LEAP: A Virtual Platform Architecture for FPGAs

Angshuman Parashar⋆, Michael Adler⋆, Kermin E. Fleming†, Michael Pellauer† and Joel S. Emer⋆†

⋆VSSAD Group
Intel Corporation
Hudson, MA
{angshuman.parashar, michael.adler}@intel.com

†Computation Structures Group, CSAIL
Massachusetts Institute of Technology
Cambridge, MA
{kfleming, pellauer, jsemer}@csail.mit.edu

***Abstract*—FPGAs are known to be very effective at accelerating certain classes of algorithms. A variety of FPGA platforms are available today, but because of the absence of a standardized platform architecture, each platform comes in the form of a board with a diverse set of devices and communication endpoints. Therefore, FPGA programmers typically have to spend significant effort in building interfaces to devices and adapting their applications to work with the semantics of these devices. Further, an FPGA board by itself is in many cases incapable running full real-world applications – software support is required. Working out communication protocols between the FPGA and software is another unnecessary time sink for programmers who would rather focus on the high-level functionalities of their applications. Finally, there is little support for building and allocating flexible memory hierarchies on FPGA platforms. All of these problems are further exacerbated by the fact that switching FPGA platforms usually requires the programmer to re-do a significant portion of this work.**

These are all non-issues for software programmers who live in a world of block and character devices, hardware-managed memory hierarchies with rich memory management libraries, and a plethora of portable communication protocols.

We attempt to bridge this gap between platform support for software and FPGA application development by proposing LEAP Virtual Platforms. LEAP (Logic-based Environment for Application Programming) provides an FPGA application with a consistent set of useful services and device abstractions, a memory hierarchy and a flexible communication protocol across a range of FPGA physical platforms. Tying these functionalities together is a modular development and build infrastructure. In this paper we describe the services provided by LEAP and explain how they are implemented using a multi-layered stack of abstractions.

I. Introduction

Software programmers today enjoy a rich programming environment comprised of sophisticated tools and platform abstraction layers that allow them to express the high-level semantics of algorithms without having to worry about the quirks of the hardware platform their code is going to run on. These include high-level language compilers, device abstractions, automated resource management (e.g. memory allocators, CPU schedulers), standardized communication protocols and a plethora of libraries.

This is in stark contrast to the programming environment from a few decades back, when programmers coded in assembly and had to learn how to interact with each new platform device via a thin layer of abstraction. The increasing speed and capabilities of hardware platforms also made them more complex over the years, which necessitated the development of abstraction layers in order to maintain the tractability of programming complex real-world applications.

Today, the power and thermal walls that processors are running into are motivating researchers to explore alternative fabrics for computation. Due to their capacity to support fine-grained parallel computation, FPGAs are known to be compelling as compute fabrics for certain classes of algorithms. Because FPGAs have traditionally been used in ASIC logic replacement and prototyping roles instead of algorithmic computation, the middleware, abstraction layers and tools that facilitate ease of programming are nowhere close to the level of sophistication found in the software world.

Thus, FPGA programmers have to write applications in languages more analogous to assembly than C. They have to deal with complex, quirky, low-level and often non-standardized interfaces to talk to devices and communication endpoints. Application programmers cannot be effective when they have to worry that the constraints for a DDR2 memory controller's DQ read capture flops are correct, instead of being able to malloc() a region of hardware-cached memory.

Unfortunately, this is a chicken-and-egg problem. Until FPGA use for application programming gains widespread adoption, there isn't much incentive to develop standardized layers of middleware. On the other hand, the absence of such middleware is one of the factors that hinders the widespread adoption of FPGAs for computing.

In this paper we present LEAP (Logic-based Environment for Application Programming), our attempt at breaking this deadlock by re-creating a small but critical subset of the software programming environment on FPGA platforms. LEAP is a Virtual Platform that provides a consistent set of interfaces and functionalities to an FPGA application across a range of physical FPGA platforms. Some of the core functionalities provided by LEAP are:

- A library of services and device abstractions that allow the FPGA application to talk to software processes and hardware devices using stylized interfaces without worrying about particular platform-specific device semantics (analogous to standard libraries and block/character device drivers in software).
- A multi-layered distributed and flexible cache/memory hierarchy for the FPGA that can make use of on-chip Block RAM, on-board memory banks and host memory with support for dynamic allocation and de-allocation (analogous to multi-level hardware caches and libc mem-

ory management in software).
- A protocol that allows programmers to communicate between an FPGA and a software process using user-defined types and method interfaces (analogous to Remote Procedure Calls [7] in software).

LEAP is implemented as a hierarchy of abstraction layers that have been carefully delineated to maximize code reuse across implementations on multiple FPGA platforms. LEAP has been implemented for a range of FPGA platforms using Bluespec System Verilog (BSV) [8] for FPGA code and C++ for software. All implementations are currently available for download [1] under the GNU General Public License (GPL) version 2.

II. Motivation

The richness and sophistication of the software programming environment is evident even from the following simple C program.

```
int main (int argc, char* argv[]) {
    int n = atoi(argv[1]);
    for (int i = 0; i < n; i++)
        printf("Hello, world!\n");
    return 0;
}
```

The compiler, standard C libraries and operating system provide a large amount of behind-the-scenes support to execute this simple-to-write program. The loader loads the program into memory and starts execution at the `main` symbol. Command-line arguments parsed in from the shell are passed into the program. The `printf` routine makes a system call to add the string to an output buffer that eventually gets channeled to the standard output stream.

An FPGA "program" to achieve the same results is far more involved in the absence of similar platform support. Using a high-level description language such as Bluespec System Verilog [8] goes a long way towards simplifying the specification of the program behavior itself. LEAP attempts to provide the missing platform support to further bridge the gap in development effort between FPGA and software application development. A BSV Hello World program using LEAP would look like the following (slightly stylized for presentation):

```
def STREAMS.HELLO "Hello, World!\n";
...
%param --dynamic N "number of iterations"
...
module mkApplication();
    ParamNode pnode <- mkParamNode();
    Param#(8) n <- mkParam(`PARAM_N, pnode);
    StreamsNode out <- mkStreamsNode();
    Reg#(bool) initialized <- mkReg(False);
    Reg#(Bit#(8)) count <- mkReg(0);

    rule init (!initialized);
        initialized <= True;
        count <= n;
    endrule

    rule hello (initialized && count != 0);
        out.makeRequest(`STREAMS_HELLO);
        count <= count - 1;
    endrule
endmodule
```

This example demonstrates the use of two LEAP services, *Parameters* and *Streams*. An array of such useful virtual services forms the standardized cross-platform interface that LEAP provides to an FPGA programmer on every supported FPGA platform.

We describe these services and their functionalities in greater detail in Section III. Next, we describe how these services are implemented in a layered manner in Section IV, following which we discuss our development and build infrastructure in Section V. Finally, we cover some related work and conclude the paper.

III. LEAP Services

A. Standard Platform Services

LEAP offers a library of cross-FPGA-platform services to an FPGA application. Some services provide platform support for tasks such as application initialization, parameterization, assertion-handling and dumping of statistics. Others serve as virtual device abstractions such as output streams and a front panel. LEAP also provides a service for organizing memory blocks on the FPGA itself, the FPGA board and host memory into a flexible shared cache hierarchy. We describe a few key services in this paper and encourage the reader to download LEAP from [1] and experiment with the full range of services.

1) Initialization and Startup: Almost all LEAP applications are hybrid FPGA/software applications that are launched from a software terminal. LEAP first programs the target configuration bitstream onto the FPGA using a platform vendor-provided API hook or script. Unnecessary re-configuration can avoided on some platforms by maintaining a hash of the last-loaded bitfile. If the hash matches, LEAP sends a soft-reset signal to a monitor module on the FPGA. All of this is completely hidden from the end user.

Post-initialization, LEAP provides the user with a Starter service that an FPGA module can use to synchronize with software. The FPGA receives a Start() method call through the service that indicates that the software is ready. At the end of application execution, an FPGA user module can call the Stop() method on the Starter to terminate the software process.

2) Parameters: As demonstrated in the BSV example from the previous section, the Parameters service receives dynamic parameters from the command line at application launch. This allows for dynamic parameterization of the application without re-synthesizing the design.

Multiple FPGA user modules could desire their own set of dynamic parameters. LEAP provides a clean and intuitive way to accomplish this. A user first declares a module's parameters in a meta language. Inside the module, the user instantiates a parameter receiver node, and then instantiates a data structure (essentially a typed register) for each parameter and associates it with the receiver node. Dynamic parameters automatically get filled into the corresponding register at initialization.

Behind the scenes, the software side of the LEAP Parameters service reads in the parameters from the command line, tags them with the recipient's module ID on the FPGA and sends them to the FPGA via LEAP communication channels (described shortly). On the FPGA, LEAP instantiates a centralized Parameters controller. During compilation, FPGA user modules that instantiate a parameter receiver node are collected together using Soft Connections [18], an abstraction for automatically generating networks using static elaboration. These nodes are connected to the central Parameters controller, from which they receive the stream of parameters. Many other LEAP services (e.g., statistics, assertions, debugging messages, etc.) also make use of the automated network generation using Soft Connections.

B. Scratchpads

The state of memory management on reconfigurable logic is woefully unadvanced. FPGA synthesis tools support relatively easy management of on-die memory arrays. But what if an algorithm needs more memory than is available on-die? At best, designers are offered low-level device drivers for embedded memory controllers, embedded PCIe DMA controllers or some other bus. Building an FPGA-side memory hierarchy is treated as an application-specific problem. On general purpose CPU-based hardware the memory hierarchy is invisible to a software application, except for timing. A similar memory abstraction, identical to the interface to on-die RAM blocks but implementing a full storage hierarchy, is equally useful for a range of FPGA-based applications.

LEAP provides a service called Scratchpads [6] that dynamically allocate and manage multiple, independent, memory arrays in a large backing store. Scratchpad accesses are cached automatically in multiple levels, ranging from shared on-board, RAM-based, set-associate caches to private caches stored in FPGA RAM blocks. In the LEAP framework, scratchpads share the same interface as on-die RAM blocks and are plug-in replacements. Additional libraries support heap management within a storage set. Like software developers, accelerator authors using scratchpads may focus more on core algorithms and less on memory management.

LEAP defines a single, timing insensitive, interface to memory. The same write, read request and read response interface methods are used for any memory implementation defined by the platform, along with the predicates governing whether the methods may be invoked in a given FPGA cycle. The simplest memory device allocates an on-die RAM block. However, LEAP memory stacks sharing the same interface can be configured for a variety of hierarchies. The most complicated has three levels: a large storage region such as virtual memory in a host system, a medium sized intermediate latency memory such as SDRAM controlled by an FPGA, and fast, small memories such as on-FPGA RAM blocks. Converting a client from using on-die memory to a complex memory hierarchy is simply a matter of invoking a different memory module with identical connections.

C. Dictionaries

LEAP provides a namespace management tool for generating unique identifiers. This dictionary tool was originally conceived for mapping integer identifiers to strings in order to trigger printing of messages on a host from an FPGA without having to specify hardware logic for passing variable length strings. We have extended it to solve the general problem of managing identifier spaces, including syntax for managing numerically dense subspaces.

An example of a dictionary specification is:

```
def MESSAGE.HELLO
    "Hello, world!\n";
def MESSAGE.ERROR
    "Encountered error %d in cycle %d.\n";
```

The dotted notation represents numerically dense subregions.

Dictionaries are internally used by several LEAP services such as Parameters, Streams and Scratchpads for naming and identification. For example, using LEAP dictionaries and conventions, an implementor allocating scratchpad IDs would specify:

```
def VDEV.SCRATCH.FBUF_Y "Frame buffer Y";
def VDEV.SCRATCH.FBUF_U "Frame buffer U";
def VDEV.SCRATCH.FBUF_V "Frame buffer V";
```

in order to allocate a group of scratchpads named *FBUF_Y*, *FBUF_U* and *FBUF_V*.

D. Remote Request-Response

While platform services like scratchpads and parameters together provide a number of utilitarian services for FPGA applications, there are often occasions where a more general communication protocol between an FPGA module and a software module prove to be useful.

Most of LEAP's services are in fact implemented on top of a typed asynchronous request-response protocol [17] called RRR (for Remote Request Response) that allows typed method-call-like communication between an FPGA and a software process. Similar to Remote Procedure Calls [7], the user defines *services* whose servers reside on either the FPGA or in software, with the client residing at the opposite end. The user defines the interface exported by each server, as shown in the following example:

```
service ISA_EMULATOR {
    server fpga <- cpu {
        method UpdateRegister(in REGINFO rinfo);
    };
    server cpu <- fpga {
        method Sync(in REGINFO rinfo);
        method Emulate(in IINFO iinfo,
                       out IADDR newPc);
    };
};
```

At compile time, RRR stub compilers generate the marshaling, demarshalling and multiplexing code that plumb the user code into underlying LEAP communication channels.

RRR is also exposed to the programmer as part of the standard LEAP interface.

RRR is implemented on top of a set of fixed-width bidirectional virtual communication channels between the CPU and the FPGA that we call *ChannelIO*. This is explained in more detail in the next section on LEAP implementation.

IV. LEAP Implementation

LEAP is built up as a stack of abstraction layers starting with low-level FPGA platform devices and ending with the high-level services described in the previous section. A pictorial representation of the hierarchy is depicted in Figure 1.

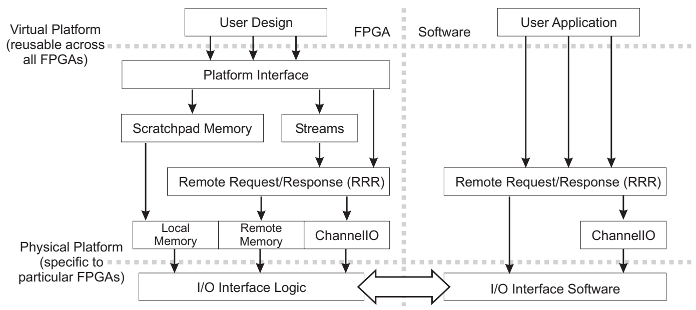

Fig. 1. LEAP abstraction hierarchy. I/O Interface Logic refers to the collection of physical devices available on the platform.

A. Physical Devices

Every production FPGA platform ships with a set of peripheral interfaces which include devices such as LEDs and switches, on-board memory banks and communication interfaces such as ethernet, PCI-express or serial I/O endpoints. The platform vendor usually includes a layer of gateware to communicate with these devices. This gateware could be a anything from direct connections to devices such as LEDs and switches, to sophisticated state machines like memory controllers.

We wrap the vendor-provided gateware in thin layers of abstraction called *physical devices*. The entire LEAP hierarchy is built on top of the interfaces exported by these devices. Because the set of devices on each FPGA platform could be very different and use a diverse range of semantics, user applications in the absence of a virtual platform could require significant work to port across multiple FPGA platforms. Our goal is to build one or more layers of abstraction such that a change in the physical device interfaces of the platform has minimal impact on the functionality of the user application. At the same time, mandating a re-write of the entire LEAP implementation for each FPGA platform is also a non-solution. We therefore partition the LEAP implementation into a carefully-organized hierarchy of abstraction layers that are intended to minimize their own re-implementation across multiple platforms.

B. Physical Platform Interfaces

The Physical Platform Interfaces are a collection of modules built on top of a subset of physical devices, in particular, devices providing communication and memory-access services. These modules expose the functionalities of the underlying platform-specific devices in a more stylized form by wrapping them in general cross-platform interfaces. These modules thus form an abstraction layer that separate the platform-specific layers of LEAP from the platform-independent layers.

1) ChannelIO: Using one or more physical devices available on an FPGA platform (e.g., a PCI-Express bus), we implement a primitive fixed-width bi-directional communication channel between the FPGA and a host software process. We call this a *physical channel*. The interface to a physical channel's endpoint consists of simple fixed-width read() and write() methods on the FPGA side, and both blocking and non-blocking versions of the same methods on the software side.

A working physical channel between the FPGA and a software process acts as a bootstrap for the remainder of the LEAP layers. Almost all of the LEAP functionality becomes available – most notably the RRR communication protocol – the moment this channel starts functioning. This greatly eases the development and debugging effort in bringing up other platform functionalities.

The physical channel can be multiplexed into multiple virtual channels. Sharing a communication medium between multiple clients necessitates some form of flow control in order to avoid deadlocks. Our channel implementations leave the burden of ensuring deadlock-free operation to the channel users because we found the buffering requirements for flow control to be prohibitively expensive. However, alternative buffered channel implementations could be shipped with a LEAP distribution and users could pick an implementation suitable to their design. We call the LEAP module exporting the set of virtual channels *ChannelIO*.

2) Local Memory: FPGA boards often ship with banks of on-board SRAM or DRAM. The controllers required to interact with these devices are instantiated in FPGA gateware and in LEAP are organized as part of the memory's physical device module. The interface exported by various flavors of SRAM and DRAM controllers are sufficiently different for higher-level LEAP layers to benefit from a thin layer of abstraction that we call Local Memory. The interface consists of methods to read and write at parameterizable word and line granularities. Reading uses a request-response protocol. Masked writes are also supported.

The availability of on-board memory varies from platform to platform. LEAP provides multiple implementations of the local memory abstraction to cater to these platform variations. The interface stays the same but the application writer may tailor physical storage to different hierarchies.

Higher-level LEAP services such as Scratchpads use this Local Memory interface to talk to on-board memory banks.

3) Remote Memory: Remote Memory provides an interface for the FPGA to talk to host system memory on platforms that support direct memory access, e.g., PCI-Express based FPGA platforms and Intel Front Side Bus (FSB) based platforms such as the ACP M2 [15]. The interface supports both line read/write as well as a larger granularity burst read/write for long sequential DMA transactions.

Depending on the organization and semantics of the physical devices on the particular platform, Remote Memory and Physical Channel may have to share the use of one or more devices to implement their functionalities. For example, on an

FSB-based FPGA platform, certain regions of host memory could be used to implement a channel while other regions could be exposed to the FPGA via Remote Memory. In such a scenario, partitioning of the address space could take place either within the physical device implementations, or in the remote memory and physical channel implementations.

C. Remote Request-Response

A set of virtualized channels between the FPGA and a software process provides a primitive communication medium between the two compute nodes. The RRR framework is built on top of these channels.

The RRR implementation instantiates a client and server manager on the FPGA and in software and uses one LEAP virtual channel to communicate between each client-server pair. Each manager module multiplexes its virtual channel further into the number of services it manages. The managers can choose from among a number of arbitration protocols to multiplex traffic from the numerous services into the virtual channels. Our current implementation supports static priority and round-robin scheduling.

The client and server managers instantiate a client or server stub for each service. These FPGA- and software-side stubs are generated during compile time based on the set of RRR specifications given by the user across the entire application. The stubs are responsible for marshalling and demarshalling the multiple typed arguments and return values for each method call, and packetizing and de-packetizing them to stream them through the virtual channels.

Implementation of high-level LEAP services such as the Starter, Parameters, Scratchpads, etc., is facilitated by the naming, multiplexing and typed method-call-style communication services provided by RRR.

D. Supported FPGA Platforms

LEAP currently supports the following FPGA platforms:

- Nallatech ACP M2 [15]
- HiTech Global v5 PCI-Express [3]
- Xilinx XUPv2 and XUPv5 [5]
- Xilinx ML605 (work-in-progress) [4]
- Altera DE2 and DE3 [2]
- Altera DE4 (work-in-progress) [2]
- Altera Arria II GX [2]
- Hybrid Bluesim/Software simulation

To support these different platforms, a LEAP distribution contains a number of modules providing alternative implementations of the same interface, particularly for the lower-level platform-specific physical device interfaces. Managing these modules is a non-trivial problem. Another problem is that swapping device modules is not sufficient for seamless FPGA platform portability if the FPGAs are from different vendors because each vendor has their own synthesis, place and route tool-chain.

LEAP addresses these problems by using a sophisticated set of infrastructural tools for modular application development and building. We discuss these in the following section.

V. LEAP Infrastructure

A. Architect's Workbench

The Architect's Workbench [11], [10] or AWB is a structured hybrid application development framework that focuses on supporting modularity and code reuse. AWB was originally conceived as a framework for the development of software performance models. It was later extended to support FPGA-based modeling [9], [20], [19] in specific and hybrid FPGA-software application development in general. Applications are represented in AWB as a hierarchical tree of modules, where each module can be replaced with alternative implementations that satisfy the interface requirements of the module. This "plug-N-play" functionality allows a variety of applications to be constructed out a common pool of modules. AWB allows these modules to be obtained from an arbitrary set of independently-maintained source-code repositories.

AWB's support for modular plug-N-play synergizes nicely with LEAP's goal of providing a common set of virtualized interfaces across a range of platforms via alternative implementations of the same interface. A LEAP configuration for a particular FPGA platform is represented by an AWB module hierarchy comprising of module selections to implement the required functionalities on that platform (though many of these modules are shared across multiple platforms). Instances of these configurations for every supported FPGA platform are provided standard as part of a LEAP distribution. Porting a user application from one FPGA platform to another requires a single double-click to switch the platform configuration and a button click to start the automated build/synthesis process.

Swapping platform configurations via AWB plug-N-play is also useful for debugging. *Hybrid Simulation* is one of the available platform configurations, which runs the FPGA part of the application in an RTL simulator that communicates with a software process via a UNIX pipe. Users can debug their applications in this environment without having to go through a synthesis process.

B. Build Pipeline

The proper use of the EDA tools, particularly the use FPGA tools, is difficult. Each of the several tools in a tool chain may have dozens of options, some of which are critical and some of which are obsolete. Makefiles for these processes, when extension is possible or even considered, quickly become incomprehensible. High-level FPGA IDEs automatically produce makefile schema, but these are difficult to extend, even when using tools from the same vendor. These concerns represent a serious impediment to the FPGA designer, particularly new users or those accustomed to more refined software build systems. LEAP addresses these issues by providing a parallel, extensible, transparent and, best of all, automated build system.

LEAP achieves these goals by requiring each FPGA environment to provide its own, environment-specific build specification. For example, each Xilinx environment carries with it a build specification targeting Xilinx tools like XST.

Users need not concern themselves with specifying a build procedure beyond selecting the platform which they are targeting. Clicking on configure and build buttons within the AWB GUI will produce a functional FPGA implementation for an FPGA environment. For those few users extending the build system, LEAP provides proper programming abstraction with an accompanying API and data-structures which permit users to integrate new tools and to compose tool chains. For example, the LEAP build pipelines for Xilinx and Altera may share tool modules like Synplify, but may also use their own vendor-specific synthesis tools, in the same plug-n-play manner as hardware modules. We view this extensibility as a major feature. By creating a new build specification in LEAP, a tool developer automatically gains access to the large and growing number of designs implemented on the LEAP infrastructure. Of course, to the end user, these changes are observable in the quality of results.

The basic build unit in LEAP is the "Synthesis Boundary", a stylized hardware module having only LEAP-managed latency insensitive FIFO connections its I/O interface. All tools operate on this abstraction, though they may also aggregate the several modules of a design together in their operation, for example linking object code to produce a final binary. By operating on synthesis boundaries as first class objects, we actually obtain a good degree of parallelism in the build process. For all but a few processes, the tasks for each synthesis boundary remain independent, permitting LEAP to stripe jobs across cores in a single machine or to run jobs in a batch system such as Condor.

VI. Related Work and Concluding Remarks

Many of the services provided by LEAP (such as RRR and Scratchpads) have been inspired by decades-old innovations in the software programming environment. There have been several related efforts at creating FPGA middleware for hardware abstraction and/or communication [21], [13], [23], some of which are targeted at domain-specific applications. BORPH [22] maps FPGA registers and RAM blocks into registers and memory regions in the host operating system kernel. RDL [24] provides a language and framework for describing module interfaces and timing characteristics of an FPGA-based performance model.

We believe LEAP is unique because it provides a cohesive package of cross-FPGA-platform functionalities to virtualize the FPGA platform starting with device abstraction layers and extending to a communication protocol, a range of useful platform services and a memory hierarchy, all tied together with an easy-to-use development and build infrastructure.

LEAP currently supports a range of FPGA platforms (listed in Section IV) and is being actively used at Intel, MIT, and Seoul National University for a variety of FPGA projects including the HAsim performance modeling framework [19], an H.264 decoder[12], an OFDM framework [16], an SSD [14], and several class projects.

References

[1] [Online]. Available: http://asim.csail.mit.edu/redmine/projects/show/leap

[2] "Altera development and education boards." [Online]. Available: http://www.altera.com/education/univ/materials/boards/unv-dev-edu-boards.html

[3] "Hitech global." [Online]. Available: http://www.hitechglobal.com/

[4] "Virtex-6 fpga ml605 evaluation kit." [Online]. Available: http://www.xilinx.com/products/devkits/EK-V6-ML605-G.htm

[5] "Xilinx university program." [Online]. Available: http://www.xilinx.com/university/

[6] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer, "Leap scratchpads: Automatic memory and cache management for reconfigurable logic," in *19th Annual ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2011)*, February/March 2011.

[7] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39–59, 1984.

[8] Bluespec, Inc., "Bluespec system verilog reference guide," *http://www.bluespec.com/*, 2007.

[9] N. Dave, M. Pellauer, and J. Emer, "Implementing a functional/timing partitioned microprocessor simulator with an fpga," in *2nd Workshop on Architecture Research using FPGA Platforms (WARFP 2006)*, February 2006.

[10] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan, "Asim: A performance model framework," *Computer*, vol. 35, no. 2, pp. 68–76, 2002.

[11] J. Emer, C. Beckmann, and M. Pellauer, "Awb: The asim architect's workbench," in *3rd Annual Workshop on Modeling, Benchmarking, and Simulation (MoBS 2007)*, June 2007.

[12] K. Fleming, C. Lin, N. Dave, Arvind, G. Raghavan, and J. Hicks, "H.264 decoder: A case study in multiple design points," in *MEMOCODE*, 2008.

[13] S. H. Kim, W. H. Tranter, and S. F. Midkiff, "Middleware for a distributed reconfigurable simulator," *Simulation Symposium, Annual*, vol. 0, p. 0253, 2002.

[14] S. Lee, K. Fleming, J. Park, K. Ha, A. Caulfield, S. Swanson, Arvind, and J. Kim, "Bluessd: An open platform for cross-layer experiments for nand flash-based ssds," in *The 5th Workshop on Architectural Research Prototyping*, ser. WARP, 2010.

[15] Nallatech, "Intel xeon fsb fpga socket fillers," *http://www.nallatech.com/intel-xeon-fsb-fpga-socket-fillers.html*.

[16] M. C. Ng, K. Fleming, M. Vutukuru, S. Gross, Arvind, and H. Balakrishnan, "Airblue: a system for cross-layer wireless protocol development," in *6th Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '10, 2010.

[17] A. Parashar, M. Adler, M. Pellauer, and J. Emer, "Hybrid cpu/fpga performance models," in *3rd Workshop on Architectural Research Prototyping (WARP 2008)*, June 2008.

[18] M. Pellauer, M. Adler, D. Chiou, and J. Emer, "Soft connections: Addressing the hardware-design modularity problem," in *46th Design Automation Conference (DAC 2009)*, July 2009.

[19] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "Quick performance models quickly: Timing-directed simulation on fpgas," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2008.

[20] M. Pellauer, M. Vijayaraghavan, M. Adler, . Arvind, and J. Emer, "A-ports: an efficient abstraction for cycle-accurate performance models on fpgas," in *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, 2008.

[21] X. Reves, V. Marojevic, R. Ferrus, and A. Gelonch, "Fpga's middleware for software defined radio applications," aug. 2005, pp. 598 – 601.

[22] H. K.-H. So and R. Brodersen, "Improving usability of fpga-based reconfigurable computers through operating system support," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, 2006, pp. 1 –6.

[23] F. Villanueva, D. Villa, F. Moya, J. Barba, F. Rincon, and J. Lopez, "Lightweight middleware for seamless hw-sw interoperability, with application to wireless sensor networks," apr. 2007, pp. 1 –6.

[24] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic, "Ramp: Research accelerator for multiple processors," *IEEE Micro*, vol. 27, no. 2, pp. 46–57, 2007.