*Dedicated to my parents**

*without whom all my aims, aspirations and dreams would be non-existent

Acknowledgements

There are a number of people without whom this work might have been a distant reality, and to whom I am greatly indebted.

To my Masters Thesis advisor Prof. Katherine Morrow, for providing me with continual guidance and support in completing this work. Her effective management style ensured that steady progress was made every week. I couldn't have asked for a better advisor.

To my parents, Prof. Rakesh Chadha and Prof. Punam Chadha, for their excellent genes and the incredible examples they set that inspires me to emulate them.

To Anthony Gregerson for his help, guidance and mentorship from the very beginning of this work. His deep insight in possibly every technical concept on this planet made it a pleasure to work with him.

To my friends, Chetan Suresh and Sai Prasanth, thanks for being there through thick and thin. I appreciate all your support when I needed it and will remember every bit of our memories about the time we spent together -- both the countless hours of technical discussions and the post-exam TV show marathons!

I am also indebted to the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison for fostering an environment in which I could excel. The quality of the program made it an honor to have studied here.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Symbol	Explanation
ASIC	Application Specific Integrated Circuit
BLIF	Berkeley Logic Interchange Format
BRAM	Block Ram
CAD	Computer Aided Design
CLB	Configurable Logic Blocks
DIMACS	Discrete Mathematics and Theoretical Computer Science
DSP	Digital Signal Processing
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IC	Integrated Circuit
IMC	Internet Measurement Conference
IP	Intellectual Property
ISCAS	International Symposium on Circuits and Systems
ITC	International Test Conference
IWLS	International Workshop on Logic and Synthesis
LUT	Look Up Table

Symbol	Explanation
MAC	Multiply Accumulate
MCNC	Microelectronics Center of North Carolina
PREP	Programmable Electronics Performance Corporation
RAM	Random Access Memory
RTL	Register-Transfer Level
VHDL	Very High Speed Integrated Circuits Hardware Description Language

1 Introduction

Modern FPGA architectures incorporate heterogeneous resources as their building blocks, including configurable logic blocks (CLBs), Multipliers, RAM blocks, IP Cores [13], etc. The Xilinx Vertex II is an example of such an FPGA architecture which contains specialized resources such as multipliers and RAM blocks interspersed among CLBs. Designs created for such modern FPGAs can make use of the specialized resources to provide better performance.

A by-product of heterogeneity is the opportunity to implement certain computations in a variety of ways, using different combinations of resources. A node in a circuit netlist that has multiple possible implementations can be said to have multiple "personalities". Prior partitioning techniques have not focused on exploiting this aspect of resource mapping. However, there is an effort under development [1] that incorporates aspects of resource mapping into partitioning and exploits this "multi-personality" nature of nodes. This approach, however, requires a set of suitable benchmarks for evaluation: large circuits where at least some of the computations have multiple possible implementations.

HDL benchmarks provide information about both the structure and the function of the graph. This allows us to extract real personality information from the nodes by analyzing different ways to implement a node's function on a given device. However, in order to assign multiple personalities, the HDL designs must contain a significant number of elements that can be implemented using multiple resource types on the device. For instance, on an FPGA this could include functions such as arithmetic operations or data storage. These additional constraints, combined with the fact that many established HDL-based benchmark sets do not adequately represent the size of modern circuit partitioning problems, limit our ability to use an existing HDL benchmark set. Existing graph partitioning benchmark suites in non-HDL format, on the other hand, cover a broad range of sizes, including graphs large enough to approximate the size of modern circuit partitioning problems. However, because these graphs are in non-HDL format and do not express the functionality of the nodes, we can only assign synthetic personality information to them. It is unclear how well synthetically assigned personalities represent the properties of real multi-personality circuits. This work documents efforts to assemble a suitable set of benchmarks that meet both criteria -- circuit size and multi-resource utilization -- for accurate evaluation of a new multi-personality partitioning algorithm. It also documents initial efforts to assemble a set of benchmarks for a content-aware partitioning algorithm, which uses information about the characteristics of communication along edges to determine where to make partition cuts.

The remainder of the work is outlined as follows: Section 2 introduces circuit partitioning, defines the concepts of multi-personality logic and personality selection, and touches upon data-intelligent partitioning. Section 3 initially discusses benchmark sources followed by an explanation of requirements from our benchmarks. It then surveys standard FPGA benchmarking suites, presents the selected HDL and Non-HDL benchmarks, and finally describes the process of adapting benchmarks for use in evaluating multi-personality partitioning. Section 4 discusses specific requirements for content-aware partitioning, surveys publicly-available processor designs for suitability in evaluation of content-aware partitioning and talks about adapting these benchmarks for our use. Lastly, section 5 concludes the work.

2 Background

2.1 Partitioning

In circuit design, partitioning is the process of dividing a circuit into a set of two or more smaller sub-circuits. Fig. 1 depicts partitioning as one of the steps in the physical design phase of the IC design flow. During partitioning, a circuit is represented by a netlist, containing nodes of logic or computations, connected by graph edges that represent communication. Common partitioning goals and constraints include:

- Minimizing interconnections between partitions
- Minimizing circuit delay (the number of times a path crosses between partitions)
- Ensuring that partitions do not exceed per-partition resource capacities
- Balancing nodes across partitions

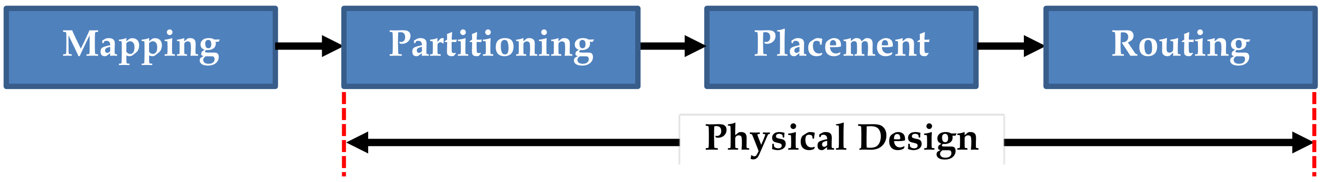


Figure 1: Typical CAD flow that includes a partitioning step.

Most variants of the partitioning problem are known to be NP-hard [6] and focus on minimizing the number of edges linking vertices from different cells (the *cut size*). Fig. 2 demonstrates a partitioning solution for the given circuit with a minimum cut size.

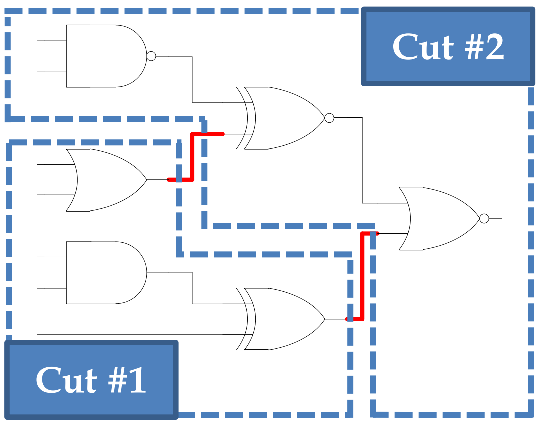


Figure 2: Illustration of cut size.

In recent years, performing partitioning prior to placement has become important for implementing large ASIC designs. This approach allows a very large problem to be broken down into a set of smaller, more tractable ones [10, 11]. As FPGA densities increase, this approach becomes more important in this domain as well [12]. Unlike ASICs, FPGAs provide the additional constraint that, for a given device, the number and type of resources available to implement netlist nodes is fixed for each region of the device. This constraint is easily handled in a homogeneous device, such as an FPGA entirely composed of LUTs. However, with heterogeneous resources, such as BRAMs and DSP blocks, the partitioning problem becomes more complex to avoid over-subscribing any type of resource in any of the partitions. For a multi-FPGA partitioning, for example, exceeding capacity in any of these resources prevents a partition from being feasibly implemented in a single device.

2.2 Multi-Personality Partitioning

Originally, FPGA devices essentially contained a single type of resource - configurable logic blocks (CLBs), which were uniformly distributed throughout the chip. Modern FPGAs, however, contain a variety of resource types, opening the opportunity to implement some computations in different possible ways within the same device. For instance, multiply-accumulate (MAC) operations can be implemented as either DSP blocks or CLBs, and data can be stored in BRAMs or distributed memories composed of CLBs. Each implementation results in different resource utilization and/or timing characteristics [1].

In this work, we refer to logic having multiple implementations as “multi-personality logic” and the process of choosing a specific implementation for multi-personality logic as “personality selection” or “implementation selection”. While CAD tools perform personality selection during the mapping stage of synthesis, multi-personality partitioning incorporates mapping computations to particular resources within the partitioning process. This can improve both the cut-size and heterogeneous resource utilization, thus leading to a better partitioning solution [1]. The possible implementations can be created by a designer during design-tradeoff exploration or by automated synthesis tools.

2.3 Content-Aware Partitioning

While most partitioning algorithms account for the topography of the circuit by focusing on minimizing the cut-size, another aspect that could be considered is the communication of information between partitions, particularly in the case of a multi-FPGA implementation. Further, redundancy amongst the data being transmitted at the edges can help reduce the number of signal transitions eventually leading to reduced transmission power. For instance, temporal correlation between sequential addresses on the address bus of a processor is one such scenario where redundancy can be exploited.

The details of this content-aware partitioning are beyond the scope of this work, but have the potential to improve effective bandwidth and/or robustness by exploiting data entropy and redundancy along communication paths. It may be that, in this case, minimizing cut size (so long as it remains below a threshold related to the number of available pins on the targeted device) is less critical than improving other characteristics of inter-partition communication. Developing a content-aware partitioner also requires large, sophisticated benchmarks. Unlike the multi-personality partitioning, however, the characteristics of the data being transmitted across the partitions needs to be known and taken into account during partitioning. Thus we require HDL benchmarks that can be simulated to process realistic data so that we can determine the values communicated across different connections within the circuit.

3 Benchmarks for Multi-Personality Partitioning

The development of a new multi-personality partitioning algorithm [1] requires benchmarks for testing and evaluation. This work documents efforts to create a set of benchmarks for this purpose by adapting existing benchmarks from several different sources. We examine a wide variety of real-world circuit designs and publicly-available benchmarks, with a specific emphasis on the benchmark size and the possibility of multi-personality nodes.

In this section, we focus our initial discussion towards potential sources of FPGA benchmarks and then develop a series of requirements that were used to evaluate the suitability of benchmarks for the purpose of evaluating multi-personality partitioning algorithms [1] and content-aware partitioning algorithms currently under development.

3.1 Potential Benchmark Sources

A multitude of benchmark suites specifically intended for evaluation of hardware and software solutions developed using FPGAs are freely available in the public domain. These benchmarks essentially hail from conferences, open source repositories, synthetic benchmark generators and industrial organizations [24].

Conference Benchmarks

Most of the benchmarks in academia originate from conferences and workshops. These benchmarks typically consist of designs derived from diverse industrial domains that can be potentially supported by FPGA systems. Conference benchmarks tend to be widely circulated as they are publicly available. For instance, the Microelectronics Center of North Carolina (MCNC) benchmark suite [30], International Workshop on Logic and Synthesis (IWLS) 2005 benchmark suite [29] and the Toronto 20 benchmark suite [28] originated from conferences. An in-depth discussion of benchmark suites is presented in Section 3.3.

Open-Source Benchmarks

Open source benchmarks enable end-users to duplicate methodologies and results. OpenCores [27] is an example of an open source community that offers digital modules, i.e., Intellectual Property (IP) cores, with a similar ethos to the free software movement.

Synthetic Benchmarks

To evaluate new architectures which cannot be efficiently tested using existing real-design-based benchmarks, designers turn to automatic generation of synthetic circuits. Numerous approaches to generate synthetic benchmark circuits for evaluating new architectures and tools have been developed [25, 26].

Industrial Benchmarks

FPGA vendors emphasize benchmarking systems using real customer designs. Work in circuit-partitioning literature has made use of industrial benchmarks in the past [2]. Unfortunately, since industrial designs are bound by non-disclosure agreements, they are generally confidential and not publicly available.

3.2 Benchmark Requirements

Effectively characterizing the performance of a partitioning algorithm requires circuits of a suitable size, large enough to require porting over multiple FPGAs. However, benchmarking a new multi-personality partitioner required a set of benchmarks that satisfy certain additional criteria, apart from circuit size. Below, we enlist the requirements that we focused on while seeking suitable benchmarks to test a multi-personality partitioning algorithm:

Circuit Size

The first requirement was size -- having benchmarks that are large enough to reflect the need for a multi-FPGA scenario are of critical importance. We thus sought netlists having a relatively large number of design modules (i.e., nodes to partition). While replicating multiple instances of the same circuit on an FPGA can help overcome the drawbacks of small benchmarks, this does not always reflect the way an FPGA is used in practice. Testing the partitioning algorithms over a set of circuits with different sizes/characteristics allows us to examine the capabilities (and possible weaknesses) of the algorithms.

Heterogeneous Resource Use

A particularly important requirement to profile the effectiveness of the multi-personality nature of partitioning was the property of heterogeneity. Netlists which contained nodes that could be synthesized using multiple resource types were sought.

Circuit Diversity

We sought circuits from a variety of different application areas such as computational algorithms, signal processing, communications, high-energy physics etc. We derived circuits from various sources: benchmark suites, open source repositories and graph archives, described in Section 3.1. Evaluation using multiple circuits from different domains and sources can help demonstrate the versatility of a partitioning algorithm.

3.3 Benchmark Evaluation

FPGA benchmarks can be broadly classified as HDL Benchmarks or as Non-HDL Benchmarks. These designs in particular are also desired for planned future research in content-aware partitioning that was described in Section 2.3.

3.3.1 HDL Benchmarks

This categorization includes benchmarking suites with circuit netlists in Verilog, VHDL and other custom formats such as NET, YAL, etc.

VTR Benchmarks

The Verilog To Routing (VTR) project [19] is a large scale, publicly available software suite that can synthesize circuits into easily-described hypothetical FPGA architectures. It consists of three core tools: ODIN II for Verilog Elaboration and front-end hard-block synthesis, ABC for logic synthesis, and VPR for physical synthesis and analysis. VTR provides a set of 19 Verilog benchmarks from diverse sources. The circuits range in size from 170 to 99,700 6-LUTs. They also exhibit heterogeneity, with several circuits utilizing memory and multiplier blocks. Even though some circuits are of an acceptable size for benchmarking partitioning algorithms, they are still very small compared to the size of the modern FPGAs which contain roughly 500,000 6-LUTs [19].

MCNC Benchmarks

The Microelectronics Center of North Carolina (MCNC) benchmark suite [3] includes logic synthesis and optimization benchmark sets from the International Symposium on Circuits and Systems (ISCAS) ′85 and ′89, in addition to some other benchmarks collected from industry and academia. The benchmark suite has standardized libraries with a wide variety of representative circuit designs ranging from simple to advanced circuits obtained from the industry [24]. MCNC benchmarks are very popular in academic research. For instance, in [4], the runtime performance of the proposed optimization approach is evaluated using MCNC benchmarks. A number of benchmark suites have been derived from the original MCNC benchmark suite which led to the creation of several MCNC-based benchmarking suites, including the MCNC Golden 20, NCSU-CBL′s benchmark mix, and some Berkeley Logic Interchange Format (BLIF) circuit compilations.

MCNC Golden 20 Benchmark Suite

The MCNC Golden 20/MCNC Big 20/Toronto 20 benchmark suite originated from an “FPGA Place-and-Route Challenge” that aimed at encouraging researchers to benchmark their CAD tools using large circuits [28]. The benchmark suite consists of twenty of the largest MCNC circuits in “.net” format used as benchmarks for the place-and-route challenge. The benchmarks within this suite range in size from 1047 to 8383 4-LUTs. Some academic researchers have adopted these benchmark designs to evaluate their work [24]. For instance, comparison of area ratios of the CMOL technology with CMOS and nanoPLA circuit architecture technologies using the MCNC Golden 20 benchmark suite was performed by Strukov et al. [5]. Similarly, in [7], the MCNC Golden 20 circuits are used to evaluate performance of T-VPack, a timing-driven packing algorithm on various FPGA architectures based on area-delay product evaluation metric. Even though the benchmarks were amongst the largest circuits from the MCNC suite, they failed to comply with our size requirements. We did not explore their multi-resource utilization aspect in depth.

NCSU-CBL's MCNC-Based Benchmark Mix

The Collaborative Benchmarking Laboratory at North Carolina State University (NCSU-CBL) put together a mix of benchmarks compiled from the original MCNC suite with a focus on floorplanning and placement [8]. The benchmarks are in ".yal" format and are categorized as block netlists, mixed netlists and standard cell netlists. With regards to suitability for benchmarking purposes, since even the largest MCNC circuits (the Golden 20) were insufficiently large for our purposes, we did not include any circuits from this suite.

BLIF Circuits

Berkeley Logic Interchange Format (BLIF) is a hardware description language designed for the hierarchical description of sequential circuits, which serves as an interchange format for synthesis and verification tools [33]. The goal of BLIF is to describe a logic-level hierarchical circuit in textual form [34]. BLIF is mainly used as an academic standard. A set of 200 BLIF-based benchmarks compiled from various sources including the MCNC benchmark suite was retrieved and evaluated for benchmarking-suitability for this work after converting them to Verilog using BLIF2Verilog [32]. A CAD package to perform re-clustering of large system-on-chip designs with interconnect variation, viz., Un/DoPack and Congestion VPR [40] contained 10 BLIF benchmarks derived from the MCNC benchmark suite by “stitching” them together. Unfortunately, the benchmarks did not meet our size requirements.

UofT Benchmarks

A number of benchmarks [21] can be derived from an implementation of a ray-tracing processor in VHDL developed by the University of Toronto. The processor design being hierarchical in nature, allows for a subset of its 50 constituent circuits to be used independently. Benchmarks having an LUT utilization ranging from 2 to 30,000 can be thus derived. We adopted the ray-tracer as a benchmark due to its size and also due to its utilization of embedded RAMs. Another VHDL circuit for real-time distance measurement using stereo vision from the same suite was also adopted [35].

IWLS 2005 Benchmarks

The International Workshop on Logic and Synthesis (IWLS) 2005 benchmark suite [9] contains 84 benchmarks compiled from OpenCores, Gaisler Research, Faraday Technology Corporation along with benchmark sets from the International Test Conference (ITC) ’99 and the International Symposium on Circuits and Systems (ISCAS) ’85 and ’89. The benchmarks are provided in two formats: Verilog and OpenAccess. The suite offers benchmarks in synthesized form, mapped using Cadence RTL Compiler to a 180 nm library. Mishchenko et al. use IWLS 2005 benchmarks to demonstrate the performance of their proposed technique for combinational logic synthesis [14]. These benchmarks did not meet our size requirements, since their size was too small for inclusion even in the smallest-size class of circuits. Their heterogeneity aspect was thus not explored in detail. Consequently, they were not used.

ERCBench

ERCBench [22] is a freely-available, open-source benchmark suite by the University of Wisconsin geared towards embedded and reconfigurable computing research. It features Verilog circuits from a variety of application areas, such as audio processing, image processing, cryptography, and wireless communications. Table 1 summarizes the circuits included as part of the ERCBench package and their corresponding application domains.

Domain	**Benchmarks**
Audio processing	Fast Fourier Transform (FFT) and Ogg Vorbis
Image processing	Face detection and JPEG2000
Wireless communications	Low-Density Parity-Check (LPDC), turbo codes and Viterbi decoders
Cryptography	Advanced Encryption Standard (AES), Blowfish, Data Encryption Standard (DES), Elliptic Curve Cryptography (ECC) and Secure Hash Algorithm (SHA)

Table 1: ERCBench circuits and their application areas.

ERCBench's circuits are specifically intended for benchmarking modern FPGA-based systems, and thus most circuits were found to be suitable for our application. The circuits also demonstrated significant heterogeneity as they were based on computational operations. A number of circuits from ERCBench were thus used in this work.

RAW Benchmark Suite

The RAW benchmark suite, published by MIT's reconfigurable architecture workstation project, consists of twelve programs designed to facilitate comparing, validating, and improving reconfigurable computing systems

[16]. These benchmarks run the gamut of algorithms found in general purpose computing including sorting, matrix operations, and graph algorithms. The suite includes an architecture-independent compilation framework, which allows each benchmark to be portably designed in both C and Behavioral Verilog and scalably parameterized to consume a range of hardware resource capacities. Benchmark results are reported using the following metrics: solution speed (kHz), speedup relative to reference software, and speedup per FPGA [56]. The benchmarks, developed in 1997, are relatively old for modern FPGA technology. Lacking in size requirements, they do not prove to be an effective choice for benchmarking.

PREP Benchmark Suite

The PREP benchmark suite, published by the Programmable Electronics Performance Corporation (PREP) sought to demonstrate the performance and capacity of programmable logic devices [16]. PREP benchmark circuits mainly consist of simplistic datapaths, counters and state machines. PREP benchmarks enable designers to estimate target devices that best suit a particular application early on in the design process [24]. The minimal benchmarks of PREP were insufficient in terms of both size as well as multi-resource utilization requirements.

3.3.2 Non-HDL Benchmarks

This categorization includes graph benchmarks that were not based on circuit netlists, but rather were distributed as graphs.

Walshaw's graph partitioning archive

Chris Walshaw's graph partitioning archive [31] contains 34 graphs ranging from 2395 to 448695 vertices aggregated from various sources. These have been very popular as benchmarks for graph partitioning algorithms since they cover a wide range of size. Some of these obey our size constraints and were found suitable for our application.

DIMACS

The 10th DIMACS Implementation Challenge Workshop [43] sought to address problems on graph partitioning and graph clustering. The testbed for the challenge comprised a large number of different graphs, both synthetic and real-world ones, coming from different applications, as shown by Table 2. In particular, we evaluated Kronecker and citation network graphs and found them to be suitable due to their sheer size.

Graph Type	**Description**
Kronecker graphs (Graph500)	Synthetic graphs created with the Kronecker generator [46] of the Graph500 benchmark [47].
Frames from 2D Dynamic Simulations	Synthetic meshes generated as part of dynamic mesh sequences which resemble adaptive 2D numerical simulations.
Delaunay Graphs	Graphs generated as Delaunay triangulations of random points in the plane.
Co-author and Citation Networks	Real-world social networks are created from co-authorships and citations.
Street Networks	Graphs that model real-world street networks.
Sparse Matrices	A small subset of the Florida Sparse Matrix Collection [18].
Random Geometric Graphs	Graphs generated from random points in a unit square.
Clustering Instances	Real-world graphs which are often used as benchmarks in the graph clustering and community detection communities.
Numerical Simulations	Graphs used in numerical simulation applications.

Graphs	
Erdös-Rényi Graphs	Random graphs from the G(n,p) model [45].
Computational Task Graphs	Computational task graphs of streaming applications generated using a specialized generator [48].
Redistricting Graphs	Graphs representing the US states in the redistricting problem.
Star Mixtures	Star-like structures of different graphs with different types.

Table 2: DIMACS graph categories [43].

IMC 2007 Data Sets

The online social networks research team at the Max Planck Institute for Software Systems has put together a series of data-sets published in the Internet Measurement Conference (IMC '07). The data-sets comprise of a list of users, links, groups and group members, obtained by crawling online social networks such as Flickr, LiveJournal, Orkut and YouTube [17]. The data-sets are very large in size -- for instance, the Orkut links dataset has over 3 million nodes -- and thus satisfy our requirements.

3.4 Selected Benchmarks

Table 3 lists the benchmarks that satisfy the requirements described in Section 3.2, and thus were chosen for use in evaluating the multi-personality partitioner.

Benchmarks	**Description**
mcml	An application that uses Monte Carlo simulation of photons which could be used as part of a photo-dynamic therapy-based cancer treatment plan [19].
raygen	A ray-tracing engine, which is a circuit for generating an image by tracing the path of light through pixels in an

	image plane and simulating the effects of its encounters with virtual objects [21].
fft128 sha	Netlists obtained from ERCBench [22], essentially computational algorithms.
boundtop blob diffeq1	Netlists obtained from VTR [19], originally derived from various university research projects.
rct jet isolation	Circuits designed for multi-FPGA high-energy physics experiments such as calorimetry-based triggering, jet reconstruction and particle isolation [23].

Table 3: Selection of benchmarks.

Table 4 characterizes the selected benchmarks in terms of their size and multi-resource utilization. We classified netlists in three categories on the basis of their size: small (1000 - 10000 nodes), medium (10000 – 100000 nodes) and large (greater than 100000 nodes). It should be noted that since nodes can have multiple personalities, some nodes are counted in multiple columns, making the sum of the percentages greater than 100%. Moreover, although the percentage of non-CLB implementations is relatively low, DSP and BRAM nodes tend to be much coarser than CLB-only nodes; DSP nodes range from the equivalent of 2-150 CLBs and BRAM nodes range from the equivalent of 40-760 CLBs [1].

Circuit Size	**Benchmark**	**Total Nodes**	**% CLB Nodes**	**% DSP Nodes**	**% BRAM Nodes**	**Ref.**
Large	mcml	346248	100	15	0.1	[20]
	rct	241349	100	31	12	[23]
	jet	189579	100	32	0	[23]
	isolation	187766	100	1	0	[23]
Medium	fft128	91590	100	9.5	0.5	[22]

	boundtop	29582	100	0.5	14	[19]
	blob	11842	100	24	0	[19]
	raygen	11457	100	25	0.1	[21]
Small	diffeq1	4292	100	38	0	[19]
	sha	3669	100	15	0	[22]

Table 4: Characterization of selected benchmarks.

3.5 HDL Benchmark Processing

Since partitioning algorithms eventually operate on graphs, the first step in enabling them to operate on HDL benchmarks involves HDL-to-graph conversion. Widely-used conventional graph formats such as the Chaco format [44] contain information about nodes and their respective connections (edges). They serve well for conventional partitioning algorithms that do not require personality-specific information embedded within the graphs. In the process of HDL-to-graph conversion for such partitioners, although nodes and their edges are retained, personality information is obliterated.

To enable the multi-personality partitioner to operate on the selected HDL benchmarks, we performed a multi-stage conversion from the HDL format to an intermediate format and then to graph format.

The circuit in HDL form is first passed through an in-house parser which creates an intermediate HDL representation that caters specifically to multi-personality partitioning algorithms by retaining nodes' personality information. The key difference between the Chaco format and our format is that it is specialized for functionality-related information necessary for personality-based partitioning algorithms. This allows us to later assign resource costs for different resource types on the basis of the selected FPGA device.

3.5.1 HDL Synthesis

If a circuit is described in structural HDL, which includes resource instantiation and specification of the connections between resources, it is already in a graph-style format. However, circuits described in behavioral HDL must first be synthesized using the GTECH library to a structural netlist before they can be further processed. The technology-independent GTECH library provides a "common platform" for any design to be synthesized to, and eases the process of assigning resource weights later on, rather than using a technology-specific library for synthesis.

3.5.2 HDL Parsing

HDL syntax can be complex to parse. It can include name-based and positional connections, connections utilizing escaped identifiers for flattened hierarchies etc. These benchmarks needed to work with a custom HDL parser for the Verilog 2001 standard that parsed an HDL file into an intermediate representation format which was essentially a log of module names and their port connections of the following format:

```
module_begin
<gtech_library_module> <connection1> <connection2> … <connectionN>
module_end
```

Fig. 3(b) shows an example of an intermediate HDL representation outputted for the selected diffeq1 circuit. The parser supports the following features:

(i) Connect-by-name and connect-by-reference type of connections.

(ii) Disambiguation between a bus referenced with its entire range (without using slice-syntax) from a bus referenced using an [a:b] range.

(iii) When a bus is referenced using slice-syntax, splits up a bus into its constituent signals to enable each signal of the bus to be individually identified as a graph edge.

(iv) Parsing of escaped identifiers[1]. These typically feature in module instantiations or signal names when the hierarchy of a synthesized netlist is flattened.

(v) Outputs a catalogue on completion, with a list of unique modules and a count of the number of instantiations of each module to identify the size of the benchmark. Fig. 3a shows the catalogue output of the parser.

3.5.3 Graph Formation

Once the textual HDL-based intermediate representation is generated at the output of the parser, formation of a graph involves characterizing nodes and their respective edges from the parser's output. A snippet from the parser's output, as demonstrated in Fig. 3b, is shown below:

```
module_begin
GTECH_ADD_ABC \add_43/Ul_l y_var[l] temp[l] add_43/carry[1]
module end
```

In the above example, "module_begin" and "module_end" demarcate a node. The first element within the node, "GTECH_ADD_ABC" denotes the

[1] Verilog provides a means of including any of the printable ASCII characters in an identifier by escaping the identifier. Escaped identifiers begin with a backslash and are terminated by white space. Characters such as commas, parentheses, and semicolons become part of the escaped identifier unless preceded by a white space.

type of the node. We may cross-reference this node-type against the implementation table for personality information, included as Table 6 in the appendix. Two entries in the table corresponding to "GTECH_ADD_ABC" denote multi-personalities for the node. Additional details about assigning resource weights for different personalities follow in the subsequent sections. The elements after the node-type delimited by space, viz., "\add_43/Ul_l", "y_var[l]", "temp[l]" and "add_43/carry[1]" are the connections of the node. Each of these represent the edges of the graph corresponding to the node. A graph can then be synthesized by coupling vertices of the graph (nodes) and their corresponding edges (connections).

3.5.4 Resource Weight Assignment

We assign node weights based on synthesis results. We initially synthesized the netlists using the technology-independent GTECH library. LUT, DSP and BRAM resource estimates for GTECH library components were calculated on the basis of the number of inputs, number of outputs, width of inputs and the width of outputs [42]. Accordingly, we assigned node weights and personalities using experimental synthesis in Xilinx ISE.

3.6 Non-HDL Benchmark Processing

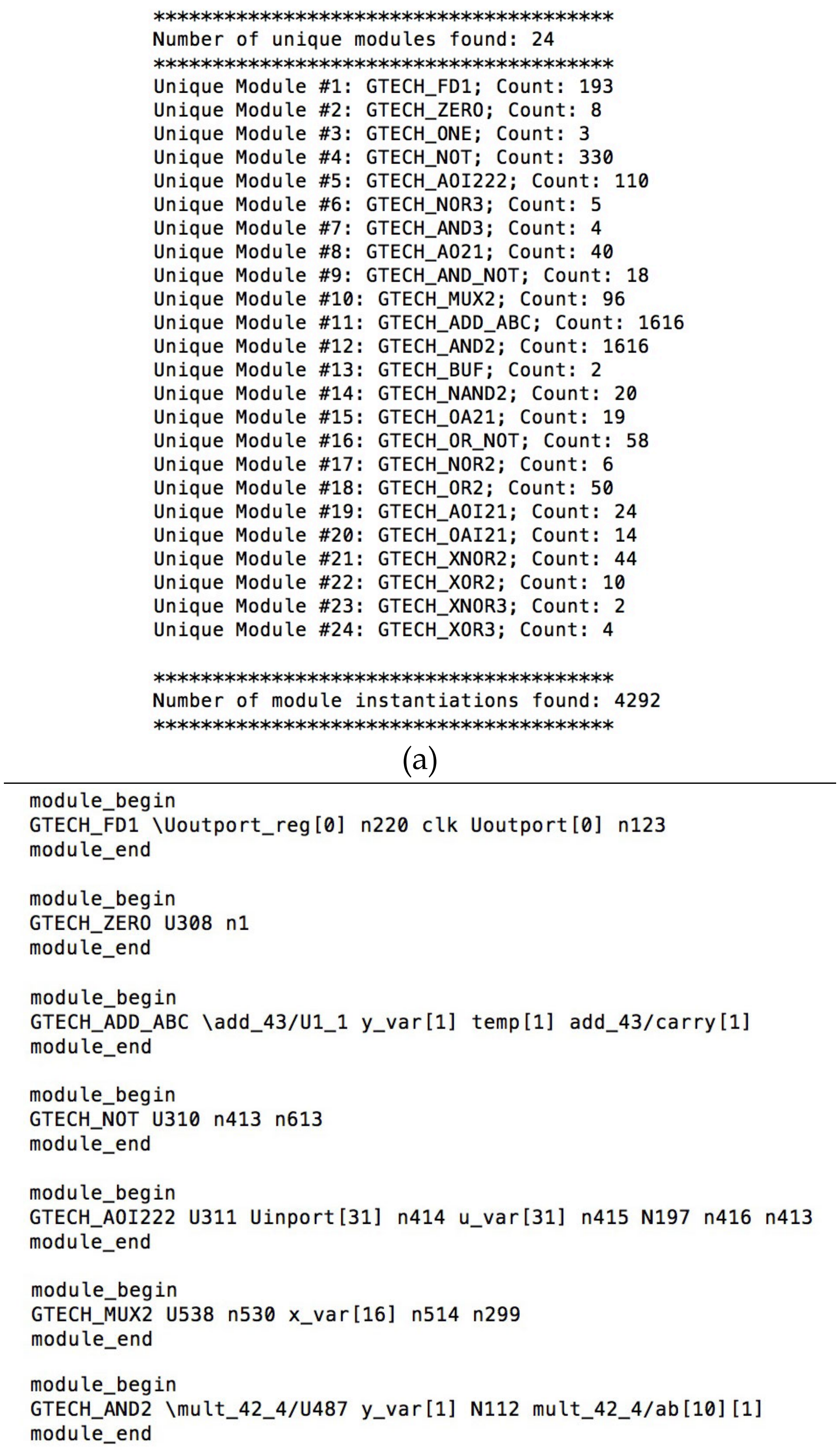
Although the chosen non-HDL benchmarks are generally suitably extensive in size, they lack personality information. We synthetically attach personality information to groups of nodes within a graph to adapt them for our application. This can be achieved by characterizing existing circuits and attempting to replicate their properties. For instance, while some circuit elements such as memory blocks, caches, ALUs, and IOs have clustered nodes, others such as high energy physics and signal processing logic blocks typically have sparse nodes. While an in-depth discussion of the implementation details of the process is beyond the scope of this work, we describe two forms of distance-based clustering to achieve personality modeling. Node weights can be assigned using a random number generator and a probability distribution. The distribution can be constrained to assign node weights for both clustered and sparse nodes. Alternatively, we may analyze the topology of the graph and do a Breadth-First-Search across the graph to characterize nodes within a particular distance, and assign personalities based on the distance.

3.7 Benchmark Use

These benchmarks were used to evaluate a set of multi-personality partitioning algorithms [1]. It was found that:

(i) Multi-personality partitioning achieves up to a 27% mean improvement in partition cut size compared to partitioning a statically-mapped circuit for our set of selected benchmarks.

(ii) Dynamically selecting node personalities for ratio control can achieve up to a 15x mean improvement in deviation in target resource utilization compared to post-partition resource mapping.



```
***************************************
Number of unique modules found: 24
***************************************
Unique Module #1: GTECH_FD1; Count: 193
Unique Module #2: GTECH_ZERO; Count: 8
Unique Module #3: GTECH_ONE; Count: 3
Unique Module #4: GTECH_NOT; Count: 330
Unique Module #5: GTECH_AOI222; Count: 110
Unique Module #6: GTECH_NOR3; Count: 5
Unique Module #7: GTECH_AND3; Count: 4
Unique Module #8: GTECH_AO21; Count: 40
Unique Module #9: GTECH_AND_NOT; Count: 18
Unique Module #10: GTECH_MUX2; Count: 96
Unique Module #11: GTECH_ADD_ABC; Count: 1616
Unique Module #12: GTECH_AND2; Count: 1616
Unique Module #13: GTECH_BUF; Count: 2
Unique Module #14: GTECH_NAND2; Count: 20
Unique Module #15: GTECH_OA21; Count: 19
Unique Module #16: GTECH_OR_NOT; Count: 58
Unique Module #17: GTECH_NOR2; Count: 6
Unique Module #18: GTECH_OR2; Count: 50
Unique Module #19: GTECH_AOI21; Count: 24
Unique Module #20: GTECH_OAI21; Count: 14
Unique Module #21: GTECH_XNOR2; Count: 44
Unique Module #22: GTECH_XOR2; Count: 10
Unique Module #23: GTECH_XNOR3; Count: 2
Unique Module #24: GTECH_XOR3; Count: 4

***************************************
Number of module instantiations found: 4292
***************************************
```

(a)

```
module_begin
GTECH_FD1 \Uoutport_reg[0] n220 clk Uoutport[0] n123
module_end

module_begin
GTECH_ZERO U308 n1
module_end

module_begin
GTECH_ADD_ABC \add_43/U1_1 y_var[1] temp[1] add_43/carry[1]
module_end

module_begin
GTECH_NOT U310 n413 n613
module_end

module_begin
GTECH_AOI222 U311 Uinport[31] n414 u_var[31] n415 N197 n416 n413
module_end

module_begin
GTECH_MUX2 U538 n530 x_var[16] n514 n299
module_end

module_begin
GTECH_AND2 \mult_42_4/U487 y_var[1] N112 mult_42_4/ab[10][1]
module_end
```

(b)

Figure 3: (a) Catalogue output of the parser; (b) Intermediate netlist representation sample.

4 Benchmarks for Content-Aware Partitioning

This section describes work begun to assemble a set of benchmarks to test future content-aware partitioning algorithms. Although initial efforts have been made, much of this is left to future work.

4.1 Benchmark Requirements

Unlike multi-personality partitioning, which requires heterogeneous nodes, content-aware partitioning focuses more on the edges in the netlist, and the data communicated along those edges. Benchmarks for content-aware partitioning require HDLs with representative input data sets that can be simulated to get communication information. If representative input data sets are unavailable for a particular design, the inputs can be characterized based on the HDL port specifications in order to constrain them accordingly.

4.2 Benchmark Evaluation

Processor designs tend to be a good match for benchmarks for content-aware partitioning, since processors may be implemented, emulated, and/or prototyped on FPGAs or multi-FPGA systems. These benchmarks may also be incorporated into the set used for multi-personality partitioning in future work.

VIPERS Soft Vector Processor

Vector ISA Processors for Embedded Reconfigurable Systems (VIPERS) [36] is a family of soft vector processors designed in Verilog with varying performance and resource usage, and a configurable feature set to suit different applications. VIPERS offers a highly parameterized HDL source code to help generate an application or domain-specific processor instance. This configurability gives designers flexibility to trade-off performance and resource utilization. By customizing the amount of parallelism, feature set, and instruction support needed by the application, VIPERS seeks to exploit the configurability of FPGAs. The processor, ISA specification, assembler, and benchmark code are available online [37]. A successor to VIPERS, namely VIPERS II with single-copy data scratchpad memory is also available [38].

VIPERS adapts traditional vector architectures to exploit FPGA-specific resources such as DSP blocks, BRAMs and multipliers, owing to its use of:

(i) Multiply-accumulate (MAC) units for vector reduction,

(ii) A partitioned register file,

(iii) Local memory in each vector lane for table-lookup functions and,

(iv) A large number of vector registers.

VIPERS exhibited significant heterogeneity and was selected as an appropriate benchmark for content-aware partitioning.

8085-Based Model

Using Intel's 8085AH-2 as the underlying baseline instruction set, a Verilog-based microprocessor design with various enhancements is available [39] and can be used as a benchmark, particularly due to its use of hardware multipliers and dividers leading to multi-resource utilization.

Amber

The Amber processor core [41] is an ARM v2a Instruction Set Architecture (ISA) based 32-bit RISC processor, supported by the GNU Toolset. The Amber project offers a complete embedded system incorporating the Amber core and a number of peripherals, including a UART, a timer and an Ethernet MAC.

The Amber project is designed in Verilog and provides two versions of the core:

(i) The Amber 23 with a 3-stage pipeline, a unified data and instruction cache and a 32-bit Wishbone interface,

(ii) The Amber 25 with a 5-stage pipeline, separate data and instruction caches and a 128-bit Wishbone interface.

Both cores implement the same ISA and are 100% software compatible. The cores do not contain a Memory Management Unit (MMU) so they can only run the non-virtual memory variant of Linux. Amber has been verified by booting a Linux 2.4 kernel. In addition, the cores are also optimized for FPGA synthesis. Amber met all of our requirements, and thus we decided to include it, in particular for evaluating content-aware partitioning, described in Section 2.3. The processor, ISA specification and user-guide are available online [41]. The Amber project also provides a basic set of workloads to test the core.

4.3 Benchmark Processing

Initially, we synthesized our selected processor designs for benchmarking content-aware partitioning. In particular, we focused on Amber, since it offered opportunities to experiment with custom workloads that could be run on the core. Amber thus provided a better context for evaluating content-aware partitioning in comparison to other designs by allowing us to tailor workloads to offer data redundancy that could be exploited by content-aware partitioning. After synthesis, we performed post-synthesis verification of the design using the included workloads to ensure functional correctness. We then experimented with external workloads for Amber, by sourcing generic benchmarks from diverse repositories and adapting them for use with the core. For external workloads which involved several function calls from diverse libraries, we observed that linking suitable libraries with definitions of all functions was a tedious task. We then explored developing custom workloads and running them on the core. These custom workloads, which were simplistic in nature, offered easy compilation and linking. A detailed description of the process has been explained in Appendix 2.

5 Conclusion

The work documented efforts to assemble and modify a set of benchmark circuits for testing a new class of circuit partitioning algorithms designed for heterogeneous FPGAs. Often, computations can be implemented using different types of resources within these devices; the new partitioning algorithms incorporate the circuit mapping step, where computations are mapped to specific resource types, into the partitioning algorithms themselves. This form of partitioning, called "multi-personality" partitioning was discussed in detail in this work. Remapping provided a great deal of flexibility to the partitioner to modify the implementation of circuit nodes in order to meet the desired partitioning criteria. However, testing this new partitioner requires large, heterogeneous netlists. In this work we developed a list of requirements for the needed benchmarks, investigated existing benchmarks to determine their suitability, and documented how we adapt the chosen benchmarks for use in testing multi-personality partitioning algorithms. Finally, we also discussed initial efforts in assembling and developing a set of benchmarks for testing content-aware partitioning.

6 References

[1] A. Gregerson, A. Chadha, and K. Morrow, "Multi-Personality Partitioning for Heterogeneous Systems," International Conference on Field-Programmable Technology (ICFPT), pp. -314-317, 2013.

[2] N. Selvakkumaran, A. Ranjan, S. Raje, and G. Karypis, "Multi-resource aware partitioning algorithms for FPGAs with heterogeneous resources," DAC, pp. 741-746, 2004.

[3] S. Yang, "Logic synthesis and optimization benchmarks user guide: version 3.0," Microelectronics Center of North Carolina (MCNC), 1991.

[4] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," IEEE Design Automation Conference, 2006.

[5] D. Strukov and K. Likharev, "A Reconfigurable Architecture for Hybrid CMOS/Nanodevice Circuits," In Proceedings of International Symposium on Field Programmable Gate Arrays, pp. 131-140, 2006.

[6] M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," W. H. Freeman and Company, 1979.

[7] A. Marquardt, V. Betz, and J. Rose, "Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density," In Proceedings of International Symposium on Field Programmable Gate Arrays, pp. 37-46, 1999.

[8] MCNC Benchmark Netlists for Floorplanning and Placement, Retrieved May 8, 2014, from: http://lyle.smu.edu/~manikas/Benchmarks/MCNC_Benchmark_Netlists.html

[9] C. Albrecht, "IWLS 2005 Benchmarks," 2005. Retrieved May 8, 2014, from: http://iwls.org/iwls2005/benchmark_presentation.pdf

[10] A. Caldwell and I. Markov, “Can recursive bisection alone produce routable placements?,” In Proceedings of Design Automation Conference, pp. 477–482, 2000.

[11] A. Soper, C. Walshaw, and M. Cross, “A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph Partitioning,” J. Global Optimization, vol. 29, no. 2, pp. 225-241, 2004.

[12] A. Wang and S. Raje, “Multi-million gate FPGA physical design challenges,” ICCAD, 2003.

[13] P. Zuchowski, C. Reynolds, R. Grupp, S. Davis, B. Cremen, and B. Troxel, “A hybrid ASIC and FPGA architecture,” ICCAD, pp. 187–194, 2002.

[14] A. Mishchenko, S. Chatterjee, and R. Brayton, “DAG-aware AIG rewriting a fresh look at combinational logic synthesis,” IEEE Design Automation Conference, 2006.

[15] C. J. Alpert, J. H. Huang, and A. B. Kahng, “Multilevel circuit partitioning,” DAC, 1997.

[16] S. Kliman, “Prep Benchmarks reveal Performance and Capacity Tradeoffs of Programmable Logic Devices,” IEEE International ASIC Conference and Exhibit, 1994.

[17] A. Mislove, M. Marcon, K. Gummadi, P. Druschel and B. Bhattacharjee, “Measurement and Analysis of Online Social Networks,” Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07), 2007.

[18] T. Davis and Y. Hu, “The University of Florida sparse matrix collection,” ACM Transactions on Mathematical Software (TOMS), Vol. 38, No. 1, 2011.

[19] J. Rose, J. Luu, C. Yu, O. Densmore, J. Geoders, and A. Sommerville, K. Kent, P. Jamieson, J. Anderson, "The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing," International Symposium on Field Programmable Gate Arrays (FPGA), pp. 77-86, 2012.

[20] J. Luu, K. Redmond, W. Lo, P. Chow, L. Lilge, and J. Rose, "FPGA-based Monte Carlo computation of light absorption for photodynamic cancer therapy," IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 157–164, 2009.

[21] J. Fender and J. Rose, "A high-speed ray tracing engine built on a field-programmable system," International Conference on Field-Programmable Technology, pp. 188-195, Dec. 15-17, 2003.

[22] D. Chang, C. Jenkins, P. Garcia, S. Gilani, P. Aguilera, A. Nagarajan, M. Anderson, M. Kenny, S. Bauer, M. Schulte and K. Compton, "ERCBench: An Open-Source Benchmark Suite for Embedded and Reconfigurable Computing," International Conference on Field Programmable Logic and Applications (FPL), pp. 408-413, Aug. 31-Sept. 2, 2010.

[23] A. Gregerson, et al., "FPGA design analysis of the clustering algorithm for the CERN Large Hadron Collider," FCCM, pp. 19-26, 2009.

[24] R. Njuguna, "A Survey of FPGA Benchmarks", Project Report, November 24, 2008. Retrieved May 3, 2014, from: http://www.cse.wustl.edu/~jain/cse567-08/ftp/fpga

[25] P. Verplaetse, J. Campenhout, and D. Stroobandt, "On Synthetic Benchmark Generation Methods," IEEE International Symposium on Circuits and Systems, Vol. 4, pp. 213-6, 2000.

[26] C. Chang, J. Cong, M. Romesis, and M. Xie, "Optimality and Scalability Study of Existing Placement Algorithms," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 23, No. 4, pp. 537-49, 2004.

[27] OpenCores. Retrieved May 1, 2014, from: http://www.opencores.org

[28] V. Betz, "The FPGA Place-and-Route Challenge," Retrieved May 3, 2014, from http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html

[29] C. Albrecht, "IWLS 2005 Benchmarks," 2005. Retrieved May 3, 2014, from http://iwls.org/iwls2005/benchmark_presentation.pdf

[30] S. Yang, "Logic Synthesis and Optimization Benchmarks, Version 3.0," Technical Report, Microelectronics Center of North Carolina, 1991. Retrieved May 3, 2014, from http://jupiter3.csc.ncsu.edu/~brglez/Cite-BibFiles-Reprints-home/Cite-BibFiles-Reprints-Central/BibValidateCentralDB/Cite-ForWebPosting/1991-IWLSUG-Saeyang/1991-IWLSUG-Saeyang_guide.pdf

[31] A. Soper, C. Walshaw, and M. Cross, "A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph Partitioning," J. Global Optimization, vol. 29, no. 2, pp. 225-241, 2004.

[32] BLIF Benchmark Suite, Retrieved May 4, 2014, from http://cadlab.cs.ucla.edu/~kirill

[33] D. Basin, S. Friedrich, and S. Mödersheim. "B2M: A semantic based tool for BLIF hardware descriptions," Formal Methods in Computer-Aided Design. Springer Berlin Heidelberg, 2000.

[34] Berkeley Logic Interchange Format (BLIF), University of California Berkeley, July 28, 1992. Retrieved May 4, 2014, from: http://www.ece.cmu.edu/~ee760/760docs/blif.pdf

[35] HDL Benchmark Circuits Designed at UofT, Retrieved May 4, 2014, from: http://www.eecg.toronto.edu/~jayar/benchmarks/bench.html

[36] J. Yu, C. Eagleston, C. Han-Yu Chou, M. Perreault, and G. Lemieux, "Vector Processing as a Soft Processor Accelerator," ACM Trans. Reconfigurable Technol. Syst., Vol. 2, No. 2, pp. 1-34, 2009.

[37] VIPERS Soft Vector Processor, Retrieved May 4, 2014, from: http://www.ece.ubc.ca/~lemieux/downloads

[38] C. Han-Yu Chou, "VIPERS II: a soft-core vector processor with single-copy scratchpad memory," M. A. Sc. Thesis, University of British Columbia, 2010.

[39] Verilog Design for Reuse Course, Retrieved May 4, 2014, from: http://saxelec.com

[40] M. Tom, D. Leong, and G. Lemieux, "Un/DoPack: Re-Clustering of Large System-on-Chip Designs with Interconnect Variation for Low-Cost FPGAs," IEEE/ACM International Conference on Computer-Aided Design, pp. 680-687, 2006.

[41] Amber ARM-compatible core, Retrieved May 4, 2014, from: http://opencores.org/project,amber

[42] DesignWare GTECH Library Databook, Release G-2012.06, June 2012.

[43] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "Graph Partitioning and Graph Clustering," 10th DIMACS Implementation Challenge Workshop, 2012. Retrieved May 4, 2014, from: http://www.cc.gatech.edu/dimacs10

[44] B. Hendrickson and R. Leland, "The Chaco user's guide: Version 2.0. Vol. 4," Technical Report, SAND95-2344, Sandia National Laboratories, 1995.

[45] P. Erdős, and A. Rényi, "On the evolution of random graphs," Magyar Tud. Akad. Mat. Kutató Int. Közl 5, pp. 17-61, 1960.

[46] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," The Journal of Machine Learning Research, Vol. 11, pp. 985-1042, 2010.

[47] The Graph 500 List, Retrieved May 5, 2014, from: http://www.graph500.org

[48] D. Ajwani, S. Ali, and J. Morrison, "Application-agnostic generation of synthetic task graphs for stream computing applications," Technical Report RC25181, IBM Research Reports, 2011.

7 Appendix 1: LUT, DSP and BRAM resource estimates for GTECH library components

Table 5 shows the LUT estimate calculations for GTECH components for Xilinx Virtex 7.

Component	Description	No. of Inputs	No. of Outputs	Input Width	Output Width	No. of LUTs
GTECH_ZERO	Logic Low	0	1	0	1	1
GTECH_AND2	2-Input AND Gate	2	1	1	1	1
GTECH_AND_NOT	2-Input AND Gate Inverted Input	2	1	1	1	1
GTECH_OAI21	2-Input OR into 2-input NAND	3	1	1	1	1
GTECH_OA21	2-Input OR into 2-input AND	3	1	1	1	1
GTECH_NOT	Inverter	1	1	1	1	1
GTECH_OR_NOT	2-Input OR Gate Inverted Input	2	1	1	1	1
GTECH_NAND3	3-Input NAND Gate	3	1	1	1	1
GTECH_NOR8	8-Input NOR Gate	8	1	1	1	2
GTECH_OR8	8-Input OR Gate	8	1	1	1	2
GTECH_XOR2	2-Input XOR gate	2	1	1	1	1
GTECH_NAND2	2-Input NAND Gate	2	1	1	1	1
GTECH_AND3	3-Input AND Gate	3	1	1	1	1
GTECH_AND8	8-Input AND Gate	8	1	1	1	2
GTECH_OR2	2-Input OR Gate	2	1	1	1	1
GTECH_NOR2	2-Input NOR Gate	2	1	1	1	1
GTECH_NOR3	3-Input NOR Gate	3	1	1	1	1
GTECH_AOI22	2-Input ANDs into One 2-input NOR	4	1	1	1	1
GTECH_AND4	4-Input AND Gate	4	1	1	1	1
GTECH_AO22	2-Input ANDs into One 2-input OR	4	1	1	1	1
GTECH_OR4	4-Input OR Gate	4	1	1	1	1
GTECH_AOI21	2-Input AND into 2-input NOR	3	1	1	1	1

Component	Description	No. of Inputs	No. of Outputs	Input Width	Output Width	No. of LUTs
GTECH_AO21	2-lnput AND into 2-input NOR	3	1	1	1	1
GTECH_MUX2	2-Bit Multiplexer	3	1	1	1	1
GTECH_AOI222	Three 2-lnput ANDs into One 3-input NOR	4	1	1	1	1
GTECH_OR3	3-Input OR Gate	3	1	1	1	1
GTECH_NOR4	4-Input NOR Gate	4	1	1	1	1
GTECH_OAI2N2	2-lnput OR and 2-lnput NAND into One 2-input NAND	4	1	1	1	1
GTECH_AND5	5-Input AND Gate	5	1	1	1	2
GTECH_NAND4	4-Input NAND Gate	4	1	1	1	1
GTECH_AOI2N2	2-lnput AND and 2-lnput NOR into One 2-input NOR	4	1	1	1	1
GTECH_MUXI2	2-Bit Multiplexer	3	1	1	1	1
GTECH_BUF	Non-Inverting Buffer	1	1	1	1	1
GTECH_NOR5	5-Input NOR Gate	5	1	1	1	2
GTECH_NAND5	5-Input NAND Gate	5	1	1	1	2
GTECH_OAI22	2-lnput ORs into One 2-input NAND	4	1	1	1	1
GTECH_OR5	5-Input OR Gate	5	1	1	1	2
GTECH_XNOR2	2-Input Exclusive-NOR Gate	2	1	1	1	1
GTECH_ADD_AB	Half Adder	2	2	1	1	2
GTECH_ADD_ABC	Full Adder	3	2	1	1	2
GTECH_ISOLATCH_EN1	Isolation Latch - One Enable	2	1	1	1	1
GTECH_ISOLATCH_EN0	Isolation Latch - Zero Enable	2	1	1	1	1
GTECH_ISO1_EN1	Isolation Buffer-Forced to 1	2	1	1	1	1
GTECH_ISO1_EN0	Isolation Buffer-Forced to 1	2	1	1	1	1
GTECH_ISO0_EN1	Isolation Buffer-Forced to 0	2	1	1	1	1
GTECH_ISO0_E	Isolation Buffer-	2	1	1	1	1

Component	Description	No. of Inputs	No. of Outputs	Input Width	Output Width	No. of LUTs
N0	Forced to 0					
GTECH_ZERO	Logic Low	0	1	0	1	1
GTECH_ONE	Logic High	0	1	0	1	1
GTECH_LSR0	SR Latch	2	2	1	1	2
GTECH_LD4_1	D Latch, Active Low, Single Output with Clear	3	1	1	1	1
GTECH_LD4	D Latch, Active Low with Clear	3	2	1	1	2
GTECH_LD3	D Latch with Clear	3	2	1	1	2
GTECH_LD2_1	D Latch, Active Low, Single Output	2	1	1	1	1
GTECH_LD2	D Latch, Active Low	2	2	1	1	2
GTECH_LD1	D Latch	2	2	1	1	2
GTECH_FJK3S	JK Flip-Flop with Clear, Set, and Scan	7	2	1	1	2
GTECH_FJK3	JK Flip-Flop with Clear, Set	5	2	1	1	2
GTECH_FJK2S	JK Flip-Flop with Clear, Scan	6	2	1	1	2
GTECH_FJK2	JK Flip-Flop with Clear	4	2	1	1	2
GTECH_FJK1S	JK Flip-Flop with Scan Test Pins	5	2	1	1	2
GTECH_FJK1	JK Flip-Flop	3	2	1	1	2
GTECH_FD4S	JK Flip-Flop with Set, Scan	5	2	1	1	2
GTECH_FD48	D Flip-Flop with Set - 8 Bit	5	6	1	1	6
GTECH_FD44	D Flip-Flop with Set - 4 Bit	4	4	1	1	4
GTECH_FD4	D Flip-Flop with Set	3	2	1	1	2
GTECH_FD3S	D Flip-Flop with Clear, Set, and Scan	6	2	1	1	2
GTECH_FD38	D Flip-Flop with Clear, Set - 8 Bit	6	6	1	1	6
GTECH_FD34	D Flip-Flop with Clear, Set - 4 Bit	5	4	1	1	4
GTECH_FD3	D Flip-Flop with	4	2	1	1	2

Component	Description	No. of Inputs	No. of Outputs	Input Width	Output Width	No. of LUTs
	Clear, Set					
GTECH_FD2S	D Flip-Flop with Clear, Scan	5	2	1	1	2
GTECH_FD28	D Flip-Flop with Clear - 8 Bit	5	6	1	1	6
GTECH_FD24	D Flip-Flop with Clear - 4 Bit	4	4	1	1	4
GTECH_FD2	D Flip-Flop with Clear	3	2	1	1	2
GTECH_FD1S	D Flip-Flop with Scan Test Pins	4	2	1	1	2
GTECH_FD18	D Flip-Flop - 8 Bit	5	8	1	1	8
GTECH_FD14	D Flip-Flop - 4 Bit	5	8	1	1	8
GTECH_FD1	D Flip-Flop	2	2	1	1	2
GTECH_INOUTBUF	Input-Output Buffer	2 or 3	1 or 2	1	1	1
GTECH_OUTBUF	Output Buffer	2	1	1	1	1
GTECH_INBUF	Input Buffer	1	1	1	1	1
GTECH_TBUF	Non-Inverting 3-State Buffer	2	1	1	1	1
GTECH_MUX8	8-Bit Multiplexer	10	1	1	1	3
GTECH_MUX4	4-Bit Multiplexer	6	1	1	1	2
GTECH_MAJ23	Two-of-Three Majority Function	3	1	1	1	1
GTECH_OA22	2-lnput ORs into One 2-input AND	4	1	1	1	1
GTECH_XNOR4	4-lnput Exclusive-OR Gate	4	1	1	1	1
GTECH_XNOR3	3-lnput Exclusive-NOR Gate	3	1	1	1	1
GTECH_XOR4	4-lnput XOR Gate	4	1	1	1	1
GTECH_XOR3	3-lnput XOR Gate	3	1	1	1	1
GTECH_NOR5	5-lnput NOR Gate	5	1	1	1	2
GTECH_NAND8	8-lnput NAND Gate	8	1	1	1	3

Table 5: LUT Estimate Calculations for GTECH components.

Table 6 shows the LUT, DSP and BRAM estimates for GTECH components. Components with numbered suffixes indicate multi-personality implementations.

Component	LUT Estimates	DSP Estimates	BRAM Estimates
GTECH_AND2	1	0	0
GTECH_AND_NOT	1	0	0
GTECH_OAI21	1	0	0
GTECH_OA21	1	0	0
GTECH_NOT	1	0	0
GTECH_OR_NOT	1	0	0
GTECH_NAND3	1	0	0
GTECH_NOR8	2	0	0
GTECH_OR8	2	0	0
GTECH_XOR2	1	0	0
GTECH_NAND2	1	0	0
GTECH_AND3	1	0	0
GTECH_AND8	2	0	0
GTECH_OR2	1	0	0
GTECH_NOR2	1	0	0
GTECH_NOR3	1	0	0
GTECH_AOI22	1	0	0
GTECH_AND4	1	0	0
GTECH_AO22	1	0	0
GTECH_OR4	1	0	0
GTECH_AOI21	1	0	0
GTECH_AO21	1	0	0
GTECH_MUX2	1	0	0
GTECH_AOI222	1	0	0
GTECH_OR3	1	0	0
GTECH_NOR4	1	0	0
GTECH_OAI2N2	1	0	0
GTECH_AND5	2	0	0
GTECH_NAND4	1	0	0
GTECH_AOI2N2	1	0	0
GTECH_MUXI2	1	0	0
GTECH_BUF	1	0	0
GTECH_NOR5	2	0	0
GTECH_NAND5	2	0	0
GTECH_OAI22	1	0	0
GTECH_OR5	2	0	0
GTECH_XNOR2	1	0	0

Component	LUT Estimates	DSP Estimates	BRAM Estimates
GTECH_ADD_AB (1)	2	0	0
GTECH_ADD_AB (2)	0	1	0
GTECH_ADD_ABC (1)	2	0	0
GTECH_ADD_ABC (2)	0	1	0
GTECH_ISOLATCH_EN 1	1	0	0
GTECH_ISOLATCH_EN 0	1	0	0
GTECH_ISO1_EN1	1	0	0
GTECH_ISO1_EN0	1	0	0
GTECH_ISO0_EN1	1	0	0
GTECH_ISO0_EN0	1	0	0
GTECH_LSR0	2	0	0
GTECH_LD4_1	1	0	0
GTECH_LD4	2	0	0
GTECH_LD3	2	0	0
GTECH_LD2_1	1	0	0
GTECH_LD2	2	0	0
GTECH_LD1	2	0	0
GTECH_FJK3S	2	0	0
GTECH_FJK3	2	0	0
GTECH_FJK2S	2	0	0
GTECH_FJK2	2	0	0
GTECH_FJK1S	2	0	0
GTECH_FJK1	2	0	0
GTECH_FD4S	2	0	0
GTECH_FD48	6	0	0
GTECH_FD44	4	0	0
GTECH_FD4	2	0	0
GTECH_FD3S	2	0	0
GTECH_FD38	6	0	0
GTECH_FD34	4	0	0
GTECH_FD3	2	0	0
GTECH_FD2S	2	0	0
GTECH_FD28	6	0	0
GTECH_FD24	4	0	0
GTECH_FD2	2	0	0
GTECH_FD1S	2	0	0
GTECH_FD18	8	0	0
GTECH_FD14	8	0	0
GTECH_FD1	2	0	0

Component	**LUT Estimates**	**DSP Estimates**	**BRAM Estimates**
GTECH_INOUTBUF	1	0	0
GTECH_OUTBUF	1	0	0
GTECH_INBUF	1	0	0
GTECH_TBUF	1	0	0
GTECH_MUX8	3	0	0
GTECH_MUX4	2	0	0
GTECH_MAJ23	1	0	0
GTECH_OA22	1	0	0
GTECH_XNOR4	1	0	0
GTECH_XNOR3	1	0	0
GTECH_XOR4	1	0	0
GTECH_XOR3	1	0	0
GTECH_NOR5	2	0	0
GTECH_NAND8	3	0	0

Table 6: LUT, DSP and BRAM Estimates for GTECH components.

8 Appendix 2: Synthesizing and running workloads with Amber, an ARM-based core

This appendix details the process of synthesis and post-synthesis verification of the core, compiling in-package and custom workloads, and adapting them for use with Amber, an ARM v2a ISA based 32-bit core [1]. All directory paths in this Appendix are relative to the Amber package source obtainable from [1], unless otherwise mentioned.

8.1 Synthesis

We modified the Makefile included in `trunk/hw/fpga/bin` as part of the Amber package and created specific Makefile targets to perform FPGA synthesis using Xilinx ISE [14]. A copy of our Makefile has been provided [8]. Also, a copy of our synthesized Amber core, along with its dependencies, has been provided [10].

Before performing the steps below, ensure that Xilinx ISE [14] on a UNIX based system preferably Ubuntu, has been setup, environment-variables related to Xilinx ISE have been registered and the `XILINX_PATH` variable in the Makefile points to the location of the Xilinx ISE installation.

Creation of the NGC file

This synthesizes the top-level Verilog file in `trunk/hw/vlog/system/system.v` and its constituent modules.

To perform this step, do a `make xst` in `trunk/hw/fpga/bin`.

Creation of the NGD file

This creates an initial FPGA netlist based on the Xilinx UNISIM primitives library.

To perform this step, do a `make ngdbuild` in `trunk/hw/fpga/bin`.

It is recommended to retain the design hierarchy during synthesis. This can be done by setting the `keep_hierarchy` option in the Makefile to `YES`.

8.2 Post-Synthesis Verification

We performed post-synthesis verification by comparing the results of the pre-synthesis RTL design with that of the synthesized design. This was accomplished by simulating the same workload on both designs using ModelSim [13], ensuring it ran to completion in either case and comparing the contents of the general-purpose registers for both designs.

We faced issues using the synthesized design directly for simulation. We tracked the error at a module-level by using the process of elimination and replacing modules in the synthesized design with the corresponding RTL modules. In particular, we found mismatched port connections for the `system` module, which serves as the top-level module of the Amber design. After referring the RTL design and comparing port connections, we narrowed down on specific ports that were mismatched and fixed the issue. A copy of our synthesized design has been provided [9].

8.3 Compiling In-Package Workloads

The in-package workloads are located at `trunk/hw/tests`. In the steps below, we demonstrate how to compile and adapt the `add.S` workload included in the Amber package, in a format acceptable for execution on the core.

Compilation requires a GNU Toolchain intended for ARM cores installed on a UNIX based system, preferably Ubuntu. We used Sourcery CodeBench [12] for this purpose.

Get .o file from the .S file using gcc

In this step, we use the `-c` flag of `gcc` to perform only compilation (and not linking) to obtain the object file.

```
arm-none-linux-gnueabi-gcc -c -Os -march=armv2a
-mno-thumb-interwork -ffreestanding add.S -o add.o
```

Get .elf file from the .o file using ld

This step, which leads to an Executable and Linkable Format (ELF) file, can be performed using one of the following three ways:

(i) No explicit library linking: Generally suitable for workloads which do not require any external dependencies.

```
arm-none-linux-gnueabi-ld -Bstatic -Map add.map -o
add.elf -T sections.lds add.o
```

(ii) Link mini-libc explicitly: Intended for workloads which only require linking of function definitions declared in the mini-libc library included as part of the Amber package in `trunk/sw/mini-libc`.

The library includes definitions for the following functions:

`printf()`, `sprintf()`, `print()`, `prints()`, `printi()`, `memcpy()`, `_outbyte()`, `_inbyte()`, `strcpy()`, `strncpy()`, `strncmp()` and `_div()`.

```
arm-none-linux-gnueabi-ld -Bstatic -Map add.map -o
add.elf -T sections.lds trunk/sw/mini-libc/printf.o
trunk/sw/mini-libc/libc_asm.o add.o
```

The above command links `printf.o` and `libc_asm.o` as external libraries.

(iii) Link additional libraries: For workloads which require linking of function definitions not included in mini-libc.

```
arm-none-linux-gnueabi-ld -Bstatic -Map add.map -o
add.elf -T sections.lds add.o _aeabi_unwind_cpp_pr0.o
Sourcery_CodeBench/arm-none-linux-
gnueabi/libc/usr/lib/libm.a
Sourcery_CodeBench/lib/gcc/arm-none-linux-
gnueabi/4.8.1/libgcc.a
```

In the above command, we are linking `libm.a` (intended for mathematical/transcendental functions), `libgcc.a` (intended for standard C library functions), and `__aeabi_unwind_cpp_pr0.o` (exception unwinding, required in in cases of an exception, for instance, a potential divide-by-0 operation in the code). A copy of our `__aeabi_unwind_cpp_pr0.c` file has been provided [8]. `Sourcery_CodeBench` represents the path to the Sourcery CodeBench installation.

Get .mem file from the .elf file

The `.mem` format serves as a functional format which can directly be read by the core for execution. The contents of a `.mem` file are arranged in the following format:

<memory_address>	<contents>

```
cd trunk/sw/tools
amber-elfsplitter add.elf > add.mem
```

8.4 Custom Workloads for Amber

In this section, we discuss about workloads obtained from external sources (referred to as custom workloads), compiling and adapting them for Amber.

8.4.1 Workload Sources

We narrowed down on several general-purpose workloads that could be adapted for use with Amber. A copy of the workloads listed below is also available at [9].

- Program to compute PI by probability [2].
- LINPACK_BENCH, a C program which measures the time needed to factor and solve a large dense linear system of equations [3].
- SUM_MILLION, a C program which sums the numbers from 1 to 1,000,000, times the computation and computes the corresponding MegaFLOPS rate [4].
- Dhrystone integer benchmark [5].
- Whetstone floating point benchmark [6].
- LMS adaptive signal enhancement used in Real-Time DSP [7].

Alternatively, one may also develop custom workloads in C, tailored to test specific components of the core, such as a simplistic sequential number summation program [15], similar to SUM_MILLION.

8.4.2 Adapting Custom Workloads for Amber – I

This step involves removal of system calls from the workload, inclusion of Amber-related header files and adding test-status code to the workload. While removal of system-calls and inclusion of Amber-related header files can be done using the `.c` source of the workload, adding test-status code requires conversion to a `.S` source file from the `.c` source. We thus split this section into two parts: Part I (code-modifications before conversion to `.S`) and Part II (code-modifications after conversion to `.S`).

Removal of system calls

While booting a non-virtual memory variant of Linux on the core is possible [1], it is a tedious task. We may avoid loading an OS on the core by ensuring that the workloads are suitable for execution using the "bare metal" environment of the core. To this effect, we perform certain code modifications in the workloads, mainly elimination of system calls. Common system calls found in workloads include input and output buffering functions.

Programming constructs that interact with `stdin` and `stdout` which consist of function calls such as `printf()`, `puts()`, `scanf()`, `getch()`, `gets()` etc. as well as `ifstream` and `ofstream` objects, should be removed. File I/O operations should also be eliminated. For instance, input data can be hardcoded to be passed on directly through the code rather than being read from a file. Similarly, code which writes the program output to a file should be removed. Dynamic memory allocation using `malloc` or `new` should be replaced by static memory allocation. In addition, any other system calls should be eliminated.

Including Amber-related header files

The header files listed below, located at `trunk/hw/tests`, need to be included in the `.S` or `.c` source of the workload.

```
#include "amber_registers.h"
#include "amber_macros.h"
```

8.4.3 Workload Conversion from C to Assembly

Custom workloads are generally available as a `.c` source rather than the assembly-based `.S` format. Conversion to assembly facilitates test-status addition described in Section 8.4.4 and permits using the procedure for workload compilation outlined in Section 8.3.

Get .S file from the .c file

This step uses the `-S` flag of `gcc` to obtain the assembler output in the form of a `.S` file.

```
arm-none-linux-gnueabi-gcc -c -S -march=armv2a
-mno-thumb-interwork -ffreestanding add.c -o add.S
```

8.4.4 Adapting Custom Workloads for Amber – II

This step involves adding test-status code to enable the core to identify if the workload ran to completion. The code snippet below should be appended at the end of the `.S` file of the custom workload. An in-package workload such as `add.S` within `trunk/hw/tests` can be referred to understand the implementation of this step.

```
testpass:
ldr r11, AdrTestStatus
mov r10, #17
str r10, [r11]
b   testpass
/* Write 17 to this address to generate a Test Passed
message */
AdrTestStatus:  .word  ADR_AMBER_TEST_STATUS
```

8.4.5 Compiling Custom Workloads

After converting the custom workloads to `.S` format and adapting them for use with Amber, we may compile the workloads using the same steps as in the case of in-package workloads, outlined in Section 8.3.

8.5 Running Workloads on Amber

Once the workloads have been compiled and the `.mem` file for the workload has been generated, we can run the workloads on the core by editing the `system_config_defines.v` file in `trunk/hw/vlog/system` and changing the `BOOT_MEM_FILE` parameter to reflect the absolute or relative path of the `.mem` file.

We can then simulate the design using the testbench for the RTL design (`tb.v`) or that for the synthesized design (`tb_post.v`). A copy of `tb.v` [11] and `tb_post.v` [10] has been provided.

8.6 Setup Summary

In this section, we summarize the process of setting up an environment that would enable simulating the core by providing a step-by-step process below.

(i) Obtain the source files of the RTL design from [11] or those of the synthesized design from [10].

(ii) Install Sourcery CodeBench [12] to enable compilation of workloads. Compile an in-package workload using the instructions in Section 8.3 or a custom workload following the instructions in Section 8.4.

(iii) Upon compiling the workload to a `.mem` file, follow the instructions in Section 8.5 to point the design to the `.mem` file of the workload. Post this step, use an HDL simulator such as ModelSim [13] to simulate the corresponding testbench file as discussed in Section 8.5.

(iv) Run the simulation to completion, i.e., until the test-bench reaches a `$finish`, which occurs when the status-code described in Section 8.4.4 finishes execution at the end of the workload.

(v) Once the simulation terminates, the contents of the core's general-purpose registers can be viewed in the simulation transcript.

8.7 RAM Generation

RAM generation for the core can be done using Xilinx CORE Generator [16] by following the instructions at `trunk/hw/vlog/xs6_ddr3/README.txt`. This step is only required for implementing Amber on an FPGA development board.

8.8 Environment Details

ModelSim PE Student Edition 10.3 [13]
Revision: Jan 6, 2014 (2014.01)

Xilinx ISE 14.5 [14]
Revision: Apr 3, 2013

Sourcery CodeBench IDE [12]
Revision: Nov, 2013 (2013.11-50)

8.9 References

[1] Amber ARM-compatible core, Retrieved May 4, 2014, from: http://opencores.org/project,amber

[2] Program to compute PI by probability, Retrieved May 19, 2014, from: http://www.60bits.net/msu/mycomp/bench.htm

[3] The LINPACK Benchmark, Retrieved May 21, 2014, from:

http://people.sc.fsu.edu/~jburkardt/cpp_src/linpack_bench/linpack_bench.html

[4] SUM_MILLION, Retrieved May 21, 2014, from:

http://people.sc.fsu.edu/~jburkardt/c_src/sum_million/sum_million.html

[5] Dhrystone integer benchmark, Retrieved May 21, 2014, from:

http://classes.soe.ucsc.edu/cmpe202/workloads/standard/dhrystone.c

[6] Whetstone floating point benchmark, Retrieved May 22, 2014, from:

http://www.netlib.org/benchmark/whetstone.c

[7] LMS adaptive signal enhancement, Retrieved May 22, 2014, from:

http://www.mrtc.mdh.se/projects/wcet/wcet_bench/lms/lms.c

[8] Makefile, Retrieved May 22, 2014, from: http://amanchadha.com/projects/amber/Makefile

[9] Workloads, Retrieved May 22, 2014, from: http://amanchadha.com/projects/amber/Workloads

[10] Synthesized Amber Repository, Retrieved May 22, 2014, from: http://amanchadha.com/projects/amber/Synthesized_Amber

[11] RTL Amber Repository, Retrieved May 26, 2014, from: http://amanchadha.com/projects/amber/RTL_Amber

[12] Sourcery CodeBench - Mentor Graphics, Retrieved May 26, 2014, from: http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/overview

[13] ModelSim PE Student Edition - Mentor Graphics, Retrieved May 26, 2014, from: http://www.mentor.com/company/higher_ed/modelsim-student-edition

[14] Xilinx ISE 14.5, Retrieved May 26, 2014, from: http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools/v2012_4---14_5.html

[15] Custom Sequential Number Summation Program, Retrieved May 26, 2014, from: http://amanchadha.com/projects/amber/Workloads/sum_million_custom.c

[16] Xilinx CORE Generator System, Retrieved May 27, 2014, from: http://www.xilinx.com/tools/coregen.htm