Quantitative Global Dataflow Analysis on Virtual Instruction Set Simulators for Hardware/Software Co-Design

Carsten Gremzow
Faculty of Computer Science and Electrical Engineering
Berlin University of Technology
gosper@cs.tu-berlin.de

***Abstract*— One of the main challenges in system design whether for high performance computing or in embedded systems is to partition software for target architectures like multi-core, heterogeneous, or even hardware/software co-design systems. Several compiler techniques handle partitioning and related problems by using static analysis and therefor have no means to capture the global data flow in quantity and its dynamics which is essential for extracting tasks or exploiting coarse grained parallelism. We present a novel solution for capturing and analyzing an application's quantitative data flow in this paper. The core part is the LLILA (Low Level Intermediate Language Analyzer) tool set, which automatically generates and augments self-profiling instruction set simulators from assembly level descriptions for a virtual machine. During run-time of the augmented program several properties (frequency, quantity and locality reflecting inter-procedural communication) of data exchange are captured at instruction level and as a consequence in the highest possible degree of accuracy.**

I. Introduction

The necessity to spread applications among distributed, heterogeneous resources such as standard processors in conjunction with programmable hardware is growing steadily. This common trend can be attributed to a change in the main stream computing paradigm towards multi-core systems as well as to the fact that there's a growing need for certain parts of a software system to be executed on dedicated, customized hardware. Not only does the user of standard desktop systems currently benefit from applications executed on distributed, heterogeneous resources where e.g. multimedia applications can be accelerated using Graphics Processing Units (GPUs) but also the domain of embedded computing where systems usually consist of a number of components including application specific processors (ASPs) which at the present need to be design manually. To achieve optimal partitioning of an individual application onto the target hardware automatically, it is mandatory do have in depth knowledge of coarse grained parallelisms at the system level. The smallest common denominator in hardware / software Co-design environments is the use of standard programming languages such as C or C++. Compile- and link time optimization techniques known from compiler engineering can be used as a preparational means e.g. to reduce design space complexity, but they turn out to be inadequate to a get a full understanding of the complete data flow in quantity between the separate elements of a complex program system which is essential for system partitioning. The latter is especially true for the case of fully automatic partitioning without the specific use of additional libraries for describing coarse grained parallelisms such as MPI, OpenMP or SystemC. In the following, a framework for in depth extraction and analysis of global data flow will presented. Part of the acquired global data flow information includes the quantity of data exchanged between program parts at the inter-procedural level as well typical memory access patterns and dominant path of program execution.

A. Related Work

A common approach to analyze static and dynamic properties of a program system or algorithm is simulating its implementation on a specific instruction set architecture. There is a broad spectrum of ISA simulation techniques ranging from the extremely flexible but slow interpretive approach to fast, efficient compiling simulators. A prominent member of the category mentioned first is SimpleScalar [3]. Increased simulation performance can be expected from Shade [4], Embra [7] and FastSim [6] employing dynamic binary translation as well as event buffering techniques. The majority of system design frameworks in the hardware software Co-design community such as FACILE [6], SimnML [8], ISDL [9], MIMOLA [10], LISA [11][12][13] and EXPRESSION [14] are based on these simulators to allow fast, quantitative evaluation of particular architecture in a Co-design scenario. The key aspect shared by all of the aforementioned tools is the a priori assumption that the program code under investigation shows static runtime behavior only. Almost all of the present concepts for dividing complex systems into a combination of hardware and software aspects perform partitioning either at or even before compile time of the application. Traditional use of dedicated Co-design languages such as SystemC [2], HandleC, SiliconC, SA-C [15] or StreamC leave the task of application partitioning either partially or completely up to the designer. Other approaches allow execution on application specific hardware for certain parts of the software and vice versa. This can be accomplished at binary level [1] e.g. by extracting computationally expensive kernels and moving them to application specific Co-processors. Methods for auto parallelization have emerged from the field of compiler engineering quite some time ago [16] and provide techniques to automatically identify and handle synchronization dependencies in complex software systems [17].

978-1-4244-2658-4/08/$25.00 ©2008 IEEE

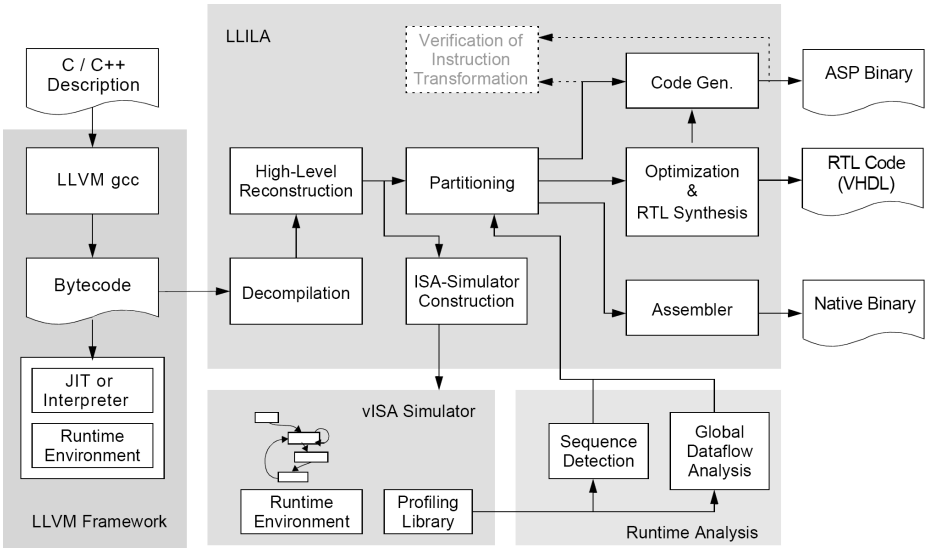

Fig. 1. Schematic outline of the LLILA simulation, profiling and synthesis flow. The three right grey areas denote our actual tool flow boundary whereas program compilation is performed using the LLVM gcc tool chain on the left.

The remainder of this paper is structured as follows: Section two presents a framework for generating self-profiling instruction set simulators automatically from arbitrary application code compiled into byte-code for a virtual machine. The related subsections will focus on techniques of code instrumentation for the target simulator in such a fashion that the global data flow can be recorded. Section three presents quantitative results from experiments with data flow intensive applications. The paper will conclude with final remarks and an outlook on future work.

II. Simulation of Virtual Instruction Set Architectures

Our ASP/ASIP synthesis and runtime analysis environment *Synphony* is centered around the LLVM (Low Level Virtual Machine) [18] which is a compiler framework designed to support program analysis and transformation for arbitrary programs. The LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form with a simple, language-independent type-system that exposes the primitives commonly used to implement high-level language features. In our work the LLVM has been given preference over other virtual machines due to its simplicity, the notion of SSA representation and the possibility to compile and analyze complex software systems through its seamless integration into the GNU gcc tool chain.

A. System Outline

Our overall simulation and synthesis flow depicted in Figure 1 consists of a number of stages which will be discussed in the following. The application's source under investigation is compiled into the LLVM's bytecode representation using llvm-gcc. Additional libraries in question need also be compiled into LLVM code for proper static binding. As a result the compilation process generates a single assembly suitable for execution using LLVM's runtime environment which was designed to provide either Just-In-Time translation to the host's native binary format or plain interpretive execution. Note that byte code references to the standard C library and systems calls need to intercepted by the execution environment and forwarded to the host's native environment. After parsing an LLVM byte code static program analysis starts with extracting type declarations and globally defined memory and function objects which are held for later reference.

B. High-Level Reconstruction

From the instruction stream of each individual function, the basic block structure and corresponding control flow graph is reconstructed by correlation of basic block addresses with conditional and unconditional jump instructions. To reconstruct the data flow graph on a per basic block basis, each instruction's data dependencies are computed and inbound flow is connected to source instructions generating it. Source and destination flow of an instruction can be identified due to LLVM's static single assignment notation. SSA notation normally assigns unique identifiers for left hand side data so tracking inter-instruction data flow should boil down to tracking identifiers. Unfortunately this is not always the case with LLVM: Variable identifiers must not be unique and are frequently reused with different type signatures. Therefor LLILA also checks the type signature of each instruction and performs variable renaming to uniquify identifiers whenever necessary. High-Level programming languages as well the LLVM assembly language representation for virtual machines are not suitable for expressing parallelisms. Detection and exploitation of operator parallelisms on a basic block level can be accomplished by analysis of the above mentioned data flow graphs. In order to gather the data flow on a procedural level and to extend the detection and exploitation of operator parallelisms and movability beyond the scope of a single basic block, LLILA folds the above control and data

flow graphs into a singular flat graph representation. This step is essential for further analysis and it supports increased throughput during ASP/ASIP synthesis.

C. Call Sequence Graphs

In order to track complete runtime behavior across the procedural data flow level, additional static data needs to be retrieved from the LLVM byte code representation. Traditional inter-procedural dependencies are usually captured in the static call graph which denotes a function's procedural dependencies in a directed acyclic graph which can easily be extracted from the instruction flow by recording subprogram call instructions. For the purpose of partitioning a software system into a distributed system, a call graph is insufficient for it only states a set of sub functions called by a parent. In order to discover the actual flow of data in between function calls, we also need to include the timely sequence in which subroutine calls can be issued during program execution. For this purpose the LLILA systems constructs a *Call Sequence Graph (CSG)*, which represents a reduced version of a function's control flow graph only denoting all possible sequences of subprogram calls a singular function can issue. During compiled program simulation CSG edges will be annotated with the actual amounts of data transfered between function caller and callee.

D. Quantitative Dataflow Tracking

Extracting an application's typical communication behavior involves two aspects of relevance for constructing a parallelized version both for distributed and multicore architectures:

- The Interprocedural Data Flow Graph (IDFG) links data sources (production) and data sinks (consumption) of computations over execution time;
- The amounts of data transfered along individual Interprocedural Data Flow Graph Edges denote the Quantitative Flow (QF) over execution time.

Static interprocedural dataflow analysis is challenging and in the presence of function pointers and dynamically created objects often impossible. Data exchange via global objects is fairly easy to track both in quality and quantity during simulation simply by associating identifiers with sources and sinks over time. Tracking data passed via function parameters turns out to be more complicated: Information passed on via the stack during function calls will eventually experience renaming and can refer to different data objects of different origin over time. In order to correlate sources and corresponding sinks communicated via function parameters correctly at all times, indirect dataflow via the stack must be analyzed at runtime first in order to uncover the actual interprocedural data flow. In the following sections, we will discuss how a combined static and dynamic analysis approach in the LLILA framework is employed to solve the above outlined profiling tasks.

1) Reverse Dataflow Analysis: The LLVM architecture defines a simple 3-address load/store machine with an infinite number of registers. The latter meaning that there are no register resources in a technical sense - only differently scoped variables. The machine's only means of moving data between variables are the load and store instructions. Since the focus of this work is on capturing the global dataflow in quality and quantity, observation and correlation of issued store and load instruction pairs for any given variable over simulation time should yield the desired graph. To further clarify the process, we will examine the following piece of hypothetical C code in detail:

```
void scale (int *buf, int len, int fac) {
  for (int i = 0; i < len; i++)
    buf[i] *= fac;
}
```

The function *scale* multiplies any arbitrarily sized array of integers pointed to by the argument *buf* with the factor *fac*. Thus, a significant amount of interprocedural data flows into and out of the function via the specified pointer *buf*. Yet, this simple example shows, that with static a priori code analysis exact quantitative dataflow figures are impossible to obtain. Also, if *buf* refers to an object created at runtime e.g with *malloc* determining the origins or more specific the owner of the object turns out to be equally as difficult. If we wish to determine the exact interprocedural flow for the above example, we need to identify all relevant load and store for proper instrumentation. The LLVM assembly code of the loop's body is the following:

```
no_exit:
  %tmp.4 = load int* %i
  %tmp.5 = gep int* %buf, int 0, int %tmp.4
  %tmp.6 = load int* %i
  %tmp.7 = gep int* %buf, int 0, int %tmp.6
  %tmp.8 = load int* %tmp.7
  %tmp.9 = load int* %fac_addr
  %tmp.10 = mul int %tmp.8, %tmp.9
  store int %tmp.10, int* %tmp.5
  %tmp.11 = load int* %i
  %inc = add int %tmp.11, 1
  store int %inc, int* %i
  br label %loopentry
```

One can see that relating read and write accesses to *buf* on the source code level with the corresponding load instruction on the assembly level is obfuscated due to the pointer arithmetic instruction *gep* (get element pointer) inserted by the compiler. Looking at load and store instructions during loop execution does not reveal any direct relation to the variable *buf*. Instead, load and store is performed on temporary variables resulting from the address computation process. To uncover the actual global variable or function argument in question reverse data flow analysis is performed on the flat control / data flow graph. The latter was gathered earlier during static program analysis.

The loop's control and data flow graph is depicted in Figure 2. While bold edges denote actual data flow the dotted edges represent index data used to compute effective addresses during *gep* instructions. The load instruction on

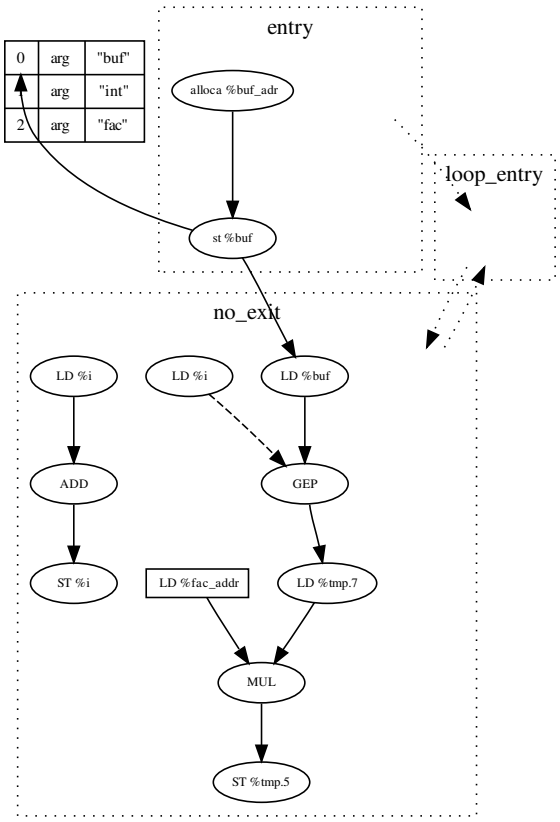

Fig. 2. Control and data flow graph reconstructed from the loop's instruction stream. Bold edges in the *no_exit* block denote the effective data flow along the variables *buf_adr* and the loop index *i*. Reverse inter block data flow analysis reveals a relation trace to the function's first argument *buf* for the load and store instructions on variable *%tmp.5* during loop execution.

tmp.7 features a data dependency all the way to the *gep* instruction computing the initial pointer of *buf*. Therefore it will be marked as a read reference to *buf* as part of the runtime profiling process. Following the same procedure, the store instruction of *%tmp.10* at address *%tmp.5* can be identified as a write access on *buf*.

2) Communication Analysis during Simulation: The previous section introduced a technique how to statically identify load and store instructions of relevance to the interprocedural data flow analysis tasks and how to extract the effective variables names associated with them. Next the mechanism for collecting the actual interprocedural data exchange in quantity and the respective sources and sinks of computation will be discusses. The following hypothetical piece of C code shall serve as a simple example:

```
void normalize (int *buf, int len, int to);
int sum (int *buf, int len);
void scale (int *buf, int len, int fac);
```

The top level function *normalize* takes an integer array of arbitrary length an will normalize its values to the specified normal value *to*. It will call the remaining two functions *scale* as outlined above as well *sum* with obvious semantics. For the purpose of this example we'll assume that data passed to these functions is stored in an array named *buffer* created on the stack by the function *main*. Even though static interprocedural dataflow analysis would suffice to reveal a dependency between *buffer* and function argument *buf* of *sum* and *scale* in this example, the dynamic dataflow trace analysis implemented in the LLILA framework will now be presented.

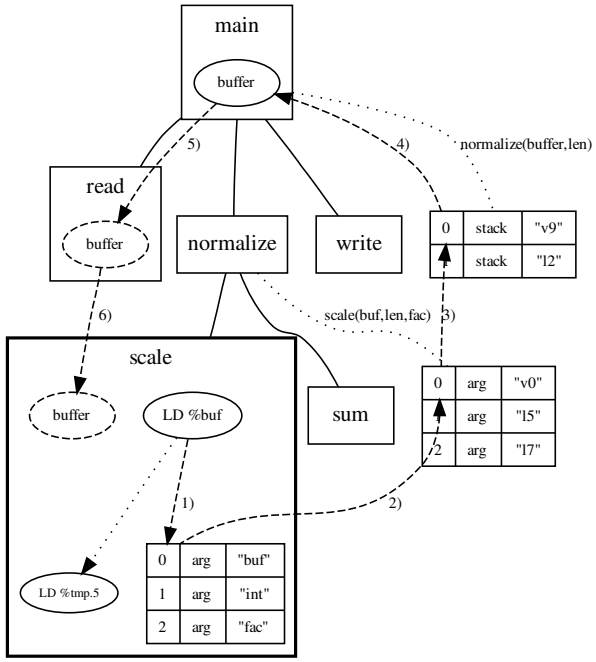

Fig. 3. Correlating store and load instruction pairs dynamically via call graph annotation during runtime. Program execution at call graph node scale requests the location and last write access to the object passed via function parameter *buf*. The call frames created at runtime lead back to the local variable *buffer* which is owned by call node *main* and was last written by the call graph node *write*.

Figure 3 depicts the call graph for the above example with additional data structures created during simulation. We assume that program execution has reached the function *scale* (bold call graph node) called from *main* via *normalize* after a prior call to *read* which has filled the integer array *buffer* local to function *main*. The previous write access to *buffer* by call graph node *read* has been stored in *main* as a *last writer* edge. Prior to *main* handing over the thread of execution to *normalize* LLILA's code instrumentation created a call frame providing the following information on the current function parameters for the callee: a) the argument's position, b) the argument's identifier in the caller's name space, c) if the parameter could be traced back to another object either in the same basic block scope, function local scope, function argument scope or in the global scope and e) if a trace exists, the objects identifier at the end of the trace. As Figure 3 suggests, a similar frame was created when *normalize* called *scale*. The information passed on from caller to callee is the essence of the reverse data flow analysis discussed earlier on.

We now assume that during the simulated execution of *scale* a load instruction marked as relevant to global data flow analysis is issued. The simulation and profiling environment computes the interprocedural data flow in the following manner (the steps are referenced in Figure 3):

1) Static analysis of the load instruction instrumented the operation as an effective read access to the function's first argument *buf*.
2) The simulator consults the top call frame created by *normalize* on the call stack and tries to resolve the argument identifier *buf*. The call frame indicates that the object is locally known as the argument *buf* at position zero and represents an incomplete trace.
3) The stack frame created by *main* indicates that the argument *buf* can be traced back to the locally scoped object *buffer*. Hence, *main* is the owner of the object pointed to by *buf* in function *scale*. The trace is now complete and call node *scale* will be marked as a consumer to *buffer*.
4) The call node for *main* is requested to resolve the call node with write access to *buffer* the last time.
5) Call node *read* has been recorded to modify *buffer* the last time and denotes the source of the next interprocedural data flow edge via *buffer*.
6) We have identified an effective interprocedural data flow edge from call node *read* to call node *scale* via *buffer* located at *main*.

In case of a store instruction dynamic backtracking would operate in a similar fashion: Once the origin of an object passed through one or more function calls has been identified, the owning call node would be instructed to redirect the "last writer" edge to the call node issuing the back trace.

If the process of locating and updating sources and sinks of computation on *buffer* is continued throughout simulated program execution for the above example, runtime analysis will yield the interprocedural data flow graph depicted in Figure 4. One can see that for the effective communication the actual location or owner of the storage associated with *buffer* is actually irrelevant. Although *buffer* is located at call node *main* no data is ever pass through this function.

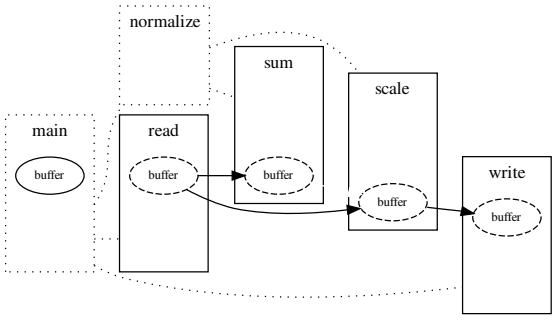

Fig. 4. Effective global data flow graph constructed by combining static reverse data flow analysis and runtime source and sink identification. In this particular example a strictly directed flow from the primary source *read* can be observed via *sum* and *scale* towards the sink *write*.

E. Simulator Generation

We have mentioned earlier on, that our primary focus in instruction set architecture simulation is detailed profiling of runtime behavior with a minimum effort of moving an application into the simulation and profiling environment. Also, we need to achieve moderate to high executing performance since an application may need to run on large amounts of "test vector" data before the acquired profiling data can be augmented to represent typical program behavior in a statistical sense. As a consequence, the LLILA tool will generate an ANSI C Program from the instruction sequence earlier on and will insert additional profiling code in the sense of augmentation or instrumentation described in section two. Hence, it can be said that LLILA generates a self profiling ISA simulator from LLVM byte code. The main obstacles in the translational process from LLVM instruction level to a C program have already be indicated earlier on: For producing correct, compilable programs identifier name de-mangling needs to be performed and function calls to external entities be identified. Note that LLILA treats external function calls in the simplest of fashion by simply inlining them into the generated C program. The task of resolving them is left up to the C compiler.

III. Experimental Results

The LLILA project is at an intermediate state of realization: Byte code and assembly analysis as well as compiled instruction set simulator generation cover the full semantic scope of the LLVM framework. In order to evaluate our approach of using virtual machine architectures for ASP/ASIP synthesis and quantitative global data flow analysis for code partitioning, several "real world" applications from the domain of digital video signal processing have been investigated. Our main test-case is currently the MPEG2 video decode and encode reference implementation officially released by the MPEG Software Simulation Group [5]. Program analysis, compiled ISA simulator generation and simulator execution with all profiling options turned on were performed on an 8 core 3 GHz Intel Xeon System with 32 Gigabytes of main memory under Ubuntu Linux 7.10.

Figure 5 features the call sequence FSM of the MPEG2 decoder's main decoding loop with all its immediate callees. It was annotated with runtime call frequency data from decoding exactly one frame from the test data stream. Figure 6 shows a typical execution sequence of subroutine calls issued by the decoder with a total length of over 644858 calls for decoding a single picture. The graph represents the shortest possible encoding of the call sequence and exposes a maximum level of three nested loops. The bold directed edge in Figure 5 denote consecutive read after read accesses and read after write accesses to data object number 182 (an MPEG layer data descriptor structure) which is passed from the main function onto the main decoding loop all the way to the function writing out the decoded picture. This is only an example of tracking one single variable instance across multiple functions by profiling load and store operations on call by reference or global objects. For the purpose of data

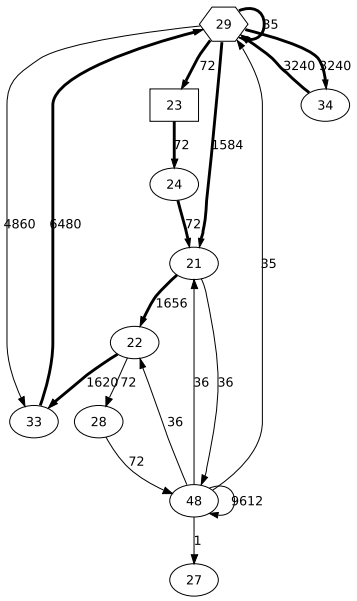

Fig. 5. Automata of the MPEG2 decoder's main decoding loop *Decode-Picture* which intially calls FlushBuffer.

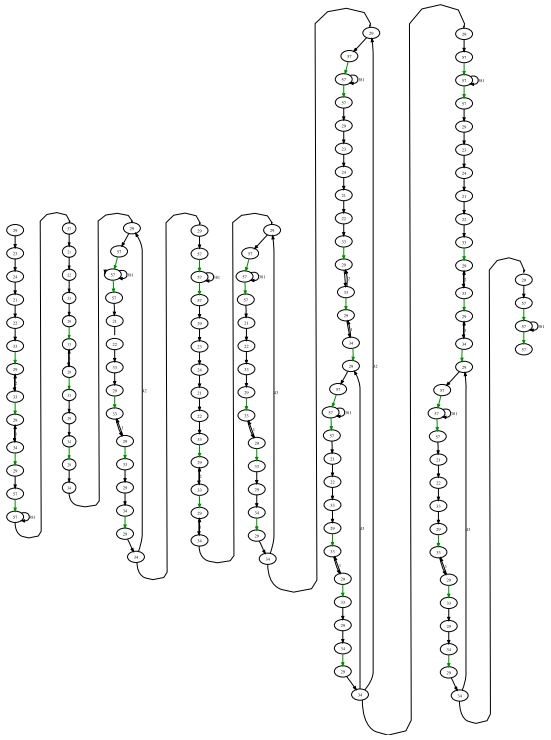

Fig. 6. One of four detected path patterns through the CSG with a total length of 644858 function calls.

flow guided partitioning of course all exchanged data - both global as well as all shared objects across the procedural level - will need to be considered for plotting a singular singular path of flow through the whole software system.

1) Memory Access Pattern Detection: During program execution, load and store operations to global data objects were captured inbetween function calls and analyzed for typical access patterns. These were recorded separately for read and write operations on individually memory addresses and differentiated by access frequency. In order to facilitate

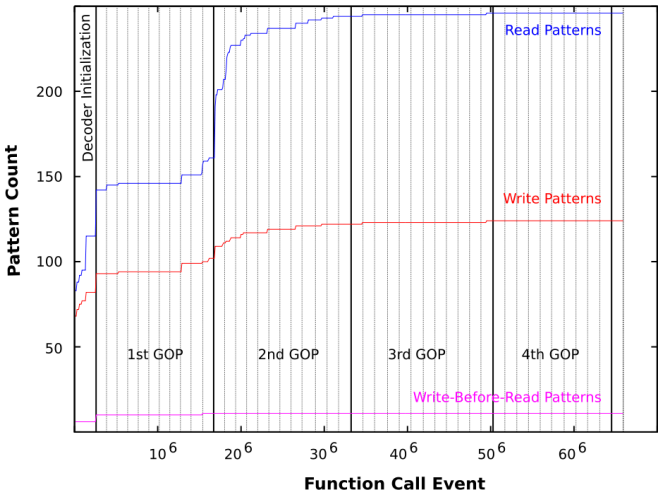

Fig. 7. Total number of read, write and write-before-read patterns recorded during the decoding of 48 frames in 4 GOPs. The x-axis denotes the executing time on a function call event scale whereas the y-axis the total number of identified access patterns. Vertical separators denote the completion of a single frame indicated.

later analysis of the lifespan of an individual computation of a given global variable, write-before-read event patterns were also extracted for all memory locations. The combination of read, write and write-before-read pattern makes up a single access pattern and is used to annotate the above mentioned path patterns through the CSG. As can be seen in Figure 7, the number of access patterns quickly converges towards an upper bound, where first patterns can be attributed to the initialization phase of the decoder (Huffman tables of the run length decoder, iDCT coefficients etc.). The steep increase in patterns between the first and second GOP (Group of Pictures) reflects the properties of the MPEG datastream: The first GOP consists of I (Inter) and P (Predicted) frames only, whereas the second also features B (Backward Predicted) frames forcing the decoder to access more reference data to decode an individual frame.

2) Quantitative Global Flow: Combining call sequence graph paths patterns and memory access patterns, one is able to compute the total number of data transfers along each path of program execution. This has been done for the path pattern depicted in figure 2 and is shown in Figure 8 as an integral plot of both read and write transfers. Even though a total of 14.5 MBytes are transfered to and from memory for a single picture to decode on average, the average global flow between function calls is quite low with only 8 bytes. However, one can clearly make out the boundaries of a stereotypical subsequence of calls which is repeated exactly 382 times. In the extracted subsequence path pattern number 117 the decoder processes motion estimation vectors (the 8 byte transfer average figure) from the variable length decoder. The latter makes a number of calls to the bitstream stream buffer function *FlushBuffer* which results in peak

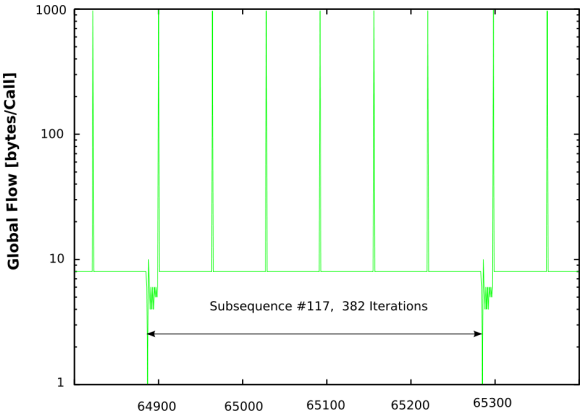

Fig. 8. Partial quantitative global dataflow along one call sequence graph pattern of the main decoder. The graph represents the total dataflow between function call 64800 and 65400. Highlighted is the stereotypical subsequence 117.

transfers from file to memory. In order to get a notion of how the call graph members communicate via global data objects, the communication graph has been outlined in Figure 9 with the total number of bytes transfered between functions and global data objects for decoding a single picture. It shows

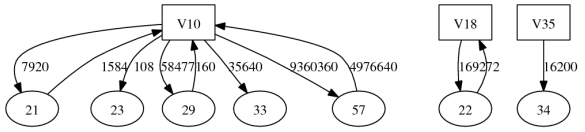

Fig. 9. Communication Graph of the MPEG decoder. Ellipse nodes represent members of the function call graph whereas square nodes represent global data objects

that very few functions are actually involved in the bulk decoding process: The majority of data is transfered through the *FlushBuffer* function (callnode 29) of the runlength decoder followed by the floor library function (callnode 57) as part of the iDCT algorithm. Interesting is also global data object V35 which is only read but never written. At a closer look, the source code reveals that V35 provides the coefficient set feeding the iDCT.

IV. Conclusion and Outlook

Simulation of virtual instruction set architectures is a powerful tool for extensive runtime program analysis without the overhead of full scale virtualization. Experiments indicate that combining code instrumentation of virtual instructions with additional simulator infrastructure for tracking inter-procedural data exchange provides a feasible environment in which both classic execution traces as well as quantitative global data-flow analysis can be conducted. Data gathered simulation experiments of this work feature all of the relevant data in order to automatically partition a strictly sequentially formulated application into a coarse grained parallel, distributed one based on its communication behavior. However this insight comes at a cost: First of all, code instrumentation and load and store instruction tracking as indicated above causes a great amount of overhead in the simulator. When compared to a none profiling simulator the overall execution time increases by a factor of roughly 500 (the additional memory overhead is negligible) due to the large number of updates of simulator internal data structures. Also, the amount of profiling data for call sequence graph and global communications graph analysis are overwhelming. One hour of MPEG2 video decoding and profiling "costs" roughly 2 weeks of computation time with almost 2 terabytes of compressed data gathered from instruction instrumentation.

At the present simulation and analysis of profiling data is performed in a single-threaded simulator instance which accounts for most of the runtime budget. Current work focuses on a multi-threaded version of the simulation environment which dedicates a single thread to the application execution where as the gathered profiling data is spread evenly among the remaining processor cores resources. With respect to the task of actually partitioning an application for a multi-core or distributed, embedded environment, work is on the way to break up the code into a multiple thread description for posix pthreads as well as SystemC.

References

[1] G. Stitt, F. Vahid, *A Decompilations Approach to Partitioning Software for Microprocessor/FPGA Platforms*, Proceedings of the Design, Automation and Test in Europe Conference, 2005

[2] T. Grotker, *System Design with SystemC*, Kluwer Academic Publishers, 2002.

[3] Simplescalar Home *www.simplescalar.com*.

[4] B. Cmelik et al., *Shade: A Fast Instruction-Set Simulator for Execution Profiling*, ACM SIGMETRICS Performance Evaluation Review, Volume 22(1), pp.128-137, May 1994

[5] MPEG Software Simulation Group, *MPEG-2 Encoder / Decoder, Version 1.2*, http://www.mpeg.org/MSSG, 1996

[6] E. Schnarr et al., *FACILE: A Language and Compiler for High-Performance Processor Simulators*, PLDI, 1998

[7] E. Witchel et al., *Embra: Fast and Flexible Machine Simulation*, MMCS, 1996

[8] M. Hartoog et al., *Generation of Software Tool Sets fir Application Specific Processor Descriptions for Hardware/Software Codesign*, Proceedings of DAC, 1997

[9] G. Hadjiyiannis et al., *ISDL: An Instruction et Description Language for Retargetability*, Proceedings of DAC, 1997

[10] P. Marwedel. *The mimola design system: Tools for the design of digital processors*, DAC '84: Proceedings of the 21st conference on Design automation, 1984

[11] A. Nohl et al., *A Universal Technique for fast and Flexible Instruction-Set Architecture Simulation*, Proceedings of DAC, 2002

[12] S. Pees et al., *Retargeting of Compiled Simulators for Digital Signal Processing using a Machine Description Language*, Proceedings of DATE, 2000

[13] G. Braun et al., *Using Static Scheduling Techniques for the Re targeting of High Speed, Compiled Simulators for Embedded Processors from Abstract Machine Description*, Proceedings from ISIS, 2001

[14] P. Mishra et al., *Functional Abstraction driven Design Space Exploration of Heterogeneous Programmable Architectures*, Proceedings of ISSS, 2001

[15] W. Boehm et al., *Mapping a Single Assignment Programming Language to Reconfigurable Systems*, The Journal of Super computing, Volume 21, pp.117-130, 2002

[16] D. J. V. Evans, A. M. Goscinski *Automatic Identification of Parallel Units and Synchronization Points in Programs*, International Journal of Computer Systems Science and Engineering, 1997

[17] A. Goscinski et al., *Towards a Global Computer: Improving the Overall Distributed System Performance an the Computational Services Provided to Users by Employing Global Scheduling and Parallel Execution*, ARC Large Grant Application, Deakin Univeristy, 1994

[18] C. Lattner et al., *LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation*, Proceedings of CGO, 2004