QUAD – A Memory Access Pattern Analyser

S. Arash Ostadzadeh, Roel J. Meeuws, Carlo Galuzzi, and Koen Bertels*

Computer Engineering Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, Delft, The Netherlands
{S.A.Ostadzadeh,R.J.Meeuws,C.Galuzzi,K.L.M.Bertels}@tudelft.nl

Abstract. In this paper, we present the Quantitative Usage Analysis of Data (QUAD) tool, a sophisticated memory access tracing tool that provides a comprehensive quantitative analysis of memory access patterns of an application with the primary goal of detecting actual data dependencies at function-level. As improvements in processing performance continue to outpace improvements in memory performance, tools to understand memory access behaviors are inevitably vital for optimizing the execution of data-intensive applications on heterogeneous architectures. The tool, first in its kind, is described in detail and the benefit and the qualities of the presented tool are described on a real case study, the *x264* benchmarking application.

1 Introduction

With the increased proliferation of Chip Multiprocessors (CMP), there is a compelling need for utility tools to facilitate the application development process, tuning and optimization. This requirement becomes more critical with the introduction of hybrid architectures incorporating reconfigurable devices [1]. Since the applications form the basis of such designs, the need to tune the underlying architecture for extracting maximum performance from the software code has become imperative [2]. As the size of reconfigurable fabrics increases, mapping an entire application onto a reconfigurable device does not seem elusive anymore. Traditionally, FPGAs contained the logic and some temporary memory for a kernel, or even the logic for the whole application, but never the logic and memory for an entire application [3]. Although this in itself is an important step towards decreasing the processor-memory gap, there is an inevitable demand for tools that can help users to have a clear understanding of the memory requirements of an application. *To accomplish this goal, a thorough analysis of the memory access behavior of an application is vital*. This demand proves to be even more crucial considering the fact that the main obstacle limiting the performance of reconfigurable systems is the memory latency [4].

Even in the best case scenario, when all the memory addresses can fit on a single chip, the latency to access memory locations from different parts imposes a substantial delay. In [3], the memory access behavior of an application (*g721_e*) whose logic and

* This research has been funded by the hArtes project EU-IST-035143, the Morpheus project EU-IST-027342 and the Rcosy Progress project DES-6392.

memory was entirely mapped onto a reconfigurable device, is discussed. The idea of mobile memory is also investigated to dynamically move the memory closer to the location where it is accessed. It justifies the motivation to lessen the distance between the accessors and the memory locations.

Inspecting the behavior of an application in general, and the actual pattern of memory accesses in particular, is an essential aspect of carrying out effective optimizations for the application development of reconfigurable systems. As a result, many research initiatives are emerging that target support tools for application behavior analysis from different perspectives.

The main contributions of this paper are the following:

- the description of an efficient tool, QUAD, to provide information that can be used in addressing memory-related bottlenecks in reconfigurable computing systems
- the detection of actual data dependency between functions compared to conventional data dependency discovered by similar memory access analysers
- the validation of the proposed tool in a real case study

The rest of the paper is organized as follows. Section 2 gives an overview of the related research. In Section 3, we present an overview of the Delft Workbench (DWB) and the profiling framework which includes QUAD as a dynamic memory access profiler. Section 4 introduces QUAD and describes some of the design and implementation issues. In Section 5, a real application case study is examined. Finally, Section 6 provides concluding remarks and an outline of the future research.

2 Related Research and Problem Definition

Profilers are tools that allow users to analyze the run-time behavior of an application in order to identify the types of performance optimizations that can be applied to the application and/or the target architecture. Generally, profiling refers to a technique for measuring where programs consume resources, including CPU time and memory. General profiling tools such as *gprof* [5], can provide function-level execution statistics for the identification of application hot-spots. *However, they do not distinguish between computation time and memory access time*. As a result, they can not be employed to locate potential system bottlenecks regarding memory-related problems.

In [6,7], the authors provide target independent software performance estimations. However, they lack a thorough memory access analysis, which is vital in tuning performance optimizations in hybrid reconfigurable architectures. [8] describes an approach for evaluating the performance and memory access patterns of multimedia applications through profiling. The tool is only utilized for algorithmic complexity evaluation and its accuracy in performance estimation is not investigated. The way an algorithm interacts with memory has a large impact on performance. More precisely, the memory reference behavior of an application, at the most basic level, depends on the intrinsic nature of the application. However, the developer still has considerable flexibility in manipulating the algorithm, data structures and program structure to change the memory reference patterns [9].

Most existing memory access analysis tools only focus on detecting memory bottlenecks, or faults/bugs/leaks and provide no detailed information regarding the inherent data dependencies in a program's memory reference behavior [10,11]. One of the early simple tools developed for understanding memory access patterns of Fortran programs is presented in [12]. The tool instruments a program and produces a flat trace file of all memory accesses which can be visualized later. Similarly, a tool set is presented in [13] to reveal the pattern of memory references. It generates a set of histograms for each memory access in a program regarding the strides of references.

In [14], the authors present a quantitative approach to analyze parallelization opportunities in programs with irregular memory access patterns. Applications are classified into three categories with low, medium and high dependence densities. Similar to our work, Embla [15] allows the user to discover the data dependencies in a sequential program, thereby exposing opportunities for parallelization. Embla performs a dynamic analysis and records dependencies as they arise during program execution. *However, in this work, we intend to discover the* ***actual data dependency****, which is different from the conventional data dependency referred to in Embla and other similar tools.* By definition, **data dependency** is a situation in which a program segment (instruction, block, function, etc.) refers to the data of a preceding segment. Actual data dependency arises when a function consumes data that is produced by another function earlier. In other words, the common argument passing by the caller function to the callee regarding data distribution does not necessarily imply that the data will be used later in the called function. *Furthermore, in our approach the usual restriction of data dependency detection based on hierarchies of function calls (commonly depicted with call graphs) is relaxed as we merely trace the journey of bytes through memory addresses and do not rely on the control dependencies of tasks to detect potential data dependencies.*

In this paper, we present QUAD (Quantitative Usage Analysis of Data), a memory access tracing tool that provides a comprehensive quantitative analysis of the memory access patterns of an application with the primary goal of detecting actual data dependencies at function-level. To the best of our knowledge, this is the first tool that addresses the actual data dependency detection and abstracts away from the properties of data dependency detection of an application on a particular architecture. In addition, previous research into data dependency detection has mainly focused on the discovery of parallelization opportunities. However, we do not necessarily target parallel application development. Even though QUAD can be employed to spot coarse-grained parallelism opportunities in an application, it practically provides a more general-purpose framework that can be utilized in various reconfigurable systems' optimizations by estimating effective memory access related parameters, e.g. the amount of unique memory addresses used in data communication between two cooperating functions. QUAD can also be used to estimate how many memory references are executed locally compared to the amount of references that have to go to the main memory.

Accessing memory locations sequentially, or within predefined strides, can be considered very efficient on cache-based computing systems. This information can be useful in the presence of memory hierarchies on a system and can be a source of performance improvement. Frequently, it is possible for the programmer to restructure the data or code to achieve better memory reference behavior. Inefficient use of memory remains a significant challenge for application developers. QUAD can also be utilized to diagnose memory inefficiencies by reporting useful statistics, such as boundaries of memory references within functions, detection of unused data in memory, etc. QUAD can be easily ported to different architectures as long as there exists a primitive tool set that can provide the basic memory read/write instrumentation capabilities, like BIT [16] for instrumenting java byte codes.

The main features of the tool proposed in this paper, are listed in the following.

– QUAD detects the actual data dependency at function-level in an application, which involves a higher degree of accuracy compared to the conventional data dependency detected by other similar tools.
– QUAD does not require any modification of the binaries and it has no compiler dependence other than debug information. It also abstracts away from the properties of a particular architecture.

3 Profiling Framework in DWB

This work has been carried out in the context of the Delft WorkBench (DWB) [17] and the hArtes [18] projects. The DWB is a semi-automatic tool platform for integrated hardware/software co-design targeting heterogeneous computing system containing reconfigurable components. It aims to be a comprehensive platform supporting development at all levels starting from profiling and partitioning to synthesis and compilation. Conversely, the closely related hArtes project targets the same heterogeneous systems. However, it also takes into account digital signal processing hardware and provides its own heterogeneous platform.

The Delft Workbench focuses on four main steps within the entire heterogeneous system design.

– *Code Profiling and Cost Modeling* - focuses on identifying application hot-spots and on estimating implementation costs for different components [19].
– *Graph Transformations* - aim to use the profiling information and estimates for clustering tasks, partitioning tasks over components, optimizing tasks, or restructuring the task graph [20,21,22].
– *VHDL Generation* - When tasks need to be implemented on Reconfigurable components, this tool allows to automatically translate the high level language descriptions into VHDL [23].
– *Retargetable Compiler* - schedules and combines all the implemented parts of the application and it generates the executable binary [24].

The work in this paper mainly revolves around code profiling. Figure 1 depicts in detail the profiling framework envisioned by the DWB platform. We distinguish between static and dynamic profiling paths. Static profiling can provide estimates in a small amount of time, whereas dynamic profiling provides accurate measurements of several aspects like execution time and memory access behavior. The dynamic profiling path focuses on run-time behavior of an application and, therefore, is not as fast as the static profiling. Furthermore, the dynamic profiling requires representative input data in order to provide relevant measurements. *gprof* is used to identify hot-spots and frequently executed functions, while QUAD is concerned with tracing and revealing the pattern of

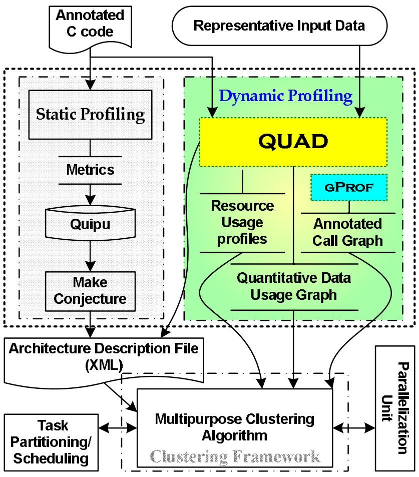

Fig. 1. Profiling Framework within DWB

memory references with the primary aim to detect actual data dependencies between functions. In this paper, we only focus on the representation and implementation details necessary for the description of QUAD.

4 QUAD Design and Implementation

4.1 Pin

QUAD is a Dynamic Binary Analysis (DBA) tool which analyzes an application at the machine code level as it runs. DBA tools can be built from scratch or be implemented using a Dynamic Binary Instrumentation (DBI) framework. Instrumentation is a technique for inserting extra code into an application to observe its behavior. This process can be performed at various stages either in the source code, or at compile-time, or at post-link time, or at run-time. QUAD is implemented as a tool using the Pin [25] run-time binary instrumentation system. By using Pin, we have the benefit of working transparently with unmodified Linux, Windows and MacOS binaries on Intel ARM, IA32, 64-bit x86, and Itanium architectures. Thanks to the instrumentation transparency, Pin preserves the original application behavior. The application uses the same addresses (both instruction and data) and the same values (both register and memory) as it would in an uninstrumented execution. This transparency, vital for correctness, results in more relevant information collected by the instrumentation. Dynamic instrumentation is particularly beneficial for this type of 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 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.

Since QUAD relies on dynamic instrumentation and it is compiler-independent, detecting the producers/consumers of the data being stored/loaded via memory addresses must be done in the absence of any kind of control/data flow or call graphs. As a consequence, the detection is based only on the dynamic execution of the program. In order to provide some degree of flexibility, QUAD also implements and maintains its own call graph during the execution of a program.

4.2 QUAD Overview

QUAD has been designed as a base system to provide useful quantitative information about the data dependence between any pair of cooperating functions in an application. Data dependence is estimated in the sense of producer/consumer binding. More precisely, QUAD reports which function is consuming the data produced by another function. The exact amount of data transfer and the number of Unique Memory Addresses (UMA) used in the transfer process are calculated. Based on the efficient Memory Access Tracing (MAT) module implemented in QUAD, which tracks every single access (read/write) to a memory location, a variety of statistics related to the memory access behavior of an application can be measured, e.g. the ratio of local to global memory accesses in a particular function call.

Figure 2 illustrates the architectural overview of QUAD along with the components in Pin. At the highest level, there is a Virtual Machine (VM), a code cache, and an instrumentation API. The main component inside QUAD is the MAT module, which is responsible for building and maintaining dynamic trie [26] data structures to provide relevant memory access information as fast as possible. The trie data structure acts as a shadow memory for each byte accessed within the address space of an application.

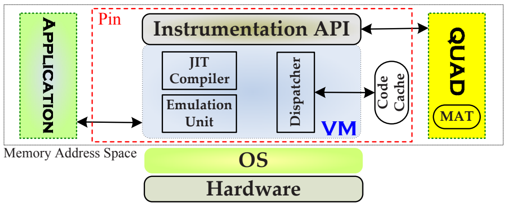

Fig. 2. Architectural overview of QUAD

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 QUAD. Pin is the engine that instruments the application. QUAD contains the instrumentation and analysis routines and it is linked with a library that allows QUAD to communicate with Pin.

4.3 QUAD Implementation

The interfaces to most run-time binary instrumentation systems are API calls that allow developers to hook in their instrumentation routines. 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. QUAD uses these two API routines to set up calls to the instrumentation routines *Instruction()* and *UpdateCurrentFunctionName()*. These two instrumentation routines, in turn, call the two main analysis routines *RecordMemRef()* and *EnterFunc()* which are responsible for updating tracing information of memory references and maintaining an internal call graph respectively. Figure 3 illustrates an implementation overview of QUAD. The detailed algorithms associated with each module are not included in this paper for brevity. Nevertheless, in the following we provide some description highlights.

The initialization process in the main module includes Pin system initialization, command line options parsing, internal call graph initialization, and some output XML file preprocessing. The *Instruction()* instrumentation routine sets up the call to *RecordMemRef()* routines every time an instruction that references memory is executed. 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 (read or write), QUAD inserts a call to *RecordMemRef()* before the instruction, passing the Instruction Pointer (IP), Effective Address (EA) for the memory operation, a flag indicating whether it is a read or write operation, number of bytes read or written, and a flag showing whether or not the instruction is a prefetch. The analysis routine returns 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 memory instruction is predicated true. There is also a separate *RecordMemRef()* analysis routine for the case that we are interested to trace local memory references within the stack region. In this case, the value of stack pointer is also passed to the analysis routine for further investigation. *Instruction()* also monitors the *ret* instruction to leave a function and upon detection calls a different analysis routine that updates the internal call graph if necessary.

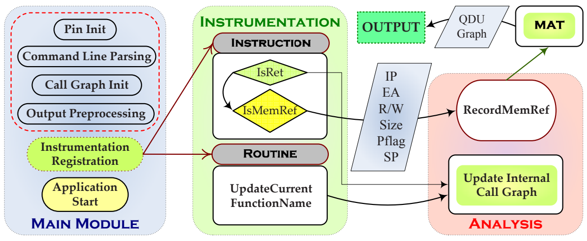

Fig. 3. Implementation overview of QUAD

The main objective of *RecordMemRef()* is to identify the function responsible for the current memory reference and to pass the required information to the MAT module. The instrumentation at the routine granularity in QUAD is responsible for pushing the name of the currently called function onto an internal call stack. Note that the respective pop operation is later performed upon detection of the *ret* instruction. QUAD needs to maintain its own call graph because a user may not be interested to dive into library routines or routines that are not included in the main binary image file. In these cases, QUAD assumes the most recent caller routine from the main binary as the one responsible for issuing memory references.

4.4 Memory Access Tracing (MAT) Module

In order to spot and to extract memory reference information during the execution of an application, an efficient memory access tracing module is implemented. The tracing process utilizes *trie* data structures for fast storage and retrieval. MAT defines trie structures with base 16 that is representative of memory addresses in hexadecimal format. Each hexadecimal digit in a 32-bit memory address corresponds to one level in the trie data structure, leaving 8 levels deep in the hierarchy for complete address tracing. The trie data structure is designed to grow dynamically on demand for reducing memory usage overhead as much as possible. This means that if a particular memory address is fed to MAT for the very first time, the levels required to trace that particular address are created in the trie. Hence, no space is allocated for unused memory addresses. The saving is considerable because the complete data structure is expected to be gigantic and may result in memory overflow in some systems.

The memory reference recording process is accomplished in two distinct phases. In the first phase, we trace an 8-level trie for a particular memory address. For each memory reference three different arguments are specified: memory address, function ID, and read/write flag. In case of a write access, the corresponding shadow memory address in the trie is labeled with the caller (producer) function ID. When a read flag is detected, the function ID responsible for the most recent write in the memory address is retrieved and passed along with the consumer function ID to the second phase where a data communication record is created.

The memory reference information gathered by QUAD during the execution of an application are reported in two separate formats. The producer/consumer binding information is saved in a text file using standard portable XML format. This makes it easy for third-party applications to import data for further interpretation and processing. The actual data dependency bindings between functions is also provided in the form of a graph data structure, which is called Quantitative Data Usage (QDU) graph.

5 Case Study

We used *x264* [27] as a benchmark to test QUAD in a series of experiments. The goal is to have an initial understanding of the application behavior regarding the data communication patterns, memory usage, and memory requirements. *The information provided by QUAD can be used later in HW/SW partitioning and mapping as well as to hint application developers how to revise and optimize the code for a specific architecture.*

x264 is a free library for encoding H.264/AVC video streams. The version used in this work is a modified *x264 r654* encoder tailored to the MOLEN [28] paradigm taking into account the restrictions in terms of coding rules accepted by the DWARV hardware compiler [23].

5.1 Experimental Setup

All the experiments were executed on an Intel 64-bit Core 2 Quad CPU Q9550 @ 2.83GHz with the main memory of 8GB, running Linux kernel v2.6.18-164.6.1.el5. The *x264* source code was compiled with *gcc* v3.4.6 and with the profiling option enabled. We need to use *gprof* as an auxiliary tool to interpret the data and to make some conclusions. The standard command line options used to run the 64-bit compiled version of *x264* was the following:

1. *–no-ssim* - to disable the computing of structural similarity (SSIM) index;
2. *rate control -q1* - to indicate almost lossless compression;
3. *–no-asm* - to disable all stream processing optimizations based on CPU capabilities.

The 64-bit version of QUAD was used with the following command line options:

1. *ignore_stack_access* - to ignore all the memory accesses to the stack region. This gives a clear view of the data transferred via non-stack region.
2. *use_monitor_list* - to include only some critically potential functions in the report files, due to the high complexity and the size of the *x264* application.

akiyo_qcif was used as the input data file for encoding. It is a raw YUV 4:2:0 file with the resolution of 176x144 pixels containing 300 frames. The output was in raw byte stream format.

5.2 Experimental Analysis

x264 contains over two hundreds functions. The set of functions to be called are determined based on different options selected by the user or by the input/output file specifications. On the basis of the computation-intensive kernels identified in the flat profile provided by *gprof*, we chose a number of functions (or series of functions) for further inspection. The main criterion adopted here was the suitability for the DWARV compilation tool. Table 1 presents part of the flat profile.

Table 1. Flat profile for *x264*

function name	% time	self seconds	calls	total ms/call	self ms/call
pixel_satd_wxh	34.51	0.49	1361024	0	0
x264_cabac_encode_decision	8.45	0.12	10808084	0	0
get_ref	7.75	0.11	1165182	0	0
block_residual_write_cabac	7.04	0.1	400643	0	0
x264_pixel_sad_x4_16x16	5.63	0.08	88506	0	0
x264_frame_filter	5.63	0.08	2700	0.03	0.03
x264_pixel_sad_x4_8x8	3.52	0.05	243588	0	0
refine_subpel	2.82	0.04	151014	0	0
motion_compensation_chroma	2.11	0.03	442213	0	0

% 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; ***total ms/call*** is the average number of milliseconds spent in the function and its descendants per call; ***self ms/call*** is the average number of milliseconds spent in the function per call.

Table 2. Summary of data produced and consumed by **satd**- and **sad**-related kernels

function name	IN	IN UMA	OUT	OUT UMA
pixel_satd_wxh	326310528	137425	91578266	5126
x264_pixel_satd_16x16	40607660	1523	26133254	1233
x264_pixel_satd_16x8	5795852	1009	2849136	722
x264_pixel_satd_4x4	34342432	2745	18099806	2318
x264_pixel_satd_4x8	3091448	1953	1610194	1644
x264_pixel_satd_8x16	6248933	1053	3152620	650
x264_pixel_satd_8x4	3019905	1937	1568928	1594
x264_pixel_satd_8x8	91709500	3307	46203830	3049

function name	IN	IN UMA	OUT	OUT UMA
x264_pixel_sad_16x16	59965275	103924	5704610	624
x264_pixel_sad_16x8	9159212	55626	1736556	480
x264_pixel_sad_4x4	0	0	0	0
x264_pixel_sad_4x8	0	0	0	0
x264_pixel_sad_8x16	8924130	53174	1709404	566
x264_pixel_sad_8x4	0	0	0	0
x264_pixel_sad_8x8	53730341	89155	10838624	664

IN represents the total number of bytes read by the function; ***IN UMA*** 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 written to those locations earlier; ***OUT UMA*** indicates the total number of unique memory addresses used in writing.

As presented in the Table 1, **pixel_satd_wxh** is the main kernel of the application accounting for 35% of the total execution time. It was initially selected as the main candidate kernel for hardware mapping along with **sad**-related functions. Although **x264_cabac_encode_decision** is the most frequently called function, each call has a smaller contribution compared to **pixel_satd_wxh**. As a result, the overall contribution of **x264_cabac_encode_decision** drops considerably. There are several **satd**-related functions defined in the form of Macros corresponding to various block sizes. These macros, when expanded, create different functions calling the main **pixel_satd_wxh** making it a very critical function on the execution path. Table 2 summarizes the results of memory access tracing for **satd**- and **sad**-related functions. As expected, **pixel_satd_wxh** is the top consumer on the list (in total more than 300MB) as all the other **satd**-related functions call this kernel to perform their primary tasks.

It is worth noting that some **sad**-related functions (the ones with 4 rows and/or columns) do not exhibit any data transfers, which is an indication that they are not called. This depends on the input file characteristics and options used. Although the kernels are intensely reading (writing) data from (to) memory, the number of unique memory addresses used in the data transfer is limited (MBs data transfer vs. KBs locations). This indicates the possibility of allocating memory buffers, e.g. on FPGA BRAMs to gain better performance. QUAD can also provide a detailed map of the used memory addresses for the examination of mapping opportunities on a target architecture. The auxiliary functions communicating with kernels are recognized and presented in the QDU graph. These auxiliary functions can be sources of further investigation. For example, one might investigate mapping tightly coupled functions on FPGA and creating a buffer to facilitate the data transfer, or merging auxiliary function(s) with the primary kernel to cut off data transfers between functions. In case of **pixel_satd_wxh**, **mc_copy_w16** is tightly coupled with the main kernel and it is responsible for producing approximately 130 MB of data (75k UMA). Further inspection of **mc_copy_w16** reveals that it belongs to the *motion compensation* library and merely calls the built-in *memcpy* routine of the C language library in a loop in order to create a block of pixels from a flat set of pixels with a predefined stride. It seems feasible to rewrite the routine from scratch and to combine it with the kernel.

sad-related routines are also defined in the form of Macros corresponding to various block sizes. Unlike **satd**-related functions, these macros, when expanded, create different functions with separate bodies. In order to evaluate the impact of an identical kernel routine for the **sad**-related functions, we created a new function called **pixel_sad_wxh** and revised all the **sad**-related functions to call this critical kernel. It is a more likely

Table 3. Flat profile for the revised *x264* (non-instrumented and QUAD-instrumented binaries)

function name	% time	self seconds	calls	rank	% time(+QUAD)	self seconds	rank(+QUAD)
pixel_sad_wxh	33.45	0.5	2646060	1	22.94	546.53	1
pixel_satd_wxh	24.32	0.36	1361024	2	7.61	181.18	4
x264_frame_filter	8.11	0.12	2700	3	9.96	237.21	3
get_ref	7.43	0.11	1165182	4	7.49	178.41	5
motion_compensation_chroma	5.41	0.08	442213	5	3.25	77.53	7
block_residual_write_cabac	4.73	0.07	400643	6	2.64	62.96	8
x264_cabac_encode_decision	2.03	0.03	10808084	8	14.62	348.32	2
x264_macroblock_cache_load	2.03	0.03	29700	9	1.49	35.39	12
x264_cabac_encode_bypass	1.35	0.02	2234007	10	4.84	115.29	6

Table 4. Data produced/consumed by **pixel_satd_wxh** & **sad**-related functions in the revised *x264*

function name	IN	IN UMA	OUT	OUT UMA
pixel_sad_wxh	816414788	245781	100154164	2976
pixel_satd_wxh	326389008	137389	91580364	5108
x264_pixel_sad_16x16	16169821	885	7275066	614
x264_pixel_sad_16x8	4588984	793	2122520	510
x264_pixel_sad_8x16	4480742	843	2066828	552
x264_pixel_sad_8x8	39174841	1011	17382666	732

candidate for implementation on FPGA devices. Table 3 depicts part of the flat profile for *x264* after the introduction of the new **pixel_sad_wxh** kernel. **pixel_sad_wxh** now gets the dominant position with the contribution of about 33.5% to the whole application's execution time. Note that it is also called nearly double of the times compared to the second dominant kernel, **pixel_satd_wxh**. The *gprof* flat profile of the QUAD-instrumented binary is also provided. The considerable increase in the self-seconds contribution of each kernel is due to the overhead introduced by the QUAD instrumentation code routines. However, the ranking provided in this respect is somehow more representative of real execution time regarding the data communication between functions via non-local memory. It is due to the fact that we do not take into consideration stack-region memory accesses and only upon detection of a non-local memory access, a time-consuming routine to parse the trace trie is called.

Table 4 summarizes the results of memory access tracing for **pixel_satd_wxh** and **sad**-related kernels in the revised version of *x264*. As expected, the communication load of **pixel_sad_wxh** dominates the former main kernel **pixel_satd_wxh**. However, there is a substantial increase in the total amount of bytes consumed by this new kernel (about 800MB) compared to the collective number of bytes consumed by the **sad**-related functions in the original version. This is due to the fact that the **sad**-related functions have to pass extra arguments to the new kernel. The new kernel uses the extra information to distinguish between different **sad**-related functions. The number of bytes consumed in **pixel_sad_wxh** can be further reduced by a revision of the code to minimize this overload. The total number of bytes produced and consumed beside the unique memory addresses used inside individual **sad**-related functions are significantly reduced since the load is shifted to the new **pixel_sad_wxh** kernel.

Including the local memory accesses in the tracing would also reveal notable observations. By including stack region accesses, **pixel_satd_wxh** becomes the dominant kernel once again (22.15% of the whole contribution). This indicates that if there is no intention to map the local temporary memory into the hardware and fetching data from external memory is expensive, there is a high probability that mapping **pixel_satd_wxh** onto hardware is preferable compared to **pixel_sad_wxh**.

6 Conclusions

The gap between processors and memory performance will be a major challenge for the optimization of memory-bound applications on hybrid reconfigurable systems. This demands the development of utility tools to help users in tuning applications for maximal performance gain of these systems. In this paper, we have presented QUAD, a tool that provides a comprehensive quantitative analysis of the memory access patterns of an application. QUAD can be employed in detecting coarse-grained parallelism opportunities as well as providing information about the memory requirements of an application. The information is particularly useful in buffer size estimation for local memory reallocation to store data in case of mapping kernels onto reconfigurable devices that initially cause memory bandwidth problems. QUAD has been tested on a real *x264* benchmarking application and a detailed discussion was presented based on the extracted statistics. In the future work, we are planning to utilize the information provided by the tool for task clustering in heterogeneous reconfigurable systems.

References

1. Kwok, T.O., Kwok, Y.K.: On the design, control, and use of a reconfigurable heterogeneous multi-core system-on-a-chip. In: Proc. of PDP, pp. 1–11 (2008)
2. Kempf, T., Karuri, K., Wallentowitz, S., Ascheid, G., Leupers, R., Meyr, H.: A sw performance estimation framework for early system-level-design using fine-grained instrumentation. In: Proc. of DATE, pp. 468–473 (2006)
3. Yan, R., Goldstein, S.C.: Mobile memory: Improving memory locality in very large reconfigurable fabrics. In: Proc. of FCCM, pp. 195–204 (2002)
4. Hauck, S., Dehon, A.: Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation (Systems on Silicon). Morgan Kaufmann, San Francisco (2007)
5. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: A call graph execution profiler. SIGPLAN Not. 17(6), 120–126 (1982)
6. Giusto, P., Martin, G., Harcourt, E.: Reliable estimation of execution time of embedded software. In: Proc. of DATE, pp. 580–589 (2001)
7. Bammi, J.R., Kruijtzer, W., Lavagno, L., Harcourt, E., Lazarescu, M.T.: Software performance estimation strategies in a system-level design tool. In: Proc. of CODES, pp. 82–86 (2000)
8. Ravasi, M., Mattavelli, M.: High-level algorithmic complexity evaluation for system design. J. Syst. Archit. 48(13-15), 403–427 (2003)
9. Martonosi, M., Gupta, A., Anderson, T.: Memspy: analyzing memory system bottlenecks in programs. In: Proc. of Sigmetrics/Performance, pp. 1–12 (1992)
10. Venkataramani, G., Doudalis, I., Solihin, Y., Prvulovic, M.: Memtracker: An accelerator for memory debugging and monitoring. ACM Trans. Archit. Code Optim. 6(2), 1–33 (2009)
11. Choudhury, A.N.M.I., Potter, K.C., Parker, S.G.: Interactive visualization for memory reference traces. Comput. Graph. Forum 27(3), 815–822 (2008)

12. Brewer, O., Dongarra, J., Sorensen, D.: Tools to aid in the analysis of memory access patterns for FORTRAN Programs. Parallel Computing 9(1), 25–35 (1988)
13. Balle, S., Steely Jr., S.: Memory Access Profiling Tools for Alpha-based Architectures. In: Kågström, B., Elmroth, E., Waśniewski, J., Dongarra, J. (eds.) PARA 1998. LNCS, vol. 1541, pp. 28–37. Springer, Heidelberg (1998)
14. von Praun, C., Bordawekar, R., Cascaval, C.: Modeling optimistic concurrency using quantitative dependence analysis. In: Proc. of PPoPP, pp. 185–196 (2008)
15. Faxén, K., Popov, K., Janson, S., Albertsson, L.: Embla–Data Dependence Profiling for Parallel Programming. In: Proc. of CISIS, pp. 780–785 (2008)
16. Lee, H.B., Zorn, B.G.: Bit: a tool for instrumenting java bytecodes. In: Proc. of USITS, pp. 7–16 (1997)
17. Bertels, K., Vassiliadis, S., Panainte, E.M., Yankova, Y.D., Galuzzi, C., Chaves, R., Kuzmanov, G.: Developing applications for polymorphic processors: the delft workbench. Technical report, CE Group (2006)
18. Bertels, K., Kuzmanov, G., Panainte, E.M., Gaydadjiev, G.N., Yankova, Y.D., Sima, V., Sigdel, K., Meeuws, R.J., Vassiliadis, S.: Hartes toolchain early evaluation: Profiling, Compilation and HDL generation. In: Proc. of FPL, pp. 402–408 (2007)
19. Meeuws, R.J., Sigdel, K., Yankova, Y.D., Bertels, K.: High level quantitative interconnect estimation for early design space exploration. In: Proc. ICFPT, pp. 317–320 (2008)
20. Ostadzadeh, S.A., Meeuws, R.J., Sigdel, K., Bertels, K.: A clustering framework for task partitioning based on function-level data usage analysis. In: Proc. of FPGA, p. 279 (2009)
21. Ostadzadeh, S.A., Meeuws, R.J., Sigdel, K., Bertels, K.: A Multipurpose Clustering Algorithm for Task Partitioning in Multicore Reconfigurable Systems. In: Proc. of CISIS, pp. 663–668 (2009)
22. Galuzzi, C., Bertels, K.: A framework for the automatic generation of instruction-set extensions for reconfigurable architectures. In: Woods, R., Compton, K., Bouganis, C., Diniz, P.C. (eds.) ARC 2008. LNCS, vol. 4943, pp. 280–286. Springer, Heidelberg (2008)
23. Yankova, Y.D., Kuzmanov, G., Bertels, K., Gaydadjiev, G.N., Lu, Y., Vassiliadis, S.: Dwarv: Delftworkbench automated reconfigurable VHDL generator. In: Proc. of the FPL, pp. 697–701 (2007)
24. Panainte, E.M., Bertels, K., Vassiliadis, S.: The molen compiler for reconfigurable processors. ACM TECS 6(1), 6 (2007)
25. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proc. of PLDI, pp. 190–200 (2005)
26. Fredkin, E.: Trie memory. ACM Commun. 3(9), 490–499 (1960)
27. x264, `http://www.videolan.org/developers/x264.html`
28. Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., Panainte, E.M.: The molen polymorphic processor. IEEE Trans. on Computers 53(11), 1363–1375 (2004)