Early-stage Automated Accelerator Identification Tool for Embedded Systems with Limited Area

Parnian Mokri
parnian.mokri@tufts.edu
Tufts University
Medford, MA

Mark Hempstead
mark.hempstead@tufts.edu
Tufts University
Medford, MA

ABSTRACT

Designers are turning toward hardware specialization through the use of application-specific accelerators to provide energy-efficiency and performance. We propose an early detection methodology to identify computationally similar and synthesize-able kernels that are used to build Shared Accelerators (SAs). SAs are specialized hardware accelerators that execute very different software kernels but share the common hardware functions between them. SAs can provide increased coverage if both data flow and control flow similarities between - seemingly very different- workloads are detected.

This work leverages abstract syntax trees (ASTs) generated from clang in LLVM to discover similar kernels among workloads. ASTs provide a level of abstraction well suited to detect commonalities between kernels. Our methodology, ReconfAST, transforms the AST into a new clustered AST (CAST) representation that further removes unneeded nodes and uses a regular expression to match common node configurations. The approach is validated using MachSuite, a HLS-ified benchmark suite designed for accelerators in C.

KEYWORDS

Application Specific Hardware, High-level Synthesis, Shared Accelerators, Abstract Syntax Tree (AST)

ACM Reference Format:
Parnian Mokri and Mark Hempstead. 2020. Early-stage Automated Accelerator Identification Tool for Embedded Systems with Limited Area . In *IEEE/ACM International Conference on Computer-Aided Design(ICCAD '20), November 2–5, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3400302.3415733

1 INTRODUCTION

Dedicated accelerators (DAs) target only one application and are used extensively across a variety of applications such as Internet of Things, wearables, and implantable medical devices due to their high performance and low power characteristics [**?**]. Industry and academia have begun research and development into hardware specialization to address the challenge of *Dark Silicon*, or the limits on parallelism enforced by constraints on chip power density [**?**].

ICCAD '20, November 2–5, 2020, Virtual Event,USA

ACM ISBN 978-1-4503-8026-3/20/11…$15.00
https://doi.org/10.1145/3400302.3415733

Current design methodologies fail to assist designers in maximizing overall system performance and workload coverage, especially under a specific area constraints. This paper proposes the use of Shared Accelerators (SAs) to find similarities between seemingly different software kernels from different domains that do not share any obvious commonality, such as libraries and common functions. The functionality of multiple kernels is combined into a single accelerator, which increases workload coverage. The process of designing SAs presents three main challenges: 1) identifying similarities in software kernels that can use shared hardware; 2) implementing the shared kernel in hardware; and 3) doing so as early in the design process as possible.

Existing methods use either dynamic traces (profilers) or analyze register transfer level (RTL) implementations (e.g. Partial Reconfiguration) to find these similarities. However, both of these approaches require in-depth knowledge of RTL and time-consuming design processes. While data-flow graphs have been used for other EDA tasks, we find that they are not suited for finding hardware similar kernels and lead to false negatives. Xilinx Vivado, an industry tool, is capable of reconfiguring FPGAs at runtime [**?**]. However, finding similarities between applications is the designer's responsibility. There has been work on leveraging functional programming characteristics to profile workloads [**?**]. With some extra processing, it is possible to find similarities between applications in this method; however, many applications are not written in functional programming languages. Further, it is difficult to leverage prior work that detects similarities between workloads at binary level because mapping binary/assembly code to RTL design, or back to the high-level source code, is an NP-problem [**?**].

Designing and implementing SAs can also be an obstacle. High-Level Synthesis (HLS) tools are introduced to ease the process of designing accelerators, but they do not help the designer to identify similarities between programs. Other conventional approaches involve modeling workloads as directed acyclic graphs (DAGs), which can be used for scheduling and logic synthesis [**?**]. Unfortunately, all of the works above suffer from long execution time and the cost of writing RTL.

We leverage other work in the literature that detects similarities in hardware or software kenerels. These works range in scope from gate-level to compiler-level solutions. Rao and Kurdahi explored the clustering of a digital circuit by extracting regularities both at the gate and system-level RTL[**?**]. In contrast, our methodology works with HLS tools and source code written in C, C++, or OpenCL. Hassoun and McCreary looked at gates and RTL similarities to find structural regularities in circuits to allow minimization of synthesis, optimization, and layout efforts [**?**]. We present a methodology to do the same but at source-code level for High-Level Synthesis

tools. Moreano et al. use Data Flow Graphs (DFGs) to find common data paths; however, based on our experiments and literature in the compiler community, Abstract Syntax Trees are much more efficient to use to find similarities mainly because DFGs are based on assembly/instruction representations (IRs) [**?**]. MH: check citation, this points to the wrong paper I think Also, the systematic classification of false positives and negatives for DFGs is difficult and relies on heuristics [**? ?**].

Dreweke et al. find duplicate assembly code segments to reduce the code size, which is useful for embedded processors [**?**]. However, source code has many lines that don't translate into hardware moreover, traditional clones don"t support hardware similarities for example, add and sub run on the same hardware.

We propose *ReconfAST*, a methodology which expresses applications by an annotated graph, a Clustered Abstract Syntax Tree (CAST), and then identifies similarities among applications using a sub-tree isomorphism algorithm. In addition, ReconfAST complements existing FPGA tools that enable partial reconfiguration by aiding designers so that they will not have to manually identify which kernels to share and reconfigure. Figure 3 provides an overview of *ReconfAST* and Sections 2.1 and 3 explain the methodology in detail. Finding similarities between kernels using Abstract Syntax Trees (AST) is computationally less expensive than using DFG and CDFGs**??**. Missing reference! As mentioned above, the EDA community has used tree representations of control and computation behavior of a workload since the late 90s [**?**]. `ReconfAST` expands on their work and provides a method to mark the source code and use HLS to design SAs.

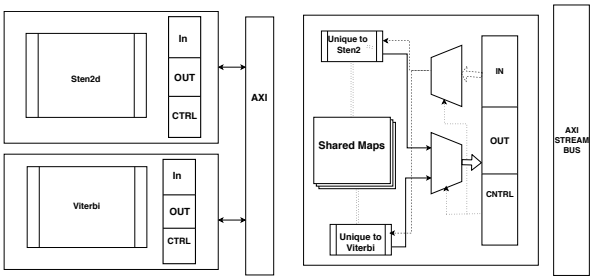
List of contributions:

(1) Introduction of **shared accelerators** (SAs).
(2) **ReconfAST**, an early-stage tool for extracting SA candidates by detaching common kernels using AST representations of source-code.
(3) Introducing Clustered-AST (**CAST**): A transformed AST representation that removes unnecessary syntax, summarizes common patterns, and aids in the efficient comparison of hardware similar workloads.
(4) Study of ReconfAST on MachSuite (**A HLSfied benchmark suite in c**) demonstrating the potential of SAs to increase coverage.
(5) FPGA implementations of example SAs and analysis of hardware costs. Classification of workload patterns that result in good and poor performing SA implementations.

2 WHAT NEEDS TO BE ACCELERATED AND HOW TO FIND IT?

2.1 Introducing Shared Accelerators

We introduce *Shared Accelerators*, which resemble the structure of an ASIC implementation of one software kernel but can accelerate two or more distinctly different kernels. Figure 1 shows a simplified example of a system with accelerators for two MachSuite benchmarks, *Stencil2D* and *Viterbi*. Instead of building a separate dedicated-accelerator for each kernel, a single shared accelerator includes hardware for both Kernels. Common hardware, in this case a loop with an array multiplication and accumulation, is identified from application source code, using our automated ReconfAST methodology.



(a) Dedicated Accelerators (DAs) **(b) Shared Accelerator (SA)**

Figure 1: Improving workload coverage and area efficiency in many-accelerator systems with Shared Accelerators. Instead of a system with an accelerator dedicated for each kernel, shared accelerators can execute multiple kernels by including all of the hardware for both kernels. Common hardware kernels are automatically discovered and shared, reducing area costs.

Covering multiple software kernels with the same piece of hardware is not new; there are accelerators in the image processing and signal processing domains that are parameterizable and capable of executing multiple standards. However, this methodology allows designers to identify similarities between very different workloads and application domains.

Shared Accelerators can significantly reduce the usage of on-chip resources. By reducing FPGA resources or ASIC area system designers will either be able to reduce system cost or add more accelerators, or higher performance accelerator implementations, to increase workload coverage and performance within the same area budget. There are system architecture, software and runtime concerns with merging multiple accelerators; these include the possibility of contention and blocking and also the need to virtualize accelerators called from separate contexts which could potentially increase communication and reduce application performance. In this work, we focus on the identification of shared kernels and leave the system-level concerns to future work.

2.2 Why Use Abstract Syntax Trees to Find Shared Kernels

Previous work has shown an Abstract Syntax Tree (AST) is a better choice to find similarities between kernels and has been used widely in the plagiarism detection community and design automation, EDA community[**?**]. Mapping from a node in an AST tree representation to the source code[**?**] is deterministic unlike other representations. In this work, we investigate how to use ASTs to detect similar hardware-patterns and overcome the limitations of common LLVM AST representations.

Designing an accelerator rich architecture is time-consuming; the designer must deliver performance under physical constraints (e.g., area, power, energy). High-level synthesis (HLS) is used to reduce the design time of individual kernels. Also, newer software profiling tools can further reduce the complexity of the accelerator selection problem by extracting applications' data dependencies and estimate the hardware implications of possible loop optimizations from the source code. These profilers are pessimistic and based on either

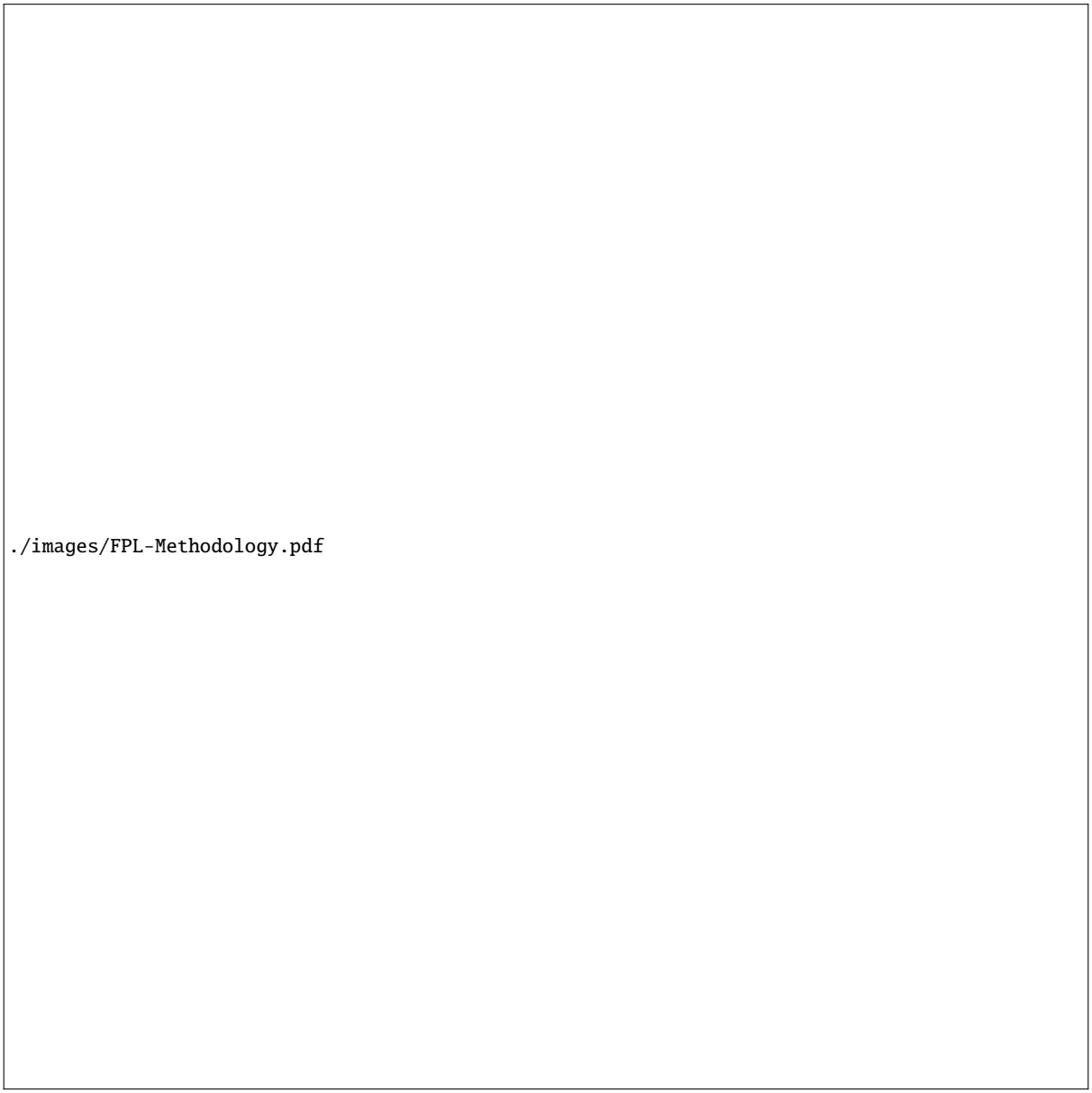


Figure 3: Block diagram of the ReconfAST methodology. Static ASTs for each workload are generated by the compiler. Before comparing workloads, a pre-processing tool transforms the AST into a Clustered AST (CAST) by removing nodes not amenable to hardware implementation and finding common AST patterns using the CAST library. Then candidate regions for shared hardware acceleration between workloads are identified using sub-graph isomorphism (VF2 algorithm).

static analysis [**?**] or dynamic analysis using dynamic data flow graphs (DDFGs) [**?**]. Although these methodologies are good for accelerator selection, they are very time-consuming to find similarities between applications. In this paper, we propose an approach that uses static abstract syntax trees (ASTs) generated from the compiler. ASTs abstract structure expresses the underlying computation clearly and other information such as data-dependency can be added to the trees more efficiently because of the structure. Clang AST supports OpenCL, c, c++ and many functional programming languages such as Haskel and Scala. Therefore, our methodology is very adaptable to early-stage design exploration. Furthermore, finding similarities between `trees` is much faster and less computationally expensive than between more generalized graphs.

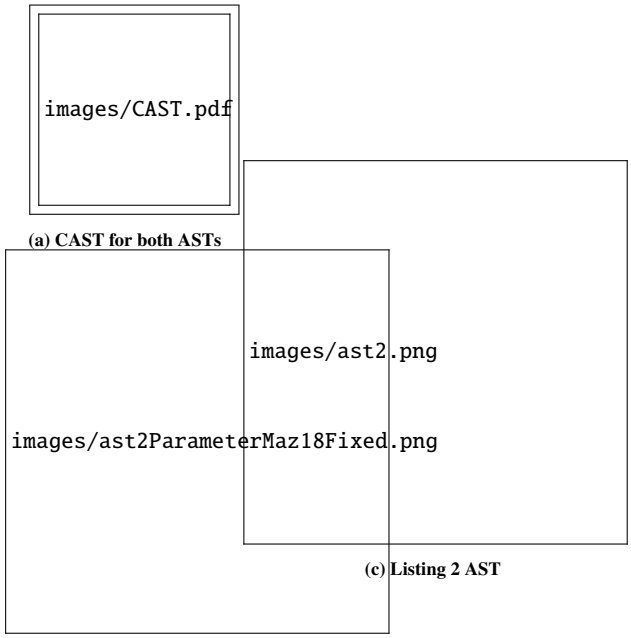
One problem with ASTs is that they can lead to verbose representations, that while, representing the same functionality will have very different AST representations. As an example we provide Listings 1 and 2 as example programs that generate very different ASTs but would result in similar hardware implementations using HLS tools. The code snippets correspond with Figure 4 the transitional control nodes are illustrated as round yellow nodes, loops that are a counter in hardware are orange rectangle, and binary computations are octagons. A graphical representation of these two ASTs can be

found in Figure 4. Clearly if an automated tool compared the two raw ASTs it would say that they are dissimilar.

2.3 What are Clustered Abstract Syntax Trees (CASTs)?

A basic AST does not provide all the information needed to identify hardware-similar software kernels in an efficient and accurate manner. Also at times ASTs, with their verbosity and syntax specific nodes, hide potential matches. We introduce **CAST** a Tree transformation method based on clang's Abstract Syntax Tree (AST). A CAST representation is designed to be compact, suscinct and easy to compare. For example, the two ASTs that represent Listings 1 and 2 are transformed into the same CAST, Figure 4 (a).

Literature from the compiler community on the concept of cloning suggests using Abstract Syntax Trees, ASTs, to detect similarities between source codes [**?**]. To design an accelerator, ReconfAST needs to find exact, or near exact, matches between arbitrary fragments of program source code. Since detection is in terms of the program structure, clones can be factored out of the source using conventional transformational methods.



(a) CAST for both ASTs

(c) Listing 2 AST

(b) Listing 1 AST

Figure 4: The same CAST can represent two different ASTs. Two snippet of code with similar functionality but have different syntax as seen in Listings 1 and 2 . Octagons are computational related nodes, rectangles are for loop and control class. The round nodes denote nodes that won't translate into hardware such as variable declaration and statements(wild-card patterns) which is about 52% of nodes in this AST and are denoted by a shadow on the nodes.

Clang ASTs are produced in two formats a simple tree with minimum information and a text file with a complete list of attributes. The ReconfAST methodology uses four steps to transform ASTs and create a data-structure that is compact and suitable for finding hardware-similar structures using graph-isomorphism. These steps are described below.

Step 1: Parsing and annotating each AST node with the following information:

Unique identifier: A unique number is assigned to each node and will be the vertices of the tree. Tree edges express relations between these identifiers. Unique identifier is used to identify matches and also aid in tying matches back to the source-code.

Node attribute- node's operational class: AST nodes have four types. They are either computation based (operation), control based (loops, function declaration, return statement, conditional statements), memory nodes that signify array and variable declaration, or nodes that serve to show the scope or parenthesis statements. We modify these classes based on Table 1 to make graph-isomorphism run faster.

Node attribute- line number in source-code: In order to calculate dynamic coverage and build a SA with HLS, we must map static nodes to the original source code. We do this by adding source-code line numbers as an attribute of each node, showing where in the source code a specific node was generated.

Node attribute- operand identifier: To find potential data dependencies, we add the name of operands as an attribute to the AST nodes.

Step 2: Pre-processing and Removal of White-Space/wild-cards

As explained in Section 2.1, many of the AST nodes do not translate directly to hardware implementation. Our pre-processing tool identifies these white-space nodes using regular expressions and labels them as *wild-card* and then removes them from the AST while maintaining connectivity of the tree. This stage is done based on our extended experiments with HLS tools. For example, we showed that all declarations become signals in HLS, implicit or explicit expression of variables, which are due to coding, results in the same hardware. Parenthesis statements signify the scope of a function or block of code which can be detected by any tree traversal and therefore these nodes add no extra information to our methodology.

Listing 1: Example of similar functionality but different ASTs. A simple for loop with an inline sum. The AST is shown in Figure 4b

```
1   int a;
2   int b=0,c=0;
3   for(a = 10; a < 20)
4   {
5         a++;
6         c = a+b ;
7
8     }
9
10    return 0;
```

Listing 2: The same functionality with a loop and function call The AST is in Figure 4c

```
1   int main () {
2    int a;
3    int b=0,c=0;
4    for(a = 10;...
5        a < 20;...
6        ++a)
7     {
8      c =
9        callSum(a,b);
10    }
11   return 0;
12  }
```

Step 3: Building CAST by Categorizing and Clustering.

CAST nodes classify a program's structure—through the use of high-level control abstractions—which in turn improves the process

Opcode Type	Statement	Reg Expression	CAST Node Name

Loop	For(i=0;i<n;i++)
	For(i=0;i<n;) i=add(i,1)
	while
		For,uOp:opr:val,BinOp:opr:int,bin:Opr,val,W*
		For,uOp:opr:val,BinOp:opr:int,bin:opr:val,W*
		while,opr:w,binOp:opr:val,W8
			Loop:For:I
			Loop:For:ExplicitAdder
			Loop:While

Operation	Binary Unary	BinW*:int,W*:int, UOp:W*:int	BinOp:<Symbol><oprnds><oprnds> UnaryOp:<Symbol><oprnds>
Control	Function Call ret ret if if	FuncCall:W*:returnStmnt with single Literal with Expression (a function or expression) without else statement(not implicit) with else statement(implicit)	Branch:Call:Noret, Bt Branch:Ret:singleLit Branch:Ret:Exp Branch:if:NoElse Branch:If:Else
WildCard	declarations, initialization	decl,Implicit, explicit, parantheisStmnt,	wildcard (W) node which will be removed

Table 1: AST library and CAST library CAST nodes contain the type of control structure and attributes that affect hardware implementation, such as operation type and the datatype. The methodology uses pattern matching to match nodes in the library because many AST patterns can represent the same CAST node.

of finding similar kernels among applications. ASTs are very sensitive to syntax and programmer's coding style; these styles need to be identified and removed when it comes to finding similarities between kernels through graph isomorphism. We match collections to a single CAST library node using regular expression matching on a depth-first traversal. Each depth-first traverse represents one instruction, therefore our methodology is flexible enough to identify similar hardware structures with only syntax differences and reduces false-negatives by finding similarities in cases where two trees are similar but have flipped subtrees.

Table 1 formally presents the CAST library. To minimize the effect of coding style on CAST, we use pattern recognition to classify nodes. We use regular expressions to find specific patterns such as operand type, function call, and loop statements. Not all patterns are included, just the AST patterns which might have slightly different AST sub-trees but result in the same hardware representation. Our transform methodology identifies all of these patterns in the library and replaces them with the appropriate CAST node. The following paragraphs describe each of the major types of CAST nodes and their features.

Loops: Whether the programmer uses a *while* or *for* loop, the hardware translation of the loop logic will be the same, essentially an adder and a comparator. The practice of discerning workload similarity based on loops has appeared in many studies [**?**]. Loops are chosen as a class in our CAST library because normally the majority of dynamic time spent in workloads is inside loops and many loops perform similar functions betwen kernels. Our CAST library also preserves the structure of nested loops.

Compute Operations : In the CAST library, we only have two types of operations: binary and unary. It is possible that one operation is repeated enough times within all kernels waiting to be accelerated that implementing that operation would speed up the entire workload. The specific operation (ADD, MULT, LOGICAL) is not typically in the AST and is added in our methodology. The datatype of an operation (e.g int or float) is important for some CAST nodes and their hardware implementations. That is why some of the nodes in Table 1 are given a unique identifier for each operand datatype. In an AST the datatype of an operation is not on the node itself, but in fact indicated by child nodes in the tree. Thus, the methodology

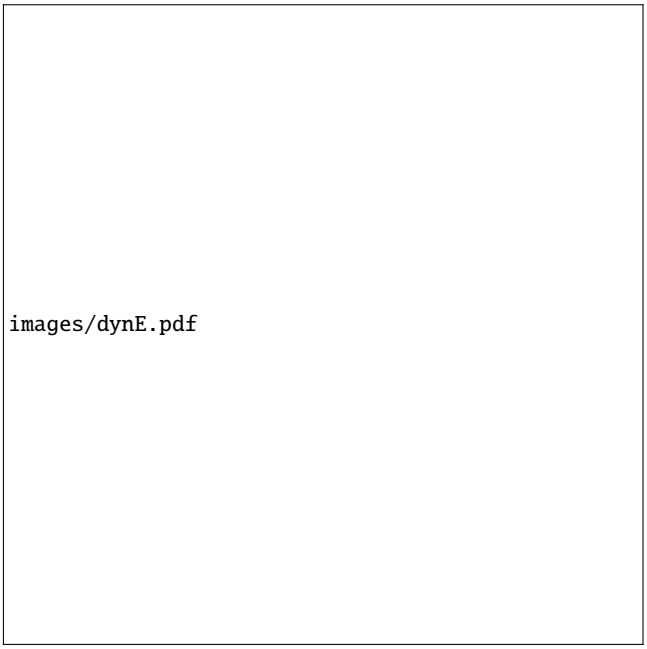


Figure 5: Maximum Dynamic Coverage (percentage of total execution time) measured of the matching (isomorphic) sub-graphs found between the CASTs of each workload. The dynamic execution time was measured using the Valgrind suit's callgrind tool. The values reported are respective to workload on the row labels total execution time, hence for the same order the results are asymmetry in the table.

matches these patterns into a single CAST node. If operand of binary operation is an array, HLS will add extra control (LUT and FF). Also, if the type of operand is a float, more resources will be used than when operands are integers. Since `ReconfAST` aims to find functionally similar hardware across kernels, in this step we differentiate between these patterns.

Control: Control nodes are a separate category in the CAST library. These nodes should be generalized to reflect hardware implementations and remove false negatives. For example, an identical series of instructions under an *if* statement in *workload1* and an *else* statement in *workload2* should match when the CAST trees are

compared. Function calls and returns are in the control category in the CAST library. These nodes will be removed later, as we show in section 3.2. With future communication modeling in mind, function calls that return values, such as integer or float, are differentiated from void functions.

Wild-cards: Parenthesis, declaration, and other statements that would show the scope of instructions/loops/functions or shows whether a variable has been defined as implicit or explicit.

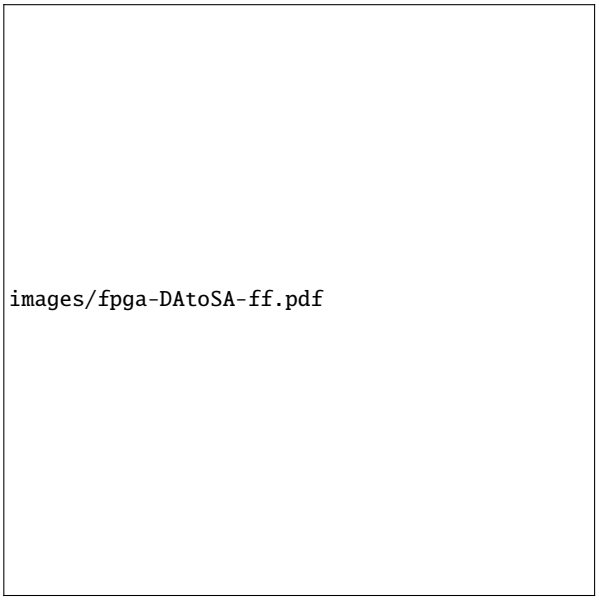
Step 4:

Data-dependency edges are annotated differently and transfer from ASTs to CAST i.e. the dashed edge in Figure 4a. We exclude DD edges from isomorphism to speed up the preliminary graph-isomorphism computation [**?**]. We conducted a study of workload coverage using the benchmarks from Machsuite. The results are found in Figure 5. We see high coverage for a range of very different workloads.

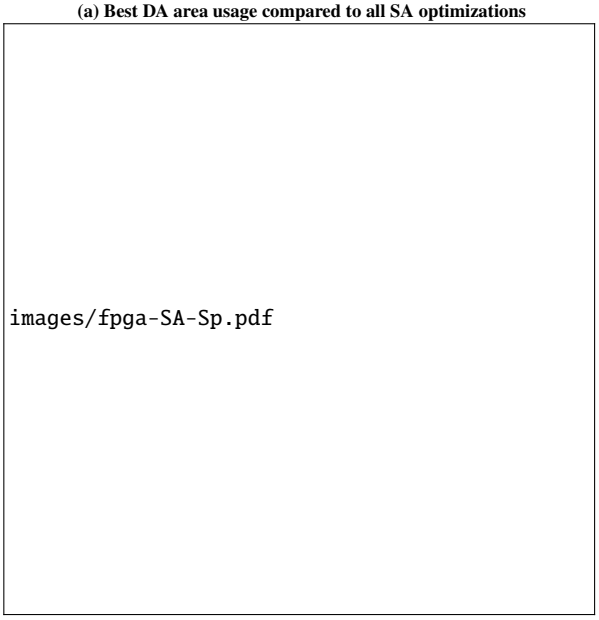
When choosing which accelerators to build, the isomorphic subtrees are only accepted if they're considered hot-code. In Figure 5 Dynamic converge is calculated by identifying the kernels that match between pairs of workloads and measuring how often that shared kernel is called running the workload. Dynamic execution time is not symetric between pairs of workloads. For example if one kernel matches in workload 1 but it called a million times vs. once in workload 2 then the kernel will contribute more to the dynamic coverage of workload 1. Our tool accepts isomorphic subtrees that have above 15% dynamic coverage.

3 IMPLEMENTATION

ReconfAST is built on the front-end of the LLVM-clang suite. Clang is used to generate the ASTs. All the workloads in our paper are written in c/c++ but the tool can support OpenCl and some functional languages. Our tool transforms ASTs into the CAST representations using python transformation scripts. The CAST of each workload is fed into a subgraph isomorphism, the VF2 library. VF2 (or other algorithms for graph isomorphism) has not been, to the best of our knowledge, ever applied to ASTs for use in high-level HLS hardware identification. We then validate the methodology by measuring the dynamic coverage using Valgrind/Callgrind and then analyze the hardware with Vivado HLS. We use Clang version 3.7.0 for static analysis, Valgrind-3.10.1 for dynamic analysis, and vivado_HLS 2019.2 for our evaluation. The VF2 algorithm, implemented in python, is used to identify similarities among CASTs [**?**]. To evaluate our methodology, we use the Machsuite benchmark suite for accelerator-based applications [**?**]. These applications range from signal processing, basic math, and linear algebra. All workloads for Machsuite have been written in HLS-compatible C code [**?**];namely, they follow suggested syntax and structure in the Xilinx HLS manual. Additionally, there are no in-line functions, and the size of the arrays are fixed. We use HLS to implement kernels. And by applying HLS optimizations we design about 7000 accelerators. Although we implemented all these kernels on a zynq board, we have also designed them on Vertex 7 and ASIC using appropriate tools. We chose the best DA (lowest latency and smallest number of DSPs) as the baseline. Nothing is manual in this work, everything is scripted.



(a) Best DA area usage compared to all SA optimizations



(b) Best DA speedup compared to all SA optimizations

Figure 6: Speedup of SA over DA with different optimizations and the area overhead that it introduces to the system. we normalize all SAs to the best case of DA design ReconfAST finds all the possible SAs, with different implementations of SAs, based on the size of shared part, how much of original kernel the shared parts cover, the data dependencies between the shared part and the unique parts of the SAs. Designers must analyze the range of potential implementations and the system resource costs.

3.1 Choosing Acceleration Candidates

In this project we automatically find all the possible matches between two workloads between two domains however not all maps result in speed up. We came up with some manual thresholds based

on our observations. These configuration can change by designers and doesn't affect the methodology itself. **CAST Coverage** We implemented maps with different coverage percentages. In our experiments none of the maps with less than 15% was efficient. Though, type of nodes was critical in our selection too. Bear in mind that each CAST node can represent much larger AST sub-trees and can contain a combination of operations and their operands or a loop and an operation and its operands. **Dynamic Coverage** We used callgrind from valgrind to separately estimated dynamic time of all possible maps between kernels. Each kernel is matched to others and has at least two maps. We show the dynamic time of the largest map between two workloads in 5. Loosely based on Amdahl's law, we only implemented maps that have more than 50% of dynamic time.

3.2 Building Workloads' CASTs

Following steps 1 to 4 we express all workloads using our graph transformation algorithm. Our primary experiments showed that workloads spend most of their time in user-defined functions and standard libraries. Therefore, in our implementation, we generate ASTs for all libraries and look for similarities in both user code and shared libraries.

Also, if the subgraph is too small—smaller than an operation and its operand subgraph in our case this measures to two CAST nodes—to design an accelerator for, the subgraph will not be considered. Considering the overhead of using the SAs, the time it takes to relay the control to SA and data movement to and from the SA, any smaller subgraph would be too fine-grained to design in hardware.

3.3 Maps: Isomorphic Subgraphs/Subtrees Between a Pair of Workloads

Our methodology finds statically similar software kernels in pairs of workloads by evaluating if two subgraphs are isomorphic. In this work, we use subgraph and subtrees interchangeably, since the VF2 algorithm is a subgraph-isomorphic algorithm but checks for the shape of the graph and detects trees [**?**] . ReconfAST only expresses kernels with trees, which reduces the complexity of the comparison to *(O(nm))*, where `n` is the total number of nodes in Graph 1 and `m` is total node in Graph 2 [**?**].

To design shared accelerators, we utilize all the information from ASTs structure and apply the VF2 algorithm only on depth-first subtrees with leaves in a recursive manner starting from the root of the CAST trees. The inputs to VF2 are the intermediate states, that track whether subgraph isomorphism has been detected between two subgraphs. will stop at the largest isomorphic subtree. This is a computationally effective approach for finding similarities between two workloads. These isomorphic subgraphs, referred to as *maps*, are the candidates for shared hardware structures in a SA.

To judge the efficacy of Shared Accelerator candidates we need to estimate the shared subtree's fraction of total execution-time. We flagged the beginning and the end of each of the shared maps in the workload's source code and fed them through Callgrind and Valgrind to provide an estimate of execution-time of the maps. ReconfAST finds 1.) large maps responsible for big fraction of dynamic execution time and 2.) smaller maps that are repeated across many workloads and contribute to system-wide execution-time.

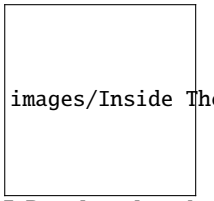


Figure 7: Data dependency between

3.4 Pruning Maps with Data Dependencies

This is the last step for pruning the maps and selecting an efficient SA. The characteristics of hardware synthesized with HLS is highly influenced by different compiler optimizations and coding styles. In addition, some patterns in the code translate to fixed hardware modules; for example, a loop often translates to a counter no matter what coding style is used. Each line of source-code translates to a depth-first-traverse sub-tree in ASTs. Using this structure our post processing script finds boundaries of functions and loops, then uses regular expression to find instances of data dependencies, and adds a different class of edge to the AST.

In our study we realized there are three main classes for data-dependency. When there is an edge from outside to the map and when the data dependency exists between the control structure and the computation inside the map, the common compiler optimizations will not be effective. However, if the data dependency edge is an output from the map to the outside it doesn't cause arbitrary behavior.

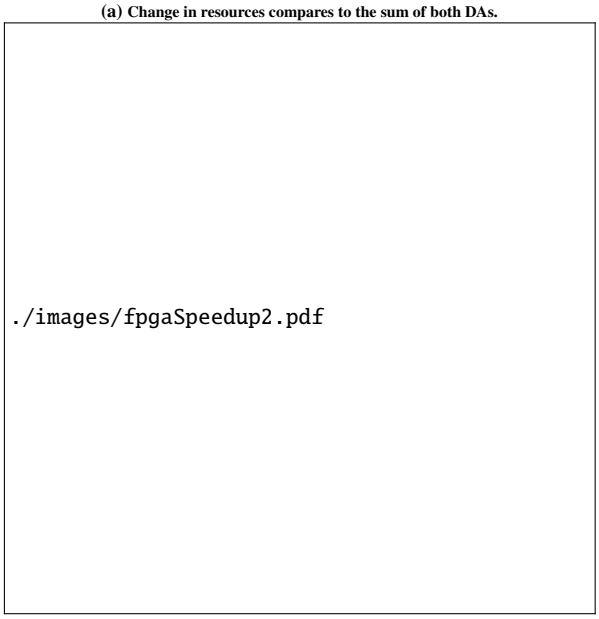
4 RESULTS AND EVALUATION

SAs in this project are implemented as blocking SAs. Once one workload is running the other is blocked. We envision a virtualize system, where multiple instance of same accelerators exist. Which will be the subject of future work. Designers must analyze the range of potential implementations and the system resource costs. In this section we focus on patterns that cause the SA to have better latency compared to their DA when no HLS optimization is applied to them. We analyze the patterns based on the size of shared part, how much of original kernel the shared parts cover, and the data dependencies between the shared part and the unique parts of the SAs. We evaluated the ReconfAST methodology to show, first, that it can be used to detect Shared Accelerators that cover a significant fraction of the workload. We presented these results already in Figure 5.

Next, we evaluate the hardware implementations of Shared Accelerators. Using HLS—vivadoHLS 2019— for FPGA, we show that the Shared Accelerators built from the matched isomorphic subgraphs reducing area usage in FPGA implementations. We discovered that some Shared Accelerator implementations are less efficient, and we analyze the software patterns that cause inefficient hardware implementations. We compared each workload in Machsuite with all other workloads in the suite using the *ReconfAST* methodology. The tool finds the largest isomorphic sub-trees between workloads, which is the way to ensure the best coverage between workloads with little hardware overhead.

We have designed SAs using ReconfAST; we applied 15 different optimizations to both SAs and DAs, then we chose the best DA (highest speedup) and normalize SAs to it. There are cases with

(a) Change in resources compares to the sum of both DAs.



(b) Speedup of each kernel is compared to the corresponding DA.

Figure 8: Examples of good SAs. An SA for the two kernels is implemented using HLS. Change in area utilization normalized to the cost of the sum of the DAs; Speedup is calculated with respect to the Dedicated Accelerator.

same speedup but multiple area cost which we chose the one with least FF; more than 10000 cases in total were considered. Figure 6 shows a selection of these designs when the resources are normalized to the best DA implementation. In most cases, the DAs had better speedup than the SAs. However, building DAs is more expensive, in most of these cases two DAs cost more resources than the SA that covers both workloads.

We noticed that designing an `efficient shared-accelerator` is possible when 1.) CAST subtrees had 2 or more nodes; 2.) CAST coverage was **continuous** and not result of multiple instances smaller subtree; 3.) The dynamic coverage of the isomorphic-subtree was more than 50% based on Figure 5.

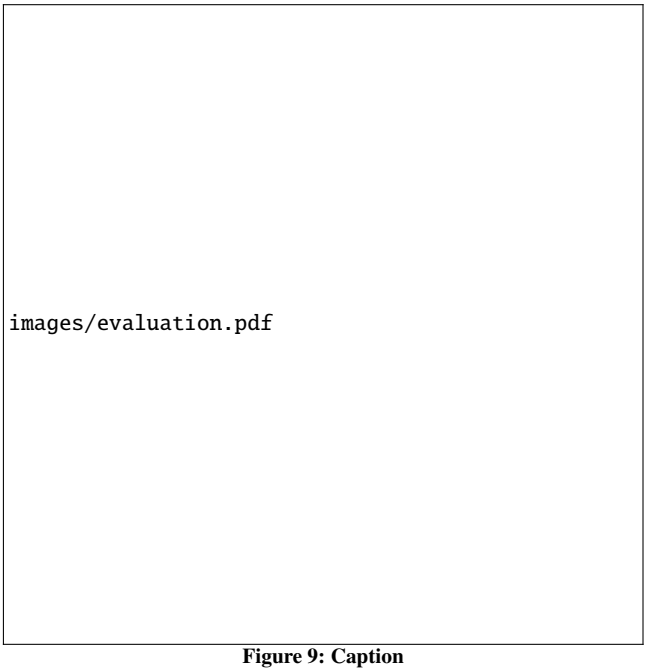


Figure 9: Caption

4.1 Efficient SAs in FPGAs

FPGAs are increasingly used to provide hardware acceleration for a range of applications from the data center to embedded systems. With on-chip SoC interconnect as well as an instruction interface with the ARM ISA, Xilinx zynq FPGAs are an excellent platform to evaluate both **loosely-coupled** and **closely-coupled** accelerators. The point of this evaluation is to show the hardware costs of SAs compared to DAs and speedup of SAs compared to dedicated accelerators.

We analyze examples of good Shared Accelerators discovered from Machsuite and present them in Figure 8. `Efficient SAs`, in our definition, are shared accelerators that either reduce resource requirements with comparable speedup to the DAs. Figure 8 shows a selection of implementations made using HLS. Figure 8 presents the change in FPGA resources with a bar for LUTs, FFs, DSP slices, and BRAMs–of a single SA compared to the sum of area for two DAs. In these examples, SAs reduce area utilization and offer similar speedup to DAs.

5 DISCUSSION

ReconfAST vs partial Reconfiguration: The EDA and FPGA community have tackled the problem by introducing partial reconfiguration in FPGAs to reduce the overhead of reconfiguring gates at the run-time. This methodology relies on hardware designer's through understanding of RTL of modules, while ReconfAST provides a methodical way to detect hot-code and similarities between workloads[**? ? ?**].

Limitations of CFG and CDFG Representations for Re-configurable Hardware:

Recent papers such as Conservation Cores, DySER, and Needle use graph representations of the workloads such as control flow graphs (CFGs), data flow graphs (DFGs), and combined CDFGs

[**? ? ?**]. These tools aim to identify hot-code in workloads, and do not find similarities between workloads. ASTs have long been used as more appropriate for cloning in compiler literature. Data flow and control flow are based on IR which is too specific and strict representation to find a large enough accelerator core which increases the communication overhead. DFG and CDFGs miss easily reconfigurable kernels with slightly different datatypes and sizes. Furthermore, for evaluation it is much easier to go from AST node (which includes line number) to the source code than its IR.

6 CONCLUSION AND FUTURE WORK

ReconfAST is an high-level tool to find similarities between kernels that enables the design of **shared accelerators**. SAs are application specific accelerators that can accelerate more than 1 workload but have an average speedup is 2x faster than common CGRAs; and are especially useful when there is a system-wide area constraint and DAs for all necessary kernels won't fit.

We have evaluated our tool by designing accelerators based on ReconfAST on Machsuite and implementing them in FPGAs using High-Level Synthesis tools We have implemented different maps and applied 15 different optimizations to both SAs and DAs; this resulted in over 10,000 implementation examples. We then analyzed the characteristics of the ReconfAST results, the isomorphic subtrees, that would result in an efficient accelerator.

Future work will focus on will expand our tools to efficiently and automatically find similarities between more than two applications and implementing the SAs as `loosely-coupled` accelerators in an SoC.