in this paper, pose serious problems in manually determining the data communication among functions. In addition, call-graph-driven application partitioning may not result in an efficient clustering of the application. This may happen because of tight bindings between functions that do not directly call one another. This phenomenon is investigated in detail in Section V.

As FPGAs contain increasing amounts of reconfigurable area, there is an increasing incentive to map larger code segments onto the reconfigurable fabric. *This implies that the focus of the reconfigurable computing research will shift from fine-grained HW/SW partitioning to coarser-grained partitioning approaches, enabling the mapping of complete functions or clusters of functions.* To the best of our knowledge, there is no other work that proposes to merge functions based on accurate data communication information and resource estimates. *The merging of tightly communicating functions into one provides a simplified and coherent view of application tasks and enables powerful optimizations with regard to memory requirements and area consumption.*

## III. Research Context

The work presented in this paper, although not restricted to any specific architecture, has been developed in the context of the Delft Workbench (DWB). The DWB is a semi-automatic tool platform for integrated HW/SW co-design, targeting heterogeneous computing systems containing reconfigurable components. It addresses the entire design cycle from profiling and partitioning to synthesis and compilation of an application. It focuses on four main steps within the entire heterogeneous system design. The first step is related to application profiling [7], [8], [9] and coarse-grained task partitioning [10], [11], which is also the focus of this work. In the second step, fine-grained hardware code segmentation is investigated along with possible parallelization of the code segments [12]. Following the decision to map particular code segments onto the hardware, a *retargetable compiler* [13] generates new object code, which contains calls to reconfigurable hardware blocks for selected segments of the code. Finally, in the last step, *VHDL generation* [14], the identified code segments are translated into hardware blocks.

## IV. $Q^2$ Profiling Framework

Fig. 1 depicts the two profiling parts in the $Q^2$ *profiling framework*. The static profiling part extracts code characteristics from the application source code. These characteristics are

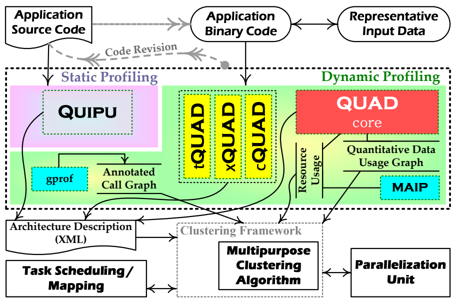

Fig. 1. The $Q^2$ Profiling framework within the DWB platform.

used by a linear model to make fast and early prediction of the FPGA area requirement for the functions in the application. The dynamic profiling part focuses on examining the runtime behavior of the application. In the dynamic part, we utilize the common general profiler, *gprof* [15] only to produce the call graph of the application. The execution timing provided by *gprof* is sample-based and, thus, it is imprecise. For this purpose, we use our own profiler, MAIP, which provides very accurate measurements of the execution time percentage for each function with respect to the whole execution time of the application. Furthermore, MAIP distinguishes between memory access and computational operations, as well as, local and non-local memory accesses. Therefore, a number of valuable parameters, such as the ratio of memory access instructions to the total instructions, can be measured. The outputs of the different tools are stored in flat profiles as well as in an XML description file for portability. This information is utilized in the clustering framework for application partitioning and subsequently mapping/scheduling on the target architecture. Furthermore, a parallelization unit examines the profiling data to discover potential concurrencies within and between the tasks of the application.

### A. Quipu *Modeling Approach*

Static profiling consists of the *Quipu* modeling approach, which is presented in [9]. The approach is generic and not limited to any particular platform or toolchain by allowing the generated models to be recalibrated for different tools and platforms, contrary to the majority of existing techniques. Furthermore, a major strength of *Quipu* models is their linear nature. Although the statistical techniques used to create the models may be very time-consuming, the resulting prediction model requires only a few multiplications in addition to parsing the source code. This allows for the integration of *Quipu* models in highly iterative design processes, where estimates are recalculated many times in a short period of time. Additionally, as *Quipu* models are based on measurements from C code, very early predictions become possible. Although such early predictions introduce a relatively large error of 10% to 20%, they allow designers to make important decisions on hardware mapping at an earlier stage.

In [16], we have introduced Software Complexity Metrics (SCMs) to quantify the characteristic aspects of software code. Examples of SCMs include the number of operators/loops, the cyclomatic complexity, or more complex metrics involving data-flow analysis. Currently, we use 58 SCMs as a base for our model. *Quipu* extracts SCMs and hardware characteristics from a kernel library *of 247 kernels spanning a wide variety of application domains, contrary to many existing techniques, which use libraries of tens of kernels at most*. This allows building *general* as well as *domain-specific* models. In the context of this paper, we have extended the kernel library with 65 floating point kernels. This greatly improves the applicability of *Quipu* models by allowing more accurate predictions for many compute-intensive applications from the multimedia, scientific, and other domains.

### *B.* Quad *Dynamic Memory Profiling Toolset*

The Quad toolset consists of several tools developed to provide a complete overview of the memory access behavior of an application, as well as, to provide fine-grained detailed memory access related statistics. The Quad [7] core module primarily detects the actual data dependencies at the function-level. The tracing module implemented in the Quad core is subsequently utilized in cQuad to reveal the data communication patterns of pairs of cooperating functions. xQuad [17] augments the memory access profiling data with extra information regarding individual data objects defined in the application source code. tQuad [8] reveals the memory bandwidth usage of each function in terms of relative execution timings. All the tools in the Quad toolset are implemented utilizing the Pin [18] Dynamic Binary Instrumentation (DBI) framework.

## V. Case Study

In this section, we present concise analyses of a real application from the speech processing domain, namely the *MELP vocoder*, to show the efficiency and applicability of the developed tools in the $Q^2$ *profiling framework*. The main goal of the case study is to have an early, yet comprehensive, understanding of the application behavior concerning memory and required area. The extracted profiling information is subsequently utilized to merge critical functions and to map the application onto the Molen reconfigurable architecture.

### *A. MELP Overview*

The MELP (Mixed Excitation Linear Prediction) vocoder is a standard [19] low rate speech coder selected by USDoD and used mainly in military/satellite communications, and secure voice/radio devices. The initial MELP algorithm was invented by Alan McCree in the mid 90s at the CSIP in Georgia Institute of Technology. Later, the coder was developed by a team from Texas Instruments Corporate Research and Atlanta Signal Processors. MELP is particularly robust in difficult background noise environments and very efficient in its computational requirements. As a result, it has low power consumption, which is a crucial consideration in embedded and dedicated hardware systems.

***Algorithm Description.*** The MELP Vocoder is based on the traditional Linear Predictive Coding (LPC) parametric model, but also includes some extra features, namely, mixed-excitation, aperiodic pulses, pulse dispersion, and adaptive spectral enhancement. These additional features mainly improve the excitation structure of LPC along with an accurate simulation of the natural speech. The mixed-excitation is implemented using a multi-band mixing model to reduce the buzz usually associated with LPC vocoders. Then, a linear prediction analysis is performed on the input speech using a hamming window. The LPC residual signal is calculated by filtering the input speech signal with the prediction filter, whose coefficients were determined by the linear prediction analysis. At this point, the final pitch estimate is calculated using the low-pass filtered residual signal. The gain is estimated afterwards. To put the speech data (LPC coefficients, pitch, gain, and bandpass voicing) into a smaller representation, Vector Quantization (VQ) is used. The encoded data, which are formed into packets, along with the VQ codebook, constitute the MELP bitstream. Channel coupling will also be generated subsequently. The decoding phase is less complex than the encoding. The packets are decomposed out of the bitstream and are processed for factors extraction and channel decoupling. Initially, the speech signal in each frequency band is recovered, and then the output speech signal is created by applying the voicing filter and adaptive spectral enhancement post filter. MELP can synthesize speech using either periodic or aperiodic pulses. Aperiodic pulses are most often used during transition regions between voiced and unvoiced segments of the speech signal. A pulse dispersion filter, based on a spectrally flattened triangle pulse, is used in the final stage to spread the excitation energy with a pitch period. This, in turn, reduces the harsh quality of the synthetic speech. A block diagram of the MELP vocoder is shown in Fig. 2.

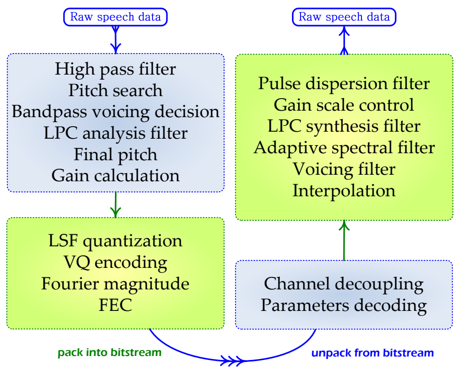

Fig. 2. The MELP vocoder block diagram.

### *B. Experimental Setup*

All experiments were performed on two different platforms. We used the Quad toolset on an Intel 32-bit Core2 Duo E8500 @3.16 GHz, running Linux kernel v2.6.34. The application was also executed on the embedded PPC 440 @400 MHz, which is integrated in a Xilinx ML510, Virtex 5 FX 130T with 1.3 MB BRAM FPGA board. On this platform, we instrumented the functions, counting the number of cycles for each function call using the PPC Time Base register which is incremented each clock cycle. The MELP application

TABLE I
RESULTS OF THE PRELIMINARY ANALYSIS OF THE MELP APPLICATION CONTAINING THE TOP 10 KERNELS.

| Kernel | Exec. time[a] | Calls | Area[b] | hMAR[c] | Speedup[d] |
|---|---|---|---|---|---|
| vq_ms4 | 23.28 | 2268 | 36337 | 18.99 | 1.30 |
| zerflt | 19.56 | 22303 | 540 | 19.40 | 1.24 |
| find_pitch | 14.88 | 9072 | 2202 | 31.21 | 1.17 |
| polflt | 11.53 | 18144 | 610 | 26.20 | 1.13 |
| frac_pch | 10.18 | 32097 | 4760 | 22.42 | 1.11 |
| fft | 8.15 | 1891 | 2903 | 23.35 | 1.09 |
| lsp_g | 2.16 | 687115 | 1053 | 7.59 | 1.02 |
| vq_enc | 2.09 | 2268 | 557 | 22.47 | 1.02 |
| envelope | 1.77 | 9072 | 810 | 26.59 | 1.02 |
| autocorr | 1.19 | 2268 | 560 | 32.95 | 1.01 |

[a]Percentage of the execution time contribution reported by MAIP.

[b]Number of slices predicted by *Quipu* model generated for Virtex5 and Delft Workbench.

[c]heap Memory Access Ratio (hMAR), reported by MAIP.

[d]Theoretical application speedup calculated with Amdahl's Law.

source code was compiled with *gcc* v4.1.1, with *-O2* compiler optimizations on both platforms. The MELP implementation consists of 25 source files with, in total, 70 functions. For the experiments, we encoded a sample voice recording of male voice fragments in 8000 sample/s raw PCM format. The *Quipu* model that we utilize in this paper was generated for a combination of the *DWARV* C-to-VHDL compiler [14] and Xilinx ISE 13.2 for the Virtex 5 FX 130T FPGA.

### C. Experimental Analysis

In Table I, we list the top 10 kernels according to our MAIP profiling tool. The second column lists the execution time percentage of each kernel reported by MAIP. The top kernel *vq_ms4*, the main part of the vector quantization stage, accounts for 23.28% of the total execution time. The subsequent four kernels in the top 5 are part of the linear prediction analysis and together account for 56.15% of the total execution time. Then, there is *fft* accounting for 8.15% and some remaining smaller kernels accounting for 7.21%. In the third column, the number of calls to each function is listed.

The *Quipu* area estimates for each kernel are listed in the fourth column. Given that the Virtex5 FPGA on the ML510 board has 20480 slices available, notice that all kernels except *vq_ms4* are predicted to fit. The reason why *vq_ms4* exceeds the size of the FPGA is the corresponding large size of the function itself. It consists of 210 lines of code, contains 12 loops, and has a nesting depth of 9. Although this function cannot be mapped to hardware, ample room remains for speedup in the remaining kernels.

From the fifth column, the hMAR, we can see that all but one kernel spend roughly 20% to 30% of their execution times in accessing the heap. One exception is the *lsp_g* kernel, which contains one loop that performs one heap load but executes several other operations per iteration. A high hMAR ratio indicates that the increased performance of hardware implementation is counteracted by latencies in accessing the heap, if we assume that such accesses cannot be further reduced. In the case of these top 10 kernels, the hMAR is low enough to allow for reasonable speedups.

In addition to the area predictions, the theoretical application speedups are also reported in the last column. These speedups are calculated using Amdahl's law, assuming an unlimited speedup for the kernel(s) in question, as follows:

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

where $p$ is the speedup factor of the accelerated part, $f$ is the percentual execution contribution of the sequential part, and $s$ is the original percentual execution contribution of the accelerated part. We can see that without merging some kernels together the maximum possible theoretical speedup would be $1.30\times$. As this speedup is bound to be much lower, it is desirable to find a combination of kernels that has a larger computational contribution and, therefore, allows for a larger speedup.

**QUAD Profiling**. We utilized QUAD to reveal the data communications between different functions of the MELP application. Due to space limitations, only a part of the resulting QDU graph is shown in Fig. 3.

Not every heavy data communication leads to a potential memory bottleneck. A more detailed investigation is required to pinpoint problems related to memory accesses. Special attention has to be given to the size of the accessed memory blocks, to the locality, to the reusability, and, most significantly, to the placement of data (on-/off-chip data allocation), where applicable. For our experimental setup, there was no off-chip data allocation due to Molen restrictions, however, this property must be considered in general.

In the top part of Table II, we see an overview of the different candidates for merging. The candidates were selected by examining functions that would fit on the FPGA and had intense communication. As expected, by inspecting the source code, we noticed that these functions were almost always called together and in similar sequences. In the analysis phase, $Q^2$ provides the computational intensity, hardware area estimates, and insight to the communication intensity. As the QDU graph suggests, an obvious merging option would be to combine *polflt* and *zerflt*. It should be noted that examining the call graph for the application displays no calling link between these two function, although they are tightly coupled. These two functions together contribute 31.09% to the execution time. They are intensively communicating, and after inspecting the code, they appear to be called in sequence in almost all cases. Row $o_1$ lists the tentative data for this merging option. Note that the merged kernel has a larger theoretical speedup than the individual kernels.

The QDU graph also reveals that two other functions, from the top 10, are intensely communicating with *polflt* and *zerflt*, namely *find_pitch* and *frac_pch*. Again, the $Q^2$ framework helps us identify these kernels, as the predicted area for each of them is relatively small, and the communication with other functions is quite intense. However, when we investigate the code, we find that these four functions are not called in a consistent order throughout the code. Sometimes only the filters are executed, sometimes *find_pitch* is also included,

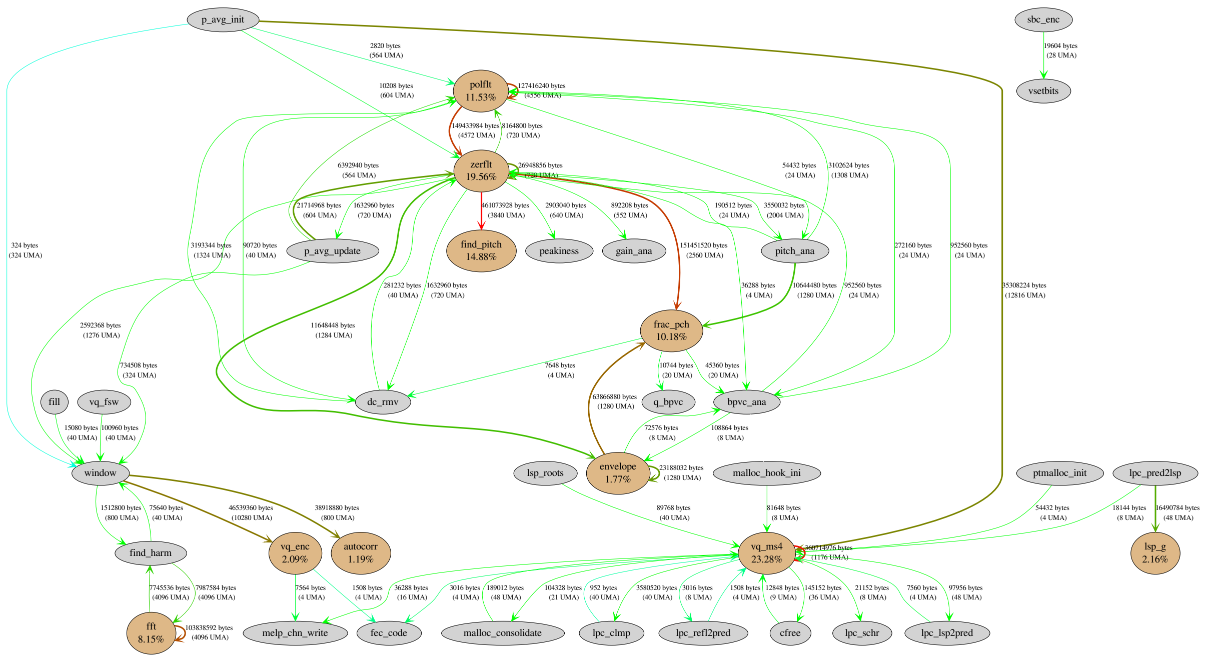

Fig. 3. The QDU graph of the MELP application before merging.

TABLE II
RESULTS OF THE ANALYSIS OF THE MERGING CANDIDATES AND FINAL MERGED KERNELS AND THE ACTUAL SYNTHESIS RESULTS.

| **id** | **Kernel** | **Area (slices)** | | **Exec. time**[a] | $t_{sw}$ **(s)**[b] | $t_{hw}$ **(s)**[c] | **Speedup** | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **Predicted** | **Actual** | | | | **Kernel** | **Application** | |
| | | | | | | | | **Theoretical** | **Actual** |
| $k_1$ | zerflt | 522 | 401 | 19.56 | 381 | 181 | 1.24 | 2.10 | 1.11 |
| $k_2$ | find_pitch | 1900 | 2205 | 14.88 | 3370 | 1824 | 1.17 | 1.85 | 1.07 |
| $k_3$ | polflt | 661 | 421 | 11.53 | 513 | 152 | 1.13 | 3.38 | 1.09 |
| $k_4$ | frac_pch | 4760 | n.a. | 10.18 | n.a. | n.a. | 1.11 | n.a. | n.a. |
| $o_1$ | $k_1$ + $k_3$ | 1183 | n.a. | 31.09 | 894 | 333 | 1.45 | 2.68 | 1.24 |
| $o_2$ | $o_1$ + $k_2$ + $k_4$ | 8112 | n.a. | 56.15 | n.a. | n.a. | 2.28 | n.a. | n.a. |
| $m_1$ | pol_vequ_zerflt | 869 | 706 | 26.81 | 894 | 365 | 1.38 | 2.44 | 1.19 |
| $m_2$ | filters_plus_pitch | 7111 | 6722 | 51.80 | n.a. | n.a. | 2.07 | 1.80 | 1.30 |
| $m_{2a}$ | ” | ” | ” | 14.84 | 4259 | 2505 | 2.07 | 1.70 | 1.07 |
| $m_{2b}$ | ” | ” | ” | 1.54 | 349 | 181 | 2.07 | 1.92 | 1.01 |
| $m_{2c}$ | ” | ” | ” | 28.33 | 1094 | 575 | 2.07 | 1.90 | 1.15 |
| $m_{2d}$ | ” | ” | ” | 7.09 | 547 | 336 | 2.07 | 1.63 | 1.03 |

[a]Percentage of the execution time contribution as reported by the MAIP profiler.

[b]Time measured on PPC on the ML510.

[c]Time measured with *Modelsim 6.5* simulator targeting ML510 Virtex5 FPGA.

or *frac_pch*, other times the filters are omitted. In total, we identified four scenarios. We merged the four kernels and added corresponding *if*-statements to switch on the correct code regions. The combined predictions for this second option are listed in row $o_2$.

After we identified the merging candidates using the $Q^2$ framework, we merged the kernels and performed a new analysis iteration. The results are presented in row $m_1$ and $m_2$ of Table II. As mentioned before, we identified four scenarios of ordering the individual kernels merged in $m_2$. For each scenario, we list detailed information in rows $m_{2a}$-$m_{2d}$. Observe that the predicted area consumption has been reduced. The reason for this is the additional opportunities for the compiler to optimize the code. We have also included parts of the new QDU graphs in Figures 4 and 5. Clearly, the communication channels between the kernels have been internalized by the merging process.

We synthesized the kernels and merged kernels in order to validate our predictions and to determine the kernel and application speedups. The results are also listed in Table II. The table includes the execution times in % and in s for the PPC and for the FPGA. The FPGA execution times were determined using the *Modelsim 6.5* simulator using the same input as on the PPC. Furthermore, we also include the

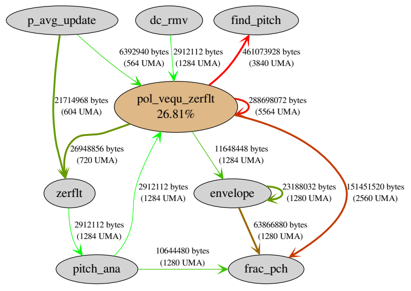

Fig. 4. The QDU graph after the first merging step.

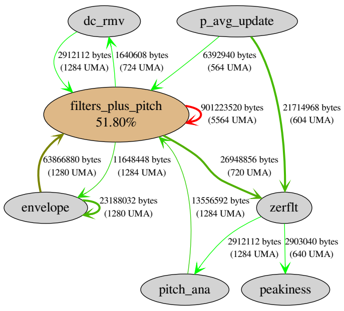

Fig. 5. The QDU graph after the second merging step

actual area for each of the kernels. In case of the *frac_pch* function no actual area is reported, as this kernel contains a function call, which *DWARV* does not support. Apart from the theoretical speedup, we also included the kernel speedup and the corresponding application speedup.

It can be seen that the *Quipu* estimates exhibited an error of 5.8% to 57%. This is an acceptable error rate for early analysis of hardware resource consumption. It should be noted that the potential for speedup has been increased to 2.07×. The actual speedup that we were able to obtain by merging was 1.3×.

## VI. Conclusions

Running existing applications on heterogeneous reconfigurable systems is a challenging task. Apart from the required system development tools to enable porting of the applications, several critical issues need to be addressed as well to fully unleash the performance gain of these systems. Among them is the HW/SW partitioning. Efficient application partitioning is only possible with careful inspection of the application behavior to make a proper separation between tasks to be mapped on various processing elements. We have shown in this paper that the advanced profiling tools developed in our $Q^2$ *profiling framework* allow effective handling of the HW/SW co-design. The valuable information extracted by the tools can be used to direct a coarse-grained partitioning of the application. In the presented case study, the result of porting a real application to the Molen reconfigurable architecture guided by the $Q^2$ profiling information was a speedup of 30 percent.

## References

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

[2] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-software co-design of embedded reconfigurable architectures," in *Proceedings of the 37th Annual Design Automation Conference*. New York, NY, USA: ACM, 2000, pp. 507–512.

[3] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. K. Brayton, and A. Sangiovanni-Vincentelli, "HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform," in *Proceedings of the 10th international symposium on Hardware/software codesign*. New York, NY, USA: ACM, 2002, pp. 151–156.

[4] M. Santambrogio, S. Memik, V. Rana, U. Acar, and D. Sciuto, "A novel soc design methodology combining adaptive software and reconfigurable hardware," in *Proceedings of ICCAD'07*, 2007, pp. 303–308.

[5] G. Wang, W. Gong, and R. Kastner, "Application partitioning on programmable platforms using the ant colony optimization," *Journal of Embedded Computing*, vol. 2, no. 1, pp. 119–136, 2006.

[6] D. Gohringer, M. Hubner, M. Benz, and J. Becker, "A design methodology for application partitioning and architecture development of reconfigurable multiprocessor systems-on-chip," *Proceedings of FCCM'10*, pp. 259–262, 2010.

[7] S. A. Ostadzadeh, R. J. Meeuws, C. Galuzzi, and K. Bertels, "QUAD - A Memory Access Pattern Analyser," in *Proceedings of the 6th international conference on Applied Reconfigurable Computing*, ser. ARC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 269–281.

[8] S. A. Ostadzadeh, M. Corina, C. Galuzzi, and K. Bertels, "tQUAD - Memory Bandwidth Usage Analysis," in *Proceedings of the 39th International Conference on Parallel Processing (ICPP)*, Sept. 2010, pp. 217–226.

[9] R. J. Meeuws, C. Galuzzi, and K. Bertels, "High level quantitative hardware prediction modeling using statistical methods," in *Proceedings of SAMOS'11*, 2011, pp. 140–149.

[10] 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 *Proceedings of the ACM/SIGDA international symposium on FPGA*. New York, NY, USA: ACM, 2009, pp. 279–279.

[11] S. A. Ostadzadeh, R. Meeuws, K. Sigdel, and K. Bertels, "A multipurpose clustering algorithm for task partitioning in multicore reconfigurable systems," in *Proceedings of International Conference on Complex, Intelligent and Software Intensive Systems (CISIS '09)*, March 2009, pp. 663–668.

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

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

[14] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, "DWARV: Delftworkbench Automated Reconfigurable VHDL Generator," in *Proceedings of International Conference on Field Programmable Logic and Applications*, August 2007, pp. 697–701.

[15] 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.

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

[17] S. A. Ostadzadeh, M. Corina, C. Galuzzi, and K. Bertels, "Runtime Extraction of Memory Access Information from the Application Source Code," in *Proceedings of International Conference on High Performance Computing and Simulation (HPCS'11)*, July 2011, pp. 647–655.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 190–200.

[19] L. Supplee, R. Cohn, J. Collura, and A. McCree, "MELP: the new federal standard at 2400 bps," in *Proceedings of International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, April 1997, pp. 1591–1594.