Synthesis of Optimized Hardware Transactors from Abstract Communication Specifications

Dongwook Lee
dongwook.lee@utexas.edu

Hyungman Park
hpark@cerc.utexas.edu

Andreas Gerstlauer
gerstl@ece.utexas.edu

Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas, U.S.A.

ABSTRACT

Increasing system complexity and heterogeneity make system integration and communication synthesis a growing concern. Even with transaction-level modeling and high-level synthesis of hardware, communication interfaces still have to be manually designed at a low protocol level. To address this challenge, we present a design flow for automatic synthesis of hardware transactors, which realize abstractly specified communication semantics on top of protocol-level transactions. Transactor synthesis is tightly coupled with high-level synthesis of computation for integrated computation/communication co-design of complete hardware processors, thus establishing a seamless path from abstract system specifications down to hardware implementations in synthesizable RTL. The flow supports a generic set of communication semantics and target implementations, where transactors are custom-generated for a specific application and architecture combination. Furthermore, we develop protocol stack optimizations that reduce the area and performance overhead of synthesized communication interfaces. We have applied our synthesis flow to several industrial-strength examples under various communication settings. Results show that synthesized interfaces are comparable to manual designs in terms of area and latency, where protocol stack optimizations can reduce area and latency overhead by up to 77% and 21%, respectively.

1. INTRODUCTION

Addressing ever-increasing complexity and heterogeneity of Multi-Processor Systems-on-Chip (MPSoCs), Transaction-Level Models (TLMs) have emerged to raise the design abstraction to higher levels [3]. Fundamentally, this is achieved by a separation of computation and communication, which enables independent design space exploration [13]. In a TLM methodology, communication details are abstracted away to the level of protocol transactions. This makes simulations faster and, on the computation side, a High-Level Synthesis

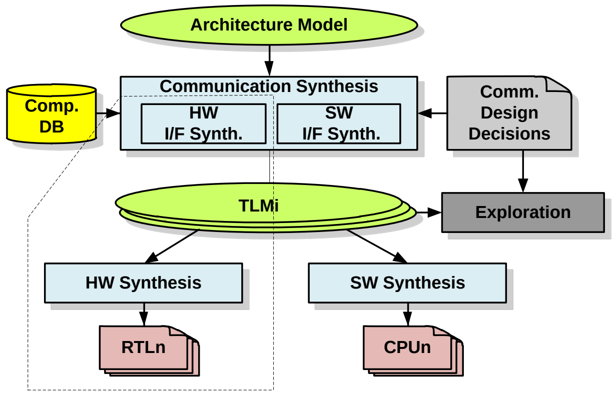


Figure 1: System design flow.

(HLS) tool can be used to refine C descriptions into RTL, enhancing design productivity as compared to a traditional design flow. However, communication interfaces still have to be designed manually at a low, detailed protocol level. Furthermore, their separation does not allow for optimizations across computation and communication boundaries.

Low-level system integration is a tedious, time-consuming bottleneck in the MPSoC design process. What is lacking are approaches that raise communication design to the same level as C-based computation, where system interactions of hardware components are abstractly specified using message-passing, queue, semaphore or other communication primitives, and *hardware transactors* are automatically synthesized to realize high-level semantics over protocol-level transactions while applying computation/communication co-design optimizations. Over the years, several approaches for system-level communication synthesis have been proposed. However, existing solutions are mostly library- or template-based and as such either limited to a single, canonical target architecture or to a generic implementation that comes with high overhead. By contrast, we aim to provide a synthesis flow that is (a) general in supporting a wide range of communication semantics and target implementations, while (b) able to custom generate communication interfaces optimized for a specific application and architecture.

In this paper, we present an approach for automatic synthesis of optimized hardware transactors from high-level communication specifications. The approach is tightly integrated with high-level synthesis of computation. Combined, this enables efficient, fully automated system integration including automatic generation of optimized bus-based communication interfaces and protocol stacks. Fig. 1 illustrates how the approach integrates into a typical system design flow based on standard system-level design languages (SLDLs), such as SpecC [7] or SystemC [10]. Starting from an architecture model in which the desired network topology and interactions between system components are specified using abstract SLDL channels, communication synthesis takes user decisions on targeted communication mechanisms and refines communication channels into a protocol-level TLM. The TLM can be used to explore the design space of possible communication implementations. In the TLM, components communicate over actual system busses using internally generated hardware or software transactor stacks. In this paper, we specifically focus on the hardware side. Synthesis of protocol stacks for hardware components is integrated with back-end hardware synthesis to automatically generate final RTL descriptions of complete hardware processors, which include all computation, transactors and external bus interfaces. In the process, hardware-specific protocol stack optimizations are applied to reduce area and communication overhead of the final RTL design.

The rest of the paper is organized as follows: following a discussion of related work and an overview of our synthesis flow, Sections 2, 3 and 4 elaborate on each step of our methodology. Section 5 shows experimental results of applying the flow to a set of industrial-strength design examples. Finally, Section 6 concludes the paper with a summary and an outlook on future work.

1.1 Related Work

Many communication synthesis approaches aim to provide integrated environments for design space exploration and mapping of applications onto custom or generic network-on-chip type of system architectures [9, 18, 15]. In almost all cases, complete system implementations are composed out of a fixed library of communication components, often limited to a single, canonical target architecture template that is proprietary and custom-designed [22] or based on an industry standard [14, 17]. Solely targeting design space exploration, several approaches support the automatic generation of TLMs for early prototyping and validation, but without the capability to generate actual implementations [20, 1, 19].

The focus in synthesis of component interfaces has been mainly on automatic generation of bridges or transducers that translate between protocols of incompatible cores. Several approaches [22, 12, 21] propose a generic architecture of the interface as a wrapper that connects processors to a common bus network. All of these approaches allow modularity in the design. However, they do not address the vertical integration path to bridge the semantic gap between abstract, high-level communication primitives and RTL. By contrast, our approach can synthesize communication from user-level channels at a range of abstraction levels.

Only recently have approaches emerged that try to synthesize communication implementations from higher-level TLMs. Cho et al. [5] describe a tool for synthesizing a TLM into a universal RTL bridge that translates the canonical TLM interface into a target bus protocol using platform definitions and protocol libraries. The bridge contains a FIFO and a dedicated controller for each bus protocol. Hatami et al. [11] and Bombieri et al. [2] propose a similar approach but with the interfaces written in SystemC TLM-2.0. Moss et al. [16] utilize communication refinement and HLS tools to

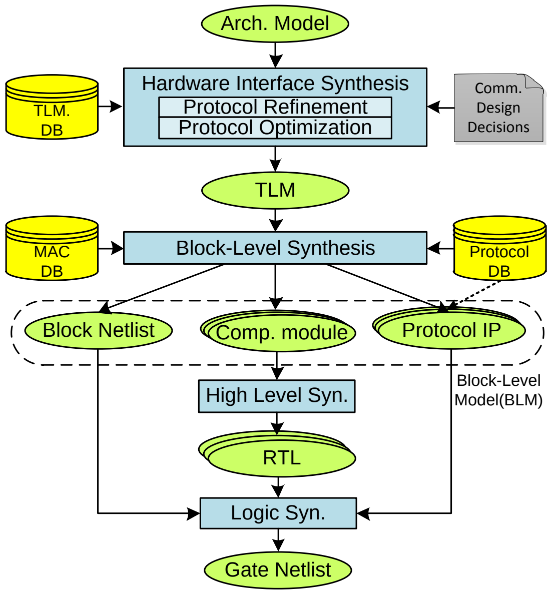


Figure 2: Hardware synthesis flow.

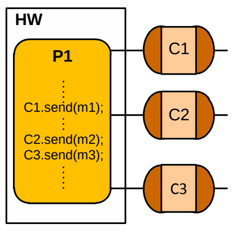
generate a data slicing transactor that realizes pin-level interfaces down to a final RTL bus adapter. Beyond academia, several commercial HLS tools [4, 6] have started to integrate similar capabilities of synthesizing interface functions written in TLM form. In all cases, however, synthesis is only supported from TLMs at a low protocol level. Designers are still required to manually implement transactors that translate from abstract communication semantics down to a protocol- or cycle-accurate level. Furthermore, existing approaches are purely database-, IP- or transducer-based, where a generic architecture template can add overhead in terms of area and latency. Our approach, on the other hand, supports tight integration with computation to synthesize a custom, application- and target-specific implementation of optimized bus transactors on top of a thin realization of the basic bus protocol taken out of a pre-designed RTL database.

1.2 Hardware Synthesis Flow

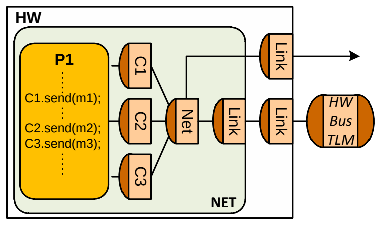
Our flow supports hardware synthesis from models at varying levels of abstraction (Fig. 2). Starting from an untimed architecture model, we follow a four-step methodology to synthesize communication down to a protocol-level TLM, transform the TLM into a block-level model (BLM), further synthesize block modules down to cycle-accurate RTL using a traditional HLS tool, and perform logic synthesis to generate a final gate-level netlist. The design flow utilizes databases in various forms through which the synthesis process can be adapted to different targets and HLS backends.

The architecture model describes the desired system topology, as well as its functional behavior. Computation is mapped to Processing Elements (PEs) that interact through logical point-to-point links. Communication is specified in abstract form, where links are user-defined through untimed channels of specific semantics, such as message-passing transfers, shared memory accesses, and event notifications.

Hardware interface synthesis uses the architecture model as input and generates the TLM. In the protocol refinement step, layers of protocol transactors are synthesized and inserted into PEs to implement communication follow-



(a) Arch. model



(b) TLM

Figure 3: Protocol refinement.

ing user-defined specifications, which include a given address map and the choice between interrupt- and polling-based synchronization mechanisms. As a result, message-passing channels of the architecture model are refined down to bus protocol transactions on top of loosely- or approximately-timed TLMs of busses brought in from a database. In the process, generated protocol stacks are flattened and optimized. Protocol optimizations are specifically developed to target synthesis of transactors down to efficient hardware.

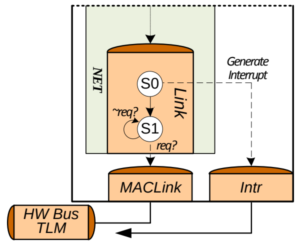
Finally, the TLM is converted into a BLM. Block-level synthesis generates synthesizable C++/SystemC code for each computation module in a hardware PE. It thereby extracts single-threaded processes and merges them with generated code for protocol transactors. In the process, thin media access (MAC) layer implementations taken out of a database translate a canonical MAC interface targeted by code generation into bus- and HLS-specific code required for further synthesis. Finally, block-level synthesis uses light-weight templates stored in a protocol database to generate both SystemC models and RTL implementations of interface IPs realizing external bus state machines. To form the BLM, generated computation blocks are combined with bus protocol IPs and a pin-level netlist that connects all blocks and external bus interface IPs.

Computation blocks are further synthesized down to cycle-accurate RTL descriptions using an external HLS tool, which transforms combined block-level computation and communication code into a merged RTL implementation of each hardware block. Generated RTL blocks, RTL bus protocol IPs, and the SystemC module netlist converted into RTL form can then be synthesized into a gate-level netlist using a traditional logic synthesis tool.

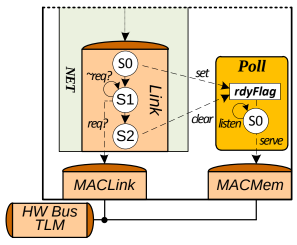
2. PROTOCOL REFINEMENT

We utilize an existing communication synthesis engine for generation of basic protocol stacks. In the following, we briefly summarize its operation, further details of which can be found in [8].

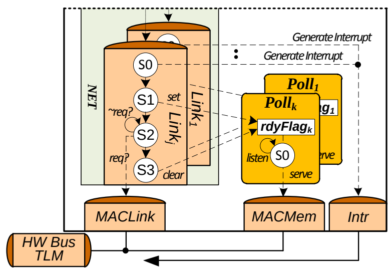
In the architecture model, hardware PEs are described in the form of C-based computation processes that communicate externally through a user-defined set of abstract, untimed messaging channels (Fig. 3(a)). We support both synchronous message-passing (double-handshake) and pure event synchronization (single-handshake). Double-handshake channels implement a synchronous, buffer-free, two-way blocking transfer of typed data streams from a sender to a receiver. By contrast, single-handshake channels perform asynchronous, data-less event notifications where the receiver blocks until the sender invokes the event. Protocol refinement then generates necessary transactor stacks to implement high-level semantics down to protocol-level transac-



(a) Interrupt

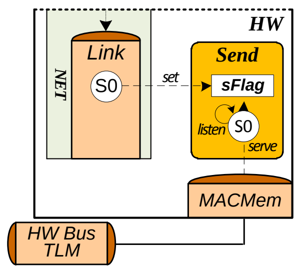


(b) Polling

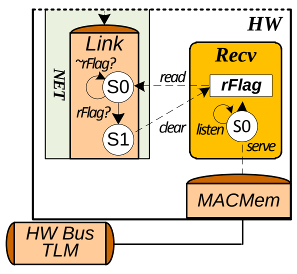


(c) Interrupt Sharing

Figure 4: Message-passing protocol stack.



(a) Polled send



(b) Receive

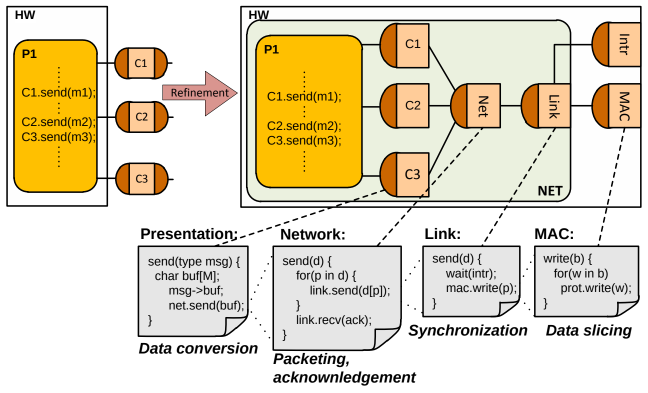
Figure 5: Event transfer protocol stack.

tions. In the TLM (Fig. 3(b)), protocol layers are inserted into an additional network level of the PE hierarchy to realize channel communication over external busses. Inserted layers include a presentation layer for data conversion, a network layer for packeting and routing, a link layer for synchronization, and a MAC layer for data slicing, alignment and media access.

Both message-passing and event communication require synchronization between slaves and masters, where we support interrupt- and polling-based mechanisms. Both types of channels can be synthesized into either implementation. Without loss of generality, we assume for the following discussions that the hardware processor is a bus slave communicating with a software processor acting as bus master.

Interrupt- and polling-based implementations of message-passing communication are shown in Fig. 4. In all cases, a link adapter is responsible for synchronizing with the external communication partner, either by generating an interrupt (Fig. 4(a)) or by setting a separate polling flag that can be accessed by the master (Fig. 4(b)). After synchronization, the actual data transfer is performed via MAC layer operations. Sharing of interrupts among multiple channels also requires a polling flag in addition to interrupt generation (Fig. 4(c)). In both cases, flags are cleared once the transaction is completed.

Pure event transfer channels are synthesized as buffered event notifications. As shown in Fig. 5, their implementation



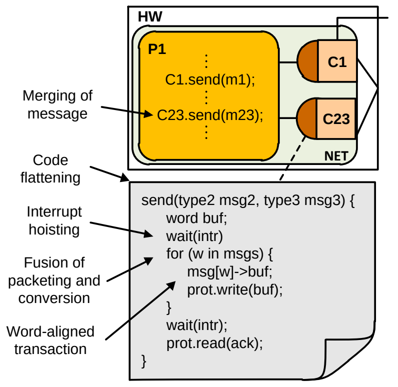
```
send(type msg) {
  char buf[M];
    msg->buf;
    net.send(buf);
}
```

```
send(d) {
  for(p in d) {
    link.send(d[p]);
  }
  link.recv(ack);
}
```

```
send(d) {
  wait(intr);
  mac.write(p);
}
```

```
write(b) {
  for(w in b)
    prot.write(w);
}
```

(a) Unoptimized layer-based code.



```
send(type2 msg2, type3 msg3) {
  word buf;
  wait(intr)
  for (w in msgs) {
    msg[w]->buf;
    prot.write(buf);
  }
  wait(intr);
  prot.read(ack);
}
```

(b) Hardware-optimized transactor.

Figure 6: Protocol stack optimizations.

is a variant of the message-passing synchronization scheme. Depending on the event direction, either master or slave can be sender or receiver. When sending an event from slave to master, the link adapter realizes interrupt, interrupt-sharing or polling-based synchronization. In a pure polling case (Fig. 5(a)), a separate *Send* process serves an event mailbox that is set by the link layer to indicate event availability to the master. When receiving an event from the master, neither interrupts nor polling can be utilized. Instead, the link layer blocks on an event flag in a separate *Recv* process, which serves the flag to be written and set by the master over the bus (Fig. 5(b)). In all cases, the *Send* behavior or link adapter will clear the flag once it has been consumed.

3. PROTOCOL STACK OPTIMIZATIONS

Basic protocol refinement generates unoptimized code for protocol stacks following a strict layer-based organization. For efficient backend synthesis, protocol code needs to be further optimized across layer boundaries.

In an unoptimized TLM (Fig. 6(a)), protocol code inserted into the PEs during basic refinement consists of four stacked adapter channels realizing layers of communication. The presentation layer converts abstract data types in application messages to untyped blocks of bytes transferred over the network layer following a canonical data layout between PEs. The network layer then optionally packetizes byte blocks into smaller chunks (packets) that are routed and transferred from PE to PE over bridges and transducers in the network. In case of synchronous message-passing over more than one hop, the network layer also performs end-to-end synchronization via acknowledge packets to restore barrier synchronicity and confirm successful completion of each message transfer. Finally, the link layer realizes point-to-point synchronization using one of the mechanisms described in Section 2, while the MAC layer slices data packets into external protocol transactions as supported by the underlying bus.

Protocol stack optimizations flatten basic communication layers and apply hardware-specific cross-layer optimizations for message merging, protocol fusion and interrupt hoisting (Fig. 6(b)). In addition, protocol coupling supports application-specific co-optimization of protocol stacks and bus protocol state machines. Each of these optimizations can be selectively enabled or disabled during synthesis. In the following, we describe optimizations in more detail.

3.1 Message Merging

Message merging combines consecutive messages transferred over one or more channels between two PEs into a single message over a common channel. This reduces the total number of channels and messages in the system. Consequently, overall resource demands are minimized.

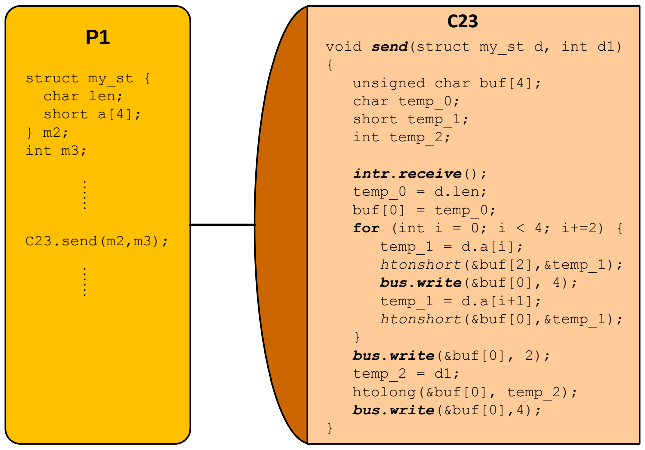
Messages of different channels can generally be re-mapped onto a single logical link if they are exchanged between the same two partners and if channels are accessed sequentially on both sides. For example, if sequential application processes exchange messages over separate application channels, and if there is no overlapped sending or receiving of such messages, the applications channels can be merged. As shown in Fig. 6(a), after conversion of data types to a canonical byte layout, this is achieved by multiplexing presentation layers to access a single untyped lower-layer stream. Overall, the number of network- and link-level streams is reduced, which in turn can lead to reduced hardware area requirements, e.g. for bus address decoding.

On top of basic channel merging, further message merging is applied (Fig. 6(b)). An analysis of data dependencies determines if two or more messages are sent (sequentially) over the same (merged) network-/link-level stream and are independent, i.e. can be considered logically consecutive with no intermediate computations. Such consecutive messages are then combined into a single, larger message. This eliminates the need for synchronization of each individual message. As such, message merging can reduce the total number of synchronization points and hence lead to less synchronization overhead in the system.

3.2 Protocol Fusion

After flattening of protocol stacks, a fusion of presentation, network and MAC layers is performed to interleave data conversion, packetizing and slicing. Protocol fusion can significantly reduce memory and area requirements, as well as generally unlock opportunities for optimization and scheduling in following high-level synthesis steps. Note that many HLS tools can internally perform loop fusion/merging. However, without further application-level knowledge, backend tools can not perform such optimizations across multiple layers of functionality in all cases.

In the unfused case, the presentation layer first converts a complete message into a block of bytes, the link layer then packetizes each byte block and the MAC layer finally slices each packet into bus words/frames. This requires buffering of complete messages and packets between layers. By



P1

```
struct my_st {
  char len;
  short a[4];
} m2;
int m3;

     ⋮

C23.send(m2,m3);

     ⋮
```

C23

```
void send(struct my_st d, int d1)
{
   unsigned char buf[4];
   char temp_0;
   short temp_1;
   int temp_2;

   intr.receive();
   temp_0 = d.len;
   buf[0] = temp_0;
   for (int i = 0; i < 4; i+=2) {
      temp_1 = d.a[i];
      htonshort(&buf[2],&temp_1);
      bus.write(&buf[0], 4);
      temp_1 = d.a[i+1];
      htonshort(&buf[0],&temp_1);
   }
   bus.write(&buf[0], 2);
   temp_2 = d1;
   htolong(&buf[0], temp_2);
   bus.write(&buf[0],4);
}
```

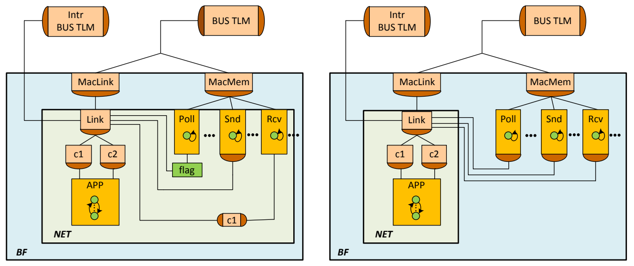
Figure 7: Example code after protocol fusion.

contrast, protocol fusion merges the loops of different layers and reschedules processing steps such that messages are processed on a word by word or frame by frame basis (Fig. 6(b)).

Within the protocol fusion process, proper alignment of message members to bus word or frame boundaries has to be considered. If even supported by the backend synthesis tool, misaligned accesses to data in PE buffers, registers or memories can lead to significant hardware and latency overhead. To combat inefficiencies, we implement an alignment algorithm in the synthesis tool. However, aligning of transactions can require padding of bus words and non-optimal utilization of available bus bandwidth, i.e. increased bus traffic. As an alternative, synthesis tools support tight packing, in which basic data members may be partially split across two bus transactions if there is leftover space in the current bus word. This is a trade-off between bus utilization and data alignment overhead. Typically, however, bandwidth losses will be smaller than the cost for restoring alignment.

Fig. 7 shows an example of protocol code after flattening and fusion, where chars and shorts are aligned and packed into half-word units while the trailing integer is aligned to the next full word boundary. In the optimized code, individual message elements of basic data type are converted into network byte format (e.g. using *hton*-type macros for endianess conversion) until enough bytes are available to fill the next bus word. Once a word is complete, it is sent or received over the bus using a protocol-level transaction. This process is repeated, possibly in a looped and hierarchical fashion, over all basic data members in a message.

All protocol fusion algorithms are realized in the synthesis tools based on PE- and bus-specific size and alignment information. Corresponding tables with size and alignment for each basic data type are stored in TLM and MAC databases. By modifying these tables, the fusion process can be adapted to different target architectures, system configurations and backend HLS/memory technology combinations. To create fused code, messages are recursively decomposed into basic data types, which are individually converted, aligned and assembled into bus words or frames. Calls to trigger bus transactions are further inserted at appropriate points in the code whenever a word/frame is complete. In the process, code optimizations are applied to maximally utilize looping structures and hence reduce overall code size.



(a) Non-Coupled TLM (b) Coupled TLM

Figure 8: TLM coupling variants.

Protocol fusion significantly reduces physical storage requirements and hence hardware overhead. Instead of buffering a complete message for conversion and packeting, the optimized code only requires a single word-sized buffer. Furthermore, interleaving across layers may lead to better opportunities for pipelining during high-level synthesis. Overall, protocol fusion is a process that can lead to significant gains, but is tedious and error-prone to perform manually.

3.3 Interrupt Hoisting

Interrupt hoisting is aimed at minimizing synchronization overhead. It is performed as a post-optimization to remove side effects of protocol fusion, and to restore synchronization points as defined after message merging. Protocol fusion effectively removes the packeting layer, which results in synchronizations being pushed down to a word or frame level. During interrupt hoisting, synchronization points are elevated up to as high a level as possible, without violating required system-wide synchronization semantics. As a result, synchronization code is hoisted to the packet or message level, and synchronization overhead is minimized.

3.4 Protocol Coupling

Protocol coupling optimizations support a tighter integration of application-specific high-level protocol stacks with synthesis of custom low-level bus protocol implementations. Link-level synchronization behaviors are either coupled with the protocol implementation or synthesized as separate computation blocks. As a result, one of two types of TLMs is generated (Fig. 8). In both cases, the TLM is hierarchically partitioned into a bus functional (BF) and a network layer. During hardware synthesis, the BF layer will be replaced with a custom IP that is generated as part of block-level synthesis (see Section 4.4). The network layer will be converted into synthesizable hardware modules that are passed through the HLS tool. In the uncoupled case (Fig. 8(a)), flag and polling behaviors are part of the network layer, where they become separately synthesized hardware blocks. By contrast, in a coupled TLM (Fig. 8(b)), synchronization functionality is tightly integrated with the protocol layer implementation, which is custom synthesized to provide a minimal set of required flags and synchronization services.

Traditional library- or transducer-based communication synthesis approaches follow an uncoupled approach in which computation blocks are integrated with general bus protocol implementations through a generic protocol-level interface. By contrast, a coupled implementation allows us to support joint co-optimization and co-synthesis of custom-generated protocol/synchronization state machines, as will be described in the following sections.

```
DESIGN-UNIQUIFY (t):
1  visited[t] ← True
2  ENQUEUE (Q, t)
3  while u ← DEQUEUE(Q)
4     for each  i Є Instances[u]
5         v = FIND-TYPE(i)
6          if visited[v] = True
7             then v = COPY(v)
8                   SET-TYPE(i, v)
9             visited[v] = True
10            ENQUEUE(Q, v)
```

(a) Isolation

```
UPDATE (t):
1 ENQUEUE (Q, t)
2  while u ← DEQUEUE(Q)
3     for each  i Є Instances[u]
4         v = FIND-TYPE(i)
5         ENQUEUE(Q, v)
6         for each p Є Ports[i]
7             m ← GET-MAP(p)
8             w = FIND-TYPE(m)
9             SET-TYPE(p, m)
```

(b) Updating

Figure 9: Pseudo-code for inlining algorithm.

4. BLOCK-LEVEL SYNTHESIS

Block-level synthesis refines the TLM down to block-level modules, protocol IPs, and a block-level netlist. Computation and synchronization behaviors in the network layer of the TLM are divided into single-threaded modules that are merged with communication adapters to become synthesizable hardware blocks. In addition, adapters and behaviors in the BF layer are combined with and transformed into custom-generated protocol IP components. Finally, pin-level connections between synthesized blocks and protocol IPs are translated into a corresponding HDL netlist.

4.1 Block Module Code Generation

A module code generator performs the necessary conversion of the TLM network layer into C++ or SystemC code that can be synthesized by a following HLS tool. In the process, it subdivides the TLM into behavioral block-level modules, inlines communication layers, and converts communication calls at block boundaries into MAC ports.

Most HLS tools can synthesize multiple single-threaded C++/SystemC modules with the capability to stitch blocks together. However, they can not automatically partition pre-existing code. Furthermore, multi-block synthesis is not universally supported and syntax and semantics vary. In order to provide a general approach that can be easily adapted to different HLS backends, we partition the code into separately synthesized modules that are integrated through our own netlisting engine. In the process, each computation or synchronization behavior in the TLM network layer is converted into a separate, synthesizable hardware block.

After block partitioning, transactor protocol stacks are inlined into each accessing computation block. This is done in such a way as to enable computation/communication co-optimizations in the following high-level synthesis step, with the scheduling freedom to overlap computation with communication and to perform general, joint optimizations, such as resource sharing or parallelization.

In the TLM, computation behaviors connect to communication adapters via an interface mechanism. Interfaces define a base signature of methods that a behavior can access without having to know the method realizations, which are provided by separate adapter channels. During block synthesis, however, interface calls need to be resolved and adapters inlined to establish the true caller-callee relationship and allow for joint high-level synthesis. Since static code analysis in most HLS tools cannot natively handle dynamic interface binding, we perform a preprocessing step to resolve interfaces and any associated polymorphism.

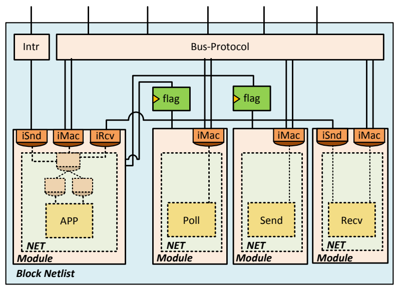
```
sc_module(MacLink) {
public:
   sc_port<sc_signal_in<…> > listen;
   sc_port<sc_signal_out<...> > serve;
   ...
   uint_t Listen(uint_t addr, bool read) {
        do { sc_bv<…> bus = listen.read();
        } while (!checkRW(bus,read) && !checkAddr(bus, addr));
        return getRdata(bus);
   }
   void Serve(uint_t data) { serve.write(data); }

   void slaveRead(uint_t addr, uint_t* data, uint_t len) {
        for( ; len; len--, data++) {
            *data = Listen(addr, false);
            Serve(0);
        }
   }
   void slaveWrite(uint_t addr, uint_t* data, uint_t len) {
         for( ; len; len--, data++) {
            Listen(addr, true);
            Serve(*data);
        }
   }
};
```

Figure 10: MAC database protocol wrapper.

The inlining algorithm in Fig. 9 traverses the module hierarchy and replaces interfaces with direct, statically bound calls to the connected adapter channel. Multiple instances of the same module type that connect to different channels are isolated into unique instance types, which include appropriate target calls. Given a top module t, the algorithm in Fig.9 (a)) visits all module classes in the TLM layer hierarchy in a breadth-first manner. For each class type u, all of its children are examined and the type v of each child instance i is determined. If another instance of v has already been encountered, i is replaced with an instance of a copy of v (lines 7 and 8). After all child instances have been examined and isolated, their types are guaranteed to be unique. In a second UPDATE algorithm, (Fig. 9 (b)) the class hierarchy is traversed again to remove all interfaces and update all port types to directly point to the connected channel instead. For each child instance in each visited class, the list of ports is traversed (line 6), the target m of the port's mapping is determined (line 7) and the port type is set to the class w of the connected instance m (lines 8-9).

4.2 MAC Layer Insertion

At the lowest level of the adapter hierarchy, behavioral interface or variable ports between blocks and bus protocol IPs have to be converted into wire-level register or FIFO interfaces as supported by the underlying HLS engine. The challenge is in providing an interfacing mechanism that is general in supporting a wide range of pre-defined bus targets while allowing for customization and co-optimization of low-level protocol implementations based on application requirements. For this purpose, MAC implementations inserted from a database provide the glue logic between application and target specifics. MAC database implementations convert functional into HLS-specific low-level interfaces. They thereby replace MAC adapters in the TLM with code that provides equivalent, canonical protocol-level TLM bus interfaces while internally translating each transaction into corresponding pin- and wire-level interactions with a target-specific bus protocol IP component. For coupling-driven op-


(a) Non-coupled BLM

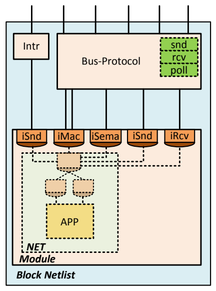

(b) Coupled BLM

Figure 11: Block-level model.

timizations, the MAC database also contains adapters that are inserted to replace synchronization behaviors in the BF layer with equivalent interfaces to flag and polling functionality directly synthesized into the protocol IP.

Fig. 10 shows an example of MAC database code for a slave protocol wrapper. The MAC adapter communicates with the protocol IP through primitive ports, which will be synthesized into a corresponding pin-level interface by the HLS tool. At the other end, it provides a TLM-equivalent, functional *Read()/Write()* interface to the calling module in which it is inlined. Internally, the MAC adapter contains functionality to translate between the two sides. In case of a bus protocol MAC adapter as shown here, this includes slicing of data packets into individual protocol transactions. To enable interleaving of communication and computation, and to expose corresponding parallelism to the HLS tool, protocol transactions are split into separate *Listen()* and *Serve()* methods. Furthermore, part of the protocol layer address decoding functionality is merged into the *Listen()* method. Combined, this provides the HLS tool the scheduling freedom to overlap computation with communication and to hide communication delays.

4.3 Protocol IP Generation

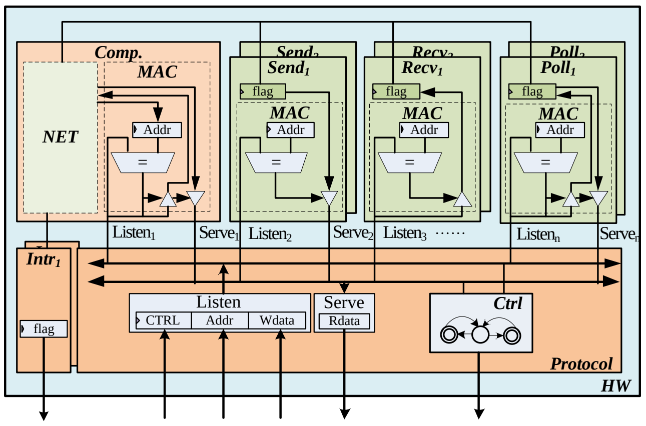
Each protocol adapter in the BF layer is synthesized into a protocol IP. Based on parameters, such as the type and number of adapters, a protocol generator creates a custom IP block from SystemC and RTL templates stored in the protocol database. The database templates are pre-written to model and implement external bus protocol and interrupt control logic. Their internal interface is designed to match associated protocol wrappers in the MAC database, which, when synthesized into other blocks, will realize appropriate pin- and wire-level interactions with the generated IP.

In a coupled model, additional parameters, such as address mapping and number and type of synchronization flags are used to synthesize custom synchronization functionality directly into the generated protocol IP. This allows for sharing of address decoding logic between synchronization and data transfers. Based on provided parameters, a protocol IP generator creates the required template instantiations, re-configures the address decoder, and provides necessary internal MAC interfaces to computation blocks.

4.4 Bus-Functional Synthesis

Bus-functional synthesis transforms the BF layer of the TLM into a final netlist to connect synthesized blocks and protocol IPs. As shown in Fig. 11, the resulting BLM combines modules to be synthesized by the HLS tool with custom-generated protocol-level IPs. As described above, application and, in the uncoupled case, synchronization behaviors are converted into synthesizable modules, where communication adapters are inlined and external interfaces are replaced with MAC implementations from the database. In a coupled model (Fig. 11(b)), synchronization functionality is realized directly in the generated protocol IP, and thin MAC wrappers are inserted to replace original synchronization behaviors with corresponding protocol IP interfaces.

Finally, the connectivity of all blocks, IPs and external ports is converted into a block-level netlist. In the process, variables and channels between modules are converted into registers and logic connecting the pin-level FIFO ports of synthesized blocks and protocol IPs. Fig. 12 shows the resulting RTL model after high-level synthesis. Protocol IPs realize state machines for driving and sampling of external bus and interrupt wires. The sampled external bus signals are passed on to MAC implementations synthesized into computation and synchronization blocks. A listen port thereby broadcasts the contents of an IP-internal bus sampling register. Synthesized MAC implementation in each block decode the listen addresses and communicate input data to the computation. The protocol IP in turn receives acknowledgements and output data from the MAC layers over a custom-generated number of serve ports. In a coupled model (Fig. 12(b)), the generated protocol IP provides additional interfaces to access synchronization flags over match-


(a) Non-coupled HW

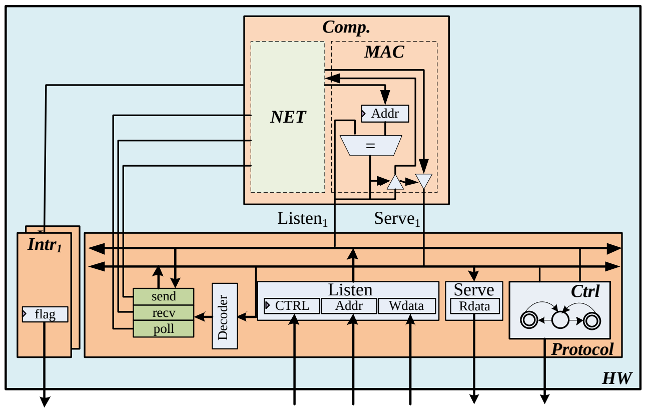

(b) Coupled HW

Figure 12: Synthesized hardware processor.

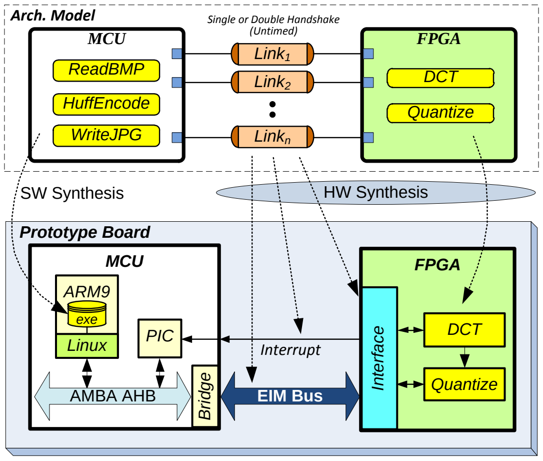


Figure 13: JPEG Encoder mapping.

ing thin MAC wrappers synthesized into each computation block. By contrast, in the uncoupled case (Fig. 12(a)), each separately synthesized synchronization block includes its own address decoders and tristate drivers that interface with the internally shared bus IP on a generic protocol level.

5. EXPERIMENTS AND RESULTS

We have implemented communication and block-level synthesis tools for our proposed approach and integrated them into an overall flow that utilizes Calypto Catapult [4] as HLS backend. Without loss of generality, we use the SpecC language to describe TLMs throughout the flow, and C++ and Verilog as input and output of the HLS tool, respectively. To demonstrate the benefits of our approach, we have applied the flow to synthesis of JPEG encoding and AC3 decoding applications onto an ARM-based target platform consisting of a Freescale i.MX21 applications processor (MCU) and a Xilinx Spartan-3 FPGA communicating over Freescale's proprietary EIM bus. The MCU contains an ARM9 that runs a Linux kernel and an EIM module that bridges between the AMBA AHB and EIM busses. We prepared database components for each tool in the flow, including TLMs of AHB and EIM busses, models of MCU and FPGA PEs, and EIM MAC layers and custom RTL bus protocol IPs.

We created architecture models for JPEG and AC3 examples with different mappings and communication semantics. As shown in Fig. 13 for the JPEG design, selected computation processes are mapped onto the FPGA while the rest of the application functionality executes on the ARM in the MCU. In the JPEG case, DCT and Quantize (Q) blocks are mapped into hardware. For the AC3, either the inverse modified discrete cosine transform (IMDCT) or the mantissa unpacking (MANT) process is in the FPGA. In both cases, the two PEs communicate through double-handshake (DH) message and single-handshake (SH) event channels. For every double-handshake channel, we configured associated implementation parameters (bus addresses and synchronization mechanism) to synthesize a polling, interrupt, or interrupt-sharing scheme for synchronization. For single-handshake channels, we applied polling for slave-to-master and master-to-slave notifications. Input architecture model configurations are summarized in Table 1.

Using our fully automated flow, we were able to synthesize input architecture models into correct RTL ready for further FPGA download within minutes, yielding substantial productivity gains compared to a manual design process. High-level synthesis was performed using Catapult's default optimization settings. Generated RTL was synthesized down to an FPGA bit file using Mentor Precision and Xilinx ISE. Designs were synthesized for target FPGA clock frequency of 50MHz. Area and latency results of synthesizing computation processes alone are summarized in Table 2. On the MCU side, software was synthesized, combined with manually written driver code, and cross-compiled into a Linux executable. The same firmware code was used for all designs. In all cases, we have successfully validated the functionality on the board. We evaluated end-to-end hardware latency by instrumenting the JPEG encoder and AC3 decoder software to record the average turnaround times over 180 DCT-Q, 1740 IMDCT or 1740 MANT invocations, including all communication and synchronization overhead.

Table 1: Input architecture models.

Application	FPGA process	DH Chs.	SH Chs.
Jpeg Encoder	DCT-Q	4	2
AC3 Decoder	IMDCT	5	0
	MANT	8	0

Table 2: Computation synthesis results.

Setup	LUTs	FFs	Muls	Memory	Latency
DCT-Q	3298	2121	16	512 bytes	6.25ms
IMDCT	6661	3673	28	11,264 bytes	1.57ms
MANT	6853	4158	1	0 bytes	0.697s

We synthesized variants of each design mapping all double-handshake message-passing channels into polling (POLL) or interrupt (INTR) implementations. In all cases, we applied message merging (MM), protocol fusion (PF), protocol coupling (PC), or all optimizations combined (ALL), and we compared FPGA resource usage and hardware latency against an unoptimized design (NOPT) and a manual implementation (MAN). The unoptimized, non-coupled implementations replicate traditional library- or transducer-based communication synthesis approaches, as realized by others and discussed in the related work. For a fair comparison, the manual design utilizes the same computation block synthesized by Catapult and the same firmware code to access hardware.

Table 3 compares designs in terms of their communication statistics and communication overhead as determined, respectively, by the number of synchronization behaviors in the TLM and by additional latency and FPGA resources of communication interfaces. Transactor overhead was measured as area and latency on top of basic computation requirements, i.e. by subtracting results in Table 2 from the final area and latency of the complete FPGA hardware.

We can observe that in all cases, the latency of final, optimized transactors synthesized with our flow is always significantly less than in a manual design. In contrast to a manual design in which transactors and computation blocks are designed separately to manage complexities, our approach is able to perform co-optimizations across computation and communication boundaries, e.g. to overlap and hide communication latency behind computation delays. In the DCT-Q case, data is processed strictly in the order it is received and sent. This allows computation and com-

Table 3: Transactor synthesis results.

Setup	Opt	TLM				Communication overhead (% difference vs. unoptimized)				
		Intr	Poll	Send	Recv	LUTs	FFs	Logic score	Mem [bytes]	Latency
DCT-Q-INTR	NOPT	2	2	1	1	2036	1076	3112	512	53.6ms
	MM	2	2	1	1	2036	1076	3112 (0.0%)	512 (0.0%)	53.6ms (0.0%)
	PF	2	2	1	1	2091	1171	3262 (4.8%)	512 (0.0%)	53.6ms (0.0%)
	PC	2	2	1	1	827	287	1114 (-64.2%)	512 (0.0%)	53.3ms (-0.5%)
	ALL	2	2	1	1	793	-75	718 (-76.9%)	512 (0.0%)	53.4ms (-0.4%)
	MAN	NA				-14	61	47 (-98.5%)	512 (0.0%)	71.4ms (33.1%)
DCT-Q-POLL	NOPT	0	4	1	1	2109	1181	3290	512	3.15ms
	MM	0	4	1	1	2109	1181	3290 (0.0%)	512 (0.0%)	3.15ms (0.0%)
	PF	0	4	1	1	2093	1161	3254 (-1.1%)	512 (0.0%)	3.15ms (0.1%)
	PC	0	4	1	1	810	264	1074 (-67.4%)	512 (0.0%)	3.08ms (-2.2%)
	ALL	0	4	1	1	835	220	1055 (-67.9%)	512 (0.0%)	3.50ms (11.2%)
	MAN	NA				-17	58	41 (-98.8%)	512 (0.0%)	13.5ms(329.1%)
IMDCT-INTR	NOPT	3	2	0	0	5035	2445	7480	41984	6.59ms
	MM	2	2	0	0	4239	1733	5972 (-20.2%)	41984 (0.0%)	6.50ms (-1.8%)
	PF	3	2	0	0	3767	1536	5303 (-29.1%)	21504(-48.8%)	6.24ms (-5.2%)
	PC	3	2	0	0	3858	1342	5200 (-30.5%)	41984 (0.0%)	6.60ms (0.4%)
	ALL	2	2	0	0	2574	699	3273 (-56.2%)	21504(-48.8%)	5.60ms(-14.8%)
	MAN	NA				522	322	844 (-88.7%)	25600(-39.0%)	6.27ms (-4.7%)
IMDCT-POLL	NOPT	0	5	0	0	4874	2181	7055	41984	5.76ms
	MM	0	4	0	0	4239	1733	5972 (-15.4%)	41984 (0.0%)	5.74ms (-0.4%)
	PF	0	5	0	0	3687	1485	5172 (-26.7%)	21504(-48.8%)	5.54ms (-3.8%)
	PC	0	5	0	0	3933	1452	5385 (-23.7%)	41984 (0.0%)	5.74ms (-0.4%)
	ALL	0	4	0	0	2463	622	3085 (-56.3%)	21504(-48.8%)	5.61ms (-2.6%)
	MAN	NA				540	319	859 (-87.8%)	25600(-39.0%)	5.82ms (1.0%)
MANT-INTR	NOPT	6	2	0	0	9164	3232	12396	23552	5.54ms
	MM	2	2	0	0	6032	4906	10938(-11.8%)	19456(-17.4%)	4.97ms(-10.2%)
	PF	6	2	0	0	8912	2987	11899 (-4.0%)	7168 (-69.6%)	5.01ms (-9.5%)
	PC	6	2	0	0	7462	3000	10462(-15.6%)	23552 (0.0%)	5.20ms (-6.1%)
	ALL	2	2	0	0	6911	3830	10741(-13.4%)	7168 (-69.6%)	4.36ms(-21.2%)
	MAN	NA				2254	2990	5244 (-57.7%)	13312(-43.5%)	4.62ms(-16.5%)
MANT-POLL	NOPT	0	8	0	0	9851	3622	13473	23552	4.41ms
	MM	0	4	0	0	6743	2640	9383 (-30.4%)	19456(-17.4%)	4.00ms (-9.2%)
	PF	0	8	0	0	7860	5543	13403 (-0.5%)	7168 (-69.6%)	3.89ms(-11.7%)
	PC	0	8	0	0	7184	3034	10218(-24.2%)	23552 (0.0%)	3.64ms(-17.4%)
	ALL	0	4	0	0	5724	1438	7162 (-46.8%)	7168 (-69.6%)	3.66ms(-17.0%)
	MAN	NA				2280	2992	5272 (-60.9%)	13312(-43.5%)	3.88ms(-11.8%)

munication to be pipelined and scheduled in parallel. By contrast, non-sequential data access patterns in the IMDCT and MANT blocks result in less latency improvements than in the DCT-Q. In both cases, computation/communication co-optimizations are also only enabled after exposing them to the following HLS tool through a protocol fusion step, as will be described below. In all cases, however, a naive, unoptimized realization of such computation/communication co-design can lead to a large increase in logic complexity, memory size and hence total area.

The proposed protocol stack optimizations prove to be efficient in reducing this area overhead as well as in further improving latency. Message merging can improve both. In the complex MANT design with a large number of messages, message merging can achieve up to 30.4% logic and 9.2% latency reduction by removing four out of eight interrupt or polling synchronizations. In the IMDCT, only one channel can be optimized away. As such, latency improvements are small but up to 29.1% less logic is required. No improvements can be seen for the simple DCT-Q design, in which only one message is exchanged at a time.

As expected, protocol fusion eliminates the need for large buffer memory between communication layers that a naive computation/communication co-design approach requires. Instead of buffering complete messages and packets, the optimized code only requires a single word-sized flip-flop. Additionally, protocol fusion can lead to latency improvements of up to 11.7% due to better interleaving and pipelining across layers during high-level synthesis. Note that in the simple DCT-Q case, Catapult is able to perform loop merging by itself. Hence, protocol fusion is not effective in optimizing the code further. However, this is not the case for more complex MANT and IMDCT designs, in which multiple levels of loop hierarchy need to be exposed for merging.

Finally, protocol coupling paired with custom-generation of protocol IP can significantly reduce resource utilization and latency. In uncoupled implementations reminiscent of traditional library- or transducer-based approaches, generality comes with high overhead and subsequent loss in optimality. By contrast, resource sharing in the coupled design results in up to 67.4% logic and 17.4% latency reduction.

Fig. 14 plots communication overhead in terms of area, memory and latency, normalized to the unoptimized case. After applying all protocol stack optimizations, logic, memory and latency overheads are reduced by up to 77%, 70% and 21%, respectively. Overall quality of results is comparable to a manual design, where on average a 21% improvement in latency can be achieved. Note that in the manual

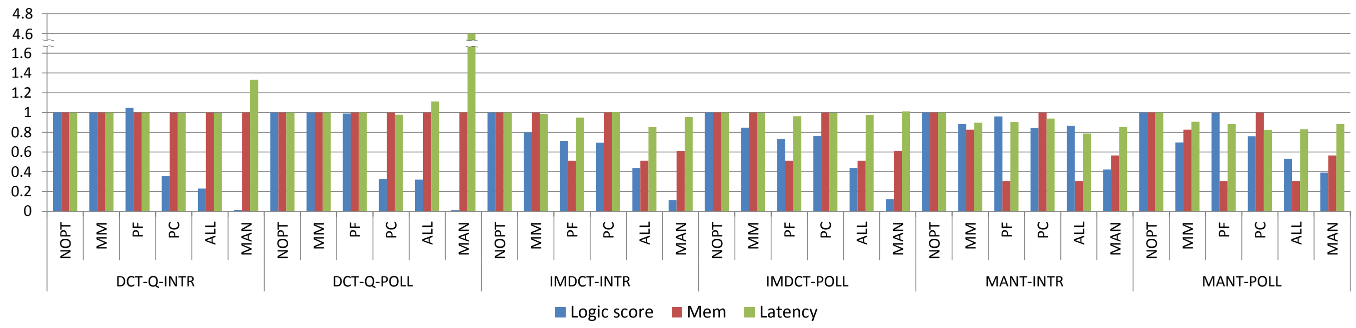


Figure 14: Comparison of communication overheads.

case, logic complexity is decreased at the expense of a higher memory requirement. This is due to less opportunities for sharing of memory among different message members. In summary, when including computation and communication, the total latency, logic score and memory size of our complete design is on average 16% faster, 20% larger and 19% smaller, respectively, than a complete hardware processor with a manually designed communication interface.

6. SUMMARY AND CONCLUSIONS

We have presented a systematic design approach to automatically synthesize custom, application- and target-specific bus-based hardware interfaces. Utilizing tools and databases along with a HLS engine, we provide a seamless flow from abstract specifications of desired communication functionality and topology down to cycle-accurate RTL. The approach supports various synchronization schemes and by replacing databases, it can be easily adapted to different HLS backends and synthesis targets. Experiments demonstrate that significant productivity gains can be achieved. Moreover, the quality of result is comparable to area and latency of manual designs, where a set of novel protocol optimizations can provide significant area and latency gains. In the future, we plan to extend our work to synthesize a wider range of communication primitives and styles (e.g. shared variables, hardware queues, master interfaces and transducers), including further optimizations of synthesized transactors. We also intend to expand support for SystemC TLM-2.0 interfaces and for utilizing other HLS backend tools.

7. REFERENCES

[1] S. Abdi et al. Automatic TLM generation for early validation of multicore systems. *IEEE D&T*, 28(3), 2011.

[2] N. Bombieri, F. Fummi, and V. Guarnieri. Automatic synthesis of OSCI TLM-2.0 models into RTL bus-based IPs. In *HLDVT*, 2010.

[3] L. Cai and D. Gajski. Transaction level modeling: An overview. In *CODES+ISSS*, 2003.

[4] Calypto Design Systems. Catapult C. http://www.calypto.com.

[5] H. Cho, S. Abdi, and D. Gajski. Interface synthesis for heterogeneous multi-core systems from transaction level models. In *LCTES*, 2007.

[6] Forte Design System. Forte Cythesizer. http://www.forteds.com.

[7] D. Gajski et al. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[8] A. Gerstlauer et al. Automatic layer-based generation of system-on-chip bus communication models. *TCAD*, 26:1676–1687, 2007.

[9] J. Gladigau et al. Automatic system-level synthesis: From formal application models to generic bus-based MPSoCs. *Transactions on HiPEAC*, 5(4), 2011.

[10] T. Grotker et al. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[11] N. Hatami, P. Prinetto, and A. Trapanese. Hardware design methodology to synthesize communication interfaces from TLM to RTL. In *AQTR*, May 2010.

[12] Y.-T. Hwang and S.-C. Lin. Automatic protocol translation and template based interface synthesis for IP reuse in SoC. In *APCCAS*, 2004.

[13] K. Keutzer et al. System level design: Orthogonalization of concerns and platform-based design. *TCAD*, 19, 2000.

[14] G. Lee et al. Automatic bus matrix synthesis based on hardware interface selection for fast communication design space exploration. In *SAMOS*, 2007.

[15] H. G. Lee et al. On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches. *TODAES*, 12(3), 2007.

[16] L. Moss et al. Automation of communication refinement and hardware synthesis within a system-level design methodology. In *RSP*, 2008.

[17] S. Pasricha et al. CAPPS: A framework for power-performance trade-offs in bus matrix based on-chip communication architecture synthesis. *TVLSI*, 18(2), 2010.

[18] A. Pinto, L. Carloni, and A. Sangiovanni-Vincentell. COSI: A framework for the design of interconnection networks. *IEEE D&T*, 25(5), 2008.

[19] K. Popovici and A. Jerraya. Flexible and abstract communication and interconnect modeling for MPSoC. In *ASPDAC*, Jan. 2009.

[20] D. Shin et al. Automatic generation of transaction level models for rapid design space exploration. In *CODES+ISSS*, 2006.

[21] S. Watanabe et al. Automatic protocol transducer synthesis aiming at facilitating IP-reuse. In *IWLS*, 2006.

[22] N.-E. Zergainoh, A. Baghdadi, and A. Jerray. Hardware/software codesign of on-chip communication architecture for application-specific multiprocessor system-on-chip. *JES*, 1:112–124, 2005.