Rapid Heterogeneous Prototyping From Simulink

Shen Feng, Chris Driscoll, Jeremiah Fevold, Hao Jiang, Gunar Schirner
sfeng@ece.neu.edu, {driscoll.ch, fevold.j, jiang.hao}@husky.neu.edu, schirner@ece.neu.edu

Department of Electrical and Computer Engineering
Northeastern University
Boston, MA, 02115

ABSTRACT
Designing embedded high-performance systems is challenging due to complex algorithms, real-time operations and conflicting goals (e.g. power vs. performance). Heterogeneous platforms that combine processors and custom hardware accelerators are a promising approach. However, manually designing HW/SW systems is prohibitively expensive due to the immense manual effort.

This paper introduces SimSH: Simulink Sw/Hw CoDesign Framework, which provides an automatic path from an algorithm captured in Simulink to a heterogeneous implementation. Given an allocation and a mapping decision, the SimSH automatically synthesizes the Simulink model onto the heterogeneous target with reconstruction of the synchronization and communication between processing elements. In the process, the SimSH detects an underutilized bus and optimizes communication by packing / unpacking. Synthesizing a heterogeneous implementation from Simulink allows the developer to focus on the algorithm design with rapid validation and test on a heterogeneous platform. We demonstrate synthesis benefits using a Sobel Edge Detection algorithm and target a heterogeneous architecture of Blackfin processor and Spartan3E FPGA. The synthesized solution is 2.68x faster (and energy efficient) over pure SW execution.

1. INTRODUCTION
Algorithm designers prototype and fine-tune algorithms using high-level environments, such as Simulink [12]. In Simulink, users benefit from large library resources and an interactive user interface for algorithm design and system modeling. Furthermore, Simulink can synthesize algorithms onto homogeneous platforms (either CPU or FPGA), through Simulink Embedded Coder or HDL Coder [12].

However, a homogeneous architecture may not meet performances or power constraints of demanding applications (e.g. streaming applications). Designer shift attention to heterogeneous Multiprocessor System-On-Chip (MPSoC) which comprises multiple CPUs, memories and hardware accelerators. MPSoCs can improve performance and power efficiency through specialized hardware components (i.e. accelerators). However, current tools to not offer an easy path from Simulink onto a general heterogeneous architecture.

While additional heterogeneous components increase performance, they widen the gap between Simulink prototyping and heterogeneous implementation. The tremendous effort of manually creating a heterogeneous solution by stitching together homogeneous synthesis results (created in isolation) becomes a bottleneck and lengthens the time-to-market. To bridge this gap, this paper introduces SimSH which provides an automatic path from a Simulink model to a heterogeneous implementation.

Fig. 1 overviews SimSH. It takes a platform architecture and a Simulink specification model as input. Using the architectural database, our SimSH generates the necessary interfaces and produces a SW and HW implementation (as binary for processor(s) and bitstream for FPGA(s)). The user analyzes the computation and communication workload of the model and determines processing element (PE) allocation and model-to-PE mapping.

The contributions are the following:
- Introducing a SimSH that provides an automatic path from Simulink to a heterogeneous platform, given PE allocation and mapping. The SimSH empowers algorithm developers rapidly synthesize the application avoiding tedious and error-prone manual implementation efforts.
- The SimSH automatically inserts necessary communication and synchronization across PEs via Communication Refinement. The synthesized layered communication is influenced by the OSI standard [5] to enable reusability and scalability over varying architectures.
- A communication optimization is introduced which detects an underutilized bus, and increases efficiency through pack/unpack to fully utilize the bus.

We demonstrate the benefits using Sobel Edge Detection [9], and map it to a heterogeneous platform of Blackfin DSP and Xilinx FPGA. The results demonstrate significant benefits in terms of (a) rapid realization (within minutes), and (b) increased performance and energy efficiency (both 2.68x over SW implementation).

This paper is structured as following: Section 2 examines the relevant research. Section 3 introduces SimSH flow in detail. Section 4 shows the experiment result. Section 5 concludes the paper and touches on future work.

2. RELATED WORK
Synthesizing Simulink algorithm models to specifications has emerged in recent research. In [6, 7], authors proposed a framework for software code generation from Simulink and validation on MPSoC architecture. In [8], authors generate software for MPSoC...
from Simulink model and map them onto virtual platform (VP) implemented in FPGA. In [4], authors generate in addition to SW the VP via a combined algorithm and architecture model (CMMMA). Unlike [6, 7, 8, 4] which only target multiprocessor architecture and SW generation, we target a general heterogeneous architecture, including CPUs and hardware accelerators (e.g. FPGA). SimSH also explicitly addresses the communication across HW and SW.

The work in [15, 14] convert Simulink models to a System Level Design Language (SLDL) for System Level Design. The work introduces an interesting profiling approach and focuses on design space exploration (DSE). However, it stays at abstract simulation level, unlike our which aims for heterogeneous target execution.

Simulink R2014a [12] also supports concurrent execution code generation. However, it does not specifically address communication optimization. Furthermore, Simulink only targets specific heterogeneous architectures (such as Zynq with single CPU and up to 2 FPGAs), while our work targets a general heterogeneous architecture. Different from the industry approach, SimSH reveals both design methods and usage. It allows users in the academic community to easily expand the tool to support other platforms.

3. HW/SW CODESIGN FRAMEWORK

The input of the SimSH is a Simulink specification model. Simulink [12] is a Model-Based Design (MBD) tool for system modeling and verification. A Simulink model is described as a set of functional blocks and subsystems, i.e. a grouping of blocks linked by signals.

Fig. 2 illustrates our SimSH in more detail. It takes a Simulink model as input and guides the user in allocating and mapping blocks based on profiling. Synthesis occurs in 3 phases: Front-end Synthesis, Communication Refinement, and Back-end Synthesis, yielding the SW/HW implementation.

SimSH includes a profiler to investigate the application’s computation and communication workload. It employs AlgoSpec [14] to generate a SLDL specification model (in SpecC), and then profiles the specification using scprof [3]. The profiler reports computation and traffic demands in terms of number of operations, individually for each operation and data type. The profiling exposes computational and communication hot spots of the application.

Guided by the profiling results, the user manually allocates processing elements (PE) and maps the model onto the PEs accordingly yielding a mapped specification model. Allocation is recorded by annotating the Simulink model. In particular, the block of a certain type of computation is likely mapped onto the corresponding PE that is designed to optimize this type of computation. Details of profiling Simulink applications and the synthesis of system specifications can be found in [3]. Conversely, they are out of scope of this article, which instead focuses on the synthesis framework.

The input Simulink application model is shown in Fig. 2. For the discussion of this example, assume that blocks B and C are computationally heavy as revealed by the profiler. User then maps them on a hardware while other blocks stay in software.

In the Front-end Synthesis, the mapped specification model is split into hardware modules and software modules and then synthesized into software implementation in C/C++ and hardware implementation in Hardware Description Language (HDL). In this step, the functionality of all blocks in the model is synthesized for different PEs while the communication across the PEs is missing. To address that, we insert the Proxy in the model that encapsulates the cross-PE communication which will be further refined.

In the Communication Refinement, the Proxy is refined and realized following the OSI standard [5]. In our case, the Proxy comprises of 4 layers: the application layer for the consistent interface, the transport layer for synchronization, the network layer for addressing and marshaling and the physical layer for interfacing with the physical bus. Then the refined communication is integrated into the software and hardware implementation yielding a complete implementation in C/C++ and HDL on all PEs.

In the Back-end Synthesis, SimSH integrates the cross-compilation environment for software compilation and Xilinx ISE [13] for high-level synthesis. It finally generates software binary for processors and bitstream for FPGAs. The work in this paper makes assumptions and restrictions: (a) the user selects allocation and mapping manually, (b) it is bounded by Simulink Embedded Coder and HDL Coder restrictions and only supports discrete event models using fixed step solver.

3.1 Front-end Synthesis

In the result of the profiler-guided allocation and mapping, a mapped annotated Simulink specification model is the input of the front-end synthesis, as shown in the upper graph in Fig. 3. While the functional blocks and inter-PE communication is mapped on a PE, the cross-PE communication is implicitly mapped on the shared bus. The front-end synthesis explores the inter-PE communication optimization and inserts Proxys for further refinement.

![Figure 3: Communication Optimization over Underutilized Bus](image)

3.1.1 Communication Optimization

Given a group of blocks mapped on each PE in the mapped model, the traffic between blocks mapped on different PEs is influential to the overall performance. It usually incurs longer latency than inter-PE communication. To achieve efficiency, it is important that user transactions (as generated by the specification) match the underlying interconnect.
SimSH detects the under-utilized bus by comparing cross PE signals’ data width and bus width. Fig. 3 shows an opportunity for communication optimization. In the original mapped specification model (upper graph), a single transaction (P-bit width from A to BC) is less than the bus width (NPQ-bit width), which under-utilizes the bus. The bus utilization can be optimized by concatenating multiple user transactions accordingly. To do this, SimSH inserts in the mapped model a pack and unpack block at both sides of a cross-PE communication. This bundles multiple user transfers utilizing the bus width (lower graph).

Figure 4: pack and unpack for Concatenating N Transfers

Fig. 4 visualizes pack and unpack as parametrizable blocks. On the top, pack buffers NPQ P-bit input and concatenates them to a single NPQ-bit output. This reduces the data rate by factor NPQ between input and output. On the bottom, unpack slices a NPQ-bit input and into NPQ P-bit outputs, increasing the data rate by NPQ. In result, while the processing blocks (A, BC, D) remain untouched, transfers are bundled and bus utilization is increased.

If the target heterogeneous architecture supports bus burst transfer or DMA, SimSH can more aggressively concatenate transactions at cost of latency. The added blocks (pack, unpack) minimally increase computation. The benefits through better utilizing the bus outweigh the minimal computation overhead as the concatenation ratio increases (see Section 4).

Overall, communication optimization updates the mapped model with fewer transfers across the blocks mapped on different PEs.

3.1.2 Model Splitting and Proxy Generation

SimSH then splits the mapped model into a set of target models, one for each PE. Types include a SW model for a processor (e.g., CPU, DSP, ) or a HW model (e.g., for FPGA). Each target model only contains the blocks mapped to the particular PE.

Fig. 5 shows in the top half the mapping annotated model. Blocks A and D are mapped to a DSP (SW) and B,C to FPGA-1 (HW). To illustrate a more general complex example, blocks (X,Y,Z) and the backward communication c3 are included. In result of mapping, the communication across PE boundaries (c1, c2 and c3) needs to be established. For this, Proxy Generation replaces blocks mapped on another PE with a local Proxy. A proxy acts as a placeholder and bundles data input and output of the current PE. Proxy Generation traces the interface types in the Simulink model, and inserts proxies maintaining the interfaces. Fig. 5 shows the results. Block that are mapped on FPGA-1 (HW), i.e., BC (Unpack NPQ:1 and Pack 1:NP) are replaced with a Proxy-BC in the SW model. Proxy-BC on the DSP sends c1 and reads c2 and c3 from HW model. In result, A and D execute as if block BC were still in SW.

3.1.3 Homogeneous Synthesis

SimSH invokes the Simulink Embedded Coder [10] and Simulink HDL Coder [11] to generate target SW and HW implementations. Each of these can optimize internally to generate efficient code. To maximize the potential, we do not synthesize each block individually. Instead, blocks mapped to the same PE are grouped into a super-block and then synthesis is invoked on that super-block. Furthermore, as each inserted Proxy retains the boundary interfaces of the blocks it replaces, it can be scheduled identically to the original specification model. All inserted blocks: pack, unpack and Proxy block are composed of synthesizable blocks (for both SW and HW). Section 3.2 discusses Proxy composition and refinement in more detail.

Manually scheduling blocks mapped on the same PE is challenging due to Simulink semantics. To circumvent scheduling ambiguities, SimSH Proxy blocks are Simulink blocks and synthesized by Simulink together with the computation modules. We observe Simulink Embedded Coder generates sequential implementations and Simulink HDL Coder generates pipelined implementations. In addition, by Simulink synthesizing the Proxy, it benefits from all optimizations of the Simulink synthesis.

Overall, front-end synthesis generates computation blocks and the communication within one PE. The inter-PE communication via Proxy needs further refinement as discussed in the next section.

3.2 Communication Refinement

SimSH automatically refines communication into a layered implementation following the OSI standard [5] as shown in Fig. 6: application layer, transport layer, network layer and physical layer. A transaction initiated at the application layer is decomposed into packets at the transport layer, converted into bus transactions in the network layer and finally transferred via the physical layer. The layered design hides the underlying hardware (from the physical layer up), as well as application specifications (from the application layer down). The OSI layered communication implementation allows a wide application of the Proxy principle to a host of heterogeneous architectures. It simplifies expanding the database for new architectures.

Figure 5: Model Split for Each PE

At Application Layer, Proxy is a placeholder for blocks mapped to other PEs. It retains identical boundary interfaces of those remote blocks, replicating each port (e.g. in direction, width, data type, and update rate). E.g. SW Proxy-BC in Fig. 5 implements input port c1, output ports c2 and c3, identical to HW block BC.

At Transport Layer, as shown in Fig. 6, the SW Proxy-BC instantiates a proxy_recv block for each input and a proxy_send block for each output port. In case of a rate change in the replaced blocks,
a Rate Adapter is inserted. In SW, proxy_recv and proxy_send are connected through the Rate Adapter to allow different read and write transactions rates. proxy_recv, proxy_send, and Rate Adapter are constructed from a Simulink synthesizable subset to simplify code generation.

Implementing the hardware proxy requires strict timing, as it interfaces with the network layer from the database. To guarantee the timing, one approach is to generate the proxy out of communication primitives. In fact, communication refinement extracts the system composition and connection from the HW target model into an XML file generally following the IP-XACT standard [2]. It captures all relevant port characteristics, which guides the communication refinement to generate a proxy_recv or proxy_send for each port in the HW Proxy. The FIFOs in proxy_recv or proxy_send decouple execution of synthesized application from communication code.

Listing 1: Proxy API at Transport Layer

```c
/* Transport Layer of SW Proxy-BC */
proxy_send0(import_c1) {
    c1 = import_c1;
    send("Proxy-BC", c1);
    proxy_recv0(export_c3) {
        recv("Proxy-BC", c3);
        export_c3 = c3;
        proxy_recv1(export_c2) {
            recv("Proxy-BC", c2);
            export_c2 = c2;
        };
    };
}
```

For synchronization across HW and SW, our transport layer uses a buffered asynchronous communication. We use this as we observed that Simulink HDL Coder can synthesize HW blocks into a pipelined design to relax the pressure for the high-level synthesis. We utilize this concept to realize synchronization across each HW/SW boundary by adding an additional cycle delay.

Listing 1 shows the pseudo API of read and write transaction in the transport layer. All the transactions at the transport layer and above are hitherto addressed by block ID and port ID from Simulink model and therefore transparent to all underlying heterogeneous architectures.

Simulink Embedded Coder synthesizes the SW model into a step function triggered by a periodic timer. After execution of A, Proxy BC issues a write transaction, immediately followed by a read transaction (as governed by the Rate Adapter). It reads the BC result of the previous iteration, and SW continues with D. Hence, HW and SW execution are overlapped. HW starts executing upon availability of the data and produces the output.

Figure 7: HW/SW Synchronization

As shown in Fig. 7, BC produces results which are read by D in the next iteration (assuming the same rate for simplicity). This additional iteration delay makes the implementation of HW completely independent of the speed of SW. Therefore, the maximum delay of HW is relaxed to be as large as the complete loop of software execution. In a result, the HW can run multiple orders of magnitude slower than the bus speed, which has a potential for more energy saving. But we don’t explore it in this case because we are targeting on FPGA.

The Network Layer of the Proxy provides addressing and data marshaling as shown in Fig. 6. We follow a two-layer addressing, similar to Simulink’s identification (block ID and port ID). The network layer maps Simulink’s addressing onto the physical address.

Figure 8: Proxy Addressing at Network Layer

Depicted in Fig. 8, block ID and port ID follow the HW address prefix from the most significant bit (MSB) to the least significant bit (LSB). The address range for block ID and port ID is dependent on the number of blocks and ports in the model.

Listing 2: Proxy API at Network Layer

```c
/* Proxy Network Layer */
send(BlockX, PortY) {
    addr = convert2addr(BlockX, PortY);
    API_BUS_SEND(addr, Port.data);
    recv(BlockX, PortY) {
        addr = convert2addr(BlockX, PortY);
        API_BUS_RECV(addr, Port.data);
    }
}
```

Listing 2 shows two steps: addressing and marshalling. The SW send and recv function in SW Proxy first convert the block ID and port ID to the physical address and then marshal the transaction payload and eventually call the bus API. In HW model, a 2-level decoder (out of the database) is instantiated to select the HW block and the Proxy FIFO.

The Physical Layer in SW model contains the bus driver from the database. It provides a set of native API for bus transactions called from the network layer. In addition, it also wraps the SW application with some top level architecture specific initialization. In HW model, the physical layer instantiates the bus Interface (IFC) component and the top level FPGA pin mapping [1] as well as the User Constraint File (UCF). The IFC component can directly read data from the bus and interprets a bus writing as signals on the bus lines.

Table 1: Timing and Dependency of Proxy

<table>
<thead>
<tr>
<th>OSI Layer</th>
<th>Timing Accuracy</th>
<th>Application Specific</th>
<th>Platform Specific</th>
</tr>
</thead>
<tbody>
<tr>
<td>App</td>
<td>Application</td>
<td>loosely</td>
<td>high</td>
</tr>
<tr>
<td>Proxy</td>
<td>Transport</td>
<td>approximate</td>
<td>medium</td>
</tr>
<tr>
<td>Database</td>
<td>Physical</td>
<td>cycle</td>
<td>accurate</td>
</tr>
</tbody>
</table>

Overall, Table 1 summarizes the layering scheme from timing and dependency aspects. The timing accuracy (precision of synthesis) increases along the top-down layering refinement of communication based on OSI model. Besides, the higher OSI layer is more application specific and less platform specific. Hence, the most timing accurate element (bus interface), instantiated from the database, is completely platform specific and independent on applications. Conversely, the application requires full synthesis (with least timing requirements), while proxies are partially parameterized.

In the result of the front-end synthesis and communication refinement, SimSH has generated a complete SW and HW implementation in bare-C and VHDL.

3.3 Back-end Synthesis

Back-end synthesis is responsible for synthesizing the C/C++ and HDL code into the appropriate target binaries/bitstream. Based on the selected target architecture, the back-end synthesis integrates cross-compilation environments (e.g. BF527) and Xilinx ISE [13] to automate the SW compilation and HW high level synthesis.
The process of back-end synthesis is automated. SimSH generates Makefiles to automate the SW cross-compilation. For HW back-end synthesis, SimSH generates Xilinx ISE project files and invokes ISE for HW high level synthesis via command line [13].

4. EXPERIMENTAL RESULTS

To demonstrate the benefits of the framework, we use the Sobel Edge Detect [9]. Sobel Edge Detect detects the edges in an image by comparing each pixel with its neighbors. It computes the gradients of the current pixel via a matrix multiplication of a Sobel operator and the matrix of current neighboring pixels. If the gradient of a pixel is larger than a certain threshold, this pixel is detected as a part of an edge.

![Figure 9: Sobel Edge Detection Algorithm](image)

Fig. 9 depicts the Simulink model, mainly as a pipeline of Image Load, Serialize, Sobel Edge Detect, Deserialize and Image Print. Image Load simply loads a 320x240 gray image (8 bits/pixel) and Serialize sends each pixel to Sobel Core. Then, Sobel Core outputs the binary decision whether the pixel is part of an edge. Finally, Deserialize assembles the image.

![Figure 10: Sobel Edge Detect Performance Estimation](image)

To analyze the computation demands and to guide the mapping process, the application is profiled. In Fig. 10, the profiling results of Sobel Edge Detect show that Sobel Core occupies 79.1% of the total computation demand. User maps Sobel Core onto the FPGA and the rest on DSP, as shown in blue in Fig. 9.

This experiment targets a heterogeneous platform of a Blackfin BFM27 Digital Signal Processor (DSP) 600MHz [1] and a Xilinx FPGA Spartan3E XC3S500E 100MHz linked by 16-bit External Bus Interface Unit (EBIU) 100MHz on chip.

4.1 Application-specific Synthesis Results

In result of the mapping, Sobel Core’s input and output cross PE boundaries (Fig. 11). For each 8-bit input pixel, the Sobel Core outputs a 1-bit result indicating the edge. Considering a 16-bit EBIU bus, only 1/2 and 1/16 of the bus width is utilized. SimSH detects the underutilized bus in the data streaming and concatenates input and output of HW block by inserting pack and unpack blocks.

To realize the input concatenation, SimSH inserts pack_SW after the Serialize in the SW model (Fig. 11 bottom). Before sending, the pack_SW marshals two 8-bit pixels into one 16-bit bus transaction. Upon receiving, unpack_HW fires the Sobel edge Detect twice, once with each pixel. This cuts the number of input transactions into half. Similarly, concatenation of 16 Sobel Core output reduces the amount of output transactions 16x.

![Figure 11: Sobel-Edge Detect Communication Optimization over Underutilized Bus](image)

SimSH splits the resulting model into a HW model and a SW model. In the SW model, it replaces the Sobel Core with emp-proxy_sobel as a placeholder which consumes pixels and outputs decisions mimicking as if Sobel Core would still be in SW. Similarly, in the HW model, SimSH replaces Image Load, Serialize and Deserialize, Image Print with two proxies to receive pixels and send results.

Then, SimSH invokes imulink Embedded Coder [10] and Simulink HDL Coder [11] to generate target SW (CIC++) and HW (HDL) implementations. Here, the generated HDL for the Sobel Core has 13 pipeline stages. Invalid outputs due to pipeline fill are discarded by the HW Proxy.

![Figure 12: Proxy Network Layer Addressing](image)

During the communication refinement, the logical addressing based on block ID and port ID is refined to physical addressing of EBIU bus. Fig. 12 depicts the address allocation following Simulink two-layer addressing: in a 32-bit address, the address range from 19 down to 16 is allocated to block ID and address range from 15 down to 12 is for port ID. The most significant 12 bits are reserved by the memory system for FPGA address range while the least significant 12 bits are still free.

Therefore, the total available 20 bit address range (excluding the reserved 12 bits) supports up to $2^{20}$ combination of block ID and port ID. In this experiment, the allocated 4-bit block ID and 4-bit port ID supports up to 16 blocks and 16 ports, while we only need three blocks and two ports.

To route bus transaction to the correct Simulink block entity on FPGA, the corresponding two level decoders are generated: two 4- to-16 bit decoder select the hardware block entity and the input/output Proxy FIFO respectively.

The physical layer refinement of the HW Proxy is dependent on the database: the EBIU bus driven by bare-C and Interface (IFC) component in VHDL [1]. We implement the IFC as the EBIU bus driver in FPGA. It reacts to EBIU control line: reading from data line during read transaction and writing to EBIU control line and data line during write transaction. Furthermore, the physical layer also encapsulates the top level FPGA pin mapping as well as the User Constraint File (UCF).

4.2 Evaluation

To illustrate the benefits of the HW/SW co-design and the communication optimization, we compare five different implementations (varying mapping and optimization level) as illustrated in Fig. 13. Pure SW Solution maps the whole specification model on BFM27 DSP. HW-SW Co-Design (no opt) maps Sobel Core on the Xilinx Spartan3E FPGA and the remaining blocks on BFM27 DSP. HW-SW input pack/unpack optimizes the input communication of HW module by inserting pack/unpack on the path from SW to HW. HW-SW output pack/unpack optimizes the output communication of the Sobel Core. HW-SW input/output pack/unpack
The baseline solution is the pure SW solution, mapping the whole Simulink model on DSP. It results in the longest execution time about 45.6 Mcycles. HW-SW [no-opt] maps the most computationally intensive Sobel Core to FPGA, with the rest running on DSP, all in a pipelined fashion. The total execution time drops to 20.3 Mcycles, yielding a 2.25x speed up. However, HW-SW [no-opt] includes a communication overhead across DSP and FPGA of 22.2% of the total execution time.

Optimizing the path from SW to HW, input pack/unpack solution reduces traffic overhead slightly but yields an longer total execution time than HW-SW [no-opt]. The overhead of executing pack (in SW) outweighs the communication performance gain which is small due to the low input concatenation ratio of 2 : 1 (pack 2 pixels). Conversely, when only optimizing the path from HW to SW in [output pack/unpack] solution, the overall performance increases, as the output concatenation ratio 16 : 1 is much higher than input concentration.

Finally, optimizing both paths, SW to HW and HW to SW, achieves a 2.68x speed up against the pure software solution. The total communication time (0.58Mcycles) of HW-SW input/output pack/unpack decreases 10 fold compared to unoptimized HW-SW [no-opt] solution (5.8Mcycles). Meanwhile, the communication time (0.58Mcycles) with 3.4% of the total execution time is no longer a significant delay contributor.

To assess power efficiency, we measure board-level power of our platform. It remains fairly constant at around 680mW regardless of load and FPGA usage. As the DSP runs at the fixed frequency, this indicates that the FPGA (whose load is changed) is a minor contributor towards the total dynamic power. Nonetheless, HW/SW Co-design and further communication optimization shorten total execution time in the heterogeneous execution. Hence, the energy efficiency improves linearly with performance speedup. Our optimized HW/SW solution is 2.68x more energy efficient.

Table 2: FPGA Utilization of HW/SW Optimized Solution

<table>
<thead>
<tr>
<th>Slice</th>
<th>Total Application</th>
<th>Proxy+Glue Pack+Unpack</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage (out of 9312)</td>
<td>547</td>
<td>170</td>
<td>177</td>
</tr>
<tr>
<td>Utilization</td>
<td>5.8%</td>
<td>1.8%</td>
<td>1.9%</td>
</tr>
</tbody>
</table>

In the HW/SW optimized solution, the FPGA utilization of the Spartan 3E is only 5.874% as shown in Table 2. The generated Proxy, pack, unpack and other glue logic (bus IF) occupy 32%. The application specific Sobel Core is small in this example. This indicates significant room for other implementations, e.g. duplicating Sobel Core on FPGA. However, this algorithm optimization is out of the scope of the SimSH. On the other hand, the DSP is fully utilized at nearly 100% for all five implementations due to the overlapped HW/SW execution.

5. CONCLUSION

This paper introduces a Simulink-based SimSH to bridge the gap between the algorithm design in Simulink and its implementation on a heterogeneous platform. Given an allocation and a mapping decision, our SimSH automatically synthesizes the Simulink model onto the heterogeneous target and refines the synchronization and communication across processing elements. Furthermore, the SimSH optimizes communication by detecting an underutilized bus and concatenating transactions accordingly. In the result, it allows the developer to focus on the algorithm exploration and tuning and rapidly prototype it on a heterogeneous target platform.

We have demonstrated the benefits using Sobel Edge Detection [9], and targeted a heterogeneous architecture with a Blackfin processor and Spartan3E FPGA. Our proposed SimSH achieves up to a 2.68x speedup and energy efficiency with communication optimization against a pure software solution. In future work, we will investigate into automatic mapping decisions for a given platform.

6. REFERENCES