Runtime Extraction of Memory Access Information from the Application Source Code

S. Arash Ostadzadeh, Marco Corina, Carlo Galuzzi, and Koen Bertels
Computer Engineering Group
Delft University of Technology, Delft, the Netherlands
{S.A.Ostadzadeh,M.Corina,C.Galuzzi,K.L.M.Bertels}@TUDelft.nl

ABSTRACT

The shift towards using increasing numbers of processing elements has placed new burdens on the programming community to fully exploit the potential performance gain of multiprocessor systems. The programming problem is even more complex in the case of systems that utilize reconfigurable devices. The increased complexity of programming necessitates the use of tools that can support programmers in migrating existing applications to these emerging systems. Programmers need increasingly sophisticated tools for profiling and analysis of applications. Particularly, tools to inspect the memory access behavior of applications become crucial due to the processor/memory communication bottleneck. In this paper, we present xQUAD, a unique extension to the QUAD dynamic profiling toolset, which augments the memory access analysis of an application by providing detailed, fine-grained intra-function information. xQUAD provides detailed memory access information on the application source code data object granularity. This information can help programmers for application optimization and revision. We also present a ranking method based on the memory access intensity of a function, which reveals more accurately the suitability of the function for hardware implementation. xQUAD is tested on a real application from the multimedia domain to describe the capabilities of the proposed toolset.

KEYWORDS: Dynamic profiling, Instrumentation, Performance analysis, Code tuning, Reconfigurable architectures, Hardware/Software partitioning.

1. INTRODUCTION

In the last decades, the rate of improvement of processor performance has greatly exceeded the rate of improvement of memory performance. This phenomenon is the main obstacle limiting the overall performance of applications on computing systems. Conversely, modern applications require an increasing amount of computing power, which can not always be delivered by conventional modern processors. As an alternative to conventional computing, the introduction of heterogeneous architectures featuring reconfigurable devices, such as FPGAs, has demonstrated to greatly accelerate a wide variety of applications [1], [2]. Furthermore, the latest developments in FPGA technology have been quite beneficial for this field, making FPGA logic the de facto standard as a co-processor component. Nevertheless, the introduction of heterogeneous reconfigurable architectures has not solved the processor/memory bottleneck yet. In fact, efficient and high-performance application results on FPGAs require careful consideration of the memory bottleneck of the underlying system. Additionally, heterogeneous architectures incorporating reconfigurable devices introduce more processing elements on the same platform, which is usually reflected by the need for an increase in I/O capabilities. As a result, the performance of an application mapped onto a heterogeneous reconfigurable platform is largely determined by how much exploitable parallelism is available, and also by the ability of the system to provide data to keep the parallel hardware operational [3].

To address this issue, there is an increasing demand in software tools to help developers in analyzing their applications in terms of memory usage. The QUAD (Quantitative Usage Analysis of Data) core tool [4] provides a comprehensive overview of the actual data communications between the functions in an application. The tQUAD tool [5] focuses on the memory bandwidth issues and delivers temporal information for the functions in an application. In order to manually and/or automatically revise application source code for increased performance on hardware accelerators, it is required to have in-depth fine-grained memory access related information on individual data

objects. This information is critical in code tuning to reduce data communications between tasks running on different processing elements. The main problem with the information extracted by the QUAD toolset is that it provides a coarse view of the intra-function memory accesses. As a result, it is very difficult to attribute the extracted memory access information to particular user-defined data objects in a program at the source code level.

In this paper, we present an extension to the QUAD toolset, called *xQUAD*, which augments the memory access analysis of an application by providing detailed, fine-grained intra-function information. It identifies and reveals the correspondence between data objects defined in the application source code and the relevant memory access information reported by the QUAD toolset.

The main contributions of this paper are the following:

- the design and implementation of an extension of the QUAD toolset, which provides fine-grained memory access information on user-defined data objects in a program source code;
- the presentation of a memory access intensity index, which helps to provide a rough estimation of hardware mapping decisions for reconfigurable architectures;
- the validation of the proposed tool on a real application.

The remainder of the paper is organized as follows. Section 2 summarizes some related profiling tools. In Section 3, we provide background and description of the basic components required for xQUAD, including the instrumentation framework and the necessary module for retrieving source code level information from binary executable files. Section 4 describes the proposed tool. Finally, Section 5 concludes the paper.

2. RELATED WORK

As computing systems grow increasingly complex, profiling tools become vital to help developers in analyzing and improving the performance of applications. *Dynamic profiling tools*, contrary to *static* profiling tools, analyze programs during their execution and provide valuable information about their runtime behavior. In the context of dynamic profiling tools, in [6], the authors present an efficient profiler and tracing system called QPT. It rewrites the executable file of a program by inserting code to record the execution frequency or sequence of each basic block. The execution cost of functions in the program can be extracted from this information. *gprof* [7] is a general profiler that estimates the execution time of each function through sampling. SpixTools [8] is a collection of programs, which allows instruction-level profiling of applications. *Spix* creates an instrumented version of the user's program. As it runs, this instrumented program keeps track of how often each basic block is executed. Several tools are provided for displaying and summarizing collected data. *Spixstats* prints tables showing opcode usage, branch behavior, register usage, and other information. *Sdas* disassembles the application program and annotates the disassembled code with instruction execution counts. *Sprint* prints the source code for the application and annotates it with statement or instruction execution counts.

MemSpy [9] instruments source programs with Tango [10], an execution-driven simulator, with calls to the memory simulator for each memory reference associated with dynamically-allocated memory or explicitly-identified address ranges. Data accumulation is indexed in a 2-D space by code objects (procedures) and data objects (data allocated by an instance of a call to malloc). CPROF [11] is a cache performance profiler that annotates source listings to identify the source lines and data structures that cause frequent cache misses. By annotating lines of source code and data structures with the corresponding number of cache misses, CPROF helps the user to focus on problematic data structures and it aids the programmer in identifying types of transformations that can improve program cache behavior. The Memory Trace Visualizer (MTV) [12] is a tool that provides interactive visualization and analysis of the sequence of memory operations performed by a program as it runs.

Current dynamic profilers lack the ability to provide runtime memory usage statistics regarding individual data objects defined in an application source code. This deficiency imposes a substantial burden on programmers to efficiently detect the memory access related bottlenecks and revise the code based on the information extracted during the execution of an application. The QUAD toolset was missing the capability for recognizing data object symbols in the program source code, which made it complicated to *discretize* where, inside a kernel, a certain memory behavior appears.

3. RESEARCH CONTEXT

The QUAD toolset is developed as the dynamic profiling framework in the context of the Delft WorkBench (DWB) [13]. The DWB is a semi-automatic tool platform for integrated hardware/software co-design, targeting heterogeneous computing systems containing reconfigurable components. It targets the Molen machine organization

[14], a heterogeneous reconfigurable platform developed at Delft University of Technology. The DWB addresses the entire design cycle from profiling and partitioning to synthesis and compilation of an application and it focuses on four main steps within the entire system design, namely:

- *the code profiling and the cost modeling* [4], [5], [15];
- *the graph transformations and optimizations* [16]–[18];
- *the retargetable compiler* [19];
- *the VHDL generation* [20].

For a given application, *code profiling and cost modeling* identify which parts of the application are good candidates for hardware implementation. This decision is based on the available hardware resources and the speed-up provided by the hardware implementation of the application, or parts of it, versus a software implementation. *Graph transformations and optimizations* analyze the candidate parts of the application for hardware implementation to find out if the code segments can be clustered/partitioned according to various objectives such as hardware resource sharing. After making the decision of which parts of the code segments to implement in hardware, the code is annotated. Subsequently, the *retargetable compiler* generates new object code. Finally, the identified instructions (code segments) pass through a *VHDL generation* phase, which generates hardware description of the instructions.

4. xQUAD

Nowadays, Dynamic Binary Instrumentation (DBI) frameworks are gaining popularity among the available methods for intercepting memory accesses. These frameworks can be used to develop dynamic profilers. xQUAD falls under this category of profilers. It is implemented as a Dynamic Binary Analysis (DBA) tool using the Pin [21] DBI framework.

4.1. Pin

DBI is a technique for analyzing the behavior of an application, by injecting extra code into the application's binary at runtime. Pin provides a DBI framework for building a variety of DBA tools for multiple architectures. Instrumentation in Pin is performed by a Just-In-Time (JIT) compiler. Pin intercepts the very first instruction of the application and re-compiles the executable generating *basic blocks* code starting at this instruction, and instrumenting the code according to the specified instrumentation type. This straight-line code sequence is almost identical to the original one, except that it runs under the control of Pin. When a branch exits this basic block, Pin generates more basic blocks code for the branch target and it continues the execution. The JIT generated code and its instrumentation are saved in a code cache for future execution of the same sequence of instructions to improve performance.

```
.debug_info
COMPILE_UNIT<header overall offset = 306>:
<0><11>  DW_TAG_compile_unit
            DW_AT_stmt_list      218
            DW_AT_high_pc        0x8048571
            DW_AT_low_pc         0x80483e4
            DW_AT_comp_dir       /comp/dir/dwarf
            <1>< 686> DW_TAG_subprogram
                      DW_AT_sibling     <867>
                      DW_AT_external    yes(1)
                      DW_AT_name        main
                      DW_AT_low_pc      0x804845e
                      DW_AT_high_pc     0x8048571
                      <2><715> DW_TAG_variable
                               DW_AT_name       s
                               DW_AT_decl_file  1
                               DW_AT_decl_line  53
                               DW_AT_type       <238>
                               DW_AT_location   DW_OP_fbreg -44
            <2><699> DW_TAG_subprogram
                      -----------
                      DW_AT_name                 function_sample
                      <2><715>  DW_TAG_variable
                                DW_AT_name          m
                                DW_AT_decl_file     1
                                DW_AT_type          <238>
```

Figure 1. A Sample Debugging Information Entry

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

4.2. DWARF2 Debugging Information

The Pin DBI framework does not provide API functions for retrieving variable information. Therefore, source-level information about variables should be extracted directly from the ELF object file. By compiling the application with debugging information flag on, *gcc* augments the ELF object file with a debugging section, whose format is, unless specified otherwise, the DWARF2 Debugging Information format [22], [23]. DWARF provides debugging entries to define low-level source code representation, like, among others, information about source code types, function and object names, line numbers information, instruction addresses and effective memory address offsets. These information are stored in different sections, all prefixed with the *debug_* keyword. The most important section is the *.debug_info* section,

which is the DWARF core data structure containing the actual debugging information. These debugging information are, in turn, contained in entities called *Debugging Information Entries* (DIE), organized in a tree structure, as depicted in Fig. 1. The data layout is modified to resemble the tree structure of the *.debug_info* section.

As most modern programming languages are block structured, DWARF also follows this model. Hence, each DIE, except the topmost root DIE representing the Compilation Unit (CU) of the source file, is contained within a parent DIE and may contain children DIEs. DIEs are tree nodes, which may represent data types, variables, functions, and everything else which participates to the formation of the object code. The DIE type is specified by a *tag*. Furthermore, a DIE is composed by a set of attributes describing the type, name, source line number, location address or references to another DIE (e.g., a variable's reference to a data type specification). Following the DWARF structure, source code level information is retrieved by using the *libdwarf consumer library interface* [24]. This library abstracts away from the DWARF low-level routines, by defining wrap functions that ease the process of retrieving debugging information from object files.

4.3. xQUAD Overview

xQUAD retrieves low-level source code representations of variables from the DWARF debugging section of an executable. This is necessary as Pin does not provide any information about data symbols. For this purpose, a module is developed for retrieving low-level source code information from the debugging section of an object file. The architectural overview of the xQUAD tool is depicted in Fig. 2. xQUAD performs two types of analysis: a *detailed intra-functional variable memory access analysis* and a *global memory usage analysis* of the entire application. Performing such an analysis at this fine level of granularity can produce results, which can become soon unmanageable, both in terms of analysis content and time. Therefore, the tool is able to selectively filter out information according to the user needs and preferences. This is done by feeding a text-based file to the tool upon analysis start, where the user can specify the functions to perform the analysis being on some specific, or all, local variable(s). Furthermore, it is also possible to include global variables in the analysis. The extracted information is output in flat text files, which allows the user to inspect these files later by searching for desired information or performing postprocessing. The extracted information can also be visualized with some third-party visualization tools to gain, at first glance, a general idea of the memory

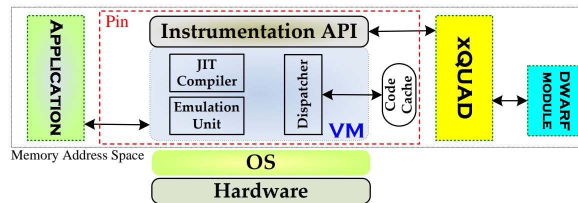

Figure 2. Architectural Overview of xQUAD

usage of an application. Moreover, the text files have a simple format layout, which renders possible to *parse* these files for extracting specific information, such as the number of accesses on the local memory, the frequency of accesses on the heap, and the ratio of usage of different memory segments.

4.4. xQUAD Implementation

xQUAD mainly consists of three parts:

- a module for retrieving DWARF debugging information;
- the Pin's instrumentation functions;
- the QUAD's analysis functions called from the instrumentation functions.

Due to space limitations, the implementation details associated with each of the above-mentioned parts are omitted. Nevertheless, we provide an overview of the most important routines and peculiarities in the xQUAD implementation. In the beginning, the tool performs some preliminary work, more specifically, the initialization of the analysis report files, the command-line parsing, the initialization of data-structures used for analysis information storage, and the initialization of the Pin framework. Following that, xQUAD processes the debugging information from the object file. Processing debugging information is *transparent*, meaning that the user is not aware of this process and the only requirement is that the object file should include debugging information, which is indeed a compulsory prerequisite. By using the *libdwarf library*, the DWARF process is initialized and, afterwards, each CU inside an application is processed and analyzed. Following the tree structure of the DWARF format, each individual CU is examined by a depth-first traversal algorithm. During the search, only the information interesting for the user (according to the previously defined list of functions and variables) is further processed. The processing involves storing variable names and their offsets w.r.t. the current frame base pointer into a table, kept temporarily in the memory.

After reading the debugging information, xQUAD starts the instrumentation process by using APIs from the Pin framework. The xQUAD tool instruments the application at two levels of granularity: at the *routine level* and at the *instruction level*, by calling the Pin API functions *RTN_AddInstrumentationFunction()* and *INS_AddInstrumentationFunction()*, respectively. *RTN_AddInstrumentationFunction()* instruments every routine in the application and its purpose is to keep its own version of the stack of routine calls. The name of each routine is retrieved by using the API function *RTN_name* and the function name is stored in a stack data structure. By keeping an owned version of the call stack, it is possible to know if a *currently executing function* has been *called* for the first time, meaning that the function is producing *fresh memory accesses*, i.e., these memory accesses are being produced for the first time. Conversely, a function may be *returning* from a callee, meaning that the data accesses currently produced must be accounted for a previous function call. The call stack is used in various aspects of the analysis, like the implementation of the *routine count functionality* and the correct implementation of the *memory map file*.

Afterwards, the *INS_AddInstrumentationFunction()* API function is called. This function allows instrumentation analysis at instruction level, meaning that every instruction of the application is inspected at the runtime. The instrumentation analysis checks for instructions operating on *memory* and the *return* instruction. The return instruction is inspected for the purpose of maintaining the earlier described call stack, while the inspection of memory related instructions is the main concern of xQUAD. Each time an instruction references memory (read or write), the tool calls an analysis function that inspects whether or not the current memory operation is done on a data object that the user has earlier defined. This is done by checking if the *current instrumented function* matches a function in the table stored by the previous DWARF information extraction step.

Pin's API function *INS_InsertPredicatedCall()* is used to call analysis functions. This API function prevents pollution of the memory analysis by calling an analysis function only if a particular instruction is actually executed, i.e., only if the instruction is *predicated true*. In case a memory access instruction is executed, various parameters are passed to the analysis function *RecordTrace()*. The *REG_GBP* parameter represents the current base pointer. This parameter is used to calculate the actual memory address of the variable in question, by adding to this base pointer the offset stored in the table of the offset-variable name pairs. Furthermore, the effective address of the instruction is passed to the analysis function, along with the current function name retrieved by using the Pin API call to *RTN_Name*. Using these parameters, it is possible to build another table consisting of pairs of *variable name* and *variable address*. Finally, the *INS_IsPrefetch()* parameter is used to indicate if the current instruction is a prefetch instruction and, if this is the case, the analysis function returns immediately as we aim at examining only the actual memory operations.

Another analysis function is defined for the purpose of keeping a count of the executed instructions. This analysis function is called for every instrumented instruction, memory related or not. The count of the executed instructions allows us to have an estimation of the time a certain memory operation is executed, effectively implementing a *time-stamp* in terms of instruction cycles. This temporal information can be helpful in gaining useful statistics about the memory usages of data objects defined in the application source code.

5. CASE STUDY

In this section, we present a case study that demonstrates the capabilities of xQUAD. For this purpose, we have chosen the *hArtes wfs* audio processing application. The main goal is to have a detailed analysis regarding the memory accesses within each kernel of the application. The ability of the tool to make a clear connection between raw memory addresses and the actual data objects defined in the source code of the application makes it practical to spot memory deficiencies within the application. It also provides valuable hints for revisions. Furthermore, a *ranking method* is presented, which extracts a *memory penalty factor* for a kernel from the execution time retrieved by *gprof*. This metric serves as an indication of the memory access intensity of a kernel relative to its execution time. However, it does not reflect any particular quantitative value of measurement. It only specifies an *index* by taking the ratio of the memory accesses over the memory access related part of the *gprof* execution time. As a result, this index is only applicable for comparison purposes between kernels, which allows to draw up a ranking method.

5.1. hArtes wfs

The Wave Field Synthesis (WFS) [25] concept is a 3D audio rendering technique characterized by the creation of a virtual source and a virtual room. WFS is based on the *Huygens principle*, which, informally, states that each point in a wavefront can be considered as a *primary source* for the creation of new *secondary waves*, which, in

turn, become a primary source for other waves. Hence, an advancing wave can be constructed by the summation of all the secondary waves arising from previously *primary source* waves. This principle is reproduced by *loudspeaker arrays* that generate a complete sound field in the listening zone, which is identical to an appropriate real sound event. The *hArtes wfs* application provided by Fraunhofer IDMT [26] implements a selfcontained wave field synthesis system.

5.2. Experimental Analysis

By inspecting the memory map profiles of the *hArtes wfs* application for the three distinctive regions namely, the stack, the heap, and the data, some observation can be derived. It turns out that from the beginning until approximately half of the execution time, there is a sparse usage of heap memory addresses, while during the second half of the execution time this usage becomes quite intense for a certain range of addresses in the heap memory. Intensive heap usage is accounted for the *wav_store* kernel, which becomes active approximately in the middle of the execution time and it is the only active function until the end of the execution. Actually, *wav_store* saves the output audio signals, produced during its execution, from the buffers allocated in the heap memory to an output file. To accomplish this, it uses mostly individual heap addresses, which explains the intense heap usage.

Table 1 summarizes all the memory references of the *hArtes wfs* application along with the number of individual memory addresses used. The detailed flat profile produced by xQUAD contains the memory address referenced during a kernel's execution, the corresponding data object name, the kernel's call number, and the time stamp when the reference is issued. The number of data objects presented in Table 1 refers to the local variables defined in the kernel excluding the formal parameters.

Table 1. Memory Access Statistics for *hArtes wfs*

Kernel	Stack			Heap		Data	
	# DO	# ADD	# ACC	# ADD	# ACC	# ADD	# ACC
wav_store	4	38	3160179	35935	291844	537	131932
fft1d	10	33	2192660	6	13	9	59
DelayLine_processChunk	18	41	893129	3	5	9	25
bitrev	2	26	922918	7	32443	10	64303
zeroRealVec	1	19	324462	3	281	7	504
AudioIo_setFrames	2	14	665	32	32	5	12
perm	4	22	126662	5	17	10	50
cadd	1	24	86742	5	16155	10	32378
cmult	3	24	123200	6	16173	10	32251
Filter_process	3	20	81054	5	19	9	47

#DO is the number of local data objects defined in the kernel; ***#ADD*** is the total number of distinct addresses referenced during the application execution; ***#ACC*** is the total number of accesses for data objects.

Table 2. *gprof* Flat Profile for *hArtes wfs*

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

% 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.

The recorded accesses are based on variables, which can have different sizes. Therefore, xQUAD does not reveal the actual number of bytes accessed during the execution of a kernel. To have an aggregate estimation of this value, tQUAD can be utilized. Due to space limitation, we have omitted the detailed memory usage information of each data object defined in the source code of the application. The data in Table 1 is recorded with a time slice length of 500 instructions, i.e., these results are describing *almost completely* the actual behavior of the application w.r.t. its memory usage. Selecting a larger time slice results in the reduction of the analysis time and the disk space usage[1]. However, the choice of a *too large* time slice should be avoided, as this would cause a loss of some valuable information.

From Table 1, we can see that the number of individual memory addresses for accessing the non-local memory is considerably higher for *wav_store* compared to the other functions. In the case of stack, more than one third of the total accesses (the data is not present in Table 1) are due to a local variable that acts as a sentinel for the main loop, which stores the specifications of wave frames to an output file. Thorough examination of the flat profile and the application source code also reveals that the main load of the local accesses in *DelayLine_processChunk* originates from a rather large data structure, which collects the necessary data for delay update. Should the kernel be implemented in the hardware, allocating the mentioned data object and two small counters on the chip memory can result in more than fifty percent reduction in the total external memory accesses.

AudioIo_setFrames is responsible for copying interleaved

1. The produced memory map file for the *hArtes wfs* application with a time slice of 500 instructions is almost 50 MB. By storing each instruction, the produced file can grow up into GBs, which may make the postprocessing of the data impractical.

audio signal parts into the corresponding audio frame in the memory. The *hArtes wfs* application uses 32 secondary sources (i.e. an array of 32 loudspeakers) and so, *AudioIo_setFrames* needs 32 distinct addresses for accomplishing its task. The data presented in Table 1 can be misleading by itself, particularly for the stack region, as it does not take into account the number of times that a specific function is called. Therefore, we used *gprof* to find out the execution frequencies of the functions. The results are presented in Table 2. At a first glance, in Table 1, *bitrev* shows a high frequency of stack usage. However, this function is called over 2 million times, which reduces the number of local memory accesses to only a few instances per call. To have an overview of the memory usage with a high level of abstraction, i.e. without diving directly into numeric values, the visualization feature of xQUAD can be used. xQUAD is able to show the memory usage progress of an application in a motion picture format.

Memory Access Intensity. *gprof* provides a cumulative execution time estimate for each kernel, failing to distinguish between the time spent on computations and the time spent on memory operations. The selection of potential candidates for hardware implementation based solely on the computational intensity can not be regarded as an accurate metric in the context of heterogeneous reconfigurable systems. The reason lies in dealing with the high data communication that must be provided to the reconfigurable devices. To have a more accurate understanding of the behavior of kernels, we tried to estimate the time spent on memory operations w.r.t. the time spent for computations. Finding the exact time that is spent on each memory access, if possible at all, is a very difficult task, as it depends extensively on the intrinsic nature of the underlying platform.

We tried to measure separately the computation- and the communication-related contributions of each kernel in the *hArtes wfs* application. Table 3 summarizes the analysis results. The values are averaged in the cases that the kernels show different behaviors in subsequent calls. Almost all the kernels had similar behaviors in recurring calls and the differences were negligible, except for *zeroRealVec*, which acted considerably different in one case. As Table 3 shows, for most of the kernels more than half of the whole executed instructions are due to memory access operations. This memory communication load can increase up to nearly the whole execution time for strictly memory-bound kernels, such as, *AudioIo_setFrames*.

We introduce a *ranking strategy* to help the estimation of the time spent on memory operations w.r.t. the time spent on calculations. We assume that for each kernel k, the time needed for accessing the memory system (τ_{comm}) is proportional to the total number of issued memory accesses during the execution of that kernel (η_{ma})

Table 3. Communication vs. Computation Time Profile of *hArtes wfs*

Kernel	MAR	NLOC MAR	Total inst.	Total MA inst.	Stk Ratio	# Uni. Exec.
wav_store	37.14	24.54	3389224874	12587888805	33.92	1
fft1d	54.43	13.06	1411388	768176	76.00	6
DelayLine_processChunk	54.24	10.30	1009733	547848	81.04	139
bitrev	51.31	5.48	264	136	89.33	4
zeroRealVec	54.59	9.16	11277	6156	83.21	6
AudioIo_setFrames	99.66	99.37	132127	131677	0.30	4
perm	65.86	11.33	70235	46260	82.81	6
cadd	60.22	18.18	86	51	69.72	4
cmult	63.46	15.40	94	60	75.75	4
Filter_process	53.08	22.45	100520	53351	57.71	7

MAR is the percentage ratio of the memory access instructions to the total instructions executed in the kernel; ***NLOC MAR*** is the same as ***MAR*** except that only references to the non-local region are considered; ***Stk Ratio*** is the percentage ratio of the memory access instructions within the local region to the total memory access instructions; ***# Uni. Exec.*** is the total number of distinct statistical data recorded for different calls of the kernel.

$$\forall k : \tau_{comm}(k) \propto \eta_{ma}(k). \tag{1}$$

Furthermore, the time needed for accessing the memory can be estimated by Equation 2, assuming a completely primitive flat memory architecture without considering any hierarchies, caches, compiler optimizations, and other complexities:

$$\tau_{comm}(k) \approx \xi \times \tau_{gprof}(k) \times MAR(k) \tag{2}$$

$$\approx \alpha \times \tau_{stk}(k) + \beta \times \tau_{heap}(k) + \gamma \times \tau_{data}(k), \tag{3}$$

where $\tau_{gprof}(k)$ is the total cumulative time reported by the *gprof* profiler for the kernel k and MAR (Memory Access Ratio) is the ratio of the total memory access instructions (η_{ma}) to the total instructions. τ_{stk} is the communication time accessing the stack region. Accordingly, τ_{heap} refers to the heap and τ_{data} to the data regions of the memory. α, β and γ reflect cost factors for accessing data objects in stack, heap, and data regions, respectively. By using the parameters presented in Table 3, we can revise $\tau_{comm}(k)$ as follows.

$$\tau_{comm}(k) \approx \tau_{gprof}(k) \times MAR(k) \times \Big(\alpha \times \frac{\eta_{stk}(k)}{\eta_{ma}(k)} + \beta \times \frac{\eta_{heap}(k)}{\eta_{ma}(k)} + \gamma \times \frac{\eta_{data}(k)}{\eta_{ma}(k)}\Big). \tag{4}$$

In the context of reconfigurable systems, it makes sense to consider two distinctive types of memory accesses. Local data objects, commonly limited in size, are allocated in the on-chip memory (BRAMs and/or LUTs) and all the other global and dynamically allocated data objects, rather

Table 4. A Ranking of the *hArtes wfs* Kernels

Kernel	η_{ma}	τ_{comm}	$(MAI)^{-1}$	Rank
wav_store	423776	0.068712	6167423	4
fft1d	72	0.032650	2205	9
DelayLine_processChunk	30	0.012360	2427	8
bitrev	96746	0.003836	25220542	3
zeroRealVec	785	0.005496	142831	5
AudioIo_setFrames	44	0.029811	1475	10
perm	67	0.002266	29568	6
cadd	48533	0.001818	26695819	2
cmult	48424	0.001540	31444156	1
Filter_process	66	0.002245	29399	7

large in size, are put on any kind of off-chip memory, e.g. SDRAMs. These can be addressed with the terms, *local* data, usually referring to the data objects in the stack region, and *non-local* data, which commonly refers to the data objects in the global scope and heap regions. In this respect, we rewrite (4) as following:

$$\begin{aligned}\tau_{comm}(k) &\approx \tau_{gprof}(k) \times \Big(\alpha \times MAR(k) \times \frac{\eta_{stk}(k)}{\eta_{ma}(k)} \\ &\quad + \zeta \times MAR(k) \times \frac{\eta_{heap}(k) + \eta_{data}(k)}{\eta_{ma}(k)}\Big) \\ &\approx \tau_{gprof}(k) \times \Big(\alpha \times MAR(k) \times StkRatio \\ &\quad + \zeta \times MAR_{nloc}(k)\Big),\end{aligned} \quad (5)$$

where $MAR_{nloc}(k)$ is the ratio of the *non-local* memory access instructions to the total number of instructions executed in the kernel k. α and ζ reflect cost factors for accessing data objects in *local* and *non-local* regions, respectively. We define *Memory Access Intensity* (MAI) of a kernel as a metric to distinguish between the memory intensiveness of kernels.

$$MAI(k) = \frac{\tau_{comm}(k)}{\eta_{ma}(k)}. \quad (6)$$

In a simple scenario, suppose that the cost for accessing the local memory region is totally insignificant compared to the non-local memory region, i.e α is close to zero. Table 4 presents an order of the kernels based on the probable suitability for mapping onto an FPGA. Lower value for MAI indicates the kernel is more appropriate for hardware implementation. The total execution time is retrieved from the *gprof* flat profile in Table 2, while the total number of memory accesses is extracted from Table 1. In Table 4, the stack accesses of the kernels are not taken into account, as most time penalty is expected to be for accessing the heap and data segments of the memory. The η_{ma} column reports the sum of the heap and the data regions memory accesses, while $\tau_{comm}(k)$ is an estimate of the time spent executing a kernel's memory access operations. The ordering is based on the inverse values of the MAI of the kernels.

As seen in Table 4, *cadd* and *cmult* get the top positions in our list. These tiny frequently used kernels are responsible to do mathematical addition and multiplication for complex numbers. A thorough inspection of the source code also justifies the placement. The reason lies in the fact that *cadd* consists of only two floating point addition operations (computation workload) and six memory access operations in total, four memory reads and two memory writes (communication workload). A similar scenario applies to *cmult* with five additions, three multiplications, and sixteen memory accesses. It should also be stressed that mapping these kernels to HW still does not affect the overall performance considerably, as they are only responsible for a very small fraction of the whole execution time of the application. Apart from these kernels, *wav_store* still accounts as one of the most appropriate kernels for mapping onto the FPGA. Interestingly, *fft1d* and *DelayLine_processChunck*, which had top positions in the *gprof* profiling information, drop down to near the last position in the list, when we consider our Indexing scheme. The distinct case of *AudioIo_setFrames*, which is the strictly memory-bound kernel in the list, is quite interesting, as it gets now its actual position in the ranking. The kernel is identified as the worst candidate for hardware mapping. Examining the source code reveals that this kernel merely copies interleaved audio signal parts to a frame by invoking the *memcpy* library function. This means that the time spent for memory accesses *relative* to the computation time is dominating, which makes the kernel an inappropriate candidate for the hardware implementation.

6. CONCLUSIONS

The primary obstacle for improving the overall performance of computing systems arises from the communication bottleneck between processing elements and memory subsystem. This bottleneck is even more evident with the introduction of heterogeneous architectures containing reconfigurable fabrics. To alleviate this problem, a thorough and detailed analysis of the memory access usage of an application is of vital importance.

This paper presents xQUAD, a unique tool that is able to perform memory access analysis at the *intra-functional* level. The tool provides detailed information regarding the memory access behavior inside a function, which delivers important information for optimization purposes on a fine-grain scale and for discovering possible unusual behavior of data objects in functions. These information are essential to have a deep understanding of the application behavior and they would not be revealed by

an analysis based on an inter-functional level. In order to accomplish this task, xQUAD extracts data objects information at the source code level during an application execution and attributes the memory access statistics to the descriptive names in the application source code. Based on the extracted information, we proposed a memory access intensity metric which, in comparison to general profilers, characterizes the kernels in an application more accurately for mapping onto the hardware in reconfigurable architectures.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surv.*, Vol. 34, No. 2, pp. 171–210, 2002.

[2] T. Todman and et al., "Reconfigurable computing: architectures and design methods," *IEE Proc. Computers and Digital Techniques*, Vol. 152, No. 2, pp. 193–207, 2005.

[3] S. Hauck and A. DeHon, RECONFIGURABLE COMPUTING: THE THEORY AND PRACTICE OF FPGA-BASED COMPUTATION. , Morgan Kaufmann Publishers Inc., San Francisco, CA, 2007.

[4] S. A. Ostadzadeh, R. Meeuws, C. Galuzzi, and K. Bertels, "QUAD - A memory access pattern analyser," Symposium on Applied Reconfigurable Computing, 2010, pp. 269–281.

[5] S. A. Ostadzadeh, M. Corina, C. Galuzzi, and K. Bertels, "tQUAD - Memory bandwidth usage analysis," ICPP, 2010, pp. 217–226.

[6] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Trans. Program. Lang. Syst.*, Vol. 16, No. 4, pp. 1319–1360, 1994.

[7] 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.

[8] R. F. Cmelik, "Spixtools: Introduction and user's manual," Mountain View, CA, Tech. Rep., 1993.

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

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

[11] A. R. Lebeck and D. A. Wood, "Cache profiling and the SPEC benchmarks: A case study," *IEEE Computer*, Vol. 27, No. 10, pp. 15–26, 1994.

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

[13] K. Bertels and et al., "Developing applications for polymorphic processors: The Delft Workbench," Tech. Rep., January 2006.

[14] S. Vassiliadis and et al., "The molen polymorphic processor," *IEEE Trans. on Comp.*, Vol. 53, No. 11, pp. 1363–1375, 2004.

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

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

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

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

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

[20] Y. D. Yankova and et al., "Dwarv: Delftworkbench automated reconfigurable VHDL generator," FPL, 2007, pp. 697–701.

[21] C. Luk and et al., "Pin: building customized program analysis tools with dynamic instrumentation," PLDI, New York, NY, 2005, pp. 190–200.

[22] The DWARF debugging standard, http://www.dwarfstd.org.

[23] M. J. Eager, "Introduction to the DWARF debugging format," February 2007, Available: http://http://dwarfstd.org/doc/Debugging%20using%20DWARF.pdf

[24] D. Anderson, "A consumer library interface to dwarf," February 2010, Available: http://reality.sgiweb.org/davea/dwarf.html

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

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