Compiled Low-Level Virtual Instruction Set Simulation and Profiling for Code Partitioning and ASIP-Synthesis in Hardware/Software Co-Design

Carsten Gremzow
Faculty of Computer Science and Electrical Engineering
Berlin University of Technology
email: gosper@cs.tu-berlin.de
WWW: http://rt.cs.tu-berin.de

Keywords: Instruction Set Architecture Simulation, Coarse-Grained Parallelism, Profiling, Hardware/Software Co-Synthesis, LLVM, Quantitative Dataflow Analysis

Abstract

We present ongoing work and first results in static and detailed quantitative runtime analysis of LLVM byte code for the purpose of automatic procedural level partitioning and co-synthesis of complex software systems. Runtime behaviour is captured by reverse compilation of LLVM bytecode into augmented, self-profiling ANSI-C simulator programs retaining the LLVM instruction level. The actual global data flow is captured both in quantity and value range to guide function unit layout in the synthesis of application specific processors. Currently the implemented tool LLILA (Low Level Intermediate Language Analyzer) focuses on static code analysis on the inter-procedural data flow via e.g. function parameters and global variables to uncover a program's potential paths of data exchange.

1. INTRODUCTION

In order to meet the increasing performance an energy demands of current embedded systems applications, hardware/software co-design approaches often partition critical sections of software onto dedicated hardware serving as an accelerator to standard microprocessor platforms. These application specific processors (ASP) feature highly application specific data- and control-paths whereas other approaches generate VLIW-like co-processors from software binaries of an application. Also, tuning the instruction set of an existing processor architecture into an application specific instruction processor (ASIP) to meet an application's particular runtime behavior and/or instruction level parallelism has been shown to be very promising in the recent past yielding execution speedups up to an order of magnitude compared to standard RISC processors. Application execution profiles are commonly used to identify critical regions of an application and guide the hardware/software partitioning process. Note however, that the term 'critical' is regularly used to primarily denote frequency of code execution. Following Amdahl's Law[2], selecting code regions based on percentage of execution time guarantees the largest potential speedup. However, one critical aspect of runtime behavior for hardware/software partitioning embedded applications to any particular system architecture (Co-Processor, ASIPs or a generic distributed system) is however seldom covered by traditional profiling tools: Extending a profiler's scope to capture the quantitative data flow profile during program execution. This will uncover the application's communication behavior and the actual amounts of data being transfered between various regions of the application and the particular mode of transfer (e.g. via stack, registers or global objects etc.).

1.1. Scope and Problem Outline

Our work has two primary goals: a) to provide the means to perform data flow driven, multi-way partitioning of software onto standard and custom application specific processors and b) to construct ASPs and/or ASIPs from static as well as dynamic program properties. For construction of the later one needs to perform static program analysis and high-level reconstruction (e.g. control-/data flow graphs) for the task of high-level synthesis. This problem can be addressed either from the source code level as well as the object code level (decompilation). In turn, profile driven partitioning depends on executing and monitoring of binary level representations (either simulated or on real hardware). Hence, the key objective is to find a proper program representation suitable both for the task of synthesis as well as the task of efficient run time analysis while maintaining a reasonable degree of abstraction from the actual target architecture. In the following we will present an approach to automatically gather static as well as dynamic runtime program behavior in detail by deriving self-profiling instruction set simulators from arbitrary ANSI C and C++ compiled to a low level virtual machine architecture. Acquired data is used to perform ASP/ASIP synthesis as well as software partitioning into coarse grained parallel units.

2. PREVIOUS WORK

In the next sections we will review recent achievements in simulated program execution, dynamic and static analysis as well as partitioning to distributed systems which we believe are of relevance to this paper.

 ISBN # 1-56555-316-0

2.1. Simulation, Augmentation and Profiling

An extensive body of recent work has addressed instruction set architecture simulation. The wide spectrum of today's instruction set simulation techniques include the most flexible but slowest interpretive simulation and faster compiled simulation. Recent research addresses retargetability of instruction set simulators using machine description languages. Simplescalar [9] is a widely used interpretive simulator that does not have any performance optimization for functional simulation. Shade[10], Embra [13] and FastSim [12] simulators use dynamic binary translation and result caching to improve simulation performance. Embra provides the highest flexibility with maximum performance but is not retargetable: it is restricted to the simulation of the MIPS R3000/R4000 architecture. A fast and retargetable simulation technique is presented in [11] and improve on traditional static compiled simulation by aggressive utilization of the host machine resources. This is achieved by defining a lower level code generation interface specialized for ISA simulation, rather then the traditional approaches that use C as a code generation interface. Retargetable fast simulators based on an Architecture Description Language (ADL) have been proposed within the framework of FACILE [12], Sim-nML [14], ISDL [15], MIMOLA [16], LISA [17][18][19] and EXPRESSION [20]. A simulator generated from a FACILE description utilizes the Fast Forwarding technique to achieve reasonable high performance. All of aforementioned approaches assume that the program code is run-time static.

The problem of gathering a program's memory typical access behavior and data layout in memory has been studied intensively both in the parallel programming on shared memory systems and cluster computing community as well as in the computer architecture community for designing memory hierarchy and cache layout. In the first case a significant amount of projects has focused on improving the data locality of running programs and further minimizing the memory access latency. The SIMT System [21] is a multiprocessor simulator for PC based clusters with NUMA interconnection fabrics. It simulates the parallel execution of shared memory programs and provides extensive and detailed information about their run-time data layout. This information allows users to analyze an application's memory access behavior and to specify an optimized data placement within the source codes resulting in a minimum of remote accesses at run-time. It is an extended version of Augmint [5], a fast execution driven multiprocessor simulation toolkit for Intel x86 architectures which applies a code augmentation techniques by inserting instrumentation code into the source object code effectively turning it into a simulator executable.

The instrumentation code makes reference to any instruction causing direct or indirect memory read or write access and forwards it to the simulator's run-time environment. A similar system for instrumenting binaries is the DIOTA system [1], which also deals correctly with programs that contain traditionally hard to instrument features such as data in code and code in data. In contrast to the aforementioned approaches DIOTA performs dynamic instrumentation which does not require reverse engineering, program understanding tools or heuristics about the compiler or linker used. The instrumented code is generated on the fly and records faulty memory accesses, data races, deadlocks as well as some basic tracing of operations and memory accesses. Another instrumentation tool for the Intel A32 architecture is Valgrind [3] which detects memory-management problems. When a program is run under Valgrind's supervision, all reads and writes of memory are checked and calls to malloc(), new(), free() and delete() are intercepted. Valgrind uses techniques that are very similar to the above mentioned methods. Valgrind attaches itself to any dynamically-linked ELF x86 executable, without modification, recompilation, or anything.

Conclusion. Using either compiling or interpretive ISA simulators to perform quantitative data flow analysis on standard applications is complicated by the circumstance that applications either need modifications to run stand alone or they need to be simulated in a complete, virtual system. Instrumenting binaries by inserting supervision code can be used to overcome this deficiency. Never the less the problem of associating objects on source code level (e.g. variables) with corresponding objects on the binary level remains with all profiling techniques on real architectures (without reverting to the overhead of debugging data formats such as DWARF and GNU STABS).

2.2. Decompilation

A decompiler, or reverse compiler, is a program that attempts to perform the inverse process of the compiler. Given an executable program compiled in any high-level language, the aim is to produce a high-level language program that performs the same function as the executable program. Thus, the input is machine dependent, and the output is language dependent. Decompilation was originally developed for purposes of translating software binaries from one instruction set architecture to another and for recovering high-level code from legacy assembly code . In the following decompilation is used for a different purpose, namely for converting a software binary into a representation suitable for synthesis.

The FREEDOM Compiler [4] automatically translates software assembly and binary codes targeted for general DSP processors into Register Transfer Level (RTL) VHDL or Verilog code to be mapped onto commercial FPGAs. The Texas Instruments C6000 DSP processor architecture has been used as the DSP processor platform, and the Xilinx Virtex II as the target FPGA. Various optimizations including loop unrolling, induction variable analysis, memory and register optimiza-

tions, scheduling and resource binding.

Recent work by Vahid and Stitt [6] also use existing decompilation methods and adapt them to to convert the software binary into a control/data flow graph (CDFG) annotated with high-level information. Initial binary parsing converts the software binary into an instruction set independent representation. Next, CDFG creation builds a control/data flow graph (CDFG) for the application. Control structure recovery analyzes the CDFG and determines high-level control structures, such as loops and if statements. After recovering a CDFG of the application, several optimizations need to be applied to eliminate overhead introduced by the instruction set.

Conclusion. Gathering high-level data from source code level description of modern programming languages for ASP/ASIP synthesis is a very complex and tedious task. Synthesis from binary level descriptions is a true alternative and while recovering the necessary high-level data it also introduces an abstraction from language specific details to the synthesis task. Yet, real general purpose and signal processing architectures feature a multitude of instructions (more than 1000 for IA32) which obfuscate the process of high-level reconstruction from object code.

2.3. Source and Instruction Level Partitioning

Nearly all software/hardware partitioning approaches partition at the source code level during or even before compilation. Software partitioning at source code level is traditionally performed manually by use of co-design-centered languages such as SystemC [8], HandleC, SiliconC, SA-C [22] or StreamC. The majority of these systems and languages require either special notations or actions on behalf of the programmer (e.g. pragmas) or a specific programming paradigm which force the parallelism to be explicitly identified by the programmer. The actual granularity of parallelism ranges from the instruction to procedural level. From an analytical point of view plain source code offers the highest degree of freedom during design space exploration for hardware synthesis (code re-timing, loop unrolling). Binary-level hardware/software partitioning is the process of extracting computational kernels of a software binary into regions that will be implemented in custom hardware and regions that will execute in the existing binary format and has been under investigation by [6] among others. Most of the current work on partitioning for hardware/software co-design focuses on a bipartite partitioning modeling where the overall software system is split into two parts: The majority of the code usually resides on a general purpose processor whereas computational intensive kernels are moved to dedicated hardware usually residing on programmable logic fabric such as FPGAs either configures statically or as currently intensively discussed in a dynamic reconfiguration environment.

However, it has been proposed more than a decade ago that a compiler can be used to detect which parts of a program, written in a procedural language can be executed in parallel [23]. Furthermore, it has been proposed that a compiler can be used to detect when program synchronization should occur. Coarse-grained parallelism based upon the logical structure of a program's components is required to get the optimal performance of program in a distributed computing environment [24], while parallelism is normally based around loops or some other programming constructs that involve iteration.

Conclusion. Identifying parallel sequences of execution and estimating global data flow and associated throughput of an application is a task usually performed manually by the designer. For automatic software partitioning to go beyond a bi-partit architecture, frequency of code execution needs to be complemented by accurate figures for data production and consumption of code as a partitioning metric.

3. LOW LEVEL VIRTUAL MACHINE SIMULATION

In order to find a common basis of program representation equally suitable for the problems mentioned above (simulation/profiling, partitioning and ASP/ASIP synthesis) we have turned to using virtual instruction set architectures as opposed to real ones. Virtual machine instructions sets are usually lean ($\leq$ 255 instructions) and either stack based (Java Virtual Machine, CIL/.NET) or use no register files at all (LLVM, see below). Hence, the reconstruction of high-level information (control- and data flow) can easily be implemented. Also, tracing data flow between instructions is not obfuscated by direct or indirect memory or register file access as in the Augmint approach. Also, execution environments for virtual machines are fairly easy to modify (excluding just-in-time code transformations in preference of lower interpretive execution) in order to extract the runtime behavior discussed above.

Our ASP/ASIP synthesis and runtime analysis environment *Synphony* is centered around two virtual architectures: The ECMA-335 Common Language Infrastructure (CLI) - also known as .NET - and the LLVM (Low Level Virtual Machine) framework [26]. The latter is a compiler framework designed to support program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations. 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 CLI 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.

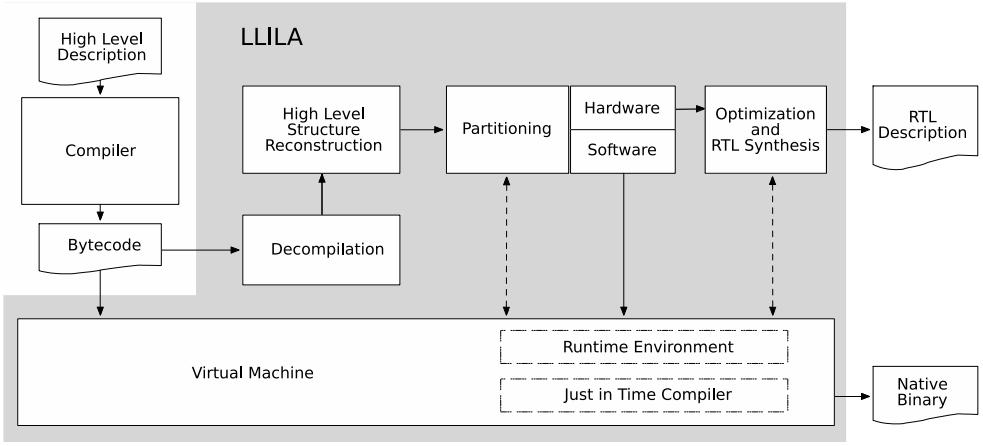

Figure 1. Schematic outline of the LLILA simulation, profiling and synthesis flow. The grey area denotes the our actual tool flow boundary whereas program compilation is performed using the LLVM gcc tool chain.

3.1. 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 hosts 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 hosts native environment.

After close examination of the LLVM execution environment we chose not to take it as a starting point for our simulation and profiling needs for a number of reasons: At the current state of implementation (LLVM 1.8) the interpretive code execution engine is incomplete with respect to the standard C library. Hence, standard benchmark suits cannot be executed. In that sense the just-in-time execution engine is complete but far to complex for our purposes. Instead our LLVM Intermediate Language Analyzer (LLILA) has no LLVM dependencies an was written from scratch to deal with LLVM bytecode or assembly files.

3.2. Static Program Analysis

After parsing an LLVM byte code program static program analysis starts with extracting type declarations and globally defined memory and function objects which are held for later reference.

3.2.1. Basic 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 fairly easily be identified due to LLVM's single static 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 signature. Therefor LLILA also checks the type signature of each instruction and performs variable renaming to unify 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 easily 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 graph "flat" representation. This step is essential for further analysis (see indirect data flow detection below) as well as to create potential for increased throughput during ASP/ASIP synthesis. For details on the graph folding process please refer to [27] for it is beyond the scope of this paper.

3.2.2. Higher Order Data Structures

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 subprograms 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.

3.2.3. Indirect Dataflow Detection

As mentioned earlier, we wish to record byte accurate figures of the amounts of data transfered into and out of an individual function during the course of its execution. A function's signature is a good starting point for pure static analysis for it defines the amounts and types of data transfered via the stack. Estimating the data flow is trivial when the programmer uses call by value only. This is of course seldom the case and intensive use of pointers and variable size arguments are used. The following piece of code which computes the sum of an integer data array of variable size is a simple example where a priori data flow estimation is impossible:

```
int simple (int len, int *buf) {
  int i, sum = 0;
  for (i = 0; i < len; i++)
    sum += buf[i];
  return sum;
}
```

In order to track the function's actual indirect data flow we need to analyze LLVM's load and store instruction which are the only means of accessing memory locations. Yet, relating an access to a variable on the source code level (in the case of the above example it would be the integer pointer "buf") with the corresponding load instruction on the assembly level is somewhat obfuscated due to the pointer arithmetic overhead inserted by the compiler. In the following lines of LLVM assembly code one can see the instruction sequence of the loop's entry part and body statement, right hand side only:

```
entry:
  store int* %buf, int** %buf_addr
  [..]
b2:
  [..]
  %tmp.4 = load int** %buf_addr
  %tmp.5 = load int* %i
  %tmp.6 = gep int* %tmp.4, int %tmp.5
  %tmp.7 = load int* %tmp.6
  [..]
```

One can clearly see that looking at the load instructions during loop execution does not reveal any direct relation to the variable "buf". In order to uncover any indirect read or write data flow either through call by reference or global variables, we employ backtracking on the flat control / data flow graph gathered earlier during static program analysis for all load and stores instructions. In the case of our example, the load instruction for "tmp.7" features a data dependency all the way from the gep (get element pointer) instruction to the initial pointer computation of "buf". It will be marked as a read reference to "buf" as part of the runtime profiling process.

3.2.4. Instruction Scheduling

Without going into too much detail it should be noted, that instruction scheduling is also performed as part of the static analysis phase. LLILA can apply simple ASAP/ALAP heuristics as well as Integer Linear Programming techniques for optimal scheduling both with and without resource constraints. As a result instructions will be assigned a virtual instruction slot even though the compiled ISA simulator processes them in strict sequential order. During program simulation instruction level parallelism and hypothetical speedup can thus be recorded and evaluated by the designer.

3.2.5. Static Feature Database

Static program analysis is completed by creating a graph database for the extracted program features (flat control/data flow graphs, call graph, call sequence graph etc.) as an XML representation. Later on it will be reread by the monitoring application of the runtime environment for annotation. The database is not compiled into the simulator itself since the enriched graphs are needed for partitioning based on the data flow characteristics.

3.3. Compiled ISA 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 statical 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. In order to get an impression of the generated C code we return to the above example and look at the loop body:

```
bb2: {
 __llila_lbid = __llila_cbid;
 __llila_cbid = 2;
 __llila_block_enter();
 __llila_inc_cstep_counter(5);
 __llila_inc_inst_counter(11);
 __llila_prof_tab[441]++;
 int* l0 = *&v1;
 __llila_prof_tab[442]++;
 int l1 = *&v3;
 __llila_prof_tab[444]++;
 int (*l2) = &(l0[l1]);
 __llila_prof_tab[444]++;
/* argument variable traceback : int *buf */
 __llila_arg_read(2);
 int l3 = *l2;
 [..]
}
```

The first three statements are used for recording the last executed basic block and the current basic block. The control flow transition is then communicated to the runtime environment for profiling through the block enter library call. The following statement advances the virtual parallel instruction execution profiler by 5 control steps (the figure has been computed during instruction scheduling based on the data flow graph's ILP) whereas strict sequential instruction execution profiling is incremented by 11 (the number of instructions in this basic block). Note that for all instruction prior to their execution a global instruction profiling table is incremented. This will yield a simple instruction execution distribution after the program has terminated and is used to gather traces later on.

The case of indirect variable access by pointer arithmetic obfuscation discussed earlier on can be seen in the last instruction where l3 is read by dereference of l2: Dataflow traceback of this instruction indicates that function argument number two (integer pointer "buf") has been accessed. The preceding profiler library call records this action.

4. EXPERIMENTAL RESULTS

Although the LLILA project is at a very early 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 [28].

Program analysis, compiled ISA simulator generation and simulator execution with all profiling options turned on were performed on an AMD Athlon 64 3000+ machine with 1 Gigabyte of main memory under Ubuntu Linux 6.10.

4.1. Compiled Simulator Size and Performance

For the native MPEG2 decoder binary it takes 10.046 cpu seconds (measured using unix command time(1)) to decode and save 72 video frames from the sample elementary bit stream. This averages to 0.139 seconds per frame and servers as a performance reference. The decoder was compiled using a native ix86 gcc from 8694 lines of C source code. The self profiling ISA simulator generated by LLILA from the llvm-gcc compiled byte code of the MPEG2 decoder results in 46797 lines of C source code. This equals an increase in code size by a factor of roughly 5.4. When we look at LLVM assembly file which amounts to an overall of 13625 instructions, we're left with an overhead of 2.5 additional statements for profiling for each instruction in the generated simulator. Unfortunately we're not able to provide execution performance figures using LLVM's own runtime engine (neither JIT or interpretive) since it fails to run the MPEG2 decoder completely. For the simulator to decode a single picture we measured an average of 59.8 seconds. During that period, the simulator executed on average over 256 million LLVM instructions which lead us to a simulation performance of over 4.2 million instructions per second on the test system with all profiling options in place. When we compare the performance of the native ix86 MPEG2 decoder to the compiled instruction set simulator, we're faced with an overall average "speed-down" of factor 430. At first sight this figure may appear to be a bit of a disappointment since it suggest, that the anticipated level of profiling detail comes at cost of 2.6 order of magnitude in speed decrease. Traditional execution profiling using *gprof* for example comes at a cost of only 1.7 percent more execution time in case of the MPEG2 decoder, yet it does not deliver the degree of profiling detail the compiled instruction set architecture simulator offers.

4.2. Tracking Global Dataflow

Figure 2 features the call sequence graph of the MPEG2 decoder's main decoding loop with all its immediate callees. It was annotated with runtime data from decoding exactly one frame from the test data stream (generating a pictorial of the flat call sequence graph is beyond the limits of the graph layout program).

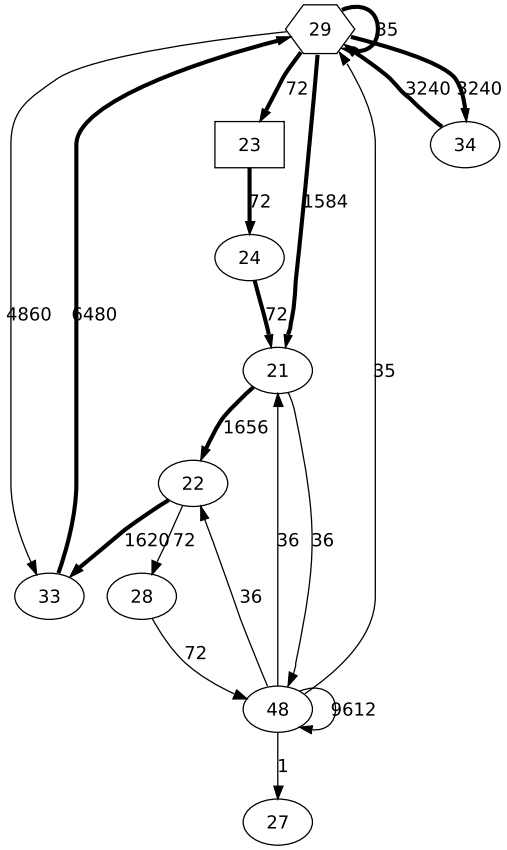

Figure 2. Call Sequence Graph of the MPEG2 decoder's main decoding loop DecodePicture which intially calls FlushBuffer. The bold directed edges denote the global flow trace of a singular variable.

The boled directed edge denotes 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 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. This has not yet been implemented.

4.3. ILP Estimation

In order to get a rough estimate of instruction level parallelism of the LLVM byte code, ASAP scheduling was performed without any resource constraints. The resulting peak parallelism of 8 ALU operations (including multiplication) with simultaneous 16 LOAD operations does not come as a surprise, since it reflects the computationally most expensive part of an MPEG decoder - the iDCT. Assuming a hypothetical ASIP layout suitable for the above mentioned peak instruction constellation, the instruction simulator records (see figure 3) an overall average speedup of 2.6 for the decoding the first frame (which is an I-frame) an then moves up to an average speedup of 2.9 for the next less costly predicted frame which benefits from the 16 LOAD/STORE ports for increased memory transfer during motion compensation. Interestingly these speedup figures are also reported by [25] who performed a bi-partit hardware/software co-design of a H.264/AVC decoder. The system consisted of an ARM920 processor core where the computational intensive tasks (also iDCT along with motion compensation) were partitioned onto a XilinX Virtex2 FPGA.

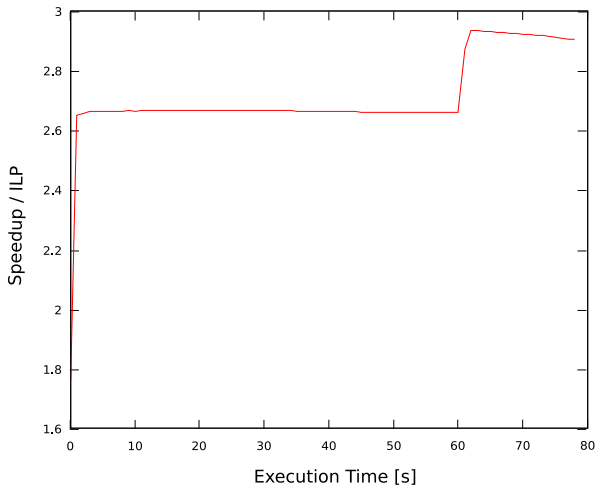

Figure 3. Average Instruction level parallelism measured for the MPEG2 decoder over execution time. Integration period has been set to 1 second.

5. CONCLUSION

First experiments show that virtual machine instruction set architectures are a promising common basis for the purpose of extensive runtime program analysis without the overhead of full scale virtualization as well as the task of ASP synthesis. Yet, effective software partitioning from the overwhelming amounts of data gathered from global data flow profiling needs to be proved in the immediate future.

REFERENCES

[1] J. Maebe, K. De Bosschere, *Instrumenting self-modifying code*, Proceedings of the Fifth International Workshop on Automated Debugging, 2003

[2] G. M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, AFIPS Conference Proceedings, 1967

[3] N. Nethercote, J. Seward, *Valgrind: A Program Supervision Framework*, Electronic Notes in Theoretical Computer Science, Volume 89, No.2, Elsevier Science, 2003

[4] G. Mittal, D.C. Zaresky, X. Tang, *Automatic Translation of Software Binaries onto FPGAs*, Proceedings of Design Automation Conference 2004, pp.389-394, 2004

[5] A. Nguyen, M. Michael, A. Sharma, J. Torellas, *The Augmint Multiprocessor Simulator Toolkit for Intel x86 Architectures*, Proceedings International Confernce on Computer Design, 1996

[6] 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

[7] T. Austin, E. Larson, D. Ernst, *Simplescalar: an infrastructure for computer system modeling*, IEEE Computer, Volume35, Issue 2, pp.59-67, 2002

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

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

[10] 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

[11] J. Zhu et al., *A Retarggetable, Ultra-fast Instruction Set Simulator*, Proceedings of DATE, 1999

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

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

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

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

[16] R. Leupers et al., *Generation of Interpretive and Compiled Instruction Set Simulators*, Proceedings of DAC, 1999

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

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

[19] 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

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

[21] J. Tao et al., *SIMT/OMP: A Toolset to Study and Exploit Memory Locality of OpenMP Applications on NUMA Architectures*, Springer Lecture Notes in Computer Science, Volume 3349, pp.41-52, 2005

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

[23] 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

[24] 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

[25] G. Stitt, F. Vahid, *Hardware/Software Partitioning of Software Binaries: A Case Study of H.264 Decode*, Proceedings of CODES+ISSS, pp.285-290, 2005

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

[27] C. Gremzow, *High-Level Syntheses aus flachen Kontroll-/Datenflussgraphe*, Doctoral Thesis at Berlin University of Technology, 2004

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