A Model of Computation and Compiler For Heterogeneous Architectures

Kermin Fleming¶ Michael Adler† Joel Emer†¶

†Intel Corporation
VSSAD Group
{michael.adler, joel.emer}
@intel.com

¶Massachusetts Institute of Technology
Computer Science and A.I. Laboratory
{kfleming, emer}
@csail.mit.edu

ABSTRACT

Heterogenous computation plays an important and expanding role in the modern world. However, developing programs that target heterogenous systems remains difficult, especially when those systems involve fine-grained parallel components like FPGAs. Design portability, design virtualization and dealing with complex and low-level platform interfaces are key challenges in programming these systems.

In this paper we describe a simple, abstract model of computation for heterogeneous systems built around the notion of *latency-insensitive programs*. This model of computation, which was inspired by our experience in developing high-performance applications for FPGAs, enables automated solutions to many of the problems facing programmers of heterogeneous systems. We present a compilation framework that automatically maps high-level latency-insensitive programs onto a variety of heterogeneous systems. We then describe several deployments of our compiler in industrial and academic projects.

1. INTRODUCTION

Heterogeneous computational platforms have become the norm rather than the exception. Heterogeneity at the low end of the computational spectrum has long been common: SoCs have included special purpose processors and hardware blocks for at least a decade. A shift towards heterogeneity is now occurring in intermediate and high-end computing, as the marginal utility of each additional homogeneous core diminishes. For example, many desktop processors include an integrated GPU which can be used for graphics or for computation. We believe that this trend in system design will continue into the foreseeable future as system architects continue to chase power-performance.

In a future comprised of largely heterogeneous computational systems, the complexity of programming these systems becomes a fundamental problem. In particular, orchestrating communication between the various computational platforms within the system is tedious. Traditionally, this communication has been managed by unportable, low-level device driver code against which programmers write similarly unportable programs. Moreover, these programming tasks are extremely error prone, and low-level programming errors of the kind likely to occur in such codes are extremely difficult to debug. Our own work [4] has focused primarily on programming heterogeneous systems of FPGAs and processors. Even as domain experts with many years of experience, it takes us about a month to fully debug the low-level drivers involved in such systems.

The difficulty in programming heterogenous systems has led to the development of several programming frameworks which attempt to facilitate writing heterogeneous programs. The best known of these frameworks is OpenCL [6], which facilitates the mapping of data parallel kernels across processors, DSPs, and GPUs. However, fine-grained parallel computation platforms, as embodied by FPGAs, CGRAs, and accelerators, which can exhibit highly non-deterministic and autonomous behavior, have not been well-addressed by OpenCL.

In this paper, we build upon the concept of latency-insensitive design [2] [3], which we have found enormously useful in programming FPGAs. In synchronous systems, latency-insensitive design seeks to decouple the notion of computation from the physical details of the system implementation by abstracting the timing of synchronous execution. We leverage this abstraction heavily to simplify heterogeneous programming.

First, we propose latency-insensitive programs as a model of computation for heterogeneous systems. In this model of computation, programs are viewed as sets of latency-insensitive modules, autonomous execution units, connected by latency-insensitive channels. Latency-insensitive channels are communications primitives which make some minimal guarantees about message delivery. Latency-insensitive modules work well for programming fine-grained parallel substrates like FPGAs because they separate the idea of communication from when and how that communication occurs. From a programming perspective, latency-insensitive channels insulate modules from the behavior of other modules, improving composability and simplifying the construction of large systems. From a design automation perspective, the programming contract of latency-insensitive modules enables the compiler to automatically solve several challenging design problems, including orchestrating heterogeneous communication.

Second, we propose an abstract machine model for heterogeneous systems, again inspired by latency-insensitive design. In the past, heterogenous computational environments have been difficult to understand and describe. Memory-mapped devices, interrupts, and other system minutiae conspire to complicate the lives of system programmers. To alleviate these issues, we present a simple declarative syntax, based around platform-level latency-insensitive communications, for describing the heterogeneous execution platforms available within a particular execution environment.

Finally, we unite the latency-insensitive model of computation and our abstract machine model in a latency-insensitive module (LIM) compiler, an extensible compilation framework which automatically maps programs consisting of latency-insensitive modules onto heterogenous execution environments. Because both the programming model and machine model are restricted to communication by latency-insensitive channels, the LIM compiler has broad freedom in mapping modules to the environment and implementing the network between modules, eliminating many of the design headaches in programming heterogeneous systems. We conclude by discussing some concrete applications of the LIM compilation framework.

2. PROGRAMMING MODEL

A good programming model for heterogeneous systems must both admit of many kinds of component platforms and simplify writing programs spanning those platforms. Latency-insensitive programs are composed of sets of latency-insensitive modules communicating by way of latency-insensitive channels. Latency-insensitive modules are computational entities, for example, a process or a hardware block, that communicate with each other only by way of primitive latency-insensitive channels. Internally, latency-insensitive modules may have any implementation that the programmer sees fit so long as that implementation honors the programming model. Externally, latency-insensitive modules present a highly-regular, channel-based interface, enabling automatic solutions to many of the complex design issues present in heterogeneous systems. These two properties make latency-insensitive modules an ideal model of computation for heterogeneous platforms.

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 observation methods, like `notFull` and `notEmpty`, to test the status of a channel. Unlike these library FIFOs, which have fixed implementations within a given library, the latency-insensitive channel denotes abstract communication and makes only two basic guarantees. First, the channel guarantees 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. When a programmer instantiates a latency-insensitive channel, he asserts that *these variations do not impact the functional correctness of his design*.

Latency-insensitive channels permit programmers to define latency-insensitive modules, computational entities that communicate only by way of latency-insensitive channels. A program composed of these modules executes on some heterogeneous environment, with the environment providing some physical communication primitives to the modules. Latency-insensitive module internals do not matter from the perspective of the LIM compiler; rather, it is the latency-insensitive interface that is fundamental. Internally, modules may be expressed in any programming language and have whatever behavior the programmer sees fit, provided that the internal implementation honors the programming contract of the external latency-insensitive interface. For example, in a fixed-function accelerator module, after data arrives through an interface latency-insensitive channel, it might pass through a latency-sensitive synchronous pipeline internal to the accelerator. For latency-insensitive modules described in sequential languages, the module internals could be implemented as a thread or process.

The latency-insensitive channel abstraction gives the compiler great freedom in choosing the physical implementation of the channels carrying data between latency-insensitive modules. For example, the compiler may choose to implement the channel using a fixed-buffer, fixed-latency RTL FIFO. Alternatively, the compiler is free to choose a more complex implementation, including implementations in which the endpoints of the latency-insensitive channel are physically far apart. This freedom of choice permits us to capture many physical systems within the programming model.

In addition to enabling design automation, the latency-insensitive modules also provide several other benefits, including portability and support for virtualization. Since inter-module communications are abstract, modules may be moved to different execution platforms of the same type by the compiler. In FPGAs, which can be area constrained, this frees programmers to scale their designs arbitrarily. The abstraction of communications also assists in the virtualization of modules. Because timing does not impact the functional correctness of latency-insensitive modules, virtual execution facilities can be added to a given heterogeneous environment if those facilities are not available natively in the environment. For example, if a given latency-insensitive program runs out of FPGA area, excess modules can be implemented in software simulation running on an general-purpose processor or a soft-core within the environment.

Latency-insensitive programs are quite general. Although we believe that the kernel-based model of computation provided by OpenCL is too restrictive for fine-grained and potentially autonomous platforms like FPGAs, we point out that latency-insensitive modules can capture the kernel-based model of computation provided by OpenCL. Kernel arguments themselves may be transported over an underlying latency-insensitive network to the latency-insensitive modules that execute the kernels.

3. ABSTRACT MACHINE MODEL

Once a program has been formulated in terms of latency-insensitive modules, it must be mapped onto a heterogeneous environment for execution. An environment is an aggregation of execution platforms, each of which can execute some kind of latency-insensitive module. These platforms are joined together by an interconnect capable of carrying latency-insensitive channel communication. Thus, our abstract machine model for executing latency-insensitive programs can be viewed as a graph of platforms connected by reliable communication channels.

The latency-insensitive abstract machine model requires execution platforms to satisfy a handful of simple requirements, primarily related to inter-platform connectivity. Each platform in the environment must be connected by a bi-directional transport link to at least one other platform, and the platform graph implied by these links must be strongly connected. Furthermore, inter-platform transport links must provide reliable, in-order delivery of messages. Examples of acceptable transport layers include PCIe and QPI for CPU-to-FPGA communication, high-speed SERDES for FPGA-to-FPGA communication, and UNIX sockets for CPU-to-CPU communication.

Figure 1 gives an example of the language used in our compilation framework to specify an execution environment. In this example declaration, there are two FPGA platforms connected by a single interconnect and a CPU connected to one of the FPGAs. The platform declaration conveys three important pieces of information: the platform type, the platform service description file, and the inter-platform interconnections declaration.

Much of the previous discussion of latency-insensitive modules has focused on the structure and properties of the input program. However, programs do not operate in a vacuum: they require access to external physical resources. Physical devices, especially those attached to FPGAs, do not necessarily conform to the latency-insensitive model of computation described in Section 2. For example, a latency-insenstive program implementing an SSD controller requires a physical, latency-sensitive interface to a set of flash chips.

Latency-sensitive physical device interfaces cannot be directly described in the user latency-insensitive programs, since latency-insensitive modules by definition are prohibited from having external latency-insensitive interfaces. We capture these non-ideal devices using the platform device description, denoted in the example as "`*.apm`". The platform device description contains all device libraries necessary to build an abstract interface to the physical devices provided by the platform. Within the device description, latency-senstive devices are wrapped in latency-insensitive interfaces. When viewed as an ensemble, the platform device description itself is a latency-insensitive module, although the non-latency

```
platform CPU CPU0 ``drivers_cpu0.apm'';
    FPGA0 -> drivers.fromFPGA0;
    FPGA0 <- drivers.toFPGA0;
endplatform

platform FPGA FPGA0 ``drivers_fpga0.apm'';
    FPGA1 -> drivers.fromFPGA1;
    FPGA1 <- drivers.toFPGA1;
    CPU0  -> drivers.fromCPU0;
    CPU0  <- drivers.toCPU0;
endplatform

platform FPGA FPGA1 ``drivers_fpga1.apm'';
    FPGA0 -> drivers.fromFPGA0;
    FPGA0 <- drivers.toFPGA0;
endplatform
```

Figure 1: A sample heterogeneous environment with two FPGA platforms and a CPU. This environment corresponds to the FPGA environment used in the SoC modeling case study.

insensitive portions of the device interface tie the device module to its particular platform.

The intention of the platform device description is to capture physical devices within the system, including any virtualized platforms like simulators. However, to synthesize a communications network, a compiler must know how platforms in the environment are physically connected to one another. We use the arrow notation to denote specific devices in the platform device description that can carry LI channel traffic to a specific platform. These devices have specific and well-known interfaces which may be used by the compiler.

Our compilation framework currently requires the precise specification of both the computational infrastructure and resources available within a target heterogeneous system. If a system has a processor, we expect the processor to be enumerated explicitly within the environment description file. Our specification requirement is entirely pragmatic – currently, we do very little program analysis to decide, given an environment or set of environments, what the best implementation of a given latency-insensitive program might be. However, one could imagine compilation tools playing a larger role in the determination of the environment, particularly if virtualization mechanisms are available for some of the physical platforms. For example, a compiler could instantiate softcores within an FPGA to run sequential codes.

Automatic SoC design takes the notion of platform virtualization to an extreme. If we consider a suite of latency-insensitive programs to be run by the SoC, a compiler could provision an SoC system with different kinds of cores and accelerators to minimize area while satisfying minimum performance constraints for each program.

4. LIM COMPILATION FLOW

In the previous sections, we described a programming model for heterogeneous platforms and an abstract machine model capable of describing generic heterogenous platforms. In this section, we will unify these two concepts in the LIM compiler, a framework for compiling heterogeneous programs. Conceptually, the LIM compiler takes sets of user source, described in terms of a set of supported programming languages, augmented with latency-insensitive channel annotations for each language. The LIM compiler then invokes an appropriate source compilation flow for each language, which produces an intermediate representation of the modules and channels present in the language source. The LIM compiler then analyzes this program representation, maps the modules onto a target heterogeneous execution environment, synthesizes a network between the modules, and produces a set of new programs, one for each platform. Finally, the LIM compiler performs a second compilation pass, producing an executable for each platform.

4.1 Building a Latency Insensitive Module Graph

LIM compilation begins with an initial compilation pass over the user source. During this compilation pass, each source compiler invoked by the LIM compiler is responsible for producing meta-data about the latency-insensitive modules present in that source. Generally, this requires some stylized syntax within the user source. For example, an RTL code might include new pragmas for declaring latency-insensitive modules and channels. This initial pass is similar to the object code generation phase of a normal compiler.

The LIM compiler gathers the meta-data produced by these initial compilation passes and uses it to produce a graph-based representation of the user program: the LIM graph. In this graph, latency-insensitive modules correspond to vertices and latency-insensitive channels correspond to edges. This graph will be processed by subsequent phases and eventually used to produce a physical implementation.

4.2 Mapping LIM Graph to Execution Environment

Once it has constructed a LIM graph representing a given program, the LIM compiler must map this graph to the target execution environment. Currently, we rely on the programmer to supply this mapping in the form of a module mapping file. Moving forward, we envision an automated flow in which the compiler analyzes design properties, including communications patterns and area, to produce an automatic mapping.

Depending on the underlying implementation of a particular module, the module may be mapped to a variety of platforms for execution using a virtualization layer. For example, software modules can be executed on a general purpose processor or perhaps a softcore on an FPGA. Hardware modules can be mapped to an FPGA or implemented in silicon. If no FPGA is available or for simulation purposes, hardware modules may also be executed on a general purpose processor inside of a hardware simulator. Currently, the LIM compiler supports mappings if they are user-declared.

4.3 Synthesizing Inter-Module Communication Network

Once vertices in the LIM graph have been assigned execution platforms, the compiler must synthesize a communications network connecting the latency-insensitive modules. This phase of the LIM compiler resembles the linking phase of a normal compiler.

Portions of the network may be simple: individual latency-insensitive channels between modules placed on the same platform can be implemented using the basic FIFO primitives provided by that platform. For channels connecting modules assigned to different platforms, the channels will be mapped to a synthesized shared network infrastructure generated by the LIM compiler. For example, all the channels crossing between a pair of connected FPGAs share the single, multiplexed physical interconnect between the FPGAs. This shared network infrastructure is built on top of the communications primitives declared in the heterogeneous environment file. Although conceptually simple, generating a high-quality, high-throughput inter-module network involves significant optimization effort on the part of the compiler. For example, the LIM com-

piler attempts to optimize network packet formats for performance. In FPGA-to-FPGA networks, the compiler packs data tightly to conserve communication bandwidth, while in CPU-to-FPGA networks, the compiler refrains from optimal packing to simplify the CPU-side packet decoding logic.

Finally, the LIM compiler generates an executable program for each platform in the execution environment. The synthesized programs consist of a source-level representation of the latency-insensitive modules mapped to the particular platform, the physical device drivers associated with that platform, and the synthesized communications network. These programs are passed through a final source compilation flow to produce executables for each platform in the heterogeneous system. The full system program may be run using a LIM-provided script which manages the loading of the various executables.

4.4 Compiler Extension

The LIM compiler currently supports three kinds of execution platforms (CPU, FPGA, and hardware simulator) and two programming languages (C++ and Bluespec). The LIM compiler can produce heterogeneous executables targeting any environment consisting of these kinds of platforms and described in these programming languages.

The LIM compiler has been designed as an extensible compilation framework, recognizing both that new kinds of execution environments may arise and that heterogeneous platforms may be programmed using diverse languages. Extending compilation support to new programming languages involves three activities. First, libraries for latency-insensitive primitives must be introduced into the language. In our experience, these libraries are not large, and the two examples we have produced are around 1000 lines of code each. Second, a compilation flow must be described for producing meta-data about the latency-insensitive modules and channels in a particular language. This flow is usually no more complicated than a typical enterprise compilation flow. Finally, a function for producing an interface between latency-insensitive channels and the platform interconnect must be provided. This function is invoked by the LIM compiler to produce the synthesized inter-module network. Adding support for a new kind of execution platform requires programming only language support for that platform and appropriate platform drivers.

5. USE CASES FOR THE LIM COMPILER

The LIM compiler is currently being used to produce implementations targeting several kinds of heterogeneous systems. In the following sections we outline some applications of the compiler.

5.1 High-performance FPGA-based Applications

As FPGAs have grown in size and capability, they have become computational platforms in their own right. Rather than solely prototyping ASICs, under this new use case, programs implemented on FPGAs are intended to compute the result of some algorithm. Unlike in ASIC prototyping, where the preservation of cycle-accuracy is paramount, algorithmic compuation requires only that the function of the algorithm is preserved. The LIM compiler facilitates the development of FPGA-based applications in two ways. First, although some FPGA applications are autonomous, most require at least some software interaction for activities like initialization. The LIM compiler's support for heterogeneous CPU-FPGA environments greatly simplifies these activities. The second way in which the LIM compiler supports FPGA-based applications is enabling large programs. Spatial architectures, like FPGAs, can run out of execution resources if the program is large. The LIM compiler frees programmers from the burden of manually partitioning large designs by enabling them to add more FPGAs to their execution environment and simply re-target their design.

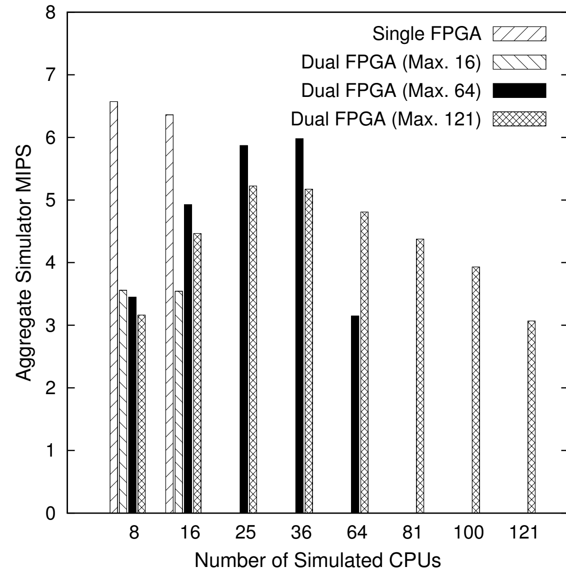

Figure 2: Performance results for one FPGA and two FPGA implementations of HAsim. Scaling HAsim to two FPGAs permits around a 10x increase in the number of cores that can be modeled.

One example of an FPGA application that benefits from both aspects of the LIM compiler is HAsim [7]. HAsim is a framework for constructing high-speed, cycle-accurate, FPGA-based chip-multiprocessor (CMP) simulators. By utilizing the FPGA, which has good support for the fine-grained parallelism common in processors, HAsim can simulate processors at speeds several orders of magnitude faster than detailed software models. HAsim is a heterogenous simulator: much of the heavy computational work is conducted on the FPGA, while rare events and instructions are modeled in software on an accompanying processor.

HAsim was designed in a highly-parametric fashion, enabling the description of enormous processor designs simply by changing a handful of parameters. However, large processor models do not fit on a single FPGA. In the case of HAsim, small CMP models, up to around 16 cores, can fit on a single FPGA. To study architectures with more cores, multiple FPGAs are required. When HAsim is mapped to multiple FPGAs, we are able to simulate CMPs up to 10 times larger than the largest CMPs that fit on a single FPGA, as shown in Figure 2.

5.2 Rack-level Local Area Network

In addition to offering high performance for fine-grained parallel applications, FPGAs are also very good at chip-to-chip and board-to-board communication. One of the more intriguing physical features of modern FPGAs is the high-speed SERDES transceivers. Modern FPGAs come with, at the high-end, dozens of these transceivers, and the bandwidth, latency, and number of transceivers appear to be scaling well with process generations. Due to the high performance of these transceivers and the flexible support for networking protocols within the logic fabric, FPGAs can be used to build very high performance local area networks at a low cost relative to traditional networking equipment.

BlueDBM [1], a distributed, rack-level storage system, highlights an emerging class of FPGA-based rack-level architectures for which the LIM compiler is ideally suited. Each CPU in the BlueDBM system is attached to an FPGA by way of PCI-E, and

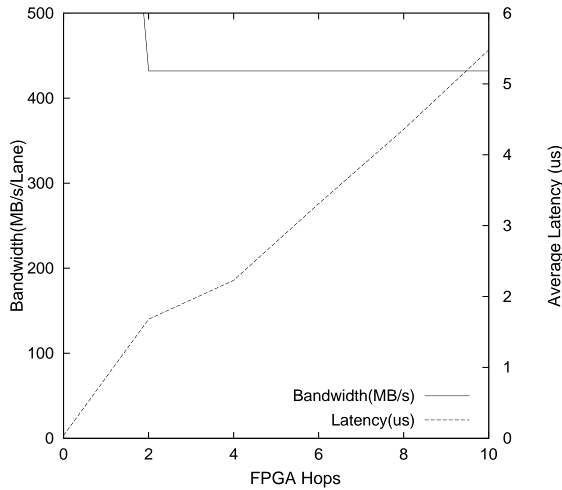

Figure 3: Network and throughput characteristics of LIM-generated networks. Networks target the Xilinx ML605 evaluation platform SERDES at 5Gbps.

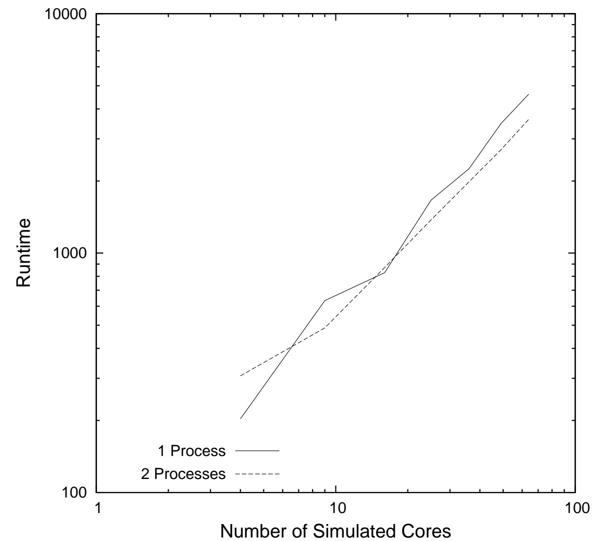

Figure 4: Parallel simulation improves the simulation speed of real programs, in this case, HAsim. The overhead of communication dominates for small models, but as compute-to-communication ratio grows, parallel simulation becomes advantageous.

each FPGA has attached an array of flash memory. The FPGAs are networked together using SERDES transceivers managed by the LIM compiler. BlueDBM attempts to leverage the relative latencies of flash and the FPGA to achieve the appearance of a large, shared store with low cost per node. The RTL and software of BlueDBM use the LIM compiler to generate the middle-ware layer between FPGAs and CPUs within the rack. Figure 3 summarizes the performance of the LIM-generated network.

BlueDBM is but one application of FPGAs to rack-level computer architecture. We view FPGA-supported rack-level computation and communication as a promising new field.

5.3 SoC Software Development Vehicles

In the past, CPUs could be validated using legacy software both because architectures were relatively stable and because the validation software was usually not complicated. However, accelerator-oriented SoCs change architecturally at each generation, resulting in the need to write and validate complicated new software before the design can be considered complete. Thus, it is necessary to provide software developers with high-quality software-development vehicles (SDVs) as early in the SoC design cycle as possible.

Unfortunately, hardware accelerators are difficult to model and slow to simulate in software. Fast emulation of most SoC blocks requires mapping the block to an FPGA. However, existing FPGA-based system emulation tools, which target cycle-accurate verification, tend to be too slow for software development. The slow execution speed of emulation solutions arises from the communication and coordination necessary to maintain cycle-accuracy across a distributed network of FPGAs.

Although most of the blocks in a given SoC are latency-sensitive, we remark that most inter-block communications in an SoC are carried over latency-insensitive networks-on-chip (NoC). The LIM compiler can exploit this property of the NoC to automate the production of FPGA-based full-system SDVs by mapping the various SoC blocks onto as many FPGAs as might be necessary to implement the system.

In viewing an SoC as a latency-insensitive program, we are trading system-level cycle accuracy for system-level functional accuracy and faster simulation speeds. Fortunately, software developers have historically been satisfied with functional accuracy as embodied by popular simulators like Simics. Should system-level cycle accuracy be required in late stages of software development and validation, we note that common cycle-accurate simulation paradigms like SCE-MI [5], which maintains cycle-accuracy algorithmically, can be layered on top of the LIM compilation infrastructure.

5.4 Parallel Logic Simulation

Software development notwithstanding, simulation times for large hardware designs can be quite long. A typical flow for such a simulation is to load an entire hardware design into some single-process software simulator. For large designs, this results in very slow simulation speeds which reduce programmer productivity during interactive debugging sessions.

Single process logic simulation is unsatisfying for two reasons: modern computer architectures offer large numbers of cores and the underlying simulation target is fundamentally parallel. The LIM compiler offers the capability to exploit this intrinsic parallelism by partitioning a latency-insensitive module graph among multiple simulators.

Figure 4 shows the results of a limit study of parallel simulation. In this test, a HAsim simulator with various numbers of cores is split in two and mapped to simulator platforms by the LIM compiler. If the amount of simulation work is small, that is, if few cores are simulated, then single process simulation is faster than parallel simulation due to communication overhead. As the number of cores is scaled, computation per cycle becomes the simulation bottleneck. In this case, parallel simulation with two processes is as much as 35% faster than a single-process simulation.

As in the SoC emulation case, our parallel simulator maintains only the functional correctness of latency-insensitive designs. However, by leveraging latency-insensitive modules for parallelism we have lost complete system-level cycle accuracy, meaning that the parallel simulator may not expose some erroneous behaviors that would arise in a cycle-accurate realization of the target system. However, because most of the simulation system retains cycle accuracy and many bugs are function-related rather timing-related, we believe that parallel simulation provides significant value to developers, especially in early, highly-interactive stages of the debugging and validation process.

Support for parallel logic simulation is also useful in constructing FPGA-accelerated SDVs. An important design consideration in deploying SDVs is cost per programmer. FPGAs tend to be expensive to deploy and, in a typical industrial setting, there are too many software developers to make personal FPGA networks or big-box simulators practical. Here, too, the LIM compiler, through its support of virtualization, offers a solution: the LIM compiler can produce programs with some modules executing on an FPGA and some modules executing in simulation. Since an individual programmer often targets only a small portion of the SoC-under-test, we can apply Amdahl's law and allocated this targeted portion to a smaller number of FPGAs. Other parts of the SoC can be simulated by bridging the FPGA and the simulator, as in the parallel logic simulation case. In this manner, it is possible for a single programmer to reap the performance benefits of FPGA accelerated simulation without the high cost typically associated with FPGA-based emulation.

6. CONCLUSION

Heterogeneity has become ubiquitous in modern computational systems. In this paper, we presented a general model of computation, latency-insensitive modules, which we believe is suitable for writing programs targeting such systems, and we presented a simple, declarative language for describing these systems. We unite these two notions in the LIM compiler, a compilation framework capable of mapping abstract representations of latency-insensitive programs and mapping them onto various heterogeneous systems.

Most of the discussion presented in this paper has focused on heterogeneous systems involving fine-grained parallel systems. However, nothing in the latency-insensitive model of computation or in the LIM compiler precludes the inclusion of other kinds of computational platforms, including GPUs and DSPs. As the architectures of these platforms trend toward fine-grained interaction with other system platforms, we believe that latency-insensitive modules will be an increasingly attractive model of computation for these systems as well.

7. REFERENCES

[1] Removed, under sumbmission.

[2] Luca P. Carloni, Kenneth McMillan, and Alberto L. Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE TRANSACTIONS on Computer-Aided Design of Integrated Circuits and Systems*, 20(9), September 2001.

[3] Josep Carmona, Jordi Cortadella, Mike Kishinevsky, and Alexander Taubin. Elastic Circuits. *IEEE Transactions on Computer-Aided Design*, 28(10):1437–1455, October 2009.

[4] Kermin Elliott Fleming, Michael Adler, Michael Pellauer, Angshuman Parashar, Arvind, and Joel S. Emer. Leveraging Latency-insensitivity to Ease Multiple FPGA Design. In *FPGA*, pages 175–184, 2012.

[5] http://www.eda.org/itc/scemi.pdf. Standard co-emulation modelling interface (sce-mi): Reference manual.

[6] http://www.khronos.org/opencl. The open standard for parallel programming of heterogeneous systems.

[7] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer. HAsim: FPGA-Based High-Detail Multicore Simulation Using Time-Division Multiplexing. In *The 17th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2011.