Integrating High-Level Synthesis into MPI

Andrew W. H. House*, Manuel Saldaña† and Paul Chow*
*Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto, Toronto, ON, Canada M5S 3G4
{ahouse, pc}@eecg.toronto.edu
†ArchES Computing Systems, Toronto, Canada
ms@archescomputing.com

Abstract

In this paper, we investigate how easily we can port existing HPC applications that use MPI to run on HPRC systems, using three commercial high-level synthesis tools in conjunction with the ArchES-MPI software/hardware communication layer. Specifically, we examine how each tool interfaces with our existing message-passing hardware, and we present a sample application that illustrates how the interface can be used.

Keywords

FPGA; high-level synthesis; MPI; programming models; parallel programming; reconfigurable computing; HPRC;

1. Introduction

High-performance reconfigurable computing (HPRC) systems using multiple FPGAs are becoming more common, but a standard programming model has yet to emerge. It would be desirable to simply take existing high-performance computing (HPC) applications and map them automatically to hardware, but current C-based high-level synthesis (HLS) tools lack support for the features software programmers rely on to build highly-parallel HPC applications, such as Message Passing Interface (MPI) libraries [1] for inter-process communication.

HLS tools intended for HPRC applications (such as Impulse-C [2]) address this problem by providing vendor-specific APIs for communication with host processors or other platform-specific resources. This allows the host processor to manage system-wide communication via MPI, while the hardware accelerator is implemented separately. Our previous work on TMD-MPI [3], [4] attempts to remove this separation between programming models by providing an implementation of MPI intended for HPRC software *and* hardware.

Since our desire is to map existing software HPC applications to HPRC systems, we are most interested in how closely existing HLS tools can use software-like application code, and—more importantly—how easily they can be interfaced with our existing ArchES-MPI platform (a more advanced version of our previous work on TMD-MPI). To that end, this paper provides a case study showing how well three current C-to-gates HLS tools (Forte Cynthesizer [5],Impulse Accelerated Technologies' Impulse-C [2], AutoESL's AutoPilot [6]) interface with our ArchES-MPI environment.

We acknowledge the CMC/SOCRN, NSERC and Xilinx for the hardware, tools and funding provided for this project, and AutoESL and Impulse Acclerated Technologies for providing evaluation licenses for their tools.

2. Background and Related Work

Tools such as Auto-Pipe (and its associated X language) [7] or SCF [8] provide new frameworks for heterogeneous application development by allowing the definition of data flow between tasks, and then mapping each task to one or more specific implementations on a particular target platform. This facilitates design exploration, but existing HPC applications would have to be extensively redesigned to work in these environments. In this context, we thus maintain our focus on using MPI as the communication model, and interfacing that model (via ArchES-MPI) with C-based HLS tools.

While interest in C-to-gates HLS tools is high, little has been published regarding existing commercial tools. A number of qualitative comparisons can be found online, such as the comprehensive evaluation presented in [9] wherein 10 different C-based HLS tools are compared and evaluated against a small benchmark set. Among the conclusions reached in that study was that I/O interfaces were a major limiting factor in ease-of-use.

In [10], three disparate high level programming models (Mitrion-C, Impulse-C, and DSPLogic) were compared on the Cray XD1 in an explicit HPRC context. They were evaluated empirically to quantify ease-of-use and efficiency of results, but in this scenario each language had a platform-specific API to interface with the XD1, and so the results are not directly applicable to our work here. Unlike these previous comparisons of HLS tools, our interest is not in comparing the tools in general, but rather evaluating how well they can interface with other hardware components and how effectively they can integrate the use of the ArchES-MPI communication layer.

3. The ArchES-MPI Platform

This section provides a brief overview of our previous work on ArchES-MPI [4]. ArchES-MPI was developed to provide a well-known programming model for specifying and developing HPRC applications by adopting MPI [1], one of the most popular and effective standards in HPC.

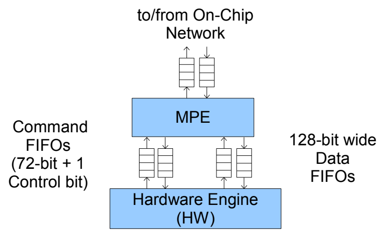

Fig. 1. FIFO Interface of the Message Passing Engine (MPE)

The current ArchES-MPI design flow involves software prototyping of a parallelized application, profiling the prototype on the target system, and then replacing selected software processes with custom hardware compute engines. These hardware engines are designed using traditional hardware design flows based around VHDL or Verilog, and this is the bottleneck in the design flow. The use of C-based HLS tools with our HPRC MPI layer would bring us closer to the desired goal of automatic retargeting of existing HPC applications.

ArchES-MPI implements a small subset of the MPI standard that provides a robust communication infrastructure. The core of its functionality is implemented as a packet-switched network used to transmit messages between processes. This network can connect multiple hardware and software processes within a single FPGA, across multiple FPGAs, on attached CPUs, or on another node in a higher-level network. The ArchES-MPI network encapsulates this information, so the routing of messages is transparent to the application programmer.

Thus, ArchES-MPI is applicable to all of the HPRC architectures we discussed in [11]. It can work in acclerated multiprocessor systems where CPUs have accelerator FPGAs connected via PCI or Ethernet, multi-FPGA application-specific reconfigurable multiprocessor systems such as the BEE3 [12], and heterogeneous peer multiprocessor systems such as those made possible by the Intel Xeon FSB FPGA socket fillers [13] from Nallatech.

Processors (hard or soft) access the communication layer via a software library; hardware compute engines connect to the network by a standardized interface to a hardware Message-Passing Engine (MPE). The MPE is a pre-designed hardware block used by compute engines that implements the protocols necessary to interface with the ArchES-MPI network. The compute engines talk to the MPE using one of the most common hardware interface abstractions: the FIFO.

The MPE interface consists of four FIFOs, split into Command and Data input/output pairs, as illustrated in Figure 1. Message transactions are initiated by writing commands to the Command FIFOs; data is read and written via the Data FIFOs. This split facilitates pipelining and stream processing by allowing hardware to overlap communication and processing, but does depart from a strict adherence to the software MPI model, which uses buffers to transfer data.

```
sc_bv<128> hwEngine::MPE_READ()
{
  sc_bv<128> din;
  {
    CYN_PROTOCOL("stream_recv");
    while (!data_in_exists.read()) {
      wait(1);
    }
    cmd_in_read = 0;
    cmd_out_control = 0;
    cmd_out_write = 0;
    data_out_control = 0;
    data_out_write = 0;
    din = data_in_data.read();
    data_in_read = 1;

    wait(1);
    data_in_read=0;
  }
  return din;
}
```

Fig. 2. Method to read from Data FIFOs in Cynthesizer

4. Interfacing With the MPE

The use of ArchES-MPI and the MPE allows for a standardized approach for system-level communication. While we do not expect to match software MPI calls exactly due to the streaming nature of the MPE, we would like to attain a similar interface in the HLS tools under consideration (Cynthesizer, Impulse-C, and AutoPilot) to ensure familiarity for MPI programmers.

Note that we are investigating this interface in the context of translating an existing MPI-based software process into a hardware engine, *not* as an approach for designing whole applications. This is meant to be the last step of the ArchES-MPI design flow, not the first, and thus the scope of our interest is quite limited.

4.1. Forte Cynthesizer (SystemC)

Forte Design Systems' Cynthesizer tool uses SystemC as its input language, and requires that users divide the application into protocol sections (timed, cycle-accurate descriptions of operations on input and output ports) and untimed data processing sections. Directives can be used in the data processing sections to suggest synthesis approaches such as pipelining, loop unrolling, or array flattening. It also supports a bit vector data type that is easy to cast to computational types, and thus a single set of non-overloaded function calls can provide an effective API, since Cynthesizer (unlike the other tools) supports synthesis of nested function calls.

Interfacing with FIFOs in Cynthesizer is relatively simple, as SystemC can describe the port interface and protocols exactly. This interface is controlled by a cycle-accurate state machine, but the C-like syntax allows for a simpler implementation than conventional HDLs. This state machine can be divided into smaller, task-specific state machines that are encapsulated in SystemC functions, as shown in Figure 2. Calls to those functions can be embedded in the data processing

```
#define READ_DATA_WORD(data, rc)                  \
    rc = co_stream_read(strm_mpeData_to_host, \
                         &(data), sizeof(data))

#define WRITE_DATA_WORD(data)                     \
    co_stream_write(strm_host_to_mpeData,         \
                     &(data), sizeof(data));
```

Fig. 3. Macros reading/writing data FIFO in Impulse-C

section, and Cynthesizer generates the master state machine that controls the MPE.

4.2. Impulse-C

Impulse Accelerated Technologies' Impulse-C is aimed at the HPRC market, with a focus on implementing algorithms. While largely similar to ANSI C/C++, it also provides its own libraries defining arbitrary-precision data types, type conversions, and other non-standard constructs. Directives can also be used to guide optimization.

Impulse-C also include an abstraction for streams: communication channels that must be opened before use, and closed when finished. Streams are read and written via explicit calls to library methods. These streams are mapped onto FIFOs in the target technology in a section of configuration code. We can implement the MPE interface by defining streams of the appropriate sizes to replicate the MPE FIFO interface. These streams can be hidden behind macros, as shown in Figure 3, to provide a software-like interface.

4.3. AutoPilot

AutoESL's AutoPilot targets system-level design. An AutoPilot C/C++ program closely resembles ANSI C/C++, albeit with limitations on how pointers can be used. It also provides tool-specific libraries for arbitrary-precision arithmetic as well as optimization directives. However, information such as target technology and all structural elements are contained in a separate file.

The interface to the MPE is defined in the program as a set of four pointers to variables of appropriate size, and then those pointer names are mapped to FIFOs. Writing or reading the FIFO in the program is then implemented as writing or reading from a pointer to a variable. This can be seen in the read/write macros shown in Figure 4. These macro calls to the MPE interface hide the pointer notation, providing an interface that looks similar to MPI software calls. As with Impulse-C and Cynthesizer, type conversion can be complicated and is non-standard.

5. Vector Accumulator Example

To evaluate the effectiveness of our interfaces, we implemented a simple vector accumulator application. The vector accumulator hardware compute engine works by waiting until a message is received and using the message tag to determine which operation is being requested: load, store, accumulate, and so on. After decoding the requested operation, the hardware engine executes it, which may include initiating additional message send or receive transactions to move data, or performing the accumulation computations.

```
#define READ_DATA_WORD(dw)                \
    dw=*mpeData_to_host;

#define WRITE_DATA_WORD(dw)               \
    *host_to_mpeData = (dw);
```

Fig. 4. Macros for reading/writing data FIFO in AutoPilot

```
sc_bv<22> actualSize = MPE_RECV_INIT("00000000",
            "0000000000000001100100", MPI_ANY_TAG);
...
else if (receivedTag.range(2,0) == C_ACC_OPCODE)
{
  sc_uint<32> d0, d1, d2, d3, a0, a1, a2, a3;
  sc_bv<128> acc;

  for (int i = 0; i < receivedVectorSize/4; i++)
  {
    CYN_INITIATE(CONSERVATIVE,4, pipe );
    din = MPE_READ();
    d0 = din.range(31,0).to_uint();
    ...

    acc = accumulator[i];
    a0 = acc.range(31,0).to_uint();
    ...

    a0 += d0;
    ...
    acc = ((static_cast< sc_bv<32> > (a3)),
           (static_cast< sc_bv<32> > (a2)),
           (static_cast< sc_bv<32> > (a1)),
           (static_cast< sc_bv<32> > (a0)));

    accumulator[i] = acc;
  }
}
...
```

Fig. 5. Cynthesizer code sample for integer vector accumulator (floating point unsupported)

As can be seen from the code listings in Figures 5, 6, and 7, the computational code for each tool looks very similar when the MPI layer is in place. Each of the tools also infers FPGA on-chip memories from arrays. The main differences are in how each tool handles its respective data types and optimization directives, but all three versions of the computational code are quite understandable to users with knowledge of ANSI C/C++. More comprehensive code samples can be found in our poster [14].

6. Conclusion

The C-based HLS tools we looked at in this paper have many similarities: libraries for arbitrary and fixed precision arithmetic, memory inference from arrays, optimization directives, and some means of external interface. Unfortunately, there are significant differences in these shared features that make porting programmer experience and code to different systems problematic. Likewise, since these tools all aim at

```
MPI_RECV(MAX_VECTOR_SIZE,0,MPI_ANY_TAG,
                                  rx_src, rx_count, rx_tag);
...
else if ( (rx_tag & C_OPCODE_MASK ) == C_ACC_OPCODE )
{
    co_float d0, d1, d2, d3, a0, a1, a2, a3;
    co_uint128 acc128;
    // accumulate
    for(i=0; i<rx_count/4; i++)
    {
        #pragma CO PIPELINE
        READ_DATA_WORD(data_wd, rc);

        d0 = to_float( (co_uint32)
                   co_bit_extract128_u(data_wd, 0, 32));
        ...
        acc128 = accumulator[i];

        a0 = to_float( (co_uint32)
                  co_bit_extract128_u(acc128, 0, 32));
        ...

        a0 += d0;
        ...

        acc128  = (co_uint128)float_bits(a0);
        acc128 |= ((co_uint128)float_bits(a1) << 32);
        ...
        accumulator[i] = acc128;
    }
}
...
```

Fig. 6. Impulse-C code sample for vector accumulator

```
MPI_RECV(MAX_VECTOR_SIZE,0,MPI_ANY_TAG,
                                rx_src, rx_count, rx_tag);
...
else if ( (rx_tag & C_OPCODE_MASK ) == C_ACC_OPCODE )
{
    uint128 data_wd, acc128;
    fp_int_t d0, d1, d2, d3, a0, a1, a2, a3;

    for(i=0; i<rx_count/4; i++)
    {
        #pragma AUTOPILOT pipeline II=1
        READ_DATA_WORD(data_wd);

        d0.u = apint_get_range(data_wd, 31, 0);
        ...
        acc128 = accumulator[i];

        a0.u = apint_get_range(acc128, 31, 0);
        ...

        a0.fp += d0.fp;
        ...

        acc128 =  0;
        acc128 |= apint_set_range(acc128,31,0,a0.u);
        ...
        accumulator[i] = acc128;
    }
}
...
```

Fig. 7. AutoPilot code sample for vector accumulator

different markets, each has some unique features not available in the others (such as floating point support, board support packages, or synthesis of nested function calls) that further emphasize their differences.

However, as we showed in the vector accumulator example, the use of the ArchES-MPI hardware communication layer with these HLS languages smooths over many of the major differences. If users can be isolated from the details of the HLS system in this way, it can make the use of the HLS tools more attractive for HPRC applications.

The HLS tools could further facilitate this integration with MPI by adopting a standardized representation for arbitrary- and fixed-precision data types and type conversions, and supporting a standard "bit vector"-style data type. Much of the difficulty in interfacing the tools with the MPE came from navigating their individual type systems, and most of the code differences in the sample implementations are related to this. A standardized data representation would simplify the integration details, and make it easier for the ArchES-MPI platform to hide the communication differences and for users to write their HPRC applications.

References

[1] *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum Std., Rev. 2.2, 4 September 2009. [Online]. Available: http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

[2] D. Pellerin and S. Thibault, *Practical FPGA Programming in C*. Prentice Hall, 2005.

[3] M. Saldaña and P. Chow, "TMD-MPI: An MPI implementation for multiple processors across multiple FPGAs," in *IEEE International Conference on Field-Programmable Logic and Applications (FPL 2006)*, August 2006, pp. 329–334.

[4] M. Saldaña, A. Patel, C. Madill, D. Nunes, D. Wang, H. Styles, A. Putnam, R. Wittig, and P. Chow, "MPI as an abstraction for software-hardware interaction for HPRCs," in *Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA) 2008*. IEEE, 16 November 2008.

[5] M. Meredith, *High-Level SystemC Synthesis with Forte's Cynthesizer*. Springer Netherlands, 2008, ch. 5, pp. 75–97.

[6] AutoESL Design Technologies, Inc, "AutoPilot High-Level Synthesis," 2009. [Online]. Available: http://www.autoesl.com/images/stories/datasheets/autopilot_datasheet.pdf

[7] M. A. Franklin, E. J. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer, "Auto-Pipe and the X language: A pipeline design tool and description langauge," in *20th International Parallel and Distributed Processing Symposium, 2006 (IPDPS 2006)*, 25–29 April 2006.

[8] V. Aggarwal, R. Garcia, G. Stitt, A. George, and H. Lam, "SCF: A device- and language-independent task coordination framework for reconfigurable, heterogeneous systems," in *HPRCTA '09: Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications*. ACM, 2009, pp. 19–28.

[9] B. Holland, M. Vacas, V. Aggarwal, R. DeVille, I. Troxel, and A. D. George, "Survey of C-based application mapping tools for reconfigurable computing," Presented at the 2005 MAPLD International Conference, 7–9 September 2005, Washington, D.C. [Online]. Available: http://klabs.org/mapld05/presento/215_holland_p.ppt

[10] E. El-Araby, M. Taher, M. Abouellail, T. El-Ghazawi, and G. B. Newby, "Comparative analysis of high level programming for reconfigurable computers: Methodology and empirical study," in *3rd Southern Conference on Programmable Logic (SPL'07)*, February 2007, pp. 99–106.

[11] A. W. H. House and P. Chow, "Investigation of programming models for emerging FPGA-based high performance computing systems," in *Proceedings of FCCM 2008, the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2008.

[12] J. D. Davis, C. P. Thacker, and C. Chang, "BEE3: Revitalizing computer architecture research," Microsoft Research, TechReport MSR-TR-2009-45, 1 April 2009. [Online]. Available: http://research.microsoft.com/pubs/80369/BEE3_TechReport.pdf

[13] "Intel Xeon FSB FPGA accelerator module." [Online]. Available: http://www.nallatech.com/index.php/Intel-Xeon-FSB-Socket-Fillers/fsb-development-systems.html

[14] A. W. H. House, M. Saldaña, and P. Chow, "Integrating high-level synthesis into MPI," poster, May 2010. [Online]. Available: http://www.eecg.toronto.edu/~pc/research/publications/house.fccm2010.poster.pdf