AUTOMATING THE DESIGN OF MLUT MPSOPC FPGAS IN THE CLOUD

*E. Cartwright, A. Fahkari, Sen Ma,
C. Smith, M. Huang, D. Andrews*

CSCE Department
Fayetteville, Arkansas
email: {eugene,dandrews}@uark.edu

Jason Agron

Intel Corporation
Santa Clara, California
email: jagron.m.agron@intel.com

ABSTRACT

Modern platform FPGAs are over the million-LUT level, large enough to support complete heterogeneous Multiprocessor System-On-Chips (MPSoCs). Constructing systems with 10's of processors is currently feasible using existing manual methods within vendor-specific CAD tools. However these manual, *by-hand*, approaches will not be feasible for constructing future systems with 100's to 1,000's of processors. Instead, new automated system assembly approaches will be required to handle these levels of system complexity and diversity. In this paper we present a new automated design flow for creating such next generation heterogeneous MPSoCs. An integral part of the MPSoPC system created is the inclusion of a general purpose PThreads-compliant HW/SW co-designed operating system and heterogeneous compiler. Our design flow has been placed in the cloud and is freely accessible across the Internet.

1. INTRODUCTION

Current Platform FPGAs contain more than a million LUTs, a sufficient density to turn a single chip FPGA into a complete multiprocessor system on chip (MPSoC). As FPGAs continue to follow Moore's law density levels will allow 100's to 1,000's of mixed-type programmable processors as well as custom accelerators to be configured within a single chip. While our current by-hand assembly methods to combine processors, busses, and support peripheral components into a multiprocessor architecture may be adequate for systems with 10's of processors they will not be sufficient for handling the complexity associated with creating next generation systems composed of 100's of heterogeneous processors. New automated architecture construction and system synthesis methods will be required to handle these levels of complexity. Platform FPGAs as MPSoPC's also bring a paradigm shift in requisite designer skills to conceptualize, design, integrate, test, and debug systems with 100's to 1,000's of heterogeneous processors configured within complex memory hierarchies, interconnect structures and

support for modern programming models. This places additional requirements on today's FPGA designers who may only possess lower level digital design skills.

Automating the assembly of soft IP processors, busses, memories, and necessary support components is an important step, but only represents a small percentage of any overall system design effort to create a usable MPSoPC system. New approaches to the run-time system itself are required to support scalability to 100's of processors, as well as resolve differences in processor ISA's, synchronization primitives, Application Binary Interfaces (ABIs), cache coherency protocols, and processor scheduling associated with processor heterogeneity. Resolving these run-time heterogeneity issues in turn requires new approaches to compilation and linking. Any new MPSoPC automation approach would not be complete without the inclusion of an Integrated Design Environment (IDE) for software development.

In this paper we present a new prototype design flow and set of automated tools that enable designers to automatically construct architectures and compile PThreads programs for next generation FPGA based MPSoPC architectures. Our design flow is vendor neutral and exists at an abstraction level above vendor supplied CAD tools. Our flow integrates the configuration of the middleware and operating system software libraries with the generation of the hardware architecture to produce a complete system that can be programmed from familiar high level programming models. Specific contributions of this paper include:

- *Automatic Generation of SMP and NUMA Multiprocessor Architectures.* The only input required is a specification of the number of processors. All busses, interconnects, distributed memories, and operating system hardware support are automatically created.
- *Vendor Neutral Intermediate Architecture Representation.* We define a vendor neutral intermediate representation that can then be used for optimizations, and vendor specific back end component generators. This allows a standard platform representation to be easily ported across different device and vendor specific

FPGAs.

- *Heterogeneous Compilation Flow.* We provide a compilation toolchain that allows users to write standard PThreads compatible applications as if they were running on a homogeneous multiprocessor system. Our toolchain resolves differences in processor Instruction Set Architectures (ISA's) and Application Binary Interfaces (ABIs) and links the different run time machine codes to our single image microkernel operating system.
- *Cloud-Based Software-as-a-Service Access.* We have placed our toolchain on the web with a simple web based interface. Designers do not need to download support packages or configure and build our tools on their local system. Instead we provide a convenient web based interface that invokes our tools running on our local server. For those who wish to download and install our tools locally, all source files are freely available and downloadable on the web.

2. ARCHITECTURE GENERATION

The goal of our automatic architecture generation is to abstract away the complexity and details of creating and configuring a complete heterogeneous MPSoPC. Our flow addresses the following key issues: creating portable architectures that can be instantiated on any FPGA platform, allowing the creation of baseline SMP and NUMA memory organizations, and provide built in support for current and evolving higher level heterogeneous programming models. Our automated architecture generation tool is called Arch_Gen. Arch_Gen partitions the process of constructing an MPSoPC into three steps, or levels of abstraction.

At the top level users select attributes using the pull down menu's shown in Figure 1 to guide the creation of either a Symmetric Multiprocessor (SMP) or Non-Uniform Memory Access (NUMA) architecture. A minimum number of attributes are needed, including the memory organization (SMP,NUMA), number of processors, and target vendor platform. Other optional parameters are set to default values but can be overridden using additional expanding menus. SMP systems rely heavily on cache support to offset main memory access times and contention across the global bus. Xilinx provides the Multiport Memory Controller (MPMC) for SMP systems that allows direct connections between the instruction caches of up to six Microblaze processors (plus host and PLB bus) to the global memory. Therefore Arch_Gen imposes a hard limit of six slave processor for SMP systems. No such limit exists for NUMA based architectures. Figure 2 shows an SMP architecture configuration for a Xilinx platform. Arch_Gen builds NUMA architectures using a distributed, hierarchical memory organization to remove the

The screenshot shows a web browser window titled 'CSDL MPSoPC' with the URL 'hthreads.csce.uark.edu/ARCHlang/buildyou'. The page has a header 'Build Your Own System' with a sub-header 'You can create your own system by entering user parameters for the system you desire. You will be able to download the design files and then import them into your version of the Xilinx tools.' The main content area is divided into several sections: 'Global Parameters' with fields for Project Name, Xilinx Tool Version (12.3), Xilinx Platform (m605), Memory Configuration (NUMA), Number of Slave Processors (6), and Number of Supported Muxes (64); 'Host Processor Parameters' with radio buttons for Default and Customize; 'Slave Processor Parameters' with radio buttons for Default and Customize; and 'UART Parameters' with fields for Baud Rate (9600), Data Bits (8), and Parity (Off, Even, Odd). A 'Submit' button is at the bottom.

Fig. 1. User Parameters

traditional SMP bottleneck associated with a single global memory and shared bus. Figure 3 shows the structure of a NUMA architecture.

The objective of the second, or Vendor-Agnostic level, is to build up the intermediate representation into a set of four distinct modules. The first module contains the VHDL hardware cores for our hthreads microkernel operating system [1]. Once synthesized, the operating system cores are small, requiring 3759 LUTs, or about 8% of a Virtex-6. The second module contains a fixed set of common peripheral soft IP components such as the clock generator, memory controller, UART, and DMA controller. The third module contains a main host processor connected to an internal memory block. The fourth module contains the slave processing units. This is the most complex of the modules.

For NUMA systems, Arch_Gen groups a variable number of processor-memory pairs into a group template, and generates a bus-memory hierarchy to connect the templates. Figure 3 shows how group templates are used to form a generic NUMA architecture. Arch_Gen attempts to minimize overall system level bus contention by first creating as many individual busses as is possible based on bus IP size and device capacity. Additional processors are then added into groups as the requested number of processors increases. Analysis of group sizing has shown that for PLB busses a group size greater than three can result in contention on the local bus. We currently build systems with up to 32 processors on a Virtex-6, which is built using eleven local busses. Unless overridden by the designer Arch_Gen will attempt to develop group sizes of up to only three. We are currently investigating approaches to optimize the number of processor-memory pairs partitioned into a group based on analysis of

Fig. 2. SMP Architecture

a specific application's code.

The objective of the lowest, or Vendor-Specific Component level, is to transform the architecture into a vendor specific CAD tool file format. This is the only level that needs to be compliant with any particular vendor CAD tool. We currently target Xilinx platforms, and generate the .mhs, .mss, and .xmp files. Once imported in XPS, the system can be immediately synthesized or modified based upon the needs of each specific user. Each IP component added into the system results in attribute information being added into the vendor files. Three classes of attributes can be produced for a component. The first are fixed attributes such as component names and configurations needed to interface into our hthreads operating system. These attributes require no user interaction and are assigned by Arch_Gen. The second are derived attributes based on the system type and numbers of slave processors. Arch_Gen uses this information to derive attributes such as assignment of address ranges and bus bridge connections for the system. The third are additional user specified attributes. These can include processor specific configuration parameters such as inclusion of multipliers, dividers, and floating point co-processor units for configurable processors.

3. HTHREADS HETEROGENEOUS MICROKERNEL

At run-time, applications call operating system library code that is typically assembly language for a specific ISA. This poses no compatibility issues for homogeneous systems. However this does prevent, a processor with a *different* ISA from directly invoking this library code. Inter-processor synchronization implemented using ISA-specific atomic operations is particularly problematic, as atomic operations are not uniform across different processor families [2]. Some processors, such as the MicroBlaze, have no built-in atomic oper-

Fig. 3. NUMA Architecture

ations at all. Existing heterogeneous systems such as IBM's Cell B.E., Intel's EXOCHI [3, 4], and ReconOS [5] resolve these differences using remote-procedure call (RPC) mechanisms that invoke services on a host processor. Although flexible, RPC mechanisms are typically implemented using heavyweight interrupt and exception mechanisms.

We developed the hthreads operating system to directly resolve heterogeneous incompatibilities and provide a precise real-time performance envelope for embedded systems [6, 7]. Hthreads resolved ISA incompatibilities by providing a system of processor-agnostic (ISA-neutral) thread management, scheduling, and synchronization services implemented as hardware cores. Figure 4 shows the 4 major hardware cores that comprise the backbone of the hthreads system: the Thread Manager, Scheduler, Synchronization Manager, and Condition Variables [6, 7]. This modular partitioning breaks up the traditional monolithic kernel structure allowing separation of concerns between different OS service cores. This is important for enabling the operating system to provide services that scale seamlessly on 100's to 1000's of heterogeneous processor cores.

Each OS IP core fully encapsulates its internal data structures and serves as the sole interface to its internal data. This fosters explicit inter-service communications and eliminates shared data structures within the operating system itself. The basic control structure of each OS core is independent of the numbers and types of processor resources and active threads in the system. This decoupling is advantageous as each generation of manycore chips will provide performance increases through the addition of cores. The operating system must provide a framework that allows application programs to be seamlessly ported between generations as well as vendor platforms.

Each processor runs a Hardware Abstraction Layer (HAL) which functions as a thin software library that enables APIs to invoke OS services. The HAL transforms APIs into a uniform set of memory mapped I/O commands; invoking the

Fig. 4. Hthreads System Block Diagram

common run-time services provided by the hthreads OS IP cores. The hardware-based services are ISA-neutral, which resolves the issues associated with processor heterogeneity. Each core has a simple memory-mapped interface that is accessed via traditional load/store instructions. This allows any processor that can master the bus to *directly* request services. This circumvents the need for slower remote-procedure call (RPC) mechanisms to provide a uniform set of efficient system services to all heterogeneous cores. Allowing each core to operate independently enables different processors to simultaneously request system services. This reduces both latency and jitter as simultaneous requests for different services do not compete for centralized services. The internal functions of each OS IP core have also been parallelized, providing low latency system calls with minimal jitter through efficient hardware implementations. More detailed descriptions of the cores and timing analysis can be found in [1]. Our build system automatically includes the hthreads cores and configures the HAL code for each processor transparently from the user.

Hthreads abstracts differences between SMP and NUMA architectures from the programmer to enable a portable and seamless multithreaded programming model. Different lower level library code is conditionally compiled for NUMA systems for API's such as `thread_create`. Within the `thread_create` API instructions are first DMA'd from global memory into a processors local instruction memory transparently to the user. This allows all processors to execute instructions out of local memories and not compete across shared busses or the global memory.

4. COMPILATION FLOW

Traditional design and compilation flows target homogeneous systems in which all compiled code is of the same ISA. In heterogeneous systems, the presence of multiple ISAs forces the development environment to make use of multiple compilation tools, or provide an interpreter or virtual machine environment. Using an interpreter or virtual machine environment can lead to execution inefficiencies due to interpreter overhead [8]. To avoid this overhead we chose to create a specialized compilation technique similar to those used by IBM's Cell and Intel's EXOCHI [4, 9] to embed heterogeneous binaries into a single executable image. An

Fig. 5. Heterogeneous Design Flow (Compilation)

advantage of our design flow is it's use of unmodified GNU compilers and linkers to produce and embed heterogeneous executables.

Our design flow allows users to work from a single application, with threads seamlessly mapped to processors with different ISAs by the run-time scheduler. The build system shown in Figure 5 allows developers to work within a single application with the thread bodies, libraries, and other support functions automatically passed through separate compilers. After being compiled into position independent code, each ISA-specific executable is embedded into a single heterogeneous executable using a set of command line tools.

The embedding process takes an executable file, or ELF, of one architecture and embeds it into the ELF of a different architecture [9]. Embedding is akin to heterogeneous linking, in which pertinent symbols within an embedded executable are made accessible to a host executable. Symbol table information, such as the addresses of thread start functions must be extracted as the standard GNU compilers used do not have heterogeneous linking capabilities. Additionally, threads are not first-class objects in languages such as C, as thread start functions are represented as function pointers. Therefore each embedded ELF must contain a set of translated function pointers, called thread handles, that correspond to the heterogeneous versions of the embedded threads. These thread handles can be used by the scheduler to reference thread implementations targeted for specific ISAs.

A linkable C-header file is produced after flattening and symbol extraction. The file contains a pre-initialized array to hold the binary version of the flattened ELF. All thread handles are added to the file, where each thread handle is a

pointer into the embedded ELF. A program can link against this C-header file and use the thread handles to create heterogeneous threads from within a single application. This design flow does not require modifications to be made to the compiler, and the resulting executable does not require run-time interpretation or RPC mechanisms to invoke code on processors of different ISAs.

Users can run our compilation flow from our web page interface. The source for our compilation flow is also available for those who wish to install and run locally or perform customizations. For these users we make available a set of utilities that enable the construction of automated or semi-automated build systems for heterogeneous systems. The process of creating heterogeneous embedded header files has been encapsulated within a new tool that can be added to traditional software build systems. Currently, the tools are implemented in Python, and contain hooks for altering the CPU specific binary utilities that are invoked. The hooks include definitions for: Input Architecture Type, Embedded Object Names, Object Copy Tools (GNU objcopy), Symbol Table Tools (GNU nm), Code Formatting and Hex Dump Tools (xxd).

The run-time system and compilation flow includes a dynamic dispatch capability that enables the run-time scheduler to treat all processors regardless of ISA as a uniform pool of available resources for scheduling threads. The scheduler maintains a dynamic dispatch table, which contains the start address, or thread handles for each ISA version of the thread, and a list of all system resources. Threads can then be transparently scheduled on any available resource, regardless of ISA. Figure 6 shows the flow for enabling dynamic dispatch.

5. RESULTS

We show the utility of our automation capabilities by highlighting partial results from two studies performed to evaluate MPSoPC architectures. No definitive conclusions should be drawn from these partial results as there intent is only to highlight the use and utility of our automated flow. The first example is from a study to understand the performance differences between SMP and NUMA architectures on three typical benchmark programs. The second example is from a study to understand the performance differences between our HW/SW co-designed microkernel and a traditional Remote Procedure Call (RPC) approach to resolve differences in processor heterogeneity. Each system was generated using Arch.Gen and programmed through our heterogeneous compilation flow. Tests were run on both Xilinx Virtex-5 and Virtex6 boards (ML507/ML605). The different hardware configurations were created in seconds, not the hours that would have been required using traditional by-hand methods within the CAD tool. No changes to the source code

Fig. 6. Autonomous Scheduling

Table 1. SMP vs. NUMA (Six slave processors).

	Memory Hierarchy	Resource Requirement		Execution Time (μs)	Sp
		LUTs	BRAMs		
MM	Shared	34,051(22.6%)	286(68.8%)	12,711,517	48
	Distr.	33,555(22.3%)	100(24.0%)	263,677	
2D FFT	Shared	34,051(22.6%)	286(68.8%)	1,431,076	40
	Distr.	34,051(22.6%)	92(22.1%)	35,685	
Canny Detection	Shared	26,015(17.3%)	192(46.2%)	300,491,488	26
	Distr.	26,633(17.1%)	116(27.9%)	11,136,755	

were required to conduct the tests on the different boards. This enabled the use of a common code base that eliminated the aliasing that occurs when hand customizing applications to run on each individual system.

5.1. SMP versus NUMA

Table 1 shows comparative area and performance results between SMP and NUMA systems running three common benchmark programs. Each MPSoPC system contained six slave processors. Again, only six slave processors were used due to the limitation of the MPMP for the cache-based SMP system. The benchmarks were written in C and PThreads and recompiled for each architecture using our heterogeneous compiler. Both systems used the common hthreads microkernel operating system. Both systems required approximately equal numbers of LUTs. The NUMA system required fewer BRAMs (35%-60%) and offered better performance compared to the SMP system in all cases.

Fig. 7. RPC versus Microkernel OS Calls

5.2. RPC versus Microkernel

Figure 7 is from a study evaluating the overhead of operating system calls. The RPC represents typical approaches such as [5] to resolve processor heterogeneity that require slave processors to request operating system services using RPC methods to a full protocol stack running on the master node. The test shows the latency of mutex operations, and how this latency effects a particular program’s scalability. The long latencies and contention for shared resources of the single hosts-node prevent the use of RPC methods from scaling beyond 8 processors. Our hardware microkernel significantly reduces the service access latency, allowing programs to achieve near linear scaling for up to the maximum of our test, 32 processors [1].

6. CONCLUSION AND FUTURE WORK

We have outlined a new, cloud-based tool flow used to automatically create complete heterogeneous Multiprocessor System-On-Chips (MPSoCs) systems for next-generation Platform FPGAs. Each generated MPSoPC includes all support peripherals, busses, processors, and a co-designed, PThreads-compliant microkernel OS. Our tool set includes a heterogeneous compiler that allows users to write high-level programs, and through re-compilation, run them on the different architectures created. This type of automation will be critical for handling the complexities of constructing future systems with 100’s-1000’s of CPUs. Programmers with no hardware expertise can quickly create and program their MPSoPC using a standard PThreads programming model. For designers seeking additional performance through customization, we provide all design files needed for vendor-specific CAD flows. Our design flow freely accessible at <http://hthreads.csce.uark.edu/ARCHlang>. Our future work focuses on automatically extracting program attributes from high-level programming models to generate tuned architectures, and automatic inclusion of custom accelerators and

vector processors. These enhancements will enable automatic creation of SIMD/MIMD/custom accelerator MPSoPC systems.

7. REFERENCES

- [1] J. Agron and D. Andrews, “Building Heterogeneous Reconfigurable Systems With a Hardware Microkernel,” in *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. New York, NY, USA: ACM, 2009.
- [2] B. Senouci, A. Kouadri M, F. Rousseau, and F. Petrot, “Multi-CPU/FPGA Platform Based Heterogeneous Multiprocessor Prototyping: New Challenges for Embedded Software Designers,” *The 19th IEEE/IFIP International Symposium on Rapid System Prototyping, 2008. RSP ’08*, pp. 41–47, June 2008.
- [3] C. H. Crawford, P. Henning, M. Kistler, and C. Wright, “Accelerating Computing with the Cell Broadband Engine Processor,” in *CF ’08: Proceedings of the 2008 Conference on Computing Frontiers*. New York, NY, USA: ACM, 2008, pp. 3–12.
- [4] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang, “EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-Core Multithreaded System,” *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 156–166, 2007.
- [5] E. Luebbers and M. Platzner, “ReconOS: An RTOS Supporting Hard- and Software Threads,” *Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL)*, August 2007.
- [6] J. Agron, W. Peck, E. Anderson, D. Andrews, E. Komp, R. Sass, F. Baijot, and J. Stevens, “Run-Time Services for Hybrid CPU/FPGA Systems On Chip,” in *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS)*, December 2006.
- [7] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, “Achieving Programming Model Abstractions For Reconfigurable Computing,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 34–44, January 2008.
- [8] M. A. Ertl and D. Gregg, “The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures,” in *Euro-Par ’01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*. London, UK: Springer-Verlag, 2001, pp. 403–412.
- [9] A. E. Eichenberger, J. K. O’Brien, K. M. O’Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo, “Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine™ Architecture,” *IBM Systems Journal*, vol. 45, no. 1, pp. 59–84, 2006.