2010 39th International Conference on Parallel Processing Workshops

tQUAD - Memory Bandwidth Usage Analysis

S. Arash Ostadzadeh, Marco Corina, Carlo Galuzzi, and Koen Bertels
Computer Engineering Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, Delft, the Netherlands
Email: {S.A.Ostadzadeh,M.Corina,C.Galuzzi,K.L.M.Bertels}@tudelft.nl

***Abstract*—One of the main issues in heterogeneous reconfigurable computing is the well-known processor/memory bottleneck. Due to the memory bandwidth limitations, the performance of execution of an application can dramatically increase via the efficient usage of the memory. In this paper, we present tQUAD, a new tool for the memory bandwidth usage analysis. This tool is capable of delivering detailed temporal memory bandwidth usage information for the functions in an application throughout a comprehensive analysis of the memory access patterns of individual functions. This tool, first in its kind, provides an accurate analysis of the task execution and memory bandwidth usage which in the end leads to a sophisticated partitioning of the tasks into different phases during the execution span of an application. Together with an accurate description of the tool, the paper presents a real case study from the multimedia domain to detail all features of the proposed tool.**

I. Introduction

Heterogeneous reconfigurable systems enable the utilization of multiple types of processing elements within a single platform, allowing each element to perform the task(s) to which it is best suited. They may contain, for instance, Application-Specific Integrated Circuits (ASICs), Field Programmable Gate Arrays (FPGAs), Graphic Processing Units (GPUs), Digital Signal Processors (DSPs), and the conventional commodity processors. These systems are increasingly gaining popularity due to their ability to speed up applications from many different domains. The well-known processor/memory bottleneck drastically limits the system performance. As a result, the tuning of the application code has become imperative to extract the maximum performance from the target architecture. Furthermore, the widespread utilization of such systems through the industry seems to be inconvenient due to the shortage of tools guiding developers throughout the entire development process. *In this paper, we focus on the memory bandwidth issues and present a new tool that delivers detailed temporal memory bandwidth usage information for the functions in a given application.*

The analysis of the behavior of tasks during the execution of an application is an important aspect of the application development and optimization. When porting an existing application to a heterogeneous system, one of the main problems is the partitioning of the application. Depending on their characteristics, tasks can be executed in software (on a general-purpose processor) or hardware (on a reconfigurable device). Additionally, there are tasks that can be implemented both in software and in hardware, depending on the availability of hardware resources. These decisions are part of the task scheduling and mapping process. In many computing fields, like embedded and real-time systems, software scheduling is of vital importance and it is a well studied topic [1], [2]. Partial reconfiguration on the reconfigurable hardware adds even more complexity to the scheduling problem [3]. As a consequence, efficient scheduling and task placement algorithms become a must. To cope with these issues and efficiently utilize the hardware resources, a detailed analysis of the tasks execution behavior must be carried out. In [4], we have presented a tool that performs a thorough analysis of the memory access behavior of an application with the primary goal of providing detailed quantitative information of actual data dependencies between a pair of communicating functions.

In this paper, we continue our analysis of the tasks execution behavior by presenting a new tool, called tQUAD, for the memory bandwidth usage analysis. The tool aims to present a detailed timing information of tasks execution via a thorough analysis of the data usage of each individual task. As a result, partitioning, scheduling and mapping of these tasks onto reconfigurable architectures can be performed in a more efficient way. The presented tool is general and not restricted to any particular architecture. Even the extracted information can be used with different objectives, such as general application revision for performance improvement or coarse grain parallelism detection. Nevertheless, here we present our application of the proposed tool in the context of the Delft WorkBench project [5], which targets the Molen reconfigurable architecture [6].

The main contributions of this paper are the following:

- the description of an efficient tool, tQUAD, to provide the timing of the tasks execution and memory bandwidth usage information that can be utilized in task scheduling and mapping process on heterogeneous reconfigurable systems;
- the recognition of the main phases in the execution time of an application that can be used to identify related kernels for task clustering purposes;
- the presentation of the memory bandwidth traffic estimations which can provide a clear platform-independent insight of an application for code revising to solve memory access related problems;
- the validation of the proposed tool on a real case study with a detailed analysis.

The remainder of the paper is organized as follows. Section 2 gives an overview of the related research. In Section 3, we present the context of our research. Section 4 introduces tQUAD and describes some of the design and implementation issues. In Section 5, a real case study is examined in detail. Finally,

1530-2016/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPPW.2010.39

Section 6 provides concluding remarks and an outline of the future research.

II. Related Work

The execution time analysis is a well-known topic. To have an estimation of the execution time of an application is vital for the acceptability of many systems, especially for systems delivering real-time services. There are two main execution time analysis: static analysis and dynamic analysis. Static techniques do not rely on the execution of the application on real hardware. They rather analyze the source code, or some form of object code, to decide the set of possible execution paths and to obtain an upper bound on the execution in a specified hardware model. Dynamic techniques instead, take into consideration a specific hardware and derive the execution time from the recorded results [7].

Hard real-time systems require a deterministic timing behavior of the application to guarantee the termination of the task execution. This is guaranteed by estimating the Worst Case Execution Time (WCET) of the application. Commercial and research prototype tools for WCET analysis estimation are available. These include aiT [8], Bound-T [9], Chronos [10], Heptane [11], SWEET [12], and Symta/P [7].

Usually WCET tools work on binary executables, as these contain all the information needed for the analysis. First, the Control-Flow Graph (CFG) is constructed. This graph is used to determine the possible program paths. Next, a simplified model of a microarchitecture is required to produce timings for the program paths using the information on caches, memory, pipelines, branch prediction schemes, and other hardware components. This model can be built-in inside the tool or, in case of a new target processor, it can be constructed and feeded to the tool by the user. Finally, by using the information acquired during the first two phases, the final bound calculation is performed, producing the WCET estimation in terms of cycles which could be then, if supported by the tool, converted to seconds.

Most vendors do not disclose enough information on their microarchitecture. As a result, the WCET estimation is likely to become a tedious task to perform as a model of the target architecture, although a simplified one, is required for the analysis. In any case, the results of the tool must be validated by measurements before considering the calculated WCET reliable. Additionally, nowadays, the hardware complexity is very high and, in many cases, it is difficult to extract an accurate model which makes the estimation unreliable. Therefore, a measurement-based analysis beside a static analysis is a common practice when estimating the WCET. As a matter of fact, when the real execution time of a certain application on a certain hardware is desired, and not necessarily an upper bound on the execution time, the user performs a measurement-based analysis of the system. To cope with this necessity, hybrid approaches combining measurement and static program analysis have been developed. Examples are the Rapitime [13] and the MTime [14] tools.

The hardware complexity and the fact that static WCET analysis can deliver an over-pessimistic timing estimation cause this method to be inefficient when applied to heterogeneous reconfigurable devices. Scheduling and mapping algorithms must strive to make the reconfigurable hardware area usage as efficient as possible. Hence, the need for dynamic analysis methods which aim at providing precise information on the relative timing behavior of an application is critical for developers.

Profilers like *gprof* [15] are a kind of dynamic analysis tools. *gprof* performs the analysis based on source code instrumentation, providing information about the execution time of functions in an application. However, *gprof* does not distinguish among CPU time and memory access time. The implementation of a task in reconfigurable hardware based on this information, may result in an inefficient usage of the hardware resources. This eliminates the advantages of a hardware implementation because the processing performance gain of a computationally intensive task may, in reality, be bounded by its memory access inefficiency.

Some dynamic analysis tools integrate hardware event monitoring, such as counters, available on the chip. Examples are the Intel's vTune [16] and the AMD CodeAnalyst [17]. Both tools provide the user with a suite for performing application performance analysis which provides, among others, a time-based analysis that helps locating the application hot-spots and the bottlenecks as candidates for optimization. The vTune time-based sampling approach gathers information on the percentage of time spent by an application by interrupting the application's execution at regular time intervals and by recording instruction pointer addresses. Then, the most frequently executed portions of code are reported, in terms of clockticks. Besides, by finding and reporting hot-spots, vTune produces also a detailed analysis at architectural level by specifying how the application behaves in memory, and by identifying problems such as cache misses. Similar to vTune, the AMD CodeAnalyst suite performs system-wide profiling and supports the analysis of both user applications and kernel-mode software. It also collects instruction pointer addresses at predefined time intervals and it reports bottlenecks, execution penalties, and optimization opportunities. For a given application, it reports the number of CPU cycles needed by a code region, the Instruction-Per-Cycle (IPC) and its inverse Cycles-Per-Instruction (CPI), and statistics on data accesses. However, both tools are hardware dependent.

Another method to analyze the execution time is by using an Instruction-Set Simulator. In [18], a simulator is described which profiles applications according to a predefined target architecture. Although this can help programmers to gain insight on how an application behaves and possibly where this application needs to be improved, by having this information based only on simulators, misleading information can be delivered, as not all simulator are clock-cycle accurate. As mentioned before, nowadays, hardware has reached a complexity that is almost unfeasible to emulate. Hence, to retrieve exact information on application execution time, the application should be run on the target hardware.

In this paper, we present the tQUAD profiler. It analyzes the memory bandwidth usage of an application in terms of relative execution timings. The timing is based on instruction counting, and it delivers accurate information about the life of certain application tasks. This general form of execution time unit representation allows to have a *platform-independent*

implementation of the tool. By knowing the number of CPI (or its inverse, IPC), which are hardware dependent, it is possible to retrieve the conventional execution time in terms of, for instance, seconds. *This timing information together with the amount of data usage of a certain task deliver useful insights on the behavior of the application.*

III. The Profiling Framework

The proposed tool is not restricted to any particular architecture. Nevertheless, in this paper, we consider an application of the tool in the context of the Delft WorkBench (DWB) [5]. 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 [6] machine organization, which is one instance of a heterogeneous reconfigurable platform. DWB addresses the entire design cycle from profiling and partitioning to synthesis and compilation of an application and focuses on four main steps within the entire heterogeneous system design, namely:

- *the code profiling and the cost modeling* [19];
- *the graph transformations and optimizations* [20]–[22];
- *the retargetable compiler* [23];
- *the VHDL generation* [24].

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. During the *graph transformation and optimizations*, the candidate parts of the application for hardware implementation are analyzed to find out if the code segments can be clustered according to various targets as, for example, sharing common characteristics. Next, an *optimization* phase is performed to spot parallelization opportunities. Particular attention is needed during the partitioning of a certain application, as the entire cycle time can be affected. Cycle time can slow down as the code segment executed on the reconfigurable device grows in size, resulting in an overall slower execution.

After making the decision of which parts of the code segments to implement in hardware, the code is annotated. After that, the *retargetable compiler* generates the new object code which contains the call to the reconfigurable hardware for the instructions found. The *VHDL generation* phase generates hardware description of the kernels.

The main focus of the work proposed in this paper is on the profiling process. Figure 1 gives an overview of the DWB profiling framework and where the proposed tool can be positioned.

IV. tQUAD Design and Implementation

tQUAD is designed as a complementary profiler in a dynamic profiling framework along with QUAD [4] to deliver detailed temporal memory bandwidth usage information for each kernel in an application. *In QUAD, the quantitative information about data communication between kernels is revealed. With tQUAD, we aim to extract the timing information of a single kernel*

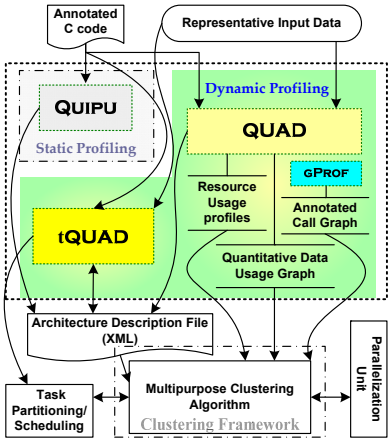

Fig. 1. Profiling framework within the DWB platform.

execution as well as its memory bandwidth usage during an application execution. The information extracted by tQUAD can lead to the recognition of the main execution phases within an application, which, in the end, can be used to identify the related kernels in each phase and their communication behaviors. The extracted information is vital for subsequent decisions in design space exploration stages of task scheduling and mapping process in heterogeneous reconfigurable architectures. It can also serve as hints for application developers for optimization opportunities, such as parallelism detection in the application code.

tQUAD is implemented as a Dynamic Binary Analysis (DBA) tool. This means that, in order to profile an application, we only need the binary machine code of the application. DBA tools are commonly developed utilizing a Dynamic Binary Instrumentation (DBI) framework. Instrumentation is a technique for injecting extra code into an application to observe its behavior. This process can be performed at various stages: in the source code, at compile-time, at post-link time, and at run-time. tQUAD, a run-time instrumentation profiler, is implemented using the Pin [25] run-time binary instrumentation framework that utilizes dynamic compilation to instrument executables while they are running.

A. Pin

Pin provides a portable, transparent, and efficient instrumentation system that works with unmodified Linux, Windows and MacOS binaries on Intel ARM, IA32, 64-bit x86, and Itanium architectures. Pin has a rich API that is designed to be architecture independent whenever possible, making a DBA tool source code compatible across different architectures. However, a tool can still access architecture-specific details when necessary. Instrumentation with Pin is mostly transparent, as the application and the tool observe the application's original, uninstrumented behavior. Dynamic instrumentation is particularly beneficial for profiling, performance evaluation, or bug detection tools. It captures the execution of arbitrary shared libraries in addition to the main program and it has no dependence on the instrumented application's compiler. Requiring only a binary and being compiler-independent does not imply that the source code is not needed for program revisions. Instead, it provides

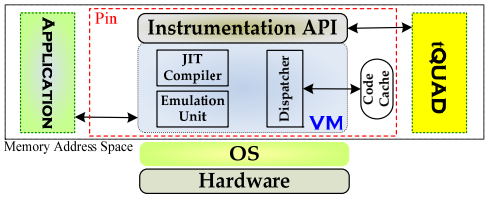

Fig. 2. Architectural overview of tQUAD.

flexibility for the tool to be language-independent and it can be used with any compiler toolchain that produces a common binary format. Furthermore, it does not require the user to modify the build environment to recompile the application with special profiling flags.

It is worth to note that, in run-time instrumentation, we do not necessarily have any kind of extra information about the structure of the program in the binary code, such as control or data flow graphs. As a result, any required information should be extracted during the dynamic execution of instructions by the tool itself. As an example, we needed to implement our own call graph. For this purpose, an internal call stack data structure is dynamically created and maintained in tQUAD.

B. tQUAD Architecture

Figure 2 shows the architectural overview of tQUAD and its interactions with the Pin's components. At the highest level, there is a Virtual Machine (VM), a code cache, and an instrumentation API. The VM consists of a Just-In-Time (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 which 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. As Figure 2 shows, three binary programs are present when an instrumented program is running: the application, Pin, and tQUAD. Pin is the engine that instruments the application. tQUAD contains the instrumentation and analysis routines and it is linked with a library that allows tQUAD to communicate with Pin.

C. tQUAD Implementation

The details associated with the modules in tQUAD are omitted from this paper for brevity. However, we provide a detailed overview of the routines in the tQUAD implementation. The interfaces to most run-time binary instrumentation systems are API calls that allow developers to hook in their instrumentation routines. Figure 3 shows the pseudocode of the main tQUAD interface in C++ style. At the beginning, there are several initializations for memory bandwidth usage data list, internal call stack and a mutual kernel-to-bandwidth data map list. *PIN_InitSymbols* must be called to access functions by name.

```
//Data Structures
class CallStack; // ADT for internal call stack
class MBWUDataList { // Memory Bandwidth Usage data
    list<MACC> mbwulst;
    // other members
    } MBW;
class K2BW; // ADT for kernel to MBW usage data mapping
//Global Variables
   string mainImg; // the main image name
   ofstream fprofile; // temporary flat profile for snapshots data
   UINT64 TSInterval; // the time slice interval
   BOOL Uncommon_Functions_Filter=TRUE;
   BOOL No_Stack_Flag = FALSE;
   UINT64 CurrentInstructions=0; // current number of executed instructions
   UINT64 SliceNumber=1; // current time slice number
   CallStack CS;
  K2BW Kernels2MBWListMap;
// Command line Options
KNOB<UINT64> KnobSlice(KNOB_MODE_WRITEONCE, "pintool","slice","500000",... );
KNOB<BOOL> KnobIgnoreUncommonFNames(KNOB_MODE_WRITEONCE, "pintool","
       filter_uncommon_functions","1", ...);
KNOB<BOOL> KnobIgnoreStackAccess(KNOB_MODE_WRITEONCE, "pintool","ignore_stack_access",
       "0",...);

int main( int argc, char *argv[] )
{
    MBW.Init();// memory bandwidth usage data list initialization
    CS.Init(); // internal Call Stack initilization
    Kernels2MBWListMap.Init(); // Kernels <--> MBWU Map
    GetMainImg(); // parse the commandline for primary image name

    PIN_InitSymbols();
    if( PIN_Init(argc,argv) ) return Usage();
    CkeckTS(TSInterval=KnobSlice.Value());
    Uncommon_Functions_Filter=KnobIgnoreUncommonFNames.Value();
    No_Stack_Flag=KnobIgnoreStackAccess.Value();

    RTN_AddInstrumentFunction(UpdateCallStack,0);
    INS_AddInstrumentFunction(Instruction,0);
    PIN_AddFiniFunction(Fini,0);
    PIN_StartProgram(); // Never returns
    return 0;
}
```

Fig. 3. tQUAD main interface pseudocode.

After initializing the Pin run-time system, the command line arguments of tQUAD are parsed to set primary parameters for the profiling process. Three options are supported, namely, the inclusion/exclusion of the local stack area memory accesses, the time slice interval setting, and the exclusion of memory bandwidth usage data caused by OS and library routine calls. When mapping a kernel on a reconfigurable device, there may be the possibility to allocate the corresponding local buffer on the hardware as well, provided that enough space is available for the size of needed memory block. In this case, all the local memory accesses should be distinguished from external memory accesses. In tQUAD, we provide the option to estimate the memory bandwidth usage including or excluding the local stack area accesses. Time slice interval is a key parameter which adjust the detailing degree of the extracted memory bandwidth usage information. With large time slices, we lose some information and a coarser view of the memory bandwidth usage of kernels, is obtained. Library and OS routines usually are not of interest to the user, therefore, tQUAD has the option to exclude them from the internal call stack.

In Pin, the API call to *INS_AddInstrumentationFunction()* allows a user to instrument programs based on a single instruction while the *RTN_AddInstrumentFunction()* provides instrumentation capability at routine granularity. We use these two API routines to set up calls to the instrumentation routines *Instruction()* and *UpdateCallStack()*, respectively.

Figure 4 shows the body of the *Instruction()* instrumentation routine. The *Instruction()* instrumentation routine sets up the call to the analysis routine *IncreaseRead()* every time an instruction that references memory read is executed. There is a similar process in the case of memory write reference. *Instruction()* also

```
VOID Instruction(INS ins, VOID *v)
{
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)IncTotalInstCount, IARG_END);
    if (INS_IsRet(ins))  // return from routines is monitored
        INS_InsertPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR)Return, IARG_INST_PTR,
                IARG_END);
    if (!No_Stack_Flag)  // stack accesses ok
    {
      if (INS_IsMemoryRead(ins) || INS_IsStackRead(ins))
        INS_InsertPredicatedCall(ins,IPOINT_BEFORE,(AFUNPTR)IncreaseRead,
                IARG_MEMORYREAD_SIZE,IARG_UINT32,INS_IsPrefetch(ins),IARG_END);
      if (INS_HasMemoryRead2(ins))
        INS_InsertPredicatedCall(ins,IPOINT_BEFORE,(AFUNPTR)IncreaseRead,
                IARG_MEMORYREAD_SIZE,IARG_UINT32,INS_IsPrefetch(ins),IARG_END);
      if (INS_IsMemoryWrite(ins) || INS_IsStackWrite(ins))
        INS_InsertPredicatedCall(ins,IPOINT_BEFORE,(AFUNPTR)IncreaseWrite,
                IARG_MEMORYWRITE_SIZE,IARG_UINT32,INS_IsPrefetch(ins),IARG_END);
    } // end of Stack is ok!
    else  // ignore stack accesses
    {
      if (INS_IsMemoryRead(ins))
        INS_InsertPredicatedCall(ins,IPOINT_BEFORE,(AFUNPTR)IncreaseReadSP,
                IARG_REG_VALUE,REG_STACK_PTR,IARG_MEMORYREAD_EA,IARG_MEMORYREAD_SIZE,
                IARG_UINT32, INS_IsPrefetch(ins),IARG_END);
      // similar calls for NS_HasMemoryRead2 & INS_IsMemoryWrite(ins)
    } // end of ignore stack
    // check for the snapshot point
    INS_InsertCall(ins,IPOINT_BEFORE,(AFUNPTR)Slice_checkpoint,IARG_END);
}
```

Fig. 4. tQUAD instruction instrumentation pseudocode.

monitors instructions for the return from a function to maintain the integrity of the internal call stack. When Pin starts the execution of an application, the JIT calls *Instruction()* to insert new instructions into the code cache. If the instruction references memory or signals the return from a function, tQUAD inserts a call to the corresponding analysis routine before the instruction, passing the required arguments which can be the Instruction Pointer (IP), the number of bytes read or written, and a flag showing whether or not the instruction is a prefetch. The corresponding analysis routines return immediately upon detection of a prefetch state for an instruction. *INS_InsertPredicatedCall()* injects the analysis routine and ensures that the analysis routine is invoked only if the instruction is predicated true. When local stack area memory accesses have to be excluded, the Stack Pointer (*REG_STACK_PTR*) is also passed as an extra argument to the analysis routine for subsequent processing. Furthermore, *Instruction()* is responsible to initiate the time simulation and memory bandwidth snapshot managements.

The code for the *UpdateCallStack()* instrumentation routine is presented in Figure 5. The *UpdateCallStack()* instrumentation routine sets up the call to the analysis routine *EnterFC()* every time a function is called during program execution. This is necessary to update the internal call stack. Since tQUAD ignores the functions which are not in the main image file of the program, *flag* is used as a signal to indicate the location of the newly-called function. The name of the function, as reported by Pin, is also passed for the internal call stack update process.

V. Case Study

tQUAD was tested on a set of real applications. Nevertheless, due to space limitations, the rest of this section presents the detailed results of only one of them, the *hArtes wfs* audio processing application. The main goal is to present a preliminary approximation of the application behavior during its execution in function of the memory bandwidth used by the kernels through time. The extracted information can be further used for critical decisions, such as HW/SW task partitioning, mapping and scheduling, in design space exploration on heterogeneous

```
VOID UpdateCallStack(RTN rtn,VOID *v)
{
    bool flag;
    char *rNtemp;
    string rName;
    flag=(!((IMG_Name(SEC_Img(RTN_Sec(rtn))).find(mainImg)) == string::npos));
    rName=RTN_Name(rtn);
    rNtemp=new char[strlen(rName.c_str())+1];
    strcpy(rNtemp,rName.c_str());
    RTN_Open(rtn);
    // Insert a call at the entry point of a routine to update Call Stack
    RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)EnterFC, IARG_PTR, rNtemp, IARG_BOOL,
            flag, IARG_END);
    RTN_Close(rtn);
}
```

Fig. 5. tQUAD routine instrumentation.

reconfigurable architectures. The information can also be useful to spot bottlenecks related to the memory usage on these systems as well as to assist the application developers to revise the application code to increase the performance gain on a particular architecture.

The Wave Field Synthesis (WFS) [26] is a spatial audio rendering technique characterized by the creation of virtual acoustic environments. It produces artificial wavefronts synthesized by a large number of individually driven speakers. Each of these speakers is activated at the exact time when the desired virtual wavefront passes through it to reproduce the original wavefront of the audio source. The *hArtes wfs* application provided by Fraunhofer IDMT [27] implements a self-contained wave field synthesis system.

A. Experimental Setup

The experiments were executed on an Intel 64-bit Core 2 Quad CPU Q9550 @ 2.83GHz with a main memory of 8GB, running Linux kernel v2.6.18-164.6.1.el5. The *hArtes wfs* source code was compiled with *gcc* v3.4.6. To use the *gprof* [15] general profiler, the program was compiled and linked with the *-pg* profiling option enabled along with the *-g* option for the debugging information to be available. The 64-bit version of tQUAD profiler was used with the command line options to include/exclude stack area memory access and to adjust the time slice interval ranging from 5000 to 10^8 instructions per time slice. The *hArtes wfs* runs in off-line mode. This means that the input audio source is read from files instead of audio devices. In all the experiments, we used one primary wavefront source and thirty two secondary audio sources (speakers).

Instrumentation-based tools can considerably slow down the execution of an application. Since memory read and write instructions are executed frequently in multimedia applications, the overhead of intercepting and checking these instructions is very high. tQUAD instruments every *load*, *store*, *call* and *return* instruction, which will result in a slowdown of the execution of the *hArtes wfs* ranging from 37.2 X to 68.95 X compared to native execution. The amount of introduced overhead is strongly dependent on the time slice and the option to include/exclude stack area accesses. Despite of the significant slowdown, the execution time is comparable with the expected slowdown [28], hence, acceptable for a realistic working set data.

B. Experimental Analysis

In this case study, we specifically aim to achieve the following goals:

- the extraction of information about the intensity of the memory bandwidth usage for each kernel during its execution. This information is critical in finding the potential memory access related bottlenecks when the application is executed on a particular heterogeneous reconfigurable architecture;
- the discovery of the exact timing and activity span (starting and ending points of a kernel execution). The extracted information is required for task scheduling and mapping on reconfigurable systems [29];
- the identification of the main phases in the application based on the activity spans of the kernels. The kernels that are active at the same time interval are possibly relevant (communicating). This information can be utilized later in the task clustering framework to efficiently partition the application [22].

In order to achieve these goals, several experiments were carried out. First, we used *gprof* to identify the top kernels in the application. Then, QUAD is used in order to have an overview of the amount of data communication between the kernels in the application. Based on the extracted data, a discussion on the potential memory access problems is subsequently presented. We also profiled the QUAD-instrumented version of the *hArtes wfs* application to understand the effect of data communication in the overall contribution of each kernel in the application. Finally, tQUAD profiler is used in a series of experiments to extract the timing information for each kernel and to identify the main phases in the application.

gprof profiling data. The *hArtes wfs* application consists of 64 functions. We used *gprof* to identify the top computation-intensive kernels for further inspection. The run-time figures that *gprof* provides are based on a sampling process. As a consequence, they are subject to statistical inaccuracy, particularly if a function runs only for a small amount of time. The sampling period, which is one hundredth of a second, is a good indication of the accuracy of a function figure regarding its total running time. If the total run-time of a program is large, a small run-time value for a function indicates that the function used an insignificant fraction of the whole execution time. We run the program fifty times to gain more accuracy. The results are summarized and presented in Table I.

wav_store and **fft1d** are the top two kernels. These two kernels take approximately sixty percent of the whole execution time of the program. **wav_store** saves the output audio signals from buffers allocated in memory to an output file in the *wav* format. **fft1d** implements a fast algorithm to compute the one-dimensional Discrete Fourier Transform (DFT) using the in-place (no additional memory allocation) butterfly *Danielson-Lanczos* method.

QUAD profiling data. Table II provides an overview of the amount of data communication between kernels in the form of producer/consumer bindings. The results take into consideration the inclusion and the exclusion of the stack area memory accesses. By comparing the data extracted from the individual cases, a lot of information can be derived. From Table II, it

TABLE I
FLAT PROFILE FOR THE *hArtes wfs* APPLICATION.

Kernel	%time	self seconds	calls	self ms/call	total ms/call
wav_store	31.91	0.28	1	277.25	277.25
fft1d	28.23	0.25	984	0.25	0.25
DelayLine_processChunk	14.23	0.12	493	0.25	0.38
bitrev	8.19	0.07	2015232	0.00	0.00
zeroRealVec	7.44	0.06	15782	0.00	0.00
AudioIo_setFrames	4.01	0.03	493	0.07	0.07
perm	2.07	0.02	984	0.02	0.09
cadd	0.79	0.01	1009664	0.00	0.00
cmult	0.73	0.01	1009664	0.00	0.00
Filter_process	0.71	0.01	493	0.01	0.73
wav_load	0.44	0.00	1	3.80	3.80
Filter_process_pre_	0.35	0.00	493	0.01	0.35
zeroCplxVec	0.28	0.00	495	0.00	0.00
r2c	0.16	0.00	490	0.00	0.00
c2r	0.14	0.00	493	0.00	0.00
AudioIo_getFrames	0.14	0.00	489	0.00	0.00
ffw	0.08	0.00	2	0.35	0.35
vsmult2d	0.02	0.00	7026	0.00	0.00
calculateGainPQ	0.02	0.00	6994	0.00	0.00
PrimarySource_deriveTP	0.02	0.00	236	0.00	0.00
ldint	0.01	0.00	1	0.10	0.10

% 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.

can be seen that in most cases the ratio between the amount of data produced/consumed for the stack inclusion to exclusion is limited. However, it is not the case with **zeroCplxVec** and **zeroRealVec** as the ratios are greater than 750 and 300, respectively. This means that the mentioned kernels are nearly reading all the time from the local memory. In other words, they can be excellent candidates for hardware mapping provided that the corresponding input buffer is also placed on the chip. However, their intense communication with the memory for writing data into the output buffers should not be ignored. The output buffers should also be instantly accessible to fully exploit the performance gain.

Approximately half of the giga bytes read by **wav_store** is from the stack memory area. This indicates that the data has been produced inside the function for further internal processing. However, the used memory addresses are almost the same. This means that a small area is locally allocated inside the function for temporary storage, compared to the global (non-stack) memory (60 MB vs. KB). The need to fetch data out of sixty five millions distinct locations into **wav_store** can pose a serious bottleneck. By examining the QDU (Quantitative Data Usage) graph[1] of the *hArtes wfs*, which is produced by QUAD, other useful information can be derived. For example, it turns out that nearly all the data produced by **wav_store** are used internally and the kernel discloses very limited amount of data for the other kernels. This remark can also be verified by the small number of Unique Memory Addresses (UnMAs) used as output buffers compared to the huge amount of data produced (hundreds of addresses per GBs). The **fft1d** case is somehow different as the ratio of stack inclusion to exclusion is

[1]It is not possible to include the large graph in this paper due to space limitations.

TABLE II
SUMMARY OF THE DATA PRODUCED/CONSUMED BY THE KERNELS IN THE *hArtes wfs* APPLICATION.

kernel	Stack area accesses excluded				Stack area accesses included			
	IN	**IN UnMA**	**OUT**	**OUT UnMA**	**IN**	**IN UnMA**	**OUT**	**OUT UnMA**
AudioIo_getFrames	2082977	2003143	2030924	4178	2193001	2003319	2132616	4290
AudioIo_setFrames	65642447	131797	64790862	64618668	66910617	131955	65875370	64618788
DelayLine_processChunk	136426363	187911	130079532	162800	1207848481	188349	1199055238	163146
Filter_process	76962891	65853	8367732	16562	166795095	66075	113578568	16744
Filter_process_pre_	8159527	16623	8288564	16480	8310811	16807	8428110	16614
PrimarySource_deriveTP	28658	271	9504	248	102558	785	81336	750
bitrev	147305084	145	64488030	86	1092514838	397	991569196	214
c2r	2062775	4231	2019224	4180	22360399	4433	22271396	4310
cadd	73825250	129	32309436	82	203213962	377	153474676	194
calculateGainPQ	654672	305	223904	270	2977380	1151	6046220	1384
cmult	73767500	137	32309306	74	235522840	393	185786118	194
fft1d	541111698	115143	348733474	86182	3377052372	115439	3178842792	86370
ffw	571706	4863	177374320	16640	832298	5496	177633766	17151
ldint	81	73	72	64	399	231	336	168
perm	15747216	55745	31271422	47762	190358486	55985	221582640	47914
r2c	2048600	4331	8028298	8458	26181770	4571	32117142	8600
vsmult2d	513564	159	224864	152	1414418	705	1807246	690
wav_load	73166075	5606	118994504	2000393	148386954	6668	194027099	2001719
wav_store	3407275698	64941803	1754503491	392	5946326334	64942676	4282480373	1115
zeroCplxVec	48499	171	8151616	41130	36631679	417	44664318	41282
zeroRealVec	1257818	219	65398908	140194	391633848	537	454905252	140406

IN represents the total number of bytes read by the function; ***IN UnMA*** indicates the total number of unique memory addresses used in reading; ***OUT*** represents the total number of bytes read by any function in the application from memory locations that the specified function has previously written to; ***OUT UnMA*** indicates the total number of unique memory addresses used in writing.

approximately ten. This indicates that most of the computations are performed inside the kernel. It is also worth noting that, the size of the locally allocated memory used for temporary results is rather nominal due to the fact that the UnMAs reported in the two cases remain identical. The immediate outcome of this observation is that **fft1d** is a better candidate than **wav_store** for hardware mapping onto a reconfigurable device. This is particularly true if there is an intention to map the corresponding local buffers as well.

As a general remark from Table II, although all the kernels are intensely communicating with memory, which is common for A/V processing applications, the size of the memory addresses used for data transfer is limited (100MB-1GB of data vs. KBs of UnMAs). However, a thorough analysis of the data in Table II discloses a critical potential bottleneck arising from the memory access pattern of **AudioIo_getFrames** and **AudioIo_setFrames**. In these kernels, the data transfer is carried out via separate memory addresses. This is the reason why the number of bytes and UnMAs are almost identical in the corresponding columns. The case is quite critical for the data written into the memory addresses in **AudioIo_setFrames** (more than 60 MB of data are saved in distinct memory addresses). This behavior, undoubtedly, will surpass any performance gain that can be achieved by running the kernel in hardware mode, for example, on an FPGA. As a matter of fact, **AudioIo_setFrames** is responsible for copying interleaved audio signal parts into relevant audio frames in the memory. This is the reason why it is saving data in completely separate locations. The detailed information in the QDU graph can even allow us to trace back the source of the data which is originating from **DelayLine_processChunk**. Later, **AudioIo_setFrames** passes the data to **wav_store** to be processed. By examining Table I, we can see that **AudioIo_setFrames** is only contributing to four percent of the whole execution time. Nevertheless, with the huge impact of the memory communication problem, it seems underestimated. *Unfortunately, general profilers like gprof are not able to provide an accurate estimation of the memory access overhead impact on the overall kernel performance when a program is profiled. In fact, the timing information estimated by gprof can not precisely describe the behavior of an application in practice, particularly when there is an extreme interaction with the memory system whose response time is influenced by some critical parameters.*

TABLE III
FLAT PROFILE FOR QUAD-INSTRUMENTED VERSION OF *hArtes wfs* APPLICATION.

kernel	% time	self seconds	rank	trend
wav_store	33.69	346.93	1	↔
fft1d	30.35	312.46	2	↔
DelayLine_processChunk	10.85	111.75	4	↓
bitrev	0.42	4.33	11	⇊
zeroRealVec	3.14	32.30	5	↓
AudioIo_setFrames	11.19	115.18	3	↑↑
perm	1.52	15.69	7	↔
cadd	0.39	0.01	13	↓
cmult	2.12	21.80	6	↑
Filter_process	0.67	7.04	8	↔

% 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; ***rank*** is the position of the function among all the kernels; ***trend*** shows the intensity to increase or decrease the function's contribution compared to the initial flat profile.

QUAD-instrumented profiling data. We profiled the QUAD-instrumented version of the *hArtes wfs* to have a more practical overview of the application's behavior. Certainly, this version tends to reveal the data communication overhead introduced by accessing individual memory addresses. Furthermore, it stresses costly global memory accesses in contrast to the less expensive local memory references during the execution of the program.

Table III summarizes the results for the previously identified top ten kernels. The considerable increase in the timing contribution of each kernel is accounted to the overhead introduced by the instrumentation and analysis routines. Meanwhile, the ranking of kernels in this version is somehow more representative of a real execution, particularly on systems that have a very expensive access cost for external memory compared to mapped on-chip local buffers. The reason is that the instrumentation routine simply discards the local stack area accesses and only upon detection of a non-local memory access, an analysis routine is called to handle a tracing process. It is worth noting that, due to the long running time of the instrumented program (a couple of hours) the statistics extracted from the flat profile show a high level of accuracy. Only very slight deviations can be detected in different runs. As expected, there is a substantial increase in the contribution of **AudioIo_setFrames** from four to eleven percent. By profiling the QUAD-instrumented version of the program, we can distinguish between local and global memory accesses. We can also take into account the size of the memory blocks used in the data transfers. For instance, **bitrev** and **DelayLine_processChunk** have more or less the same ratio of including to excluding stack area accesses. **bitrev** shows a severe drop on the execution time contribution (from 8.19 to 0.42). However, this is not the case with **DelayLine_processChunk**. This observation is justified by looking at the reported UnMA usage for the two kernels in Table II. **bitrev** only uses around one tenth of a KB as buffer, whereas **DelayLine_processChunk** accesses about 180 KB of memory locations.

The kernels in the *hArtes wfs* show a huge diversity in the number of times they are called, ranging from one to millions of calls. The huge number of calls does not necessarily mean that the corresponding kernel has a large contribution to the total execution time. Instead, the highly-called kernels have often quite a simple body. Anyhow, the case of the top kernel in *hArtes wfs* is quite interesting: **wav_store** is called only once and it has the contribution of about one third for the whole execution time. It clearly indicates that the kernel must be active in a large time span during the execution of the program.

tQUAD profiling data. We utilized tQUAD to have a clear view of the running times of the kernels in the program. The extracted information is depicted in the form of running time graphs in Figure 6. The x-axis is the execution time. Each unit represents the time slice which is set to 10^8 instructions span. The y-axis represents the intensity of the memory accesses for each kernel at a specified time slice. The graphs for different kernels are shown along the z-axis. This makes it easy to compare the memory access behaviors of the kernels in each time slice. As expected, **wav_store** is called approximately in the middle of the execution time. It is silent in the first half and it is the only kernel active in the second half.

Figure 6 also shows the memory bandwidth usage of the top ten kernels in the *hArtes wfs* related to the memory read accesses including the stack area. Memory write accesses have almost similar figures but the intensity of the data transfers is less by at least a factor of two in most kernels. The time slice interval is set to 10^8, i.e., a snapshot of the memory bandwidth

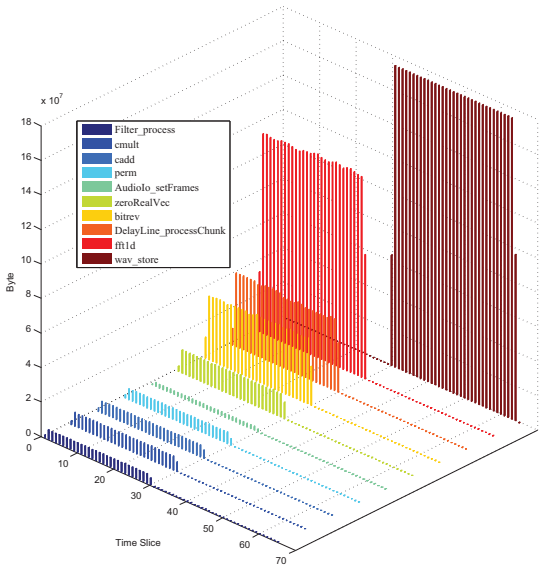

Fig. 6. Memory bandwidth usage of the kernels in the *hArtes wfs* considering only the read accesses including the stack area.

usage is recorded every hundred millions instructions. In total, 64 time slices are counted representing the execution of more than six billion instructions for the completion of the program. Setting the time slice interval to a large number causes the loss of detailed information. This is evident in the density of the produced graphs. Small time slice intervals are preferable for more accurate estimations. Figure 7 depicts the relevant graphs for the last ten kernels. Here, the time slice interval is set to $25 * 10^6$, which provides a more detailed view of the kernels' running time. The second half of the total 255 time slices is cut off, as no kernel but **wav_store** is active during this period. The graphs depict the memory bandwidth usage of the kernels regarding the memory write accesses excluding the stack area. The memory access patterns of all kernels are strictly regular in *hArtes wfs*. This is common in nearly all the applications from multimedia domain as the processing algorithms are well formulated to work on predefined data blocks.

Phase identification. tQUAD recognizes five different phases in the whole execution span of the *hArtes wfs* by the thorough examination of different graphs. Several experiments were carried out to extract the required measurements for each phase. A summary of the results is presented in Table IV. This can depict a clear image of the related active kernels in each phase including their self contributions and memory access patterns during the execution time of the program. The phases identified are mainly based on the role of the active kernels in that particular time span. Nevertheless, some kernels, such as **bitrev**, are utilized in a more general way, which causes the phases to overlap if we only consider the activity time span of the kernels. We set the time slice interval to 5000 in order to have accurate estimations of the memory bandwidth usage. However, the data presented in Table IV are still prone to slight statistical inaccuracy. Nevertheless, this inaccuracy should, in no sense,

TABLE IV
PHASES IN THE EXECUTION PATH OF THE *hArtes wfs* APPLICATION.

phase	phase span	% phase span	kernel	activity span	average memory bandwidth usage read access stack incl.	read access stack excl.	write access stack incl.	write access stack excl.	maximum memory bandwidth usage (R+W) stack incl.	stack excl.	aggregate MBW
initialization	53-144	0.007	ffw	92	1.8071	1.2422	0.4807	0.1811	2.4704	1.6376	2.6018
			ldint	1	0.0798	0.0162	0.0516	0.0176	0.1314	0.0338	
wave load	552-14660	1.1103	wav_load	14109	2.0993	1.0358	1.0355	0.9929	3.1566	2.0664	3.1566
wave propagation	540-274868	21.5891	vsmult2d	1570	0.1799	0.0655	0.1182	0.0503	0.3996	0.1548	1.4530
			calculateGainPQ	1600	0.3708	0.0815	0.2633	0.0847	0.7714	<0.2116	
			PrimarySource_deriveTP	235	0.0870	0.0240	0.0547	0.0208	0.2820	0.0980	
WFS main processing	14663-592803	45.4983	fft1d	278781	2.4179	0.3876	0.3501	0.1331	2.8738	<0.6428	<84.1862
			DelayLine_processChunk	115546	2.0859	0.2356	0.2339	0.1180	<3.3316	1.7050	
			bitrev	116755	1.8677	0.2521	0.7457	0.1934	<2.8778	0.4966	
			zeroRealVec	36304	2.1529	0.0145	0.7233	0.3610	<2.9386	<0.4028	
			AudioIo_setFrames	616	21.5553	21.8035	21.0646	21.5860	<53.2686	<52.7330	
			perm	116776	0.3252	0.0280	0.0956	0.0545	<0.6556	<0.1466	
			cadd	41076	0.9882	0.3590	0.6686	0.2753	1.6946	0.6514	
			cmult	41073	1.1456	0.3590	0.7080	0.2753	1.8946	0.6594	
			Filter_process	42583	0.7789	0.3609	0.1332	0.1141	<0.9768	0.5064	
			Filter_process_pre_	1487	1.1113	1.6384	1.6267	1.6290	<3.3862	<3.3302	
			zeroCplxVec	4132	1.7693	0.0141	0.5913	0.3926	2.6874	<0.4710	
			r2c	2716	1.9250	0.1510	0.4474	0.2983	<2.9386	<0.5642	
			c2r	2318	1.9251	0.1774	0.3549	0.1769	2.9138	0.4658	
			AudioIo_getFrames	502	0.8701	0.8296	0.8268	0.8137	1.7482	1.6866	
wave save	592804-1270674	53.3469	wav_store	677871	1.7492	1.0033	0.8064	0.7765	2.7244	1.9044	2.7244

phase span indicates the starting and ending time slices for the phase; ***% phase span*** is the percentage of the phase time interval to the program whole execution time span; ***activity span*** represents the number of time slices in which the kernel is active (accesses memory); ***memory bandwidth usage*** is measured in bytes per instruction; ***aggregate MBW*** represents the summation of all kernels' maximum memory bandwidth usages in the phase including the stack area accesses.

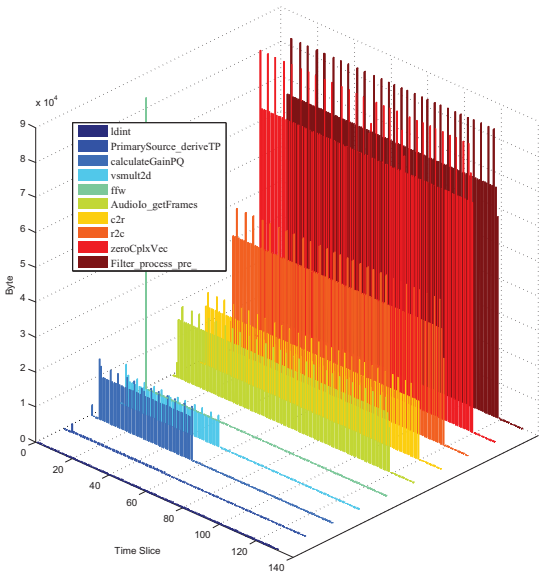

Fig. 7. Memory bandwidth usage of kernels in *hArtes wfs* considering only the write accesses excluding the stack area.

distort the overall behaviors of the kernels. 1270684 time slices were measured in total.

The *phase span* column in Table IV indicates the earliest starting point and the latest ending point in which a kernel in the phase is communicating with the memory. It should be noted that, we only consider the kernels previously selected and not all the functions in the *hArtes wfs*. Moreover, there are cases in which kernels are activated in a short period of time outside the identified span. We merely ignore these cases with respect to the overall memory access pattern of the kernels in a phase. As an example, **r2c** gets active in the $145th$ time slice for a very short time and then becomes silent until the $14663th$ time slice. Furthermore, the phase span does not necessarily mean that all the kernels within that phase are active through the whole time slices. They can be quite active, such as **fft1d** or less active, such as **AudioIo_getFrames**. This behavior has nothing to do with the intensity of the memory communications of a kernel. As an example, **perm** is moderately active in the fourth phase. However, the memory communication is not intense at all. On the other hand, **AudioIo_setFrames** is only active in rather small time intervals but acts truly intensive in memory referencing. tQUAD is capable of providing the detailed information about the exact time intervals in which a kernel is communicating with the memory. It also analyzes the data to identify the boundaries of potential phases. Since it was not possible to present in this paper the detailed profiled information due to space limitations, only a summary of the results is presented.

The average memory bandwidth usage is calculated over several passes with different time slices. The data are normalized as number of bytes-per-instruction. In this way, it is possible to have a general estimation of the kernel's architecture-independant intensity. If a more specific unit of measurement is needed, additional parameters for the target architecture should be provided for tQUAD, such as the number of PE cycles required to execute each instruction. It is also possible to derive different measurement units, such as bytes-per-cycle or bytes-per-second. The maximum memory bandwidth usage represents the maximum bytes-per-instruction measured in the peak of the communication with memory counting both read and write accesses. For some of the kernels in Table IV, the upper bounds are specified. This is due to the fact that slight inconsistencies in the measurements of the overall time slices were detected in

the experiments. Moreover, the detailed information about the timings of the maximum bandwidth usage is not presented here.

The initialization phase runs only for a very short time interval, which makes it rather uninfluential in the overall analysis. The second phase contains only one kernel which is active throughout the whole time span. The kernels appearing in the third phase are related to the implementation of a MIMO delay line and wave propagation computations for an array of speakers. Although the activity span of the kernels in the this phase cover more than one fifth of the whole execution time, they have a nominal share of the memory bandwidth traffic. The main WFS processing is carried out in the fourth phase, during which, fourteen kernels are active. As expected, this phase has the biggest share of the whole memory bandwidth traffic. **Filter_process_pre_** has almost identical amount of memory bandwidth usage in the cases of including and excluding the stack area accesses. **AudioIo_setFrames** and **AudioIo_getFrames** have similar trait. This also conforms to the information presented in Table II for the mentioned kernels. As mentioned before, **AudioIo_setFrames** shows a completely unique attribute among all the kernels in the *hArtes wfs* application. The intensity of the maximum memory bandwidth usage for this particular kernel reaches over 50 bytes per instruction, while for all the others, it is at most 3 bytes per instruction. Some kernels, such as **DelayLine_processChunk**, show a severe drop in the memory bandwidth usage by a factor of 10 when excluding stack area accesses. In the cases of **zeroRealVec** and **zeroCplxVec**, the factor is more than 125. Further investigation of the information in the flat profile (not presented here) also reveals that, by excluding the stack area accesses, the activity spans of these kernels are reduced by a factor of 2 and 8, respectively. **wav_store** is the only kernel in the last phase. It is the only kernel that is active for more than half of the whole execution time span and, yet, it can not be exclusively as influential as the main WFS processing phase.

The information about the phases and the active kernels in each phase, along with the quantitative data of memory access behavior of each kernel, provide valuable clues for the clustering framework in the DWB to partition the whole application with respect to certain criteria. Most importantly, some relevant kernels are clustered together in a sense that the intra-cluster communication is maximized whereas the inter-cluster communication is minimized.

VI. Conclusion

One of the major challenges in computing is the well-known processor/memory bottleneck. As a result, tools for the analysis of the task execution and tools for the analysis of the memory usage become vital. In this paper, we presented tQUAD, a memory bandwidth usage analysis tool which is capable of providing both information. Via a detailed analysis and classification of tasks into specific phases, the tool provides a detailed analysis of both the execution and the memory bandwidth usage for all the tasks. The proposed tool, first in its kind, is presented together with a thorough analysis of a case study form the multimedia domain. In future work, we are planning to utilize the information provided by the tool for task clustering in heterogeneous reconfigurable systems.

Acknowledgment

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

References

[1] F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-vincentelli, "Scheduling for embedded real-time systems," *IEEE Design and Test of Computers*, vol. 15, pp. 71–82, 1998.

[2] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proc. of the IEEE*, vol. 82, no. 1, pp. 55–67, 1994.

[3] A. Ahmadinia, C. Bobda, D. Koch, M. Majer, and J. Teich, "Task scheduling for heterogeneous reconfigurable computers," in *Proc. of SBCCI '04*, 2004, pp. 22–27.

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

[5] K. Bertels, et al., "Developing applications for polymorphic processors: the delft workbench," Tech. Rep., January 2006.

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

[7] R. Wilhelm, et al., "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 1–53, 2008.

[8] aiT, http://www.absint.com/ait/.

[9] Bound-T, http://www.tidorum.fi/bound-t.

[10] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," vol. 69, no. 1-3, 2007, pp. 56–67.

[11] Heptane WCET analysis tool, http://www.irisa.fr/aces/work/heptane-demo.

[12] SWEdish Execution Time tool, http://www.mrtc.mdh.se/projects/wcet/sweet.html.

[13] RapiTime, http://www.rapitasystems.com.

[14] Vienna real-time systems group, http://www.vmars.tuwien.ac.at.

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

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

[18] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," 1993, pp. 128–137.

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

[20] 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 *Proc. of FPGA '09*, 2009, pp. 279–279.

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

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

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

[24] Y. D. Yankova, et al., "Dwarv: Delftworkbench automated reconfigurable VHDL generator," in *Proc. of FPL '07*, 2007, pp. 697–701.

[25] C.-K. Luk, et al., "Pin: building customized program analysis tools with dynamic instrumentation," in *Proc. of PLDI '05*, 2005, pp. 190–200.

[26] A. J. Berkhout, D. de Vries, and P. Vogel, "Acoustic control by wave field synthesis," *The Journal of the Acoustical Society of America*, vol. 93, no. 5, pp. 2764–2778, 1993.

[27] Fraunhofer Institute for Digital Media Technology, http://www.idmt.fraunhofer.de/eng/research_topics/wave_field_synthesis.htm.

[28] G. Uh, R. Cohn, B. Yadavalli, R. Peri, and R. Ayyagari, "Analyzing dynamic binary instrumentation overhead," in *WBIA Workshop at ASPLOS*, 2006.

[29] Y. Lu, T. Marconi, K. Bertels, and G. Gaydadjiev, "A communication aware online task scheduling algorithm for FPGA-based partially reconfigurable systems," in *Proceedings of IEEE Symposium on FCCM*, May 2010, pp. 65–68.