Article

Building Complete Heterogeneous Systems-on-Chip in C: From Hardware Accelerators to CPUs

Qilin Si, Santosh Shetty and Benjamin Carrion Schaefer *

Department of Electrical and Computer Engineering, The University of Texas at Dallas, Richardson, TX 75080, USA; qilin.si@utdallas.edu (Q.S.); santosh.shetty@utdallas.edu (S.S.)
* Correspondence: schaferb@utdallas.edu; Tel.: +1-972-883-4531

Abstract: High-Level Synthesis (HLS) dramatically accelerates the design and verification of individual components within larger VLSI systems. With most complex Integrated Circuits (ICs) being now heterogeneous Systems-on-Chip (SoCs), HLS has been traditionally used to design the dedicated hardware accelerators such as encryption cores and Digital Signal Processing (DSP) image processing accelerators. Unfortunately, HLS is a single process (component) synthesis method. Thus, the integration of these accelerators has to be performed at the RT level (Verilog or VHDL). This implies that the system-level verification needs to be performed at lower levels of abstraction, which significantly diminishes the benefits of using HLS. To address this, this work presents a methodology to generate entire heterogeneous SoCs in C. This work introduces two main contributions that enable this: first, an automatic bus generator that generates a synthesizable behavioral description of standard on-chip buses and, second, a library of synthesizable bus interfaces that allow any component in the system to send or receive data through the bus. Moreover, this work investigates the generation of processors and interfaces (peripherals) at the behavioral level as these are important parts of any SoCs, but have long been thought not to be efficiently synthesizable using HLS. Generating complete SoCs in C has significant advantages over traditional approaches. First, it enables the generation of fast cycle-accurate simulation models of the entire SoC, making the verification faster and easier. Second, it allows completely isolating the bus implementation details from the developers' view, allowing the change between bus protocols with only minor changes in the designers' code. Thirdly, it allows generating different SoC variants quickly by only changing the HLS synthesis options. Experimental results highlight these benefits.

Keywords: high-level synthesis; heterogeneous SoC; bus generator; bus interfaces; fast simulation models; micro-architectural design space exploration; system-level exploration

check for updates

Citation: Si, Q.; Shetty, S.; Carrion Schaefer, B. Building Complete Heterogeneous Systems-on-Chip in C: From Hardware Accelerators to CPUs. *Electronics* **2021**, *10*, 1746. https://doi.org/10.3390/electronics10141746

Academic Editor: Akash Kumar

Received: 16 June 2021
Accepted: 16 July 2021
Published: 20 July 2021

1. Introduction

With the breakdown of Dennard's scaling, computer architectures have dramatically changed to meet stringent power budgets. Most complex Systems-on-Chip (SoC) are now heterogeneous, as shown in Figure 1. These include embedded processors, on-chip memory, different types of interfaces, and numerous dedicated hardware accelerators. These heterogeneous SoCs are faster and consume orders of magnitude less power than general-purpose systems. Unfortunately, the main problem with these heterogeneous systems is that they are much more difficult to design and verify. This is particularly important in a time where consumers demand more powerful electronic products in shorter and shorter time frames. Thus, new design methodologies are required to enable the generation and verification of these complex systems faster.

Raising the level of abstraction from the RT level to the behavioral level seems a promising approach. It has been shown that a single line of C code leads to 7× more gates than a single line of Verilog code [1]. This implies less coding and faster verification. Moreover, commercial High-Level Synthesis (HLS) tools have shown in numerous case studies that they can generate RTL code (Verilog or VHDL) that rivals hand-optimized code [1]. HLS also allows generating a variety of different micro-architectures for the given behavioral description by simply setting different synthesis options. Based on these advantages, it makes sense to build these complex heterogeneous SoCs using only HLS. Figure 1 shows the envisioned design flow with the highlighted modules in orange to be designed using HLS.

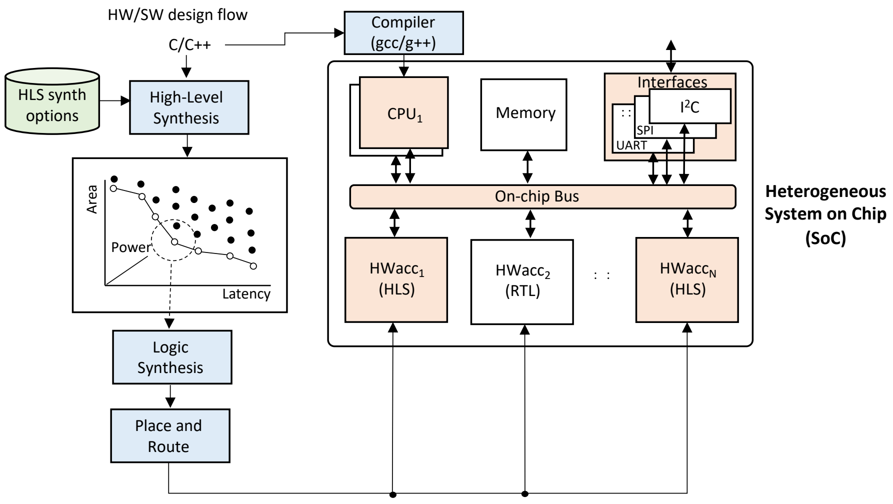


Figure 1. Overview of a typical heterogeneous SoC and its design flow.

Embedded memories shown in the figure can be easily modeled using HLS also as arrays, but would require a memory compiler. Moreover, because most SoCs are built incrementally, they might also include numerous RTL IPs such as the second hardware accelerators shown in the figure ($HWacc_2$). In this case, previous work on RTL to C compilation optimized for HLS could be used to generate synthesizable C code for these IPs [2–4]. Another approach could be to use Verilog's Programming Language Interface (PLI), which allows cosimulating Verilog with C [5]. Although faster than the RTL-only simulation, this approach is still too slow to simulate complex modern SoCs [6,7].

One of the problems with HLS is that it is a single-process synthesis method. Basically, it allows synthesizing individual modules in the SoC that later need to be *stitched* together. This implies that the system integration, and thus the full system verification, needs to be performed at a lower level of abstraction, e.g., the RT level. This defeats much of the benefits of raising the level of abstraction. It is therefore important to provide a path to allow the generation of complete heterogeneous SoCs at the behavioral level. This has numerous advantages over traditional methods based on low-level Hardware Description Languages (HDLs): First, it accelerates the design process (less coding is needed). Second, the entire system can be simulated faster making use of transaction-level or fast cycle-accurate simulation models. Third, it enables the exploration of different bus interface types by decoupling the bus interface from the functionality of each component in the system. This allows easily changing between bus types (e.g., AMBA AHB or AXI) and bus characteristics (e.g., arbiter type and bus bitwidth) and evaluating the effect of these on the overall system performance and power. Finally, it allows quickly generating SoCs with different area, performance, and power profiles by simply setting different HLS synthesis option combinations. It also allows quickly retargeting the SoCs from one hardware platform to another, e.g., ASIC to FPGA and vice versa, with minimum effort [8].

Some commercial HLS vendors already provide different types of system-level design features. The main problem is that they are proprietary. Some use dedicated pragmas to specify the bus interface (e.g., Xilinx Vivado [9,10]) or even encrypt the generated RTL code to avoid being reverse engineered [11].

This work extends the work presented in [12], which provided a basic path to enable the generation of complete SoCs through two main contributions. First is a bus generator that generates synthesizable SystemC code for a given standard bus. Currently, only AMBA AHB-LITE and AHB are supported, but in the future, other bus structures such as AMBA AXI or different types of Networks-on-Chip (NoCs) could be easily added. Second is a library of synthesizable APIs to abstract away the bus interface in each of the SoC components to read and write from and to the bus. These APIs are a library of functions that are synthesized as the bus interfaces of the masters and the slaves in the system. This facilitates the tedious and error-prone task of building the bus interfaces manually and also enables easily switching between bus types as the users only need to modify the function call. The bus generator and bus interfaces' APIs all generate synthesizable SystemC code that complies with Acellera's latest synthesizable SystemC subset specifications [13], thus ensuring that they are portable across most commercial HLS tools. It should be noted that this work assumes that the HW/SW partition has already been made a priori, although being able to build and simulate larger complex systems fast could serve to guide the system designer in determining the most appropriate HW/SW partition and evaluate the effect of this partition on the area and performance.

This previous work is extended here by presenting how to describe processors and interfaces efficiently using HLS, which has been until now perceived as not being possible or leading to suboptimal results as compared to RTL descriptions. In summary, this work makes the following contributions:

- It introduces a behavioral bus generator that generates synthesizable SystemC code for HLS for standard AMBA buses (AHB and AHB-Lite) and presents a library of synthesizable bus interface APIs that allow decoupling the bus interface for each component in the SoC from the computational part;
- It presents extensive experimental results highlighting the benefits of this approach vs. traditional approaches based on low-level HDLs.

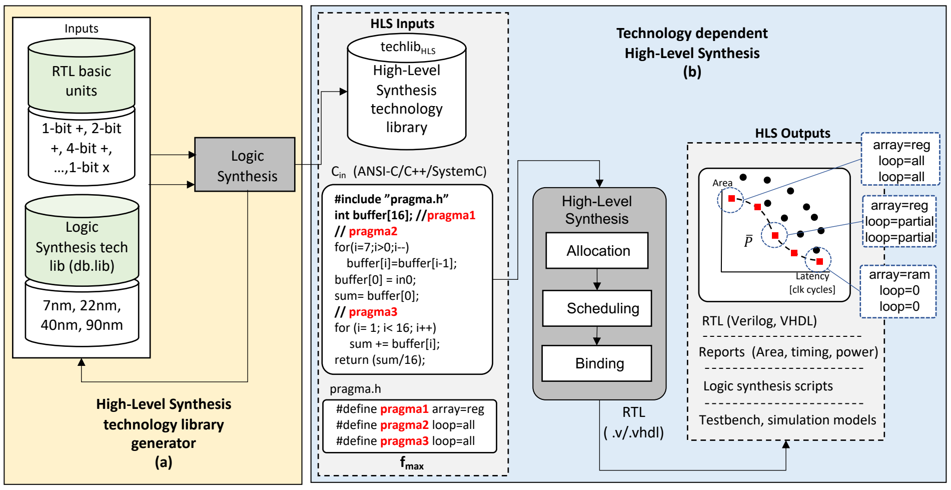
2. High-Level Synthesis

Before we proceed with describing the proposed work, it is important to review what HLS is and how it works. Figure 2 shows an overview of the complete HLS process. HLS can be described as a process to convert untimed behavioral descriptions into efficient hardware that implements that behavior. The inputs to the HLS process, as shown in Figure 2b, are the behavioral description to be synthesized in, e.g., ANSI-C, C++, or SystemC, a technology library ($techlib_{HLS}$) that contains the area and delay information of basic operations (i.e., adders and multipliers of different bitwidths), a target synthesis frequency (f_{max}), and a set of synthesis directives in the form of pragmas (pragma.h). These synthesis directives are extremely important as they allow the designer to control the synthesis process. In particular, these directives control how to synthesize arrays (RAM or registers), loops (unroll, partially unroll, not unroll, or pipeline), and functions (inline or not). In the example shown in Figure 2b, the code snippet contains one array and two loops.

The HLS process then parses the behavioral description and constraints and performs three main steps: (1) resource allocation, (2) scheduling, and (3) binding. In the resource allocation stage, the number and type of hardware resources from $techlib_{HLS}$ are extracted. In the scheduling phase, the different operations in the behavioral description are assigned to individual clock steps based on the number of available resources, and finally, in the binding stage, the hardware resources are bound to different operations in the scheduled operations.

To achieve high-quality RTL, the HLS process needs to know the accurate delay of the different functional units (FUs) that are mapped to different operations in the behavioral description. For this, commercial HLS tools provide a library characterizer shown in Figure 2a that generates the technology library for the HLS process. This library characterizer synthesizes (logic synthesis) different basic units with different bitwidths.

Basically, the library characterizer automatically generates the RTL code of these basic units, synthesizes it with the target technology specified by the user, which has to match the technology used during HLS, and back-annotates the area and delay information reported by the logic synthesis tools into the HLS technology library ($techlib_{HLS}$). This process needs to be executed before the HLS process is executed, and although time consuming, it only needs to be execute once. It should be noted that FPGA vendors do not provide this flow as they pregenerate these technology libraries for their particular FPGAs and include them with their HLS tool.


Inputs
RTL basic units
1-bit +, 2-bit +, 4-bit +, ...,1-bit x
Logic Synthesis tech lib (db.lib)
7nm, 22nm, 40nm, 90nm
Logic Synthesis
High-Level Synthesis technology library generator
(a)
HLS Inputs
techlib$_{HLS}$
High-Level Synthesis technology library
C$_{in}$ (ANSI-C/C++/SystemC)
pragma.h
f$_{max}$
Technology dependent High-Level Synthesis
(b)
High-Level Synthesis
Allocation
Scheduling
Binding
RTL
(.v/.vhdl)
HLS Outputs
Area
$\bar{P}$
Latency [clk cycles]
array=reg loop=all loop=all
array=reg loop=partial loop=partial
array=ram loop=0 loop=0
RTL (Verilog, VHDL)
Reports (Area, timing, power)
Logic synthesis scripts
Testbench, simulation models

```
#include "pragma.h"
int buffer[16]; //pragma1
// pragma2
for(i=7;i>0;i--)
   buffer[i]=buffer[i-1];
buffer[0] = in0;
sum= buffer[0];
// pragma3
for (i= 1; i< 16; i++)
    sum += buffer[i];
return (sum/16);
```

```
#define pragma1 array=reg
#define pragma2 loop=all
#define pragma3 loop=all
```

Figure 2. High-Level Synthesis (HLS) overview.

The output of the HLS process is the RTL code (Verilog or VHDL) and a set of reports that summarize the area and performance of the synthesized circuit. Commercial HLS tools also generate synthesis scripts to interface the HLS tool with the logic synthesis process and test benches to verify the generated circuit, as shown on Figure 2b. It also shows different micro-architectures generated when different combinations of the synthesis directives are specified. Out of all the combinations, the designer is typically only interested in the combinations that lead to the Pareto-optimal designs ($\bar{P}$). This unique advantage of HLS has received much attention recently. The main problem with finding these Pareto-optimal configurations is that the search space grows supralinearly with the number of explorable operations (loops, arrays, and functions), making exhaustive enumerations of all combinations impractical. Thus, multiple fast heuristic methods have been proposed to find these Pareto-optimal designs quickly. A recently published survey summarized them [14]. In this work, we use this to highlight one of the salient benefits of building a complete heterogeneous SoC in C. Using different synthesis directives, the proposed flow cannot quickly generate SoCs with different area and performance trade-offs.

Table 1 summarizes the main commercial HLS tools available to date classified in ASIC and FPGA tools. The table also shows the input language support. It can be observed that except Intel's HLS compiler, all other HLS tools accept SystemC. This is why in this work, we target generating entire SoCs using SystemC rather than C or C++, as IEEE has standardized SystemC in their 1666 Language Reference Manual (LRM), while most C or C++ languages used also include vendor specific constructs such as their own data types.

Table 1. Overview of commercial HLS tools and their supported input languages.

	HLS Tool	Input Languages
ASIC	Cadence Stratus [15]	C/C++/SystemC
	Mentor Catapult [9]	C++/SystemC
	NEC CyberWorkBench [11]	C/BDL/SystemC
FPGA	Vivado HLx [10]	C/C++/SystemC
	HLS Compiler [16]	C++

3. Related Work

The increase in the level of VLSI design abstraction, from the RT level to the behavioral level, has mainly been used to design the hardware accelerators within the SoCs. Because these have to interface with the rest of the system, different approaches have been proposed to facilitate this. Commercial HLS tools have followed two main approaches. ASIC vendors such as Cadence Stratus [15], Mentor Catapult [9], and NEC CWB [11] provide, similarly to this work, a library of synthesizable APIs that need to be included in the source code. The main problem is that these are proprietary and, in some cases, generate encrypted code [9,11]. FPGA vendors such as Xilinx follow a different approach, making use of pragmas to specify the bus interface [10]. The HLS tool parses these pragmas and generates the corresponding bus interface for the given accelerator. In all these cases, the HLS tool only generates the bus interface such that the hardware accelerator can be easily integrated within the SoC. They do not generate the bus itself, as we do in this work. In the FPGA case, this is obviously not needed as newer programmable SoC FPGAs already include the bus to connect the embedded processors with the reconfigurable fabric onto which the accelerator is mapped. To generate standard on-chip buses, companies typically rely on bus generators such as [17] that generate synthesizable RTL code, which in some cases is also encrypted. The main drawback of this approach is that it forces performing the integration and verification at the RT level. As we show in Section 6, being able to perform the integration at the behavioral level facilitates the verification of the entire system through faster simulation models. It also allows exploring different system parameters faster to optimize the SoC.

Academic efforts in this domain have mainly targeted the automatic generation of NoCs. In [18], the authors introduced FPGA-to-CUDA (FCUDA)-NoC, a NoC generator that takes in CUDA code and custom network parameters as inputs and produces synthesizable Register-Transfer Level (RTL) code for the entire NoC system. The authors in [19] targeted HLS and automated the NoC generation to improve the access to on-chip memories that cannot be determined at HLS compile time. Other approaches aim at generating customizable memory banking schemes only for a particular accelerator in the system [20]. Other recent approaches target specifically the efficient execution of neural networks in SoCs [21] and use NoCs as interconnects. The main problem with all these approaches is that they generate synthesizable RTL code and typically require some sort of RTL wrappers for the final integration. This does not allow fully leveraging the benefits of behavioral SoC design.

The proposed work allows building a complete SoC at the behavioral level, which in turn enables performing quick cycle-accurate simulations of the entire system, thus verifying functional timing issues, as well as, e.g., deciding on the overall system architecture faster.

4. Proposed Behavioral SoC Design Flow

The proposed behavioral SoC development flow was based on four main features. The first is a bus generator that generates synthesizable SystemC code for the actual SoC bus. The second is a library of synthesizable APIs to allow any component in the SoC described as a behavioral description to access the bus. These APIs are further divided into master APIs and slave APIs. These two features allow *stitching* together all the components in the SoC, simulating the system, and finally, generating the RTL code for each component in the system. The third feature that is needed to generate complete SoCs in C are embedded processors described in C. For this, we provide a library of embedded behavioral processors. Lastly, every SoC has multiple synchronous and asynchronous interfaces to communicate with the external world. This also includes memory interfaces. In this case, we made use of the ability of commercial HLS tools to time the behavioral descriptions to efficiently optimize these interfaces and create a library of these external interfaces.

Figure 3 shows an overview of the complete flow. The next subsections explain how the bus generator, the synthesizable APIs, the embedded processor, and the library of external interfaces work in detail.

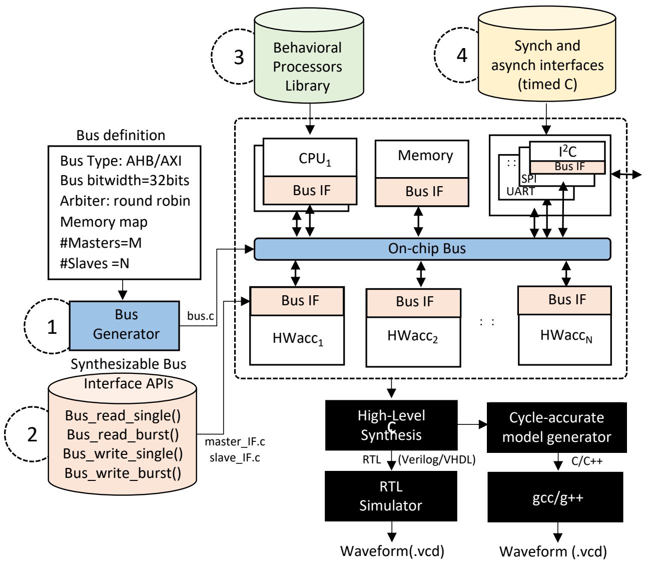


Figure 3. Overview of the proposed flow and target architecture.

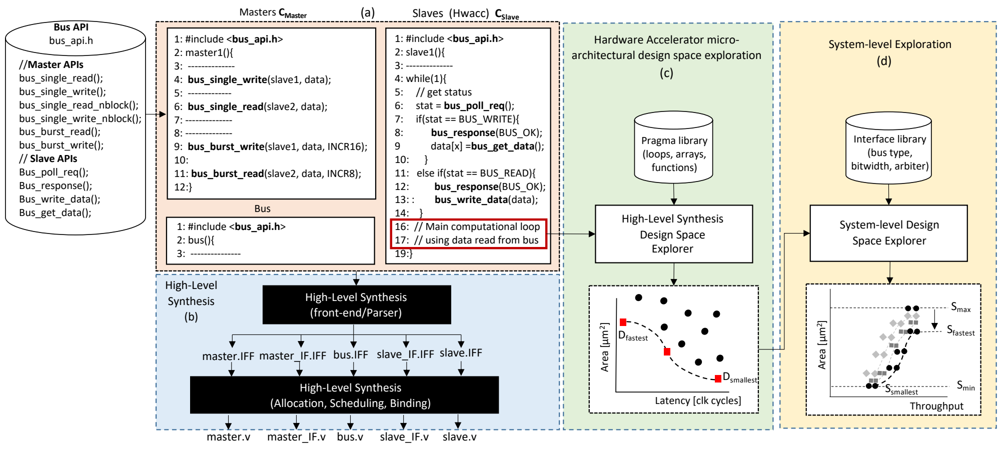
❶ **Behavioral bus generator:** The input to the behavioral bus generator is a bus specification file (.bsf) that contains all the specifications of the on-chip bus. Three main sections are required, as summarized in Table 2. Section 1 includes the bus protocol, e.g., AMBA, AHB, AHB-Lite, or AXI, although currently, only AHB and AHB-Lite are supported, the bus bitwidth, the type of arbiter (fixed or round robin), and the names and number of masters and slaves. In case the system only has a single master, then it is assumed that the SoC follows the AMBA AHB-Lite protocol (single-master system, which does not require any arbiter and, hence, is much simpler). Sections 2 and 3 describe the bus interfaces in each of the masters and slaves in the systems, e.g., if single mode or burst mode is enabled and, in the case of the slaves, the address range of each of them.

The bus generator was implemented in Python. It reads this bus specification file and outputs a synthesizable behavioral description of the on-chip bus in synthesizable SystemC code. The advantages of generating the description in SystemC is that it follows the SystemC synthesizable subset specifications published by Accellera [13]. This guarantees that every commercial HLS tool can synthesize it into efficient RTL.

Table 2. Bus specification file (.bsf).

Bus Type		
Parameter	**Options**	**Description**
Type	AHB ∣ AXI	On-chip bus protocol selection
Width	8 ∣ 16 ∣ 32 ∣ 64	Bus bitwidth
Arbiter	fixed ∣ round robin	Type of arbiter
Masters	M1, M2, . . . , Mn	Names and number of masters in the system
Slaves	S1, S2, . . . , Sn	Names and number of slaves in the system
Master definitions (one entry for each master)		
Mode	single ∣ burst	Send/receive individual data or burst mode
Slave definitions (one entry for each slave)		
Mode	single ∣ burst	Send/receive individual data or burst mode
Address	$Addr_{st} \mid Addr_{end}$	Address range of the given slave

❷ **Synthesizable APIs:** The second feature that facilitates the generation of behavioral SoCs is a library of synthesizable APIs. These APIs allow any behavioral application in the SoC to simply call the API to read from the bus or write to the bus. This approach fully abstracts away the bus interface details from the main functional description of each component. Figure 4a shows an example of a system composed of a single master and a single slave and how these two components interact with each other through the developed API.


Masters C_{Master}
(a)
Slaves (Hwacc) C_{Slave}
Bus API
bus_api.h
Bus
High-Level Synthesis (b)
High-Level Synthesis (front-end/Parser)
master.IFF
master_IF.IFF
bus.IFF
slave_IF.IFF
slave.IFF
High-Level Synthesis (Allocation, Scheduling, Binding)
master.v
master_IF.v
bus.v
slave_IF.v
slave.v
Hardware Accelerator micro-architectural design space exploration (c)
Pragma library (loops, arrays, functions)
High-Level Synthesis Design Space Explorer
Area [μm²]
$D_{fastest}$
$D_{smallest}$
Latency [clk cycles]
System-level Exploration (d)
Interface library (bus type, bitwidth, arbiter)
System-level Design Space Explorer
Area [μm²]
S_{max}
$S_{fastest}$
S_{min}
$S_{smallest}$
Throughput

```
//Master APIs
bus_single_read();
bus_single_write();
bus_single_read_nblock();
bus_single_write_nblock();
bus_burst_read();
bus_burst_write();
// Slave APIs
Bus_poll_req();
Bus_response();
Bus_write_data();
Bus_get_data();
```

```
#include <bus_api.h>
master1(){
 -------------
 bus_single_write(slave1, data);
 ------------
 bus_single_read(slave2, data);
-------------
-------------
 bus_burst_write(slave1, data, INCR16);

bus_burst_read(slave2, data, INCR8);
}
```

```
#include <bus_api.h>
bus(){
 --------------
```

```
#include <bus_api.h>
slave1(){
-------------
while(1){
   // get status
   stat = bus_poll_req();
   if(stat == BUS_WRITE){
       bus_response(BUS_OK);
       data[x] =bus_get_data();
   }
  else if(stat == BUS_READ){
       bus_response(BUS_OK);
       bus_write_data(data);
  }
 // Main computational loop
 // using data read from bus
}
```

Figure 4. Example of synthesizable communication APIs (one master and one slave) and the HLS DSE flow to optimize the micro-architecture of the slave (hardware accelerator). (**a**) Behavioral descriptions for individual components of SoC. (**b**) Result of individual HLS of each component. (**c**) Individual component HLS design space exploration. (**d**) System-level exploration by combining different micro-architectures of each individual component.

Table 3 summarizes the main APIs for the AMBA AHB bus case classified into master and slave APIs. The master has the option to send or receive individual data or multiple data at the same time through burst mode. By default, the API call is blocking. This implies that the master will be kept waiting until the data have been fully written or read. Two nonblocking additional API functions are provided to allow the master to continue its operation while the data are been sent over the bus. Figure 4a highlights how simple it is to use these APIs in the C_{master} example. The slave side is slightly more complicated, as it requires continuously monitoring the bus to check if any of the masters are trying to initiate a communication. Thus, additional APIs are needed for this, as shown in Table 3 (slave). The slave side has to continuously check for any requests from the master and respond to the request by either returning: OK, ERROR, RETRY, and SPLIT, as per the AHB specifications. Once the data have been read, the slave continues using them in its main computational loop. When finished, it then waits for the master to request the data to be sent back.

Table 3. Synthesizable communication APIs' overview.

Master	
API Name	**Description**
bus_single_read()	Blocking single data read from the slave
bus_single_read_nblock()	Nonblocking single data read from the slave
bus_burst_read()	Blocking burst read from the slave
bus_single_write()	Blocking single data write to the slave
bus_single_write_nblock()	Nonblocking single data write to the slave
bus_burst_write()	Blocking burst write to the slave
Slave	
bus_poll_req()	Poll status of the bus
bus_response()	Respond to the master's request
bus_get_data()	Read data from the bus
bus_write_data()	Write data to the bus

❸ **Behavioral processors' library:** Every SoC has at least one embedded processor. These embedded processors could again be heterogeneous such as ARM's big.LITTE processor pairs or multiple copies of the same one. It is therefore important to investigate if it is possible to design these processors also in C and include them in the heterogeneous SoC proposed.

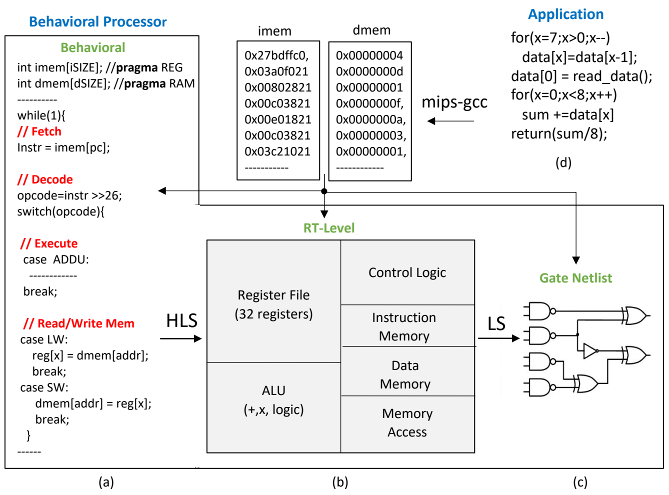
Some previous work indicated that describing a fixed architecture such as a processor using HLS leads to suboptimal implementations. Table 4 shows the results for two different types of processors implemented using HLS and manually described in Verilog. The first is the configurable NEC processor reported in [1], while the second is a scalar MIPS processor from the CHStone benchmark suite [22]. In both cases, it can be observed that the behavioral processor requires less lines of code to be described (7.6 and 6.4× less lines for the NEC and MIPS processor, respectively). The cycle-accurate simulation is also faster as HLS allows generating fast cycle-accurate simulation models, which are on average 203 and 120× faster than an RTL simulation. The drawback of describing the embedded processor in C is that HLS leads to slightly larger area overheads. 5% for the NEC processor and 2% for the MIPS processor. The authors in [23] also reported similar results for an RISC V processor compared to a manually optimized processor.

Table 4. Processor design comparison (C vs. RTL).

NEC [1]	**C**	**RTL**	**Ratio**
#lines	1300	9200	7.6×
Sim speed	61 kc/s	0.3 kc/s	203×
Size	19,000 µm^2	18,000 µm^2	1.05×
MIPS Scalar [22]	**C**	**RTL**	**Ratio**
#lines	494	3,161	6.4×
Sim speed	108 kc/s	0.9 kc/s	120×
Size	15,231 µm^2	14,913 µm^2	1.02×

Figure 5 shows an overview of how to map applications onto an embedded processor described in C. In particular, this is the MIPS processor used in this work, taken from [22] and converted to SystemC to make it portable to any HLS tool. Figure 5a shows a snippet of the behavioral processor, which uses a very simple switch-case statement to decode the different instructions. It can also be observed that the processor has two arrays, imem and dmem, which serve as instruction and data memories. One of the advantages of describing the processor in C is that these arrays can now be synthesized as registers or RAM depending on their size by setting different synthesis directives, as shown in the figure. Figure 5b shows the generated processor block diagram after HLS and Figure 5c the gate netlist after logic synthesis. Finally, Figure 5d shows the application that is compiled using a MIPS cross-compiler and stored in the instruction memory (imem).

Based on this, we can generate a library of embedded processors that currently includes the scalar MIPS processor from the CHStone benchmark [22] and an RISC V core. These processors serve as masters in our heterogeneous SoC design.



```
int imem[iSIZE]; //pragma REG
int dmem[dSIZE]; //pragma RAM
----------
while(1){
// Fetch
Instr = imem[pc];

// Decode
opcode=instr >>26;
switch(opcode){

 // Execute
 case  ADDU:
  -----------
 break;

 // Read/Write Mem
 case LW:
    reg[x] = dmem[addr];
    break;
 case SW:
    dmem[addr] = reg[x];
    break;
 }
------
```

```
imem          dmem
0x27bdffc0,   0x00000004
0x03a0f021    0x0000000d
0x00802821    0x00000001
0x00c03821    0x0000000f,
0x00e01821    0x0000000a,
0x00c03821    0x00000003,
0x03c21021    0x00000001,
-----------   -----------
```

```
for(x=7;x>0;x--)
   data[x]=data[x-1];
data[0] = read_data();
for(x=0;x<8;x++)
   sum +=data[x]
return(sum/8);
```

Figure 5. Simple behavioral MIPS scalar processor. (**a**) Behavioral description of the processor. (**b**) Register transfer Level view of the synthesized processor. (**c**) Gate netlist view of the processor. (**d**) Application example to be executed on the processor stored into instruction and data memories.

❹ **External interfaces using timed C:** HLS has traditionally been used to synthesize data-intensive applications such as filters and image processing applications, as shown in Figure 6. Numerous works have reported excellent results using HLS for these types of arithmetic intensive applications [24]. Other control-intensive applications that have complex control dependencies can also be synthesized efficiently with modern HLS tools through different types of speculation. In contrast, controller applications such as interfaces (USB, PCI, etc.) have been traditionally difficult to design and efficiently implement with HLS because they often follow specific protocols that require specific timing signals. These protocols are difficult to implement using regular untimed C/C++. To address this and enable the efficient synthesis of any application, commercial HLS tools (in particular, the ASIC ones) allow manually timing the C description, either the complete description, through what is called *manual* scheduling mode (C code is manually scheduled), or only portions, called *mixed* or *interface* synthesis mode. In this case, only a portion of the behavioral description is timed. This allows efficiently building the interface portion of the synthesized circuit, while allowing the HLS tool to automatically synthesize the rest of the description.

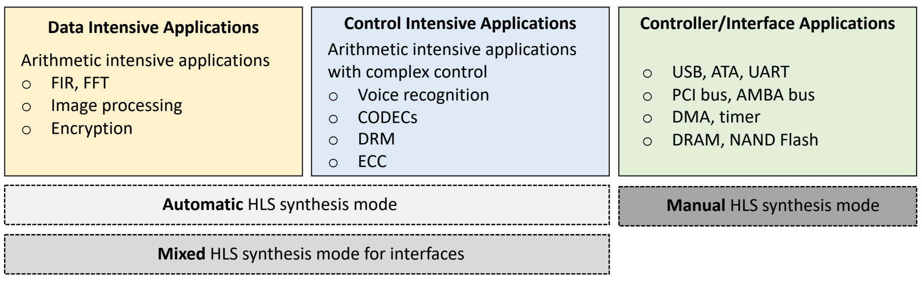

Data Intensive Applications
Arithmetic intensive applications
○ FIR, FFT
○ Image processing
○ Encryption
Control Intensive Applications
Arithmetic intensive applications with complex control
○ Voice recognition
○ CODECs
○ DRM
○ ECC
Controller/Interface Applications
○ USB, ATA, UART
○ PCI bus, AMBA bus
○ DMA, timer
○ DRAM, NAND Flash
Automatic HLS synthesis mode
Manual HLS synthesis mode
Mixed HLS synthesis mode for interfaces

Figure 6. Application types and different HLS synthesis modes.

One of the main problems with this approach is that although most ASIC HLS tools support this, the syntax is vendor specific, e.g., NEC's CWB [11] uses *$* in ANSI-C to indicate the timing boundary in manual scheduling mode. One exception is SystemC, with most of the HLS tools supporting the *wait*(*1*) statement as the clock boundary. Thus, in this work, we created a library of standard interfaces in SystemC following again the synthesizable SystemC subset to accurately model them. In particular, a UART, an SPI interface, and a VGA controller.

These four features enabled us to design complete heterogeneous SoCs at the behavioral level, which enables some interesting capabilities that we summarize in the next section.

5. Design Space Exploration: Micro-Architecture and System Level

HLS is a single-process synthesis method. This implies that every component in the system is synthesized separately. As shown in Figure 4b, this implies that once the behavioral description using the APIs is generated and after parsing the different descriptions, five separate behavioral descriptions need to be synthesized for this example: the master, the slave, the master interface, the slave interface, and the bus itself. This leads to five different RTL descriptions that can in turn be simulated and synthesized into gate netlists. Figure 2 shows these steps. After HLS, the fast cycle-accurate simulation model can also be generated. This is a model written in C/C++ that mimics the behavior of the RTL, generating as the output a waveform identical to an RTL simulation. The benefit of this approach is that the simulation time is much faster than RTL, and it also does not require any RTL simulator, as this model only needs to be compiled with a C/C++ compiler.

Based on this, the proposed flow enables a set of interesting applications. The slaves will typically be the hardware accelerators in the SoCs. Because the slave is specified in C/C++/SystemC, HLS allows the generation of different micro-architectures with unique area vs. performance and power trade-offs for any given behavioral description, as shown in Figure 2. As mentioned previously, different settings of HLS synthesis directives lead to implementations with unique area and performance trade-offs. Out of all the possible combinations, the most important ones are the ones that lead to Pareto-optimal designs ($\bar{P}$). This in turn allows generating systems with different characteristics by mixing different implementations of the hardware accelerators in the SoC using different types of interfaces, e.g., AHB, AXI, changing the bus bitwidth, arbiter type, etc. The ability to generate quick cycle-accurate simulation models allows one to quickly evaluate different configurations. The next subsections describe how this is done.

Micro-architectural Design Space Exploration: As mentioned previously, one of the advantages of the proposed flow is that HLS enables the generation of different micro-architectures with unique trade-offs without the need to modify the behavioral description. The question for the system integrator now is: Which micro-architecture for each component in the SoC should be used to meet the area, performance, and power constraints?

Much work has been done in the area of HLS DSE. A recent survey summarized the main efforts in this domain [14]. The main approach in all these methods is to explore the individual behavioral descriptions in isolation using different types of heuristics as the search space grows supralinearly with the number of explorable operations. Figure 7 shows a typical HLS DSE flow, where the design space explorer is built on top of the commercial HLS tool, seen as a black box. The explorer reads in the behavioral description to be explored and generates a unique combination of the different synthesis options available based on the commercial HLS tools. These options are often called synthesis *knobs*. In this example, these knobs can be classified into synthesis directives in the form of pragmas (k_{attr}), global synthesis options (k_{opts}), and the number of FUs (k_{fus}). Once a new combination is generated, the explorer invokes the HLS tools, extracts the synthesis results (area and latency in this case), and continues generating a new combination based on a given cost function, until a specified exit criterion is met.

In this work, we only consider the synthesis directives in the form of pragmas (k_{attr}), as it was shown in [14] that these have the largest effect on the final micro-architecture. In particular, our proposed work explores arrays, functions, and loops. Arrays are synthesized as registers or RAM or fully expanded to individual registers, functions inlined or not, and loops fully unrolled, partially unrolled, and pipelined with different initiation intervals.

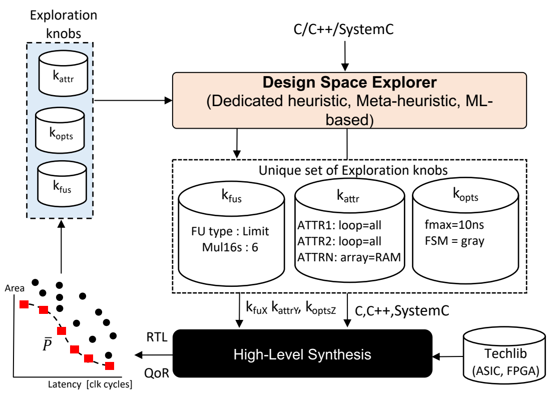


Figure 7. Micro-architectural HLS design space exploration overview.

Most previous works did not consider how the accelerator interfaces with the rest of the system. Using our proposed framework, we can further expand previous work on HLS DSE by including the interface type as an additional exploration constraint. Being able to quickly simulate the entire system is key to understanding the effect of the interface choice and pragma mixes on the area and performance of the system. In this work, we enable an easy way to extend traditional HLS DSE to include the interface type, bitwidth, and system arbiter as exploration parameters and used any of the previously developed heuristics [14] to obtain the trade-off curve of dominating micro-architectures. The additional main differences with previous work is that this explorer would need to perform a full cycle-accurate simulation for every new configuration and use system performance as the performance metric, as opposed to only using the latency or throughput of the individual accelerator.

In this particular work, we used a metaheuristic based on a genetic algorithm. Genetic algorithms have been shown to lead to very good results for these types of multi-objective optimization problems. The genetic algorithm is an evolutionary method that replicates how animals evolve. In this case, every unique pragma setting is a gene, and setting a unique pragma to every array, function, and loop conforms a chromosome. This chromosome is then combined and mutated based on predefined crossover and mutation probabilities to produce an offspring. Because metaheuristics are very sensitive to their hyperparameters (e.g., mutation rate and cross-over rate, exitcondition), we made use of a previously published method that allows automatically setting these hyperparameters in the context of HLS DSE [25].

System-level design space exploration: The HLS DSE results can now be used to generate entire systems with different trade-offs, as shown in Figure 4d. The inputs are a library of different interface parameters, including the bus type, bus bitwidth, and arbiter, and the result of the HLS DSE for each of the hardware accelerators in the system. The trade-off curve shown highlights two boundaries denoted as the system of maximum area (S_{max}) and the system with the smallest area (S_{min}). Intuitively, if the fastest, but largest micro-architectures of all the accelerators are used with the largest bus bitwidth, then the fastest, but also largest system will be obtained (S_{max}). Similarly, if the smallest micro-architectures with the smallest bus bitwidth are used, then the very smallest system should be generated (S_{min}). One observation that we have made by simulating many of these heterogeneous systems with our framework is that due to bus congestion issues in systems with multiple accelerators, choosing slower, but smaller micro-architectures might actually lead to the same overall system performance as S_{max}. This is denoted in Figure 4d as $S_{fastest}$. Thus, it is important to explore the different micro-architectures and interfaces within the entire system.

Much work has been done in the past in the domain of system-level exploration. These previous works can be classified into three different categories: (a) using aggressive pruning techniques to reduce the search space [26,27], (b) making use of metaheuristics to search the design space [28,29], or (c) using static analytical techniques to guide the explorer [30,31].

Once the candidate solutions have been generated, these have to be evaluated either through simulation (i.e., [31]) or through predictive models (i.e., [30]). Other work uses compositional techniques to explore the design space of SoCs [32].

As shown by the different trade-off curves in the system-level exploration result in Figure 4d, different heuristics will lead to slightly different trade-off curves. Thus, in this work, we simply used an exhaustive enumeration technique that allows us to compare the running time of the exploration using our behavioral SoC design framework vs. performing the exploration at the RT level. This allows us to directly compare the running times as both exploration methods generate the exact same number of configurations.

6. Experimental Results

Six SoCs of different complexities were generated to test our proposed method. These SoCs had between one and two masters and between two and five slaves. The masters were either the behavioral MIPS processor taken from the CHStone benchmarks [22] or the RISC V processor presented previously, while the slaves were all taken from the S2CBench benchmark suite [33]. AMBA AHB-Lite was selected for the SoCs with only one master and AMBA AHB for the rest. Two of the external interfaces were used to read and write data from the SoC, in particular a UART or an SPI interface, as shown.

Table 5 shows the configuration of each SoC, e.g., S1 is composed of a single master (1), which implies that it uses the AHB-Lite bus protocol and has two slaves (Sobel and snow3G). The rest of the systems contain different numbers of image processing and DSP processing hardware accelerators and two different encryption accelerators, a stream cipher (snow3G) and a block cipher (KASUMI). The main idea is that data arrive at the SoC through the external interface encrypted and need to be decrypted first by the encryption accelerator before they can be further processed. Once the data have been processed by the different accelerators, they are encrypted again and returned through the same external interface. The processors (masters) act as controllers setting the sequence of the computation steps. In all cases, the input was a 512 × 512 input image with every pixel being 32 bits. The size of the input was purposely kept small to allow the full simulation of the larger systems in RTL.

Table 5. System benchmarks' configuration overview.

System		S1	S2	S3	S4	S5	S6
Masters	MIPS	•	•		•	•	•
	RISC V			•	•	•	•
Bus	AHB-Lite	•	•	•			
	AHB				•	•	•
External Interface	UART	•	•	•			
	SPI				•	•	•
HWacc	Sobel	•	•	•	•	•	•
	fir		•			•	
	decim		•	•		•	•
	interp			•			•
	FFT					•	
	jpeg			•			•
	snow3G	•	•		•		
	KASUMI			•		•	•

Table 6 summarizes the tools used in the experiments. The HLS tool used was NEC CyberWorkBench v.6.1.1, which also allows generating cycle-accurate simulation models. The compiler used to compile the cycle-accurate simulation model was g++ 4.8.5, and the RTL simulator used was Mentor's Modelsim 10.5c. The target technology in all cases was ASIC Nangate 45 nm, and the target HLS frequency was 200MHz. The experiments were conducted on a server mounting an Intel i7-1160G7@2.60GHz processor with 32 GBytes of

RAM running CentOS 7.0. After the exploration stage, we also fully synthesized, placed, and routed all the systems to make sure that the target synthesis frequency was met.

Two set of experiments was conducted to test our proposed framework. The first compared the simulation running time of the entire SoC when we used our framework to generate cycle-accurate models vs. running the RTL simulation of the same SoCs when we applied the default HLS synthesis constraints. The second set of results investigated the ability of our proposed approach to perform efficient system-level design space exploration.

Table 6. Experimental setup.

HLS Tool	NEC CyberWorkBench 6.1.1
HLS Frequency	200 MHz
Logic Synthesis Tool	Synopsys Design Compiler 2018.06-SP1
Place and Route Tool	Cadence Virtuoso 6.1.7-64b
RTL Simulator	Mentor's Modelsim 10.5c
Synthesis Technology	Nangate 45 nm Opensource
C/C++ Compiler	g++ 4.8.5

Experiments 1: runtime comparison: All of these complex systems were fully described at the behavioral level, and we made use of our proposed flow. To test the advantages of our proposed flow, the bitwidth of the bus in each of these systems was set to 8, 16, 32, and 64 bits. The SoCs were slightly modified to break or group the data sent across the bus based on the bitwidth. One advantage of our method is that this could be accomplished within minutes. Table 7 shows the simulation runtime comparison of the entire SoC between the cycle-accurate simulation model generated by the HLS tools used in this work and the RTL simulation. The geometric mean is also shown for each benchmark to account for the size difference between the different SoCs. From the results, it can be observed that the cycle-accurate simulation was on average 4× faster, in particular 4.1, 3.4, 3.1, and 2.7× for the 8 bit, 16 bit, 32 bit, and 64 bit buses. Table 7 also shows that the 64 bit bus led to simulations that were 1.6, 1.3, and 1.2× faster than the 8 bit, 16 bit, and 32 bit bus versions. This is mainly because fewer transactions were required to complete the computation, further highlighting how simple it is to obtain these estimates quickly using our proposed flow. Moreover, these different cycle-accurate simulations all generated a waveform (VCD file) that allows the designer to analyze in detail bus congestion problems and overall system performance bottlenecks.

Table 7. Runtime comparison of RTL simulation vs. cycle-accurate simulation using our proposed approach.

Benchmark		**RTL Simulation Runtime (min)**				**Cycle-Accurate Simulation Runtime (min)**			
		8-Bit Bus	**16-Bit Bus**	**32-Bit Bus**	**64-Bit Bus**	**8-Bit Bus**	**16-Bit Bus**	**32-Bit Bus**	**64-Bit Bus**
AHB-Lit.	S1	4.2	3.1	2.6	2.5	1.5	1.2	1.0	0.9
	S2	10.5	7.2	6.5	5.8	1.8	1.5	1.3	1.1
	S3	14.3	11.2	9.1	7.4	3.9	2.9	2.5	2.3
AHB	S4	28.2	16.2	13.6	11.9	6.5	5.3	5.1	4.3
	S5	37.8	32.7	27.5	24.5	8.3	7.3	6.4	5.8
	S6	42.1	39.5	37.3	35.2	10.2	8.3	7.7	6.8
Geomean		**17.5**	**13.2**	**11.2**	**10.2**	**4.2**	**3.4**	**3.1**	**2.7**

One observation that we made during this experimental evaluation is that the compilation time of the cycle-accurate simulation for the larger systems was not negligible.

Figure 8 shows how the area of the bus and bus interfaces grew with the bitwidth of the bus and the number of masters and slaves. The 16 bit, 32 bit, and 64 bit versions of the system led to an area increase of 1.45×, 2.39×, and 4.18×, respectively, as compared to the

8 bit version. This further highlights how easy it is to analyze the impact of bus bitwidths on the area and performance of the entire SoC.

Experiments 2 : system-level design space exploration: Figure 9 shows the resultant trade-off curves from our system-level design space explorer, which fully enumerates all possible combinations. The search parameters include the bus bitwidth and the Pareto-optimal micro-architectures retrieved for each slave. Other parameters could be easily added to the search such as the type of processor and processor-specific characteristics.

Table 8 summarizes the running time of the explorer, with the total number of combinations indicated in Column 2. This search space is made out of the Pareto-optimal micro-architectures of each hardware accelerator and the different bus bitwidth. For this, each slave was explored using a genetic algorithm.

From the figure, we can clearly observe that different mixes of hardware accelerator versions combined with different bus interface specifications led to systems with dramatically different areas and throughputs.

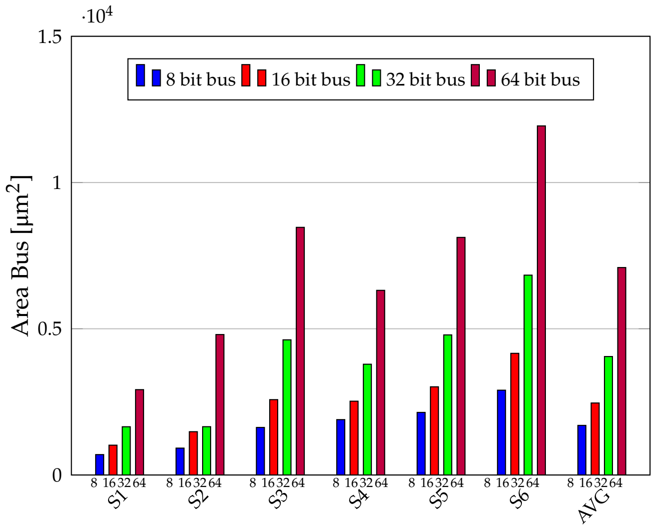


Figure 8. Bus and bus interface area for different bus bitwidths.

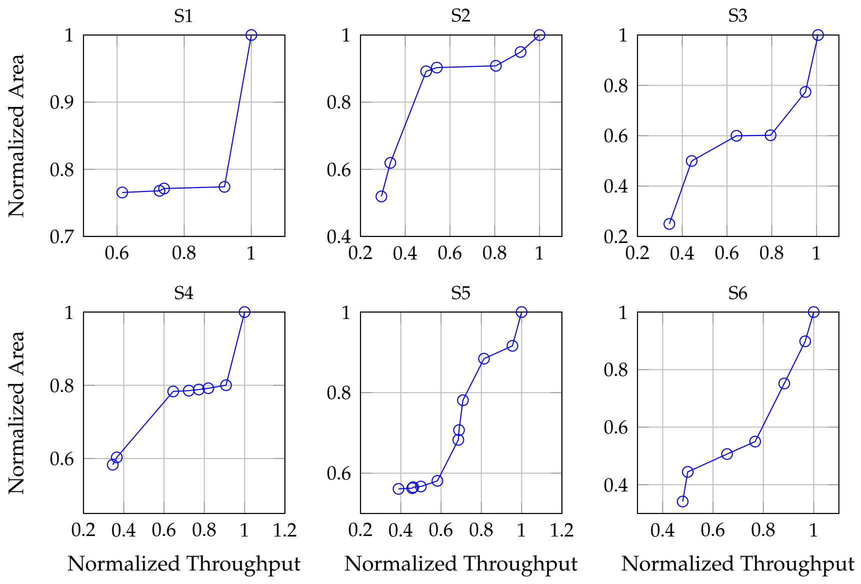


Figure 9. System exploration trade-off curves.

Table 8. System-level DSE running time results (min).

Bench	Search Space	RTL (min)	Proposed (min)	Diff
S1	40	128	23	5.61
S2	120	732	85	8.59
S3	196	1019	169	6.05
S4	128	704	87	8.09
S5	224	1456	300	4.85
S6	576	4101	806	5.09
Geomean		**858.9**	**138.0**	
Average				**6.4**

Moreover, from the results in Table 8, we can observe that using our behavioral SoC design framework led to an average speedup of 6.4× compared to performing the same exploration at the RT level.

Based on these results, we can conclude that our proposed framework works well and that it is able to simulate fully heterogeneous SoCs much faster than traditional approaches based on RTL descriptions.

7. Conclusions and Future Work

In this work, we presented a framework that enables the generation of complete heterogeneous SoCs at the behavioral level. Two main ideas enable this: a bus generator and a library of synthesizable APIs for the bus interfaces. By generating synthesizable SystemC code, any HLS tool can synthesize the bus interfaces generated and the bus into efficient RTL code. We also showed that it is possible to create embedded processors at the behavioral level, as well as external interfaces. This allows one not only to create the hardware accelerators using HLS, but also fixed architectures and controllers. Future work includes the support of other standard buses, e.g., AXI and Open Core Protocol (OCP), and even a Network-on-Chip (NoC). This will dramatically increase the flexibility of the proposed flow as only minor changes would be needed to change the topology of the entire SoC, and hence, this will enable a fast path to perform fast system-level design space explorations.

Author Contributions: Q.S. developed the behavioral CPU processor models and performed the experimental results. S.S. developed the behavioral bus generator and created the library of synthesizable interfaces APIs. B.C.S. supervised the work and summarized the work in this paper. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded by NATO Grant Number SPS Project G5640.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
CWB	CyberWorkBench
FPGA	Field Programmable Gate Array
FU	Functional Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
IC	Integrated Circuit
NoC	Network-on-Chip
OCP	Open Core Protocol

RTL	Register-Transfer Level
SoC	System-on-Chip
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver/Transmitter
VGA	Video Graphics Array
VLSI	Very Large-Scale Integration

References

1. Coussy, P.; Morawiec, A. *High-Level Synthesis: From Algorithm to Digital Circuit*, 1st ed.; Springer: Berlin, Germany, 2008.
2. Bombieri, N.; Liu, H.Y.; Fummi, F.; Carloni, L. A method to abstract RTL IP blocks into C++ code and enable high-level synthesis. In Proceedings of the 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 29 May–7 June 2013; pp. 1–9.
3. Mahapatra, A.; Carrion Schafer, B. VeriIntel2C: Abstracting RTL to C to maximize High-Level Synthesis Design Space Exploration. *Integration* **2019**, *64*, 1–12. [CrossRef]
4. Mahapatra, A.; Carrion Schafer, B. Optimizing RTL to C Abstraction Methodologies to Improve HLS Design Space Exploration. In Proceedings of the 2019 IEEE International Symposium on Circuits and Systems (ISCAS), Sapporo, Japan, 26–29 May 2019; pp. 1–5.
5. Sutherland, S. *The Verilog PLI Handbook*, 2nd ed.; Kluwer Academic Publishers: New York, NY, USA, 2002.
6. Yamamoto, H.; Chikamura, K.; Shigiya, A.; Tsujino, K.; Izumi, T.; Onoye, T.; Nakamura, Y. *System-Level Design of IEEE1394 Bus Segment Bridge*; Association for Computing Machinery: New York, NY, USA, 2002.
7. You, M.K.; Song, G.Y. Case study: Co-simulation and co-emulation environments based on SystemC SystemVerilog. In Proceedings of the 2009 IEEE Region 10 Conference (TENCON 2009), Singapore, 23–26 November 2009; pp. 1–4.
8. Liu, S.; Lau, F.C.; Schafer, B.C. Accelerating FPGA Prototyping through Predictive Model-Based HLS Design Space Exploration. In Proceedings of the 2019 56th ACM/IEEE Design Automation Conference (DAC), Las Vegas, NV, USA, 2–6 June 2019; pp. 1–6.
9. Siemens EDA. Catapult. Available online: https://eda.sw.siemens.com/en-US/ic/ic-design/high-level-synthesis-and-verification-platform/ (accessed on 19 July 2021).
10. Xilinx. Vivado HLx. Available online: https://www.xilinx.com/support/documentation/backgrounders/vivado-hlx.pdf (accessed on 19 July 2021).
11. NEC. CyberWorkbench. Available online: http://www.cyberworkbench.com (accessed on 19 July 2021).
12. Shetty, Santoshand Carrion Schafer, B. Enabling the Design of Behavioral Systems-on-Chip. In Proceedings of the ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 5–9 December 2021; pp. 1–6.
13. *SystemC Synthesizable Subset Version 1.4.7*; Technical Report; Accellera: Elk Grove, CA, USA, 2016.
14. Carrion Schafer, B.; Wang, Z. High-Level Synthesis Design Space Exploration: Past, Present, and Future. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 2628–2639. [CrossRef]
15. Cadence. Stratus. Available online: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html (accessed on 19 July 2021).
16. Intel. High-Level Synthesis Compiler. Available online: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html (accessed on 19 July 2021).
17. Synopsys. DesignWare IP. Available online:https://www.synopsys.com/designware-ip.html (accessed on 19 July 2021).
18. Chen, Y.; Gurumani, S.T.; Liang, Y.; Li, G.; Guo, D.; Rupnow, K.; Chen, D. FCUDA-NoC: A Scalable and Efficient Network-on-Chip Implementation for the CUDA-to-FPGA Flow. *IEEE Trans. Very Large Scale Integr.* **2016**, *24*, 2220–2233. [CrossRef]
19. Islam, A.; Kapre, N. LegUp-NoC: High-Level Synthesis of Loops with Indirect Addressing. In Proceedings of the 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Boulder, CO, USA, 29 April–1 May 2018; pp. 117–124.
20. Zhou, Y.; Al-Hawaj, K.M.; Zhang, Z. A New Approach to Automatic Memory Banking Using Trace-Based Address Mining. In Proceedings of the 2017 ACM/SIGDA International Symposium (FPGA '17), Monterey, CA, USA, 22–24 February 2017; pp. 179–188.
21. Giri, D.; Chiu, K.L.; Di Guglielmo, G.; Mantovani, P.; Carloni, L.P. ESP4ML: Platform-Based Design of Systems-on-Chip for Embedded Machine Learning. In Proceedings of the Conference on Design, Automation and Test in Europe, Grenoble, France, 9–13 March 2020; pp. 1049–1054.
22. Hara, Y.; Tomiyama, H.; Honda, S.; Takada, H.; Ishii, K. CHStone: A benchmark program suite for practical C-based high-level synthesis. In Proceedings of the 2008 IEEE International Symposium on Circuits and Systems, Seattle, WA, USA, 18–21 May 2008; pp. 1192–1195.
23. Liu, G.; Primmer, J.; Zhang, Z. Rapid Generation of High-Quality RISC-V Processors from Functional Instruction Set Specifications. In Proceedings of the 2019 56th ACM/IEEE Design Automation Conference (DAC), Las Vegas, NV, USA, 2–6 June 2019; Volume 122, pp. 1–6.
24. Carrion Schafer, B.; Trambadia, A.; Wakabayashi, K. Design of complex image processing systems in ESL. In Proceedings of the 2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC), San Francisco, CA, USA, 27–31 July 2010; pp. 809–814.

25. Wang, Z.; Schafer, B.C. Machine Leaming to Set Meta-Heuristic Specific Parameters for High-Level Synthesis Design Space Exploration. In Proceedings of the 2020 57th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 20–24 July 2020; pp. 1–6.
26. Givargis, T.; Vahid, F.; Henkel, J. System-level exploration for Pareto-optimal configurations in parameterized systems-on-a-chip. In Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD 2001), San Jose, CA, USA, 4–8 November 2001; pp. 25–30.
27. Fornaciari, W.; Sciuto, D.; Silvano, C.; Zaccaria, V. A Sensitivity-Based Design Space Exploration Methodology for Embedded Systems. *Des. Autom. Embed. Syst.* **2002**, *7*, 7–33. [CrossRef]
28. Erbas, C.; Cerav-Erbas, S.; Pimentel, A.D. Multiobjective Optimization and Evolutionary Algorithms for the Application Mapping Problem in Multiprocessor System-on-Chip Design. *Trans. Evol. Comp.* **2006**, *10*, 358–374. [CrossRef]
29. Ferrandi, F.; Lanzi, P.L.; Pilato, C.; Sciuto, D.; Tumeo, A. Ant Colony Heuristic for Mapping and Scheduling Tasks and Communications on Heterogeneous Embedded Systems. *IEEE Trans. Integr. Circuits Syst.* **2010**, *29*, 911–924. [CrossRef]
30. Lukasiewycz, M.; Glass, M.; Haubelt, C.; Teich, J. Efficient symbolic multi-objective design space exploration. In Proceedings of the 2008 Asia and South Pacific Design Automation Conference, Seoul, Korea, 21–24 March 2008; pp. 691–696.
31. Beltrame, G.; Fossati, L.; Sciuto, D. Decision-Theoretic Design Space Exploration of Multiprocessor Platforms. *IEEE Trans. Integr. Circuits Syst.* **2010**, *29*, 1083–1095. [CrossRef]
32. Piccolboni, L.; Mantovani, P.; Guglielmo, G.D.; Carloni, L.P. COSMOS: Coordination of High-Level Synthesis and Memory Optimization for Hardware Accelerators. *ACM Trans. Embed. Comput. Syst.* **2017**, *16*, 1–22. [CrossRef]
33. Carrion Schafer, B.; Mahapatra, A. S2CBench:Synthesizable SystemC Benchmark Suite. *IEEE Embed. Syst. Lett.* **2014**, *6*, 53–56. [CrossRef]