ACCLIB: Accelerators as Libraries

Jacob R. Stevens*, Yue Du[§+], Vivek Kozhikkottu[¶+], Anand Raghunathan*
*School of Electrical and Computer Engineering, Purdue University
[¶]Intel Corporation
[§]IBM
{steven69, raghunathan}@purdue.edu, vivek.kozhikkottu@intel.com

***Abstract*— Accelerator-based computing, which has been a mainstay of System-on-Chips (SoCs) is of growing interest to a wider range of computing systems. However, the significant design effort required to identify a computational target for acceleration, design a hardware accelerator, verify the correctness of the accelerator, integrate the accelerator into the system, and rewrite applications to use the accelerator, is a major bottleneck to the widespread adoption of accelerator-based computing. The classical approach to this problem is based on top-down methodologies such as automatic HW/SW partitioning and high-level synthesis (HLS). While HLS has advanced significantly and is seeing increased adoption, it does not leverage the ability of experienced human designers to craft highly optimized RTL, nor does it leverage the growing body of already existing hardware accelerators.**

In this work, we propose ACCLIB, a design framework that allows software developers to utilize existing libraries of pre-designed hardware accelerators automatically with no prior knowledge of the function of the accelerators, with minimal knowledge of hardware design, and with minimal design effort. To accomplish this, ACCLIB uses formal verification techniques to match a target software function with a functionally equivalent accelerator from a library of accelerators. It also generates the required HW/SW interfaces as well as the code necessary to offload the computation to the accelerator. We validate ACCLIB by applying it to accelerate six different applications using a library of hardware accelerators in just over one hour per application, demonstrating that the proposed approach has the potential to lower the barrier to adoption of accelerator-based computing.

***Keywords*—accelerator-based computing, high-level synthesis; interface synthesis; formal verification**

I. Introduction

Accelerator-based computing, a paradigm in which compute-intensive parts of an application are offloaded to specialized hardware, is an integral part of most modern computing platforms. This approach is already heavily embraced by embedded systems and System-on-Chip (SoC) designers [1]-[3], and there is widespread interest in broadening its use. However, design effort remains the major obstacle to adoption of accelerator-based computing. The causes for increased design effort include: identifying code regions amenable to acceleration, developing and verifying the accelerator, integrating the accelerator into the hardware platform, and modifying the target application to use it.

Open source and commercial hardware repositories [4]-[6] already distribute hundreds of freely available accelerators designed by expert designers for various commonly used compute-intensive functions from various application domains such as signal processing, networking, multimedia, encryption, machine learning, *etc*. We propose a framework that enables software developers to utilize such libraries of hardware accelerators with minimal design effort by simply annotating regions of interest within their programs. The framework automatically searches for a functionally equivalent accelerator within a library of existing accelerators and, if one exists, integrates the accelerator into the application.

In summary, our contributions are as follows:

- We propose a new design approach that leverages existing hardware accelerators with minimal design effort on the behalf of software developers.
- We describe ACCLIB, a tool framework that realizes this proposal by automating the identification, matching, and integration of accelerators.
- We implement the ACCLIB framework and use it to successfully accelerate six benchmarks in just over one hour per application, demonstrating the viability of our proposal.

II. Related Work

In this section, we discuss previous efforts related to ACCLIB. The dominant approach to automate the design of accelerators is high-level synthesis, wherein a behavioral description in a high-level language such as C, C++, or SystemC is synthesized into an RTL implementation. HLS tools utilize a variety of advanced techniques to generate more efficient hardware, such as intelligent memory allocation, loop pipelining, speculation, and code motion [7]. However, these advanced hardware optimizations are not turn-key; HLS tools such as Catapult [8] and LegUp [9] require user input or directives in the form of pragmas or TCL scripts to guide optimization. Despite requiring the designer to be familiar with the synthesis process to create efficient hardware, it is not always possible to match the efficiency of hand-tuned RTL. The generated hardware is integrated into the system manually or using automatic hardware-software interface generation tools [10]. Increasingly, these steps (HLS and interface generation) are combined into an integrated framework [11].

The proposed ACCLIB framework takes a complementary approach. This approach leverages the existing body of highly optimized hardware accelerators and the power of formal verification to reduce design effort without sacrificing efficiency. ACCLIB also uses automatic interface generation and software instrumentation techniques

[+] Work done while the authors were at Purdue University

to streamline the accelerator integration process. Thus, ACCLIB offers a design flow that helps to facilitate the adoption of accelerator-based computing.

III. ACCLIB Framework Overview

In this section, we provide an overview of the ACCLIB framework, which is presented in Fig. 1.

ACCLIB requires two major inputs: a software application and an accelerator library. The software application must be annotated so that the functions that the developer wishes to accelerate are encapsulated in *acceleration targets*. The annotation of acceleration targets provides the functional interface that will be used to perform equivalence checking with the hardware accelerators. The accelerator library consists of handwritten, custom accelerators specified at the register-transfer level (RTL). The library additionally contains a Timing-accurate Interface File (TIF) for each accelerator. The TIF specifies the interface of an accelerator at a cycle and pin accurate level, as well as the functional interface of the accelerator. Specifying a TIF requires the same knowledge as, but significantly less design effort than, crafting a test bench for the accelerator.

ACCLIB consists of two major steps: equivalence checking and hardware/software interface generation. In the *equivalence checking* step, the acceleration targets and hardware accelerator library are first filtered for obvious mismatches and then compared for equivalence. In the *interface generation* step, ACCLIB generates a bus wrapper, resulting in a bus-specific accelerator. It also generates the code necessary to use the accelerator, including the instrumented application and the hardware abstraction layer. The ACCLIB framework can be customized for different target platforms by specifying platform-specific features such as the bus or NoC protocol and the system-level memory map.

IV. ACCLIB Details

In this section, we provide a more detailed description of the ACCLIB framework, including inputs, outputs, verification method, interface generation, and code generation.

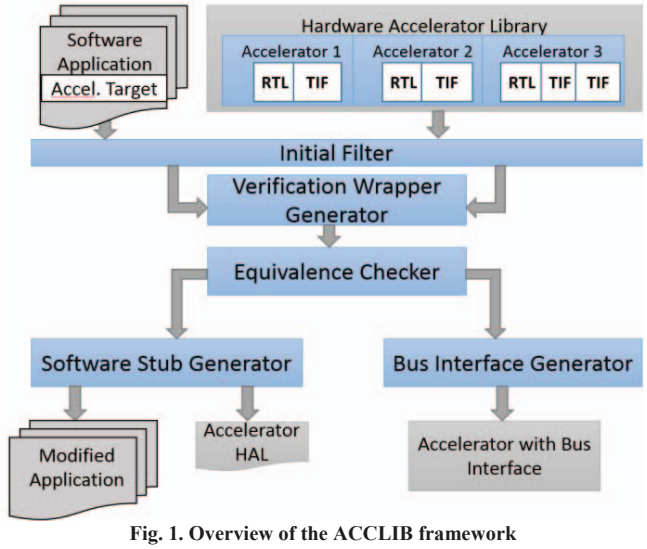

Fig. 1. Overview of the ACCLIB framework

A. Framework Inputs

1) Hardware Specification

ACCLIB assumes that hardware accelerators are described at the register-transfer level and are independent of a specific bus interface. In order to allow for equivalence checking as well as automatic bus interface generation, a description (referred to as the TIF) of the accelerator's native interface must be provided. We next describe the details of the TIF, show in Fig. 2, which consists of two parts: the interface section and the operation section. The interface section describes the functional interface (*FunctionalIF*) to the accelerator. This specifies the input data on which the accelerator will operate, as well as the output data that the accelerator produces. The interface section also describes the native data interface (*NativeDataIF*) and native control interface (*NativeControlIF*). These interfaces specify the names and bit widths of the signals that represent data and control inputs to the accelerator.

The operation section specifies, at a pin- and cycle-accurate level, how to operate a given accelerator to realize the desired functionality. It provides default values for native control signals and specifies cycle-accurate, atomic transactions. These transactions, along with statements such as `if` or `for`, can be used as building blocks to specify the operation of an accelerator, in a pseudo-C syntax.

In summary, the TIF serves two important functions:

1. It provides a high level, timing-independent interface useful in both equivalence checking and automatic hardware/software interfacing.
2. It provides a pin- and cycle-accurate mapping from the functional interface to the native interface.

2) Software Specification

In addition to a TIF for each hardware accelerator, ACCLIB requires that the target application be annotated to

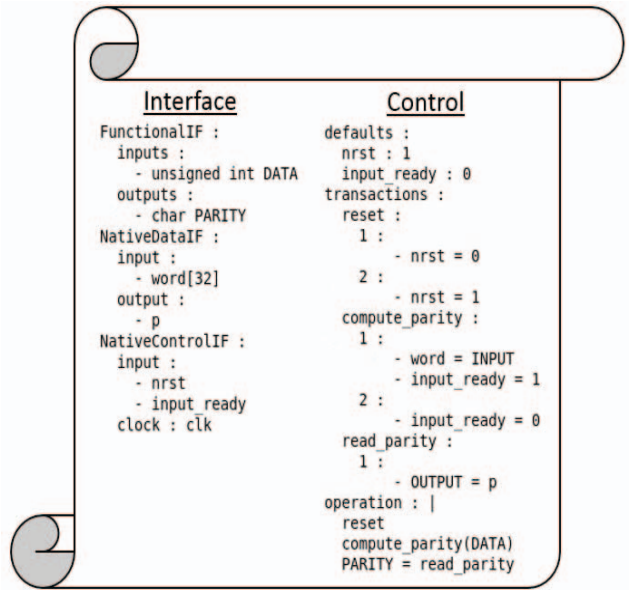

Fig. 2. Stylized example of a Timing Interface File (TIF)

indicate to the framework which regions of code are of interest for acceleration. These regions are demarcated using the `#pragma` keyword. The symbol `ACCLIB` indicates the namespace for the framework. The symbol `START_TARGET` is followed by a unique identifier used for that code section. The same identifier must be used after the `END_TARGET` symbol. Subsequent directives identify the inputs and outputs of the acceleration target.

There are a few constraints on program regions that can be declared as acceleration targets for the ACCLIB framework. The inputs and outputs of a function must consist of scalars and arrays of native C/C++[1] data types. Further, the target code must have no system calls, dynamic memory allocation, or side effects. While these restrictions may seem limiting, they are similar to those imposed by modern high-level synthesis tools, and many commonly accelerated computational kernels meet these criteria.

B. Equivalence Checking

In this subsection, we discuss the equivalence checking step of ACCLIB.

ACCLIB relies on being able to directly compare the behavior of software (acceleration targets specified in the program) and hardware implementations of accelerators across possibly many clock cycles. This results in a sequential logical equivalence checking (SLEC) problem.

SLEC is increasingly used in the industry, with offerings such as Calypto's SLEC [12], Mentor Graphics' Questa SLEC [13], and the open source tool `hw-cbmc` [14]. `hw-cbmc` is a Bounded Model Checking (BMC) based program that checks consistency between an ANSI-C program and a Verilog module. In BMC, the specification (the C program) and the implementation (the Verilog module) are temporally unrolled to a specified degree, obtaining a Boolean formula that is fed into a SAT solver. If the Boolean formula is found to be satisfiable, the specification and implementation are not equivalent.

1) Verification wrapper generation

The verification wrapper generator uses the TIF to generate a C program that emulates the cycle-accurate operation of the accelerator using data structures defined by the SLEC tool. At the end of this program, the original acceleration target is also executed with the same inputs as the accelerator. The outputs of the acceleration target and the hardware accelerator are then asserted to be equal. Logically, this creates a miter structure. This verification harness is passed to `hw-cbmc`, along with the C/C++ and Verilog codes and arguments specifying the correct unrolling factors. Since `hw-cbmc` must unroll sufficiently that the output can be obtained, an upper bound value must be used for the length of arrays that determine execution time (for example, the length of a string to encrypt). All other input values are free, *i.e.*, their values are determined by `hw-bwmc`.

`hw-cbmc` then unrolls both the hardware and software implementations, generating a SAT problem instance that attempts to determine if the assertion is ever false—that is, that the output produced by software is different from the output produced by the hardware. Proving the SAT problem unsatisfiable can be quite time consuming. Our experience suggests that for many commonly accelerated simple computational kernels, SAT problems are proven satisfiable (*i.e.*, proving the accelerator is not equivalent) or unsatisfiable in reasonable time (a little over an hour for all our benchmarks).

C. Hardware/Software Interfacing

1) Interface generation

After determining a suitable accelerator, if any, for a given software function, ACCLIB creates both the hardware wrapper necessary to encapsulate the accelerator using the desired bus protocol, as well as the software necessary to interact with the accelerator.

To generate the wrapper, each transaction specified in the TIF is represented in the HDL by a simple state machine that mimics the cycle-by-cycle behavior specified for that transaction. Each transaction is also assigned to a memory mapped register. Whenever a read or write operation is performed on a transaction's associated register, the transaction steps through its state machine. Throughout the execution of a transaction's state machine, the bus is held busy. This maintains atomic accelerator transactions, allowing for the cycle-by-cycle behavior of a transaction specification in the TIF to be reproduced. Currently, ACCLIB supports the Avalon Memory Mapped (Avalon-MM) [15] interface, but can easily be extended to other standards.

In summary, the ACCLIB framework automatically generates bus interface wrappers for accelerators using the provided TIF description.

2) Software generation

ACCLIB first generates a HAL for the accelerator to be integrated. This HAL provides mnemonics to be used in the modified application for all the memory mapped registers that belong to the accelerator and its wrapper. Before generating code, the functional interface of the accelerator is mapped to the corresponding inputs and outputs of the software application. Constructs such as `FOR`, `IF`, `ENDIF`, *etc.*, are converted to their equivalent C/C++ representation. Statements in the operation section that contain defined

Table 1. Benchmarks used to evaluate ACCLIB

Target Functions	
Function	**Domain**
Cyclic Redundancy Check	Error Detection
Parity	Error Detection
Cordic	Trigonometry
Finite Impulse Response Filter	Digital Signal Processing
Dot Product	Machine Learning
Average Pool	Machine Learning

[1] Due to limitations of the underlying formal verification tool, ACCLIB currently supports applications written in C or C++

transactions are translated into reads or writes to the corresponding memory mapped register, identified by its mnemonic.

V. Experimental Results

To benchmark the targeted applications, the Altera DE2115 board, with a Cyclone IV FPGA, was used. A simple SoC consisting of an Intel Nios II/f processor, SRAM, and a JTAG debug module, was used as the baseline hardware. Each accelerator and its auto-generated Avalon wrapper were integrated into the baseline system using Qsys (now known as Platform Designer) [16]. This process took less than five minutes for each accelerator. The runtime of the ACCLIB framework was benchmarked on a server with four Intel Xeon E5-2680 processors running at 2.7 GHz with 128 GB of RAM.

A. Benchmarks

To demonstrate the ACCLIB framework, six different benchmarks with appropriate accelerators across a variety of application domains, summarized in Table 1, were selected. Each benchmark contained at least one annotated acceleration target. In the case of the parity benchmark, both a parity32 and a parity64 function were included. In this way, we can demonstrate the ability to accelerate multiple functions with a single accelerator.

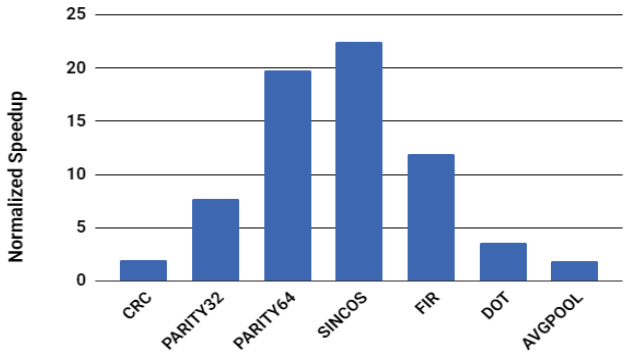

Fig. 3. Speedups achieved through automatic acceleration with ACCLIB

Each of the benchmarks were automatically accelerated using the ACCLIB framework. The performance improvement of each function as well as the execution time of the ACCLIB framework were measured. Fig. 2 presents the speedup achieved by each of the programs automatically accelerated by ACCLIB. The speedups range from 1.8x-22x as compared to the pure software implementations.

Table 2 lists the number of variables and clauses in the

Table 2. Complexity of the SAT instances for equivalence checking

Accelerator	Variables	Clauses
CRC	520,366	1,031,538
PARITY32	6,164	513,304
PARITY64	8,386	522,580
SINCOS	1,701,712	5,422,983
FIR	179,650	824,824
DOT	60,502	239,201
AVGPOOL	15,340	36,546

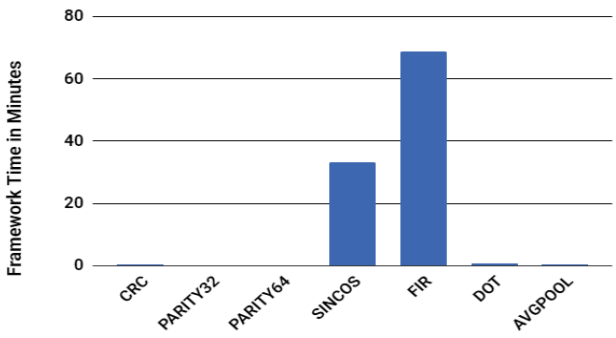

Fig. 4. Runtime of the ACCLIB framework

SAT instance generated by the equivalence checking step. Fig. 3 plots the run time of ACCLIB, which is dominated by the time it takes to solve the SAT instance generated by `hw-cbmc`; the rest of the framework takes between 1-3 seconds to complete, not including the time necessary to integrate using Qsys.

Our results suggest that for small to medium sized accelerators, the ACCLIB framework can accelerate software applications in reasonable time, with very little design effort.

VI. Conclusion

In conclusion, in this paper we have presented ACCLIB, a framework for enabling the efficient integration of existing hardware accelerators into existing software applications. We have shown that for small to medium designs, ACCLIB can automatically identify a suitable accelerator and generate the requisite hardware and software to integrate the accelerator.

References

[1] "NVIDIA Tegra K1," [Online]. Available: http://www.nvidia.com/object/tegra-k1-processor.html.

[2] "Snapdragon," [Online]. Available: https://www.qualcomm.com/products/snapdragon.

[3] "TI OMAP," [Online]. Available: www.ti.com/omap.

[4] OpenCores: Open Source Hardware Cores. http://opencores.org/.

[5] Design and Reuse. http://www.design-reuse.com/.

[6] Synopsys DesignWare IP. http://www.synopsys.com/IP.

[7] R. Nane *et al*. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Trans. CAD*, 2016.

[8] "Catapult High Level Synthesis," [Online]. Available: www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls.

[9] A. Canis *et al*. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proc. FPGA*, 2011.

[10] ChangRyul Yun *et al.* Automatic interface synthesis based on the classification of interface protocols of IPs. In *Proc. ASP-DAC*., 2008.

[11] C. Pilato *et al*. On the automatic integration of hardware accelerators into FPGA-based embedded systems. In *Proc. FPL*, 2012.

[12] "HLS Verification," [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/hls-verification.

[13] "Questa Sequential Logic Equivalence Check," [Online]. Available: https://www.mentor.com/products/fv/questa-slec.

[14] E. Clarke *et al.* Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In *Proc. DAC*, 2003.

[15] "Avalon Interface Specifications," 2017.

[16] "Platform Designer," [Online]. Available: www.altera.com/products/design-software/fpga-design/quartus-prime/