The LEAP FPGA Operating System

Kermin Fleming*, Hsin-Jung Yang†, Michael Adler*, Joel Emer*†
*VSSAD Group, Intel Corporation
{kermin.fleming, michael.adler, joel.emer}@intel.com
†Computer Science and A.I. Laboratory, Massachusetts Institute of Technology
{hjyang, emer}@csail.mit.edu

***Abstract*—FPGAs offer attractive power and performance for many applications, especially relative to traditional sequential architectures. In spite of these advantages, FPGAs have been deployed in only a few, niche domains. We argue that the difficulty of programming FPGAs all but precludes their use in more general systems: FPGA programmers are currently exposed to all the gory system details that software operating systems long ago abstracted away.**

In this work, we present the Latency-insensitive Environment for Application Programming (LEAP), an FPGA operating system built around latency-insensitive communications channels. LEAP alleviates the FPGA programming problem by providing a rich set of portable latency-insensitive abstraction layers for program development. Unlike software operating systems services, which are generally dynamic, the nature of FPGAs requires that many configuration decisions be made at compile time. We present an extensible interface for compile-time management of resources. We demonstrate that LEAP provides design portability, while consuming as little as 3% of FPGA area, by mapping several designs on to various FPGA platforms.

I. Introduction

In contrast to modern software, current FPGA programming environments remain primitive. Many of the abstractions typically available in general purpose systems: high-level languages, abundant libraries, code portability, and automatic resource management, do not exist for FPGAs. Creating a design for an FPGA often requires that programmers start from bare metal, bringing up not only their application, but significant operating infrastructure including memory controllers, I/O systems, and debugging facilities. All of these activities require large amounts of time, slowing the FPGA development process to the point that the relative simplicity of software becomes very attractive. Worse, getting an application to work once is insufficient. The lack of portability of FPGA programs between platforms and generations means that building the next generation implementation usually requires significant re-implementation. Although FPGAs offer attractive power and performance relative to traditional sequential processors, they are not commonly used in systems architecture *because they are painful to program*.

To address these development issues, we propose the LEAP FPGA operating system. Like a software operating environment, LEAP provides both basic device abstractions for FPGAs and a collection of standard I/O and memory management services. These abstraction layers shield programs from the complex details of the underlying FPGA hardware while simultaneously providing design portability across all FPGAs supported by LEAP.

LEAP resembles general-purpose operating systems in function. However, because LEAP targets FPGAs, its implementation is necessarily very different. In general-purpose systems, user programs are organized into processes and threads that share a common execution substrate with the operating system: the processor and its memory system. Interaction between the program and the operating system generally occurs by storing data in memory and then changing contexts to the operating system by a function call. FPGAs differ from this general-purpose model in two ways, both of which profoundly affect the organization of LEAP.

In the FPGA, user programs and the operating system share neither a common execution substrate nor a common, global view of storage. Instead, FPGA programs are organized as spatially distributed modules, with portions of the FPGA fabric dedicated to each of the different functions of the program and the operating system. To accommodate this distributed programming model, LEAP's fundamental abstraction is *communication*. To enable portable, abstract communications, LEAP builds upon the concept of latency-insensitive design [1]. LEAP formalizes latency-insensitive design as *latency-insensitive (LI) channels*, programming constructs that provide point-to-point, reliable communication but do not explicitly specify the timing or the implementation of the communication. By decoupling the notion of communication from the physical and timing details of data transmission, latency-insensitive channels slacken the requirements of synchronous timing while preserving the parallelism intrinsic to hardware designs. This timing relaxation is critical in providing operating system abstractions on FPGAs, since the timing behavior of operating system services must necessarily change between platform and program configurations. We will demonstrate, by example, that latency-insensitive channels can capture most important operating systems functionalities in FPGAs while providing a user-friendly programming interface.

In general-purpose systems, new instructions can be introduced into a program at any time during execution. As a result, general-purpose operating system activities occur at program run time: context-switching is lightweight and operating system instruction flows incur overhead only when those instructions are executed. FPGAs are typically programmed only once per execution. While possible, including dynamic management layers within a static FPGA program incurs fixed area and performance overhead that lasts for the life of the FPGA program. To avoid these penalties, most resource management decisions in LEAP, for example memory and clock management, are made statically at compile time. This decision results in a deeper coupling of compiler and operating system. LEAP's static incorporation of resources resembles early software operating systems, which did require re-compilation to integrate new functionalities. A key contribution of this work is an extensible compiler interface that permits the integration of new operating systems services within the LEAP framework.

In this work, we outline a philosophy for the construction of FPGA systems that allows developers to focus on core algorithms, while retaining the full flexibility of FPGAs. In building platforms for FPGAs, there is much to be learned from general-purpose operating systems. Throughout the paper, we will highlight the similarities and differences between LEAP and traditional software operating systems and the importance

of communication to FPGA design. This work will also explore the unification of operating-systems features, many of which have been previously evaluated individually, into a conceptual whole that has never been available, previously, for FPGAs.

II. Related Work

Previous work on FPGA operating systems has focused on adding communications support within existing general-purpose operating systems. BORPH [2] views FPGA programs as UNIX processes that can communicate externally by means of UNIX pipes. HThreads [3] takes a similar processor-centric approach, in which fabric-based accelerators are treated as threads that coordinate with other activities on a soft processor. FSMLanguage [4] proposes a new domain-specific language for finite state machines. FSMLanguage abstracts communications between hardware and software FSM components, using channel constructs analogous to latency-insensitive channels. LEAP generalizes the concepts put forward in these works by supporting channel-based communication between arbitrary combinations of execution platforms, including multiple FPGAs. LEAP, BORPH, HThreads, and FSMLanguage all provide strong compilation support for mapping user programs to the FPGA.

In addition to communication, researchers have investigated memory abstractions for FPGAs. CoRAM [5] proposes a cache interface similar to LEAP's Scratchpad [6] interface. Whereas LEAP allows the user RTL to control the memory interface directly, CoRAM advocates control by way of control threads programmed using a C-like language. Unlike LEAP, CoRAM does not address communication and provides no support for shared memory within or among FPGAs. FSMLanguage provides a basic memory abstraction, but its treatment of memory is limited to scheduling the ports of in-fabric SRAM resources.

Both Xilinx and Altera have produced OpenCL [7] tool flows which simplify the coupling of a processor and FPGA. These flows allow a user to specify a C kernel, which the tool will then implement on the FPGA. We believe OpenCL is a good option for a class of kernel-based programs, and LEAP's latency-insensitive primitives can capture the kernel model of computation. However, OpenCL does not expose the full flexibility of FPGAs. For example, the kernel model proposed by OpenCL does not capture any design in which a kernel makes requests back to the host for service or in which kernels communicate with each other dynamically. LEAP's interfaces are intended to facilitate the expression of a more general class of parallel programs.

III. LI Channels: The LEAP Design Philosophy

Operating systems are built on abstraction. In software, memory serves as the primary operating system abstraction layer. Core features like virtual memory, communication, I/O, and dynamic library linking all occur through memory. Even communication between the user program and the operating system, in the form of system calls, occurs through memory. Memory-as-abstraction is enabled in part by the dynamic time-multiplexing capabilities of the processor and in part by support for sharing within the processor memory system.

In contrast to general-purpose systems, FPGAs have no intrinsic concept of a globally shared storage or execution infrastructure. Moreover, constructing such an infrastructure in the FPGA as a fundamental operating system abstraction layer is inefficient. FPGAs are comprised of small state and logic elements which are typically dedicated to a single task and which communicate with each other using wires.

Communication of data, as opposed to the storage of data in memory, is fundamental to the FPGA. However the basic communication primitives provided by the FPGA, wires, are not sufficiently abstract for an operating system. Wires in the FPGA are synchronous, and clock edges have concrete semantic meaning. Synchronous semantics present a problem for operating systems: the timing behavior of operating system services necessarily changes depending on the user program and the platform targeted. Although it is possible to automatically abstract away syncronicity [8] within an FPGA program, the overhead of such abstraction is very high. Thus, communication over pure wires cannot be used as a basic interface into the operating system. To enable FPGA operating systems, a higher-level abstraction for communication is required.

LEAP introduces latency-insensitive channels as its primary communication primitive. Latency-insensitive channels have operating behaviors and interfaces similar to the concurrent FIFO modules commonly available in hardware and software programming libraries – a simple enqueue and a simple dequeue operation along with some status methods, e.g. `notFull` and `notEmpty`, for use by the user program in determining when to send and receive data. Syntactically [9], latency-insensitive channels consist of named send and receive endpoint pairs, instantiated with the following syntax:

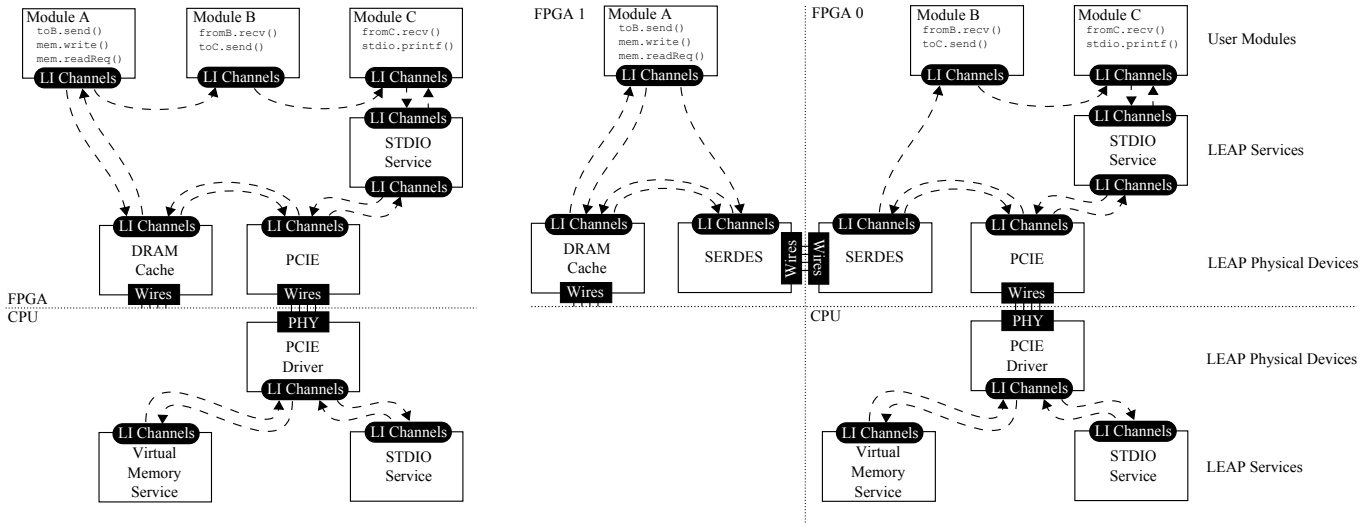
```
module mkA (Empty);
  SEND(Bit(42)) toB = mkSend("AtoB");
endmodule
module mkB (Empty);
  RECV(Bit(42)) fromA = mkRecv("AtoB");
endmodule
```

Semantically, these pairs represent a reliable, in-order channel from the sender to the receiver. Unlike library FIFOs, which have a fixed implementation within a given library, the latency-insensitive channel denotes abstract communication and makes only two basic guarantees. First, the channel guarantees reliable, FIFO delivery of messages. Second, the channel guarantees that at least one message can be in flight at any point in time. Consequently, a latency-insensitive channel *may* have dynamically-variable transport latency and arbitrary, but non-zero, buffering. At compile time, LEAP's compilation infrastructure (Section‘VIII) matches channel endpoints and produces physical implementations of the latency-insensitive channels.

LEAP leverages the indeterminate behavior of the latency insensitive channel in three ways. First, LEAP leverages the abstract nature of latency insensitive channels as a means of orchestrating communications between FPGA and CPU and across FPGAs (Section IV). Second, LEAP relies on latency-insensitive channels to safely decouple its services, like memory, from the user program (Sections V and VI). Finally, LEAP uses latency insensitive channels to abstract physical device interfaces, providing portability among platforms (Section VII).

IV. LEAP Communications

When a programmer instantiates a latency-insensitive channel, he asserts that the potentially variable timing behavior of the



(a) A Complete LEAP Program

(b) The Same Program, Mapped to Two FPGAs

Fig. 1: Two implementations of a LEAP-based program. Dotted lines represent the logical latency insensitive channels. These channels are inferred at compile time from the combination of user source and platform devices. LEAP targets multiple FPGA platforms and communicates with software by stretching channels across chip boundaries.

channel *does not impact the functional correctness of the design*. Thus, the semantics of latency-insensitive channels permit LEAP to select *any* channel implementation that preserves reliable, FIFO message delivery. Much like the POSIX `socket` interface, LEAP uses latency-insensitive channels both for communication between hardware modules within an FPGA and for communication with other types of programs on external platforms. At compile time, LEAP analyzes the user program and discovers channels that terminate externally. These channels are tied, via a synthesized communications network, to physical devices capable of communicating externally.

In addition to managing communication between hardware and software, LEAP leverages latency-insensitive channels to map user programs automatically across arbitrary sets of FPGAs [10]. This operation is analogous to the mapping of threads to a multi-core processor by a general purpose operating system. Mapping an FPGA program to multiple platforms first requires a mechanism for partitioning the program. Again, LEAP leverages latency-insensitive channels. We define a latency-insensitive module to be a region of a program that interacts externally only by way of latency-insensitive channels. LEAP views all user programs as collections of latency-insensitive modules, a subdivision that can be thought of as the FPGA equivalent of operating system process or thread management. To map a program across FPGAs, LEAP allocates the latency-insensitive modules comprising the program to the available FPGAs in the system based on area consumption and communication. As in the case of FPGA-processor communications, LEAP synthesizes a networking layer to transport messages between connected FPGAs. Figure 1(b) shows an example of a simple program that has been mapped to two FPGAs. As in software operating systems, no changes are necessary in the user program to make use of LEAP's partitioning.

Latency-insensitive modules offer a good balance between the abstraction needed for operating-system-level management and program expressivity. Moreover, this balance is achieved without a significant impact to program performance or area consumption: most latency-insensitive channels will be implemented as RTL FIFOs, preserving the performance of hand-coded RTLs. Latency-insensitive modules are already a common design paradigm in hardware systems, both at the system and micro-architectural levels. For example, SoCs targeting FPGAs are generally framed in terms of network-on-chip protocols, which are usually latency-insensitive. Within individual hardware blocks, it is a common design practice to decouple components with guarded FIFO interfaces. Because LEAP's latency-insensitive channel syntax and semantics resemble existing RTL structures, LEAP's latency-insensitive channel syntax can often be substituted directly into existing RTL.

Although LEAP primarily targets RTLs mapped to FPGAs with some support for software, latency-insensitive modules admit of a diversity of programming substrates and languages. So long as the external latency-insensitive interface semantics of a module are maintained by the programmer, modules may have any implementation internally that programmers require. Such implementations include not only RTL, but also arbitrary software either on an external host processor or an internal soft processor.

V. LEAP Scratchpads: Scalable FPGA Memory

Although we argue that abstract communication is the fundamental building block of FPGA programs, memory is critical to many applications. FPGAs offer a rich set of memory primitives: SRAM resources are available within the fabric, FPGA boards include off-chip DRAM, and host virtual memory may be accessible over a communications link. Unfortunately, these memory resources are often difficult to use, in part because FPGA programmers are fully exposed to the low-level details of the memory: the number of memory banks, the width of memory, timing, and the number of beats per memory access. Memory systems in general purpose machines are also complex, and large fractions of processor die area are consumed by memory management: caches, memory controllers, and virtualization hardware. However, most user programs in general purpose

systems are insulated from the complexity of the memory hierarchy through a simple, abstract load-store interface to virtualized memory. The internal complexity of the memory hierarchy improves throughput dramatically, but the simple interface to memory allows programmers to focus on algorithms and enables program portability.

LEAP provides a software-like in-fabric memory abstraction for FPGA programs. LEAP's memories have a latency-insensitive interface with three methods: read-request, read-response, and write. A LEAP program may instantiate as many memories as needed and these memories may have arbitrary size, even if the target FPGA does not have sufficient physical memory to cover the entire requested memory space. Programmers instantiate LEAP memories using a simple, declarative syntax:

```
// Instantiate two memory interfaces with private
// address spaces
module mkModuleA();
  MEM_IFC(ADDR,DATA) priv1 = mkMem();
endmodule
module mkModuleB();
  MEM_IFC(ADDR,DATA) priv2 = mkMem();
endmodule
```

LEAP provides two basic interfaces to memory: private and shared [11]. Each declaration of a memory resource creates a latency-insensitive interface to memory. For shared memories, programmers declare a named coherence domain and instantiate a LEAP coherence controller in addition to the memory interface itself.

LEAP's memory primitives are portable among and across FPGAs. Like the load-store interface of general-purpose machines, LEAP's memory interface admits of complex backing implementations. LEAP's memory interface does not explicitly state how many operations can be in flight, nor how quickly in-flight operations will be retired. As was the case in communications, this ambiguity provides significant freedom of implementation to the operating system. For example, a small memory could be implemented as a local SRAM, while a larger memory could be backed by a cache hierarchy and host virtual memory. LEAP leverages this freedom to build complex, optimized memory architectures on behalf of the user, bridging the simple user interface and complex physical hardware.

At compile time, LEAP aggregates the declared user memory interfaces into a cache hierarchy, making use of whatever physical memory resources are provided by the platform, as shown in Figures 1 and 2. This hierarchy is optionally backed by an interface to host virtual memory, as shown in Figure 1, which provides the illusion of an arbitrarily large address space. LEAP's backing memory hierarchy automatically multiplexes host virtual memory by explicitly assigning segments of virtual memory to each memory region declared in the user program, preventing independent memory regions from interfering with one another within the cache hierarchy.

VI. LEAP Service Libraries

One of the most attractive features of software programming is that minimal programs, e.g. `hello_world`, are small and easy to understand, even for a novice programmer. Contrast the conciseness and clarity of a basic software program with a basic RTL program running on an FPGA. The difference is stark. The

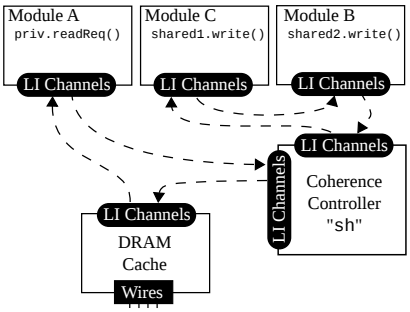


Fig. 2: The memory hierarchy inferred by LEAP for a program with a single private memory and two shared memories. Memories are organized as networks of latency-insensitive channels. All memory interfaces are able to take advantage of the large DRAM cache.

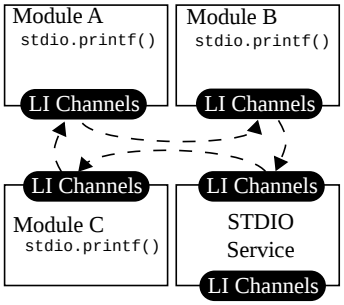


Fig. 3: STDIO, an example of a typical LEAP service library. Several user modules instantiate STDIO clients. Clients are aggregated by LEAP at compile time and attached to the STDIO server. In turn, this hardware server is connected to a software-side server, which implements the service library.

chief distinction between the software and RTL program is not the complexity of the binary: the software binary is comprised of tens of thousands of unique instructions. Rather, the perceived simplicity of software arises from the availability of good libraries, which mask system complexity through intuitive APIs and strong composability. By combining communications and memory primitives with an extensible compilation infrastructure, LEAP is able to provide libraries that rival software in their scope and simplicity. Indeed, the programmer-supplied portion of LEAP's `hello_world` application is only a dozen lines of code.

Although we argue that the LEAP libraries are as easy to use as software libraries, there are some obvious differences. Software libraries consist of instruction flows to which the user program gives control. LEAP service libraries take a distributed approach: service libraries consist of clients and servers which interact over latency-insensitive channels. The typical architecture of a service library, a central server and multiple clients, is shown in Figure 3. To make use of the service library, programmers instantiate client modules in their code. Client modules provide a latency-insensitive local interface framed in terms of request and response methods, as in Figure 4. Pairwise, these methods resemble software function calls, but are decoupled to enable hardware pipelining. At runtime, clients issue requests to the controller over the network. These requests are serviced by the controller, often by

```
interface STDIO(type data);
    // fopen is a request/response interface, returning the file handle
    Void fopen_req(GLOBAL_STRING_UID nameID, GLOBAL_STRING_UID modeID);
    STDIO_FILE fopen_rsp();

    // The list of arguments to printf may be up to STDIO_WRITE_MAX elements
    Void printf(GLOBAL_STRING_UID msgID, List(data) args);
    Void fprintf(STDIO_FILE file, GLOBAL_STRING_UID msgID, List(data) args);
endinterface
```

Fig. 4: A portion of the interface to an STDIO client. LEAP supports most common STDIO functions.

invoking a software routine using a latency-insensitive channel which terminates in software. Direct composition with software libraries, for example, using software to allocate physical memory or handling printing to a terminal, greatly simplifies the FPGA-side server implementation and underscores the value of good hardware-software communications support.

To facilitate the aggregation of library clients, LEAP provides a broadcast communication primitive: the latency-insensitive ring. Participants in the broadcast instantiate named ring stops. At compile time, ring stops are aggregated by name and connected in a ring topology via latency-insensitive channels. Rings are advantageous for service libraries both because they carry a low area overhead and provide a convenient way to describe library implementations in which the number of clients is unknown prior to compilation. Indeed, the overhead is so low that LEAP's debugger service library can instantiate debugger clients at every latency-insensitive channel in a program, enabling programmers to monitor channel state at runtime.

LEAP provides many basic service libraries, including assertions, statistics collection, command line parameters, locking, and synchronization. For brevity, we describe only Standard I/O (STDIO) in detail.

STDIO is one of the most fundamental libraries available in C and often serves as an introduction to software programming. LEAP's STDIO service, part of which is listed in Figure 4, provides the functional analog of STDIO in the FPGA. Like its software counterpart, LEAP STDIO provides a simple API which masks the complexity of actually writing formatted data to a file.

Programmers instantiate STDIO clients and invoke methods on them. Internally, STDIO clients marshal programmer commands into a packet format, which is then streamed over the service ring network to an STDIO server. The server transmits these packets to software on an attached processor. Software then invokes the appropriate C STDIO functions using the FPGA-supplied arguments.

One issue in Standard I/O is dealing with strings. Strings, which are logically unbounded, are expensive to manipulate directly in the FPGA. To avoid the area overhead of string manipulation in fabric, we borrow a tactic from C and cast strings as pointers. At compile time, string literals are replaced by pointers, the `GLOBAL_STRING_UID`. When FPGA programs manipulate strings, as in `snprintf`, they do so by passing string pointers to software. Software creates a new string and returns a pointer to the new string back to the FPGA. LEAP's string pointer management makes use of the SoftServices interface described in Section VIII to keep track of string pointers at compile time.

VII. LEAP Platforms

Physical devices are hardly abstract, whether they are attached to a general purpose processor or to an FPGA. Interfacing to these devices requires a deep understanding of device behavior, the details of which can leak back into user code if the interface programmer is not careful. This is especially true in FPGAs, where device timing details are often absorbed by user logic. For example, a design targeting an FPGA with an attached SRAM may absorb and come to rely on the fixed-latency behavior of a particular SRAM. As a result of this dependence, the design becomes unportable.

To deal with physical devices, software operating systems utilize hardware abstraction layers. Each device provides some basic API which is common among all devices of that type. LEAP adopts this approach. In LEAP, classes of physical devices each provide a uniform, abstract device interface. Rather than a call-based API, LEAP devices provide a latency-insensitive-channel-based API, usually framed in terms of request and response channels. Internally, any implementation may be chosen by the driver writer, but the external driver interface is constrained by the LEAP compilation flow to use only latency-insensitive channels.

LEAP-enabled FPGA platforms may be viewed as a collection of driver modules for these low-level devices, in much the same way that `linux/arch` is used to differentiate low-level interfaces to different processor architectures. For each target platform, LEAP accepts a platform description file which includes abstract drivers for each physical device on the platform. LEAP uses this configuration file at compile time to instantiate those drivers required by the user program. As with software, a platform description must be written only once per platform and may be shared by all programs targeting the platform. To represent systems with multiple platforms, LEAP simply bundles together descriptions of single platforms with a description of platform interconnectivity.

Some FPGA platforms will not provide all of the resources required by all of the LEAP services. To maintain compatibility with such platforms, LEAP provides virtualized device implementations. These devices may rely on support from an external platform, either FPGA or processor, which can provide the service. In the case of LEAP Scratchpads, if no backing memory is available on the local FPGA, backing memory on a remote FPGA or on a remote processor may be used instead. LEAP virtual services resemble the memory-based virtualization layers typically found in software, such as `ramdisk`. LEAP's latency-insensitive approach directly enables platform virtualization: virtual devices are functionally equivalent to physical devices but their timing and performance may be radically different.

LEAP's FPGA platform abstraction directly enables its multiple FPGA partitioning capabilites. Because operating

system interfaces are consistent across platforms, LEAP can map user modules to any platform irrespective of the services and resources provided by that platform.

VIII. LEAP Compilation

In software, applications typically access system resources by making a request to the operating system for an interface object. For example, `fopen` requests access to a file by way of a `FILE` handle. From the programmer's perspective, this interface is very clean: a request results in a simple object, which the program then manipulates. In this transaction, the operating system functions as an intermediary between the program and the underlying file system resource, and the operating system's responsibility is to provide an efficient implementation of the accessor object on behalf of the user program.

Resource access in LEAP is functionally analogous to software resources. Programmers request abstract resource accessor objects, which are then supplied by LEAP. The key difference between LEAP and software operating systems is when accessor objects are created. Since instructions incur low overhead, resource management decisions in general purpose systems can be made dynamically and the operating system has the freedom to optimize its provided implementation at runtime. Although LEAP makes FPGA resource management decisions dynamically wherever prudent, dynamic management of resources in an FPGA often requires both additional control and storage overhead. These overheads frequently outweigh the benefit of dynamism: for point-to-point communication, a FIFO is much cheaper to implement than a network router. Avoiding dynamism at runtime means that resource allocation decisions must be made at compile time, resulting in a tighter coupling of compiler and operating system than is typically found in modern general-purpose operating systems.

Consider, as an example, the problem of allocating clocking resources in an FPGA. On the surface, clocking seems like a trivial problem in FPGA design: the designer instantiates a new clock primitive and ties it to the design RTL. However, even a resource as simple as a clock benefits from abstraction. Clock primitives are finite resources within the FPGA fabric. If a program instantiates too many clocks, it will not fit in the FPGA, *even if these clocks all have the same frequency*. A second issue in clocking is portability. Since FPGA clocks are parameterized in terms of ratios, including a fixed clocking primitive in the RTL immediately renders a design unportable.

To address these issues, LEAP provides the SoftClocks service. Programs ask for a clock at a particular frequency. Internally, the service maintains a list of previously requested frequencies. In response to a user request for a clock, the service searches its list for a clock of that frequency. If found, the service returns the appropriate clock object, which may be used directly in the program. Otherwise, the service instantiates a new clock, inserts it in the clock list, and returns the new clock. All clocks generated by the service are derived from physical clocks provided by the target platform. User designs are thus portable, depending only on clocks derived from the SoftClocks service.

SoftClocks illustrates a general resource management paradigm: a resource management service gathers information at compile time about how a resource is used within an FPGA program, and then creates an efficient implementation of the resources required by the program. To manage general classes of FPGAs resources at compile time, LEAP defines a programmer-extensible compiler interface, called *SoftServices*. LEAP uses SoftServices to manage many diverse functionalities, including clocking, the implementation of the latency-insensitive channels described in Section III, and the construction of efficient scan chains for run-time debugging.

Conceptually, SoftServices are objects that contain arbitrary, service-specific state. Services provide two interfaces: a private compiler interface and a user interface, by which programmers interact with the service object and request resources. Service objects register with the LEAP compiler, and the compiler invokes the private interface at certain points during compilation. In a typical service implementation, the user program invokes the public service interface to request resource access. The service builds a representation, often a list, of these requests. At the end of program compilation, LEAP invokes a service-provided handler, which allows the service to examine its data and produce an efficient, program-specific service implementation.

SoftServices provide the following interface to the LEAP compilation flow:

```
interface SOFT_SERVICE(type service_state);
  service_state initService();
  Void finalizeService(service_state state);
  Void handleModule(service_state state);
endinterface
```

Each service may maintain whatever state information it requires, as represented by the structure `service_state`. Services must provide three functions to the compiler. `initService` is called at the start of compilation and `finalizeService` is called at the end of compilation. This permits a service to view all requests for resource accessors for the entire program, thereby permitting the construction of globally efficient management hardware. `handleModule` is called at each latency-insensitive module, allowing services to make regionally-scoped implementation decisions.

SoftServices may make use of other SoftServices. For example, the implementation of named latency-insensitive channels interacts with the SoftClocks service by automatically inserting channels with clock domain crossings when necessary. To capture these inter-service dependencies, LEAP allows system programmers to specify the order in which the private service interfaces should be invoked by the compiler, in much the same way that `init.d` manages service startup in a general purpose operating system.

LEAP's SoftService infrastructure requires that the RTL compiler have strong support for static elaboration. Although legacy compilers are limited in this respect, recent hardware-oriented compilers [12] [13] provide sufficient infrastructure for LEAP.

IX. Evaluation

The primary advantage of LEAP, the abstraction of user programs from common services and physical platforms, is qualitative. Software systems separate applications from libraries and kernels even in the most constrained embedded systems, because the gains in programmer efficiency outweigh the resource costs. The question for FPGAs is whether the cost

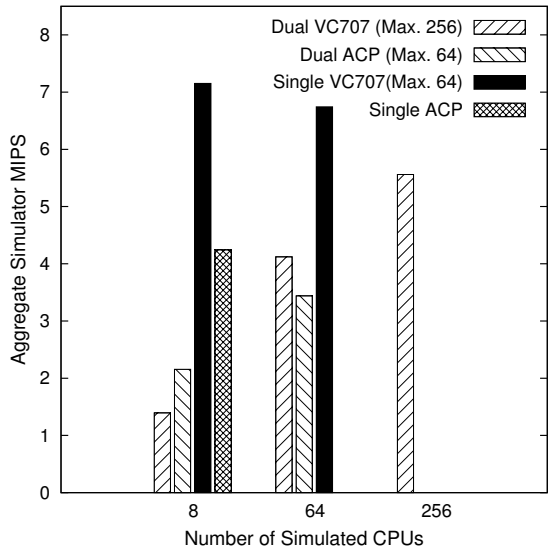


Fig. 5: HAsim Performance when mapped to different FPGA platforms. All cores in the design run a version of PARSEC's ray-trace program. Performance is reported in aggregate MIPS, the sum of throughput across all cores. When more resources are available, HAsim can scale to large numbers of cores.

of operating system abstractions is acceptable. We evaluate LEAP by measuring program performance and area overhead across a diverse set of programs and platforms.

A. FPGA Platforms

Our evaluation targets two different FPGA platforms with markedly different physical device characteristics, which we have chosen to illustrate the portability provided by LEAP. Although we will evaluate only Xilinx boards in this work, LEAP also supports a number of Altera platforms.

Nallatech ACP [14]: The Nallatech ACP module consists of a pair of Virtex-5 LX330Ts that plug directly into an Intel Front Side Bus (FSB) socket. The two FPGAs on the ACP are connected by way of a high-speed LVDS bi-directional interconnect. 8MB SRAM modules attached to each of the two FPGAs.

Xilinx VC707: The VC707 is a widely deployed evaluation board built with the Virtex-7 LX485T. The VC707 includes a 1GB DDR2 memory. The VC707 has both a PCIe interface for processor communication and numerous SERDES transceivers for inter-FPGA communication. We target both a single VC707 board and a dual VC707 configuration in which two boards communicate using multiple LEAP-managed SERDES links.

B. Benchmarks

To illustrate the broad applicability of LEAP, we examine several benchmarks of drastically different size.

Hello World: Like its well-known software equivalent, this program prints a message and then terminates. It illustrates the value of LEAP even in expressing simple programs.

Heat Transfer: A highly parallel stencil computation modeling heat transfer in a two-dimensional space. When implemented using LEAP's shared memory primitives, heat transfer can easily scale to consume all the area on a set of FPGAs.

HAsim: HAsim is a framework for constructing high speed, cycle-accurate simulators of multi-core processors. To model multiple cores, HAsim time-multiplexes components of a single processor, sharing these components among all modeled processors. HAsim is highly parametric, both in terms of the structure and the number of cores modeled. As a result of its time-multiplexed architecture and parameterization, HAsim models can scale to hundreds or thousands of cores with trivial source modifications, assuming that the operating system provides sufficient support to map large designs.

C. LEAP Overhead

Table I shows the areas of the benchmark programs when targeted to four different FPGA platforms. LEAP's overhead varies depending on the resources and services required by the user program. `Hello World` uses only two services and does not use memory, resulting in a LEAP overhead of only around 3% for a single FPGA implementation. The area overhead of LEAP in the case of `Hello World` underscores the value of LEAP: running even a baseline program requires significant support logic. Heat makes use of LEAP's shared memory. As a result, the overhead of LEAP for a small number of processing elements (PEs) is high. However, if the number of PEs is scaled, the overhead of shared memory is amortized. HAsim makes use of most LEAP services, specifies some application-specific services, and includes multiple memory interfaces to the LEAP memory hierarchy. As a result, LEAP overhead in HAsim is larger, with the cache hierarchy contributing around 12% of the area used by LEAP on each FPGA. High overhead to support memory is not unique to LEAP. Cache hierarchies and memory controllers are a large fraction of general purpose processors and many FPGA-based designs.

As a further proof of LEAP's design partitioning capabilities, we note that the area consumed by single and multiple FPGA implementation of the same design is identical, within a small margin for tool noise. LEAP and device driver area increase by a few percent due to the need to manage inter-FPGA communication. Since device usage on multiple FPGAs can be asymmetric, the area consumed by LEAP can vary between FPGAs in a multiple FPGA implementation. For example, the second FPGA in the dual VC707 configuration does not include a PCIe device.

D. Design Portability

One of the chief benefits provided by LEAP is strong design portability. In Figure 5 we show the performance of LEAP-based implementations of HAsim targeting four different FPGA platforms. Other than changing a handful of parameters to scale HAsim, the user code is unchanged across the four designs. LEAP transparently provides all of the platform-specific I/O, memory, and device management required to map HAsim to the four platforms.

On a single FPGA, HAsim scales to 8 (ACP) and 64 (VC707) cores before the FPGA runs out of resources. Using LEAP to map HAsim to two FPGAs, without changing any user code, we are able to build a partitioned model capable of supporting up to 256 cores. HAsim scaling is mainly governed by the availability of SRAM in the FPGA fabric. We achieve super linear scaling in problem size because many structures in HAsim are either time-multiplexed among all cores or

(a) ACP

	LUTS			Registers			BRAM			freq(MHz)
	User	LEAP	Devices	User	LEAP	Devices	User	LEAP	Devices	
Hello World, Single	0.2%	1.1%	1.8%	0.2%	3.0%	2.5%	0.0%	0.3%	3.7%	100
Hello World, FPGA 0	0.2%	2.9%	3.1%	0.2%	4.2%	4.4%	0.0%	0.9%	6.1%	100
Hello World, FPGA 1	0.0%	1.1%	1.8%	0.0%	1.5%	2.5%	0.0%	0.9%	3.7%	100
Heat, 2 PEs, Single	8.6%	13.5%	1.8%	4.5%	18.3%	2.5%	0.8%	0.7%	3.7%	75
Heat, 16 PEs, FPGA 0	32.4%	8.7%	3.1%	16.4%	25.1%	4.4%	2.9%	1.7%	6.1%	75
Heat, 16 PEs, FPGA 1	30.1%	13.0%	1.9%	15.9%	14.0%	2.5%	3.0%	0.1%	3.7%	75
HAsim, 8 cores, Single	72.0%	18.7%	1.8%	62.9%	19.3%	2.4%	78.0%	0.3%	3.7%	50
HAsim, 64 cores, FPGA 0	34.7%	20.5%	3.1%	25.7%	19.4%	4.4%	83.3%	2.2%	6.2%	60
HAsim, 64 cores, FPGA 1	68.9%	15.3%	1.8%	46.7%	13.8%	2.5%	88.2%	4.9%	3.7%	50

(b) VC707

	LUTS			Registers			BRAM			freq(MHz)
	User	LEAP	Devices	User	LEAP	Devices	User	LEAP	Devices	
Hello World, Single	0.2%	3.6%	1.8%	0.1%	3.3%	4.1%	0.0%	0.0%	6.8%	100
Hello World, FPGA 0	0.2%	7.4%	6.2%	0.1%	6.5%	5.1%	0.0%	3.1%	7.4%	100
Hello World, FPGA 1	0.0%	3.3%	6.2%	0.0%	3.0%	5.1%	0.0%	2.8%	7.4%	100
Heat, 2 PEs, Single	4.9%	15.1%	6.6%	1.4%	6.4%	2.6%	0.4%	0.6%	3.2%	75
Heat, 32 PEs, FPGA 0	47.8%	9.0%	7.8%	9.9%	7.9%	3.2%	1.9%	2.2%	9.7%	75
Heat, 32 PEs, FPGA 1	37.3%	7.9%	7.8%	9.7%	4.1%	3.2%	2.3%	1.6%	9.7%	75
HAsim, 64 cores, Single	73.8%	24.0%	6.6%	18.5%	9.4%	2.6%	28.0%	2.6%	3.2%	80
HAsim, 256 cores, FPGA 0	46.9%	21.5%	7.8%	8.7%	8.8%	3.2%	46.6%	3.4%	9.7%	60
HAsim, 256 cores, FPGA 1	62.0%	13.8%	7.8%	13.7%	4.1%	3.2%	72.4%	3.5%	9.7%	60

TABLE I: Synthesis metrics for single and multiple FPGA implementations. Results were obtained using the Xilinx tool chain at 14.5. Utilization results are listed as a fraction of the total FPGA resources for the target FPGA.

scale logarithmically with the number of cores. Dual-FPGA implementations of HAsim have lower performance in terms of aggregate MIPS than single FPGA implementations due to the overhead of communication between chips. While much of this latency is hidden by deeper pipelining, some critical loops are exposed.

X. Conclusion

FPGAs have great potential as platforms for many kinds of computation. However, the difficulty of programming FPGAs hinders their adoption in general systems. In this work, we presented the general philosophy and implementation of LEAP, an operating system for FPGAs. Like a software operating system, LEAP reduces the burden of programming FPGAs by providing uniform, abstract interfaces to underlying hardware resources, automatic management of these resources, and powerful system libraries that aid program design. LEAP achieves these goals through formalizing the principle of latency-insensitive design and providing strong compiler support for automating implementation decisions.

We believe that LEAP, or at least the design principles embodied in LEAP, is applicable to a wide range of other systems. We have found LEAP particularly useful in describing programs partitioned across multiple, heterogeneous platforms. The rise of programmable accelerators has made heterogeneous systems increasingly attractive. Because programming heterogeneous systems fundamentally requires strong communication support and because accelerator-based systems share many characteristics with FPGAs, we see LEAP as being valuable in programming and managing these new architectures.

LEAP is open-sourced under a BSD-style license and may be freely downloaded at `http://leap.csail.mit.edu`.

References

[1] L. P. Carloni, K. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of Latency-Insensitive Design," *IEEE Trans. on CAD*, vol. 20, no. 9, September 2001.

[2] 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.

[3] 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 Trans. on VLSI.*, vol. 16, no. 1, pp. 34–44, 2008.

[4] J. Agron, "Domain-Specific Language for HW/SW Co-design for FPGAs," in *Lecture Notes in Computer Science, Volume 56568*, 2009, pp. 262–284.

[5] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: An In-Fabric Memory Abstraction for FPGA-based Computing," in *FPGA*, 2011.

[6] M. Adler, K. Fleming, A. Parashar, M. Pellauer, and J. S. Emer, "LEAP Scratchpads: Automatic Memory and Cache Management For Reconfigurable Logic," in *FPGA*, 2011, pp. 25–28.

[7] http://www.khronos.org/opencl, "The Open Standard for Parallel Programming of Heterogeneous Systems."

[8] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. M. Hoki, and A. Agarwal, "Logic Emulation With Virtual Wires," *IEEE Trans. on CAD*, vol. 16, no. 6, pp. 609–626, 1997.

[9] M. Pellauer, M. Adler, D. Chiou, and J. Emer, "Soft Connections: Addressing the Hardware-Design Modularity Problem," in *DAC*. ACM, 2009, pp. 276–281.

[10] K. E. Fleming, M. Adler, M. Pellauer, A. Parashar, Arvind, and J. S. Emer, "Leveraging Latency-insensitivity to Ease Multiple FPGA Design," in *FPGA*, 2012, pp. 175–184.

[11] H. Yang, K. Fleming, M. Adler, and J. Emer, "LEAP Shared Memories: Automating the Construction of FPGA Coherent Memories," in *2014 Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2014, pp. 117–124.

[12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: Constructing hardware in a scala embedded language," in *DAC*, 2012.

[13] *Bluespec SystemVerilog Version 3.8 Reference Guide*, Bluespec, Inc., Waltham, MA, November 2004.

[14] http://www.nallatech.com, "Nallatech ACP module."