Chapter 1

FPGA Overlays

Developing applications that run on FPGAs is without doubt a very different experience from writing programs in software. Not only is the hardware design process fundamentally different from that of software development, software programmers also often find themselves constantly battling with the much lower *design productivity* in developing hardware designs.

To begin, first time FPGA designers are often put off by the steep learning curve of the complex labyrinth of tools involved. Beyond that, instead of spending merely seconds to compile and debug a quick-and-dirty proof-of-concept design, software programmers soon also discover that implementing even the simplest FPGA design can consume at least tens of minutes of their development time. As the size of their designs increase, the run time of these implementation tools quickly increase to hours or even days, greatly limiting the number of possible debug-edit-implement cycles per day. Indeed, to debug, they may turn to the use of a cycle-accurate simulator. Unfortunately, tracing the behaviors of even just a handful of signals through tens of thousands of cycles soon becomes a slow and intractable process.

In this chapter, we explore how the concept of FPGA overlay may be able to alleviate some of these burdens. We will look at how by using an overlay architecture, designers are able to compile applications to FPGA hardware in merely seconds instead of hours. We will also look at how overlays are able to help with design portability, as well as to improve debugging capabilities of low-level designs. Finally, we will explore the challenges and opportunities for future research in this area.

1.1 Overview

The concept of overlaying a virtual architecture over a physical system is not entirely new – in networking, for example, virtual overlay networks are routinely constructed on top of the physical infrastructure in modern systems. On the other hand, the concept of overlaying a virtual architecture over a physical FPGA is only recently gaining traction among researchers, but is already generating a lot of excitements because of their potentials.

So what is an FPGA overlay exactly? Interestingly, because of the unique nature of

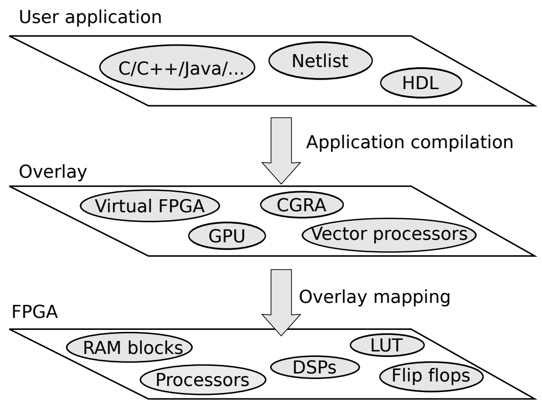


Figure 1.1: Using an overlay to form a 2-layer approach to FPGA application development. Overlay may be designed as a virtual FPGA, or it may implement an entirely different compute architecture such as a coarse-grained reconfigurable array (CGRA), vector processor, multi-core process, or even an GPU.

FPGAs as a flexible configurable hardware, drawing up a precise definition for an FPGA overlay may not be as straightforward as it may seem. In this chapter, we will use the following definition as a starting point.

> An FPGA overlay is a virtual reconfigurable architecture that overlays on top of the physical FPGA configurable fabric.

From this definition, we can see that an FPGA overlay is a machine architecture that is able to carry out certain computation. Furthermore, this architecture is *virtual* because the overlay may not necessarily be implemented physically in the final design. Finally, it is *reconfigurable* because the overlay must be able to support customization or be reprogrammed to support more than one applications.

In other words, an FPGA overlay is a virtual layer of architecture that conceptually locates between the user application and the underlying physical FPGA similar to that shown in Figure 1.1. With this additional layer, user applications will no longer be implemented onto the physical FPGA directly. Instead, the application will be targeted toward the overlay architecture regardless of what the physical FPGA may be. A separate step will subsequently translate this overlay architecture, together with the application that runs on it, on to the physical FPGA.

The easiest way to understand how an FPGA overlay operates in practice, is to consider building a *virtual* FPGA (VA) using the configurable fabric of a physical FPGA (PA). By doing so, you now have an FPGA overlay architecture in the form of VA overlaying on top of PA. We say that VA is essentially "virtual" because architectural features of VA such as its muxes or I/O blocks may not necessarily be present in PA. Yet, if the overlay is constructed correctly, any design that was originally targeting VA may now execute unmodified on PA without knowing its details.

However, the power of employing FPGA overlay is not limited to making virtual FPGAs only. On the contrary, taking advantage of FPGA's general-purpose configurable fabric, many researchers have demonstrated the benefits of overlays that implement entirely different computing architectures such as multi-core processor system, coarse-grained reconfigurable arrays (CGRAs), or even general-purpose graphic processing units (GPUs).

Now, using a multi-core processor overlay as an example, it should be apparent how overlays are able to improve a software programmer's design productivity. Instead of working with unfamiliar hardware-centric tools and design methodologies, software programmers are now possible to utilize FPGAs as accelerators simply by writing programs that target a familiar architecture. In general, one benefit of using FPGA overlay is that it is able to bridge between the often software-inclined user and the low-level FPGA hardware fabric.

Of course, if an application can be readily accelerated on a multi-core processor overlay implemented using FPGAs, then it is understandably begging the question: why not simply run the design on an actual high-performance multi-core processor instead? The answer to this question, indeed, is the key challenge that will guide the future research on overlay designs — While an overlay offers many desirable features to software programmable such as improved design productivity, the additional layer on top of the physical FPGA inevitably introduces additional performance penalty to the system. A good overlay design must therefore ensure that despite the performance penalty introduced, the overall acceleration offered by FPGA must remain competitive for the accelerator system to be worthwhile. Furthermore, it must provided added value beyond a solution with a fixed general-purpose architecture. One such added value is the ability to customize the overlay for the particular application or group of applications concerned for sake of performance or power-efficiency.

1.1.1 Coarse-Grained Architectures

In most cases, FPGA overlays are essentially coarse-grained architectures that are built on top of the physical fine-grained configurable fabric. As their names suggest, the basic idea of a coarse-grained architecture is to reduce the configuration granularity of an FPGA from its physical fine-grained configurable fabric such as LUTs to one with coarser reconfiguration granularity. In some cases, such coarse-grained blocks may refer simply to arithmetic blocks of moderate sizes such as adders, multipliers, or digital signal processing blocks of some sort. In other cases, such coarse-grained blocks may refer to very complex microprocessors connected with sophisticated network-on-chip. Regardless of the implementation, the central goal of any coarse-grained architecture remains the same: to improve power-performance of the system by trading off design flexibility.

In addition, because of their reduced configuration flexibility, coarse-grained architectures can also enable design methodologies that are more productive than traditional hardware design flow. In particular, coarse-grained architectures improve a designer's productivity in two important ways. First, by constraining the flexibility of an FPGA, a coarse-grained architecture reduces the design space significantly, which has a net effect of reducing implementation tool flow run time considerably [LPL$^+$11]. In addition, many coarse-grained architectures implement compute models that are more familiar to software designers, considerably lowering the barrier-to-entry to employ such designs. For instance, many recent overlays are implemented as coarse-grained reconfigurable arrays (CGRAs), where computation is carried out by a connected array of processing elements (PEs). Instead of implementing the user application using low-level configurable logic of the FPGA, these operations are translated into computational tasks that take place in the PEs. To many software programmers, programming a parallel processor array, while a daunting task in its own right, is arguably a lot more approachable than to implement designs on the native FPGA configurable fabric. As such, it is not surprising that many recent overlay designs are built on top of an underlying coarse-grained reconfigurable array [KHKT06, FVM$^+$11, SBB06, LS12, CA13, JFM15].

1.2 Benefits of Overlays

As a virtualization layer that sits between a user application and the physical configurable fabric, an FPGA overlay inherits many of the benefits that software programmers have learned to expect from their CPU virtualization experience — portability, compatibility, manageability, isolation, etc. On top of that, employing FPGA overlays has also been demonstrated as a good way to improve a designer's productivity through improved compilation speed and better debugging support. Along the same line, by carefully partitioning the complex hardware-software design flow around an intermediate overlay layer, it is also possible to provide separation of concerns between software and hardware engineers in the design team. The overlay essentially acts as a bridge between the two teams, while allowing the overall system to take advantage of the FPGA resources efficiently.

1.2.1 Virtualization

Virtualization of FPGA resources has long been an active area of research since the early days of reconfigurable computing. These pioneering works have demonstrated many of the possibilities as well as challenges associated with virtualizing hardware resources that are not designed to be time-multiplexed. A common trend among these early works was that virtualization can be used as a mean to provide the designers and/or tools the illusion of having infinite hardware resources. Early works by Trimberger et al [TCJW97], virtual wire[BTA93] and SCORE [DMC$^+$06], for instances, gave the users the illusion of a system with unlimited FPGA resources through carefully structured hardware/CAD system. Others have studied the problem of time-sharing of FPGA resources from an operating system's perspective as a way to provide shared accelerator resources among users/processes. [LP09, SB08, FC05].

As the concept of FPGA overlay continues to mature, the idea of virtualizing FPGAs has taken on a new focus. As an overlay, virtualizing FPGAs allows an additional benefit of providing a compatibility and portability layer for FPGA designs. In the work of Zuma [BL12], for instance, virtual, embedded FPGAs were proposed. By providing a virtual FPGA layer, the authors demonstrated that it is possible to execute the same netlist on multiple FPGAs from competing vendors using multiple different design tools.

1.2.2 Reduced Compilation Time

A key difference in design experience between software compilation and implementing FPGA designs rests on their drastically different run time of the involved tools. With modern compiler technologies, software compilation has already become a straightforward, predictable, and most importantly, very rapid process. Compiling even a relatively complex piece of software application rarely take longer than a few minutes on a reasonably fast computer. On the other hand, implementing applications for FPGAs involves a complex labyrinth of low-level tools that are convoluted, unpredictable, and takes a long time to complete. Compiling even the smallest design may take tens of minutes, while spending hours or even days on some of the largest designs are not unheard of. Unfortunately, this 2 orders of magnitude difference in run time, together with the unpredictable nature[1] and the often-mystical error reporting mechanisms[2], are all contributing to a very high barrier-to-entry that shies away most first time software programmers. Technically, this much longer run time of the tools also significantly reduces the number of possible debug-edit-implement cycles per day, causing project delays as well as lowered productivity of the designers.

By using an overlay as intermediate compilation target, together with careful crafting of the design process, researchers have demonstrated how such lengthy hardware development process can be reduced significantly. For example, in the work of Intermediate Fabric[CS10], an intermediate coarse-grained reconfigurable fabric was introduced as an overlay. With the overlay architecture, the average place and route times for the tested benchmark were significantly reduced by more than 500 times.

Similarly in the work of QuickDough, a coarse-grained reconfigurable array was used as overlay to accelerate compute intensive loop kernels. Loops are scheduled to execute on the CGRA instead of compiling to the reconfigurable fabric. As a result, when compared to manually generating custom hardware for the loops using standard hardware tools, up to 3 orders of magnitude reduction in compilation time was demonstrated.

1.2.3 Improved Debugging Capabilities

Unlike many software development frameworks where a range of debugging facilities and methodologies are readily available, tools for debugging applications that target FPGA-based systems are still in their infancy. Traditional FPGA design methodologies rely heavily on cycle-accurate simulations for application development and debugging. While such simulations are invaluable to understand the low-level operation of the FPGA, they are slow, tedious and provide only limited information about the run-time behavior of the design. To monitor run-time behavior of a design, users must rely on even more complex in-system emulation facilities or even external testing hardware. Taking advantage of FPGA overlays, researchers have demonstrated some promising results addressing the need for better debugging tools.

For instance, in a series of work by Hung and Wilton [HW14, HW13], an overlay network was incorporated into FPGA design to facilitate insertion of trace buffers after a design has been placed and routed. By carefully controlling the signal routes to utilize only unoccupied resources in the FPGA, they have demonstrated efficient ways to dynamically select and

[1]Placing and routing a design on nearly full FPGA, for example, may or may not succeed depending on the random algorithms involved.

[2]To see this, try to explain to a first-time software programmer why a functionally correct design in simulation may end up with an error message about timing violation on a net with an unknown name in the report. Then, also try to explain to the programmer how to resolve that timing error.

monitor signals during run time.

In other cases, since the overlay layer enables a virtual computing paradigm for the application developer that is different from the underlying FPGA, they also enable new debugging strategies that are more suitable to the designer. For instance in MARC, debugging the user-specified OpenCL applications that run on the generated multi-core architecture can follow traditional software debugging methodologies instead of relying on low level FPGA tools. Not only can it greatly increase the abstraction level, but it can also allow a debugging strategy that matches the user's expectation.

1.2.4 Separation of Hardware and Software Concerns

With careful planning, it is also possible to take advantage of the 2-layer design approach offered by FPGA overlays as a natural division point between hardware and software development efforts. Recall from Figure 1.1 that implementing designs via an FPGA overlay involves 2 steps: user applications must first be mapped to the overlay architecture, which is subsequently implemented to the physical fabric in a second step. In many cases, the overlay architectures are designed so they can efficiently support the computational model expected by the model. As a result, mapping of software applications to the overlay is usually more intuitive to software programmers than to map the same application to the physical FPGA fabric in one step. On the other hand, mapping the overlay and the user application to the physical fabric involves intimate knowledge about the FPGA hardware implementation process. This task is best left to a separate hardware team. Consequently, one benefit of having an overlay is that the hardware team may now devote their efforts exclusively on implementing the relatively well-structured overlay on the fabric, rather than to implement many individual applications. The hardware team can be more focused, and can perceivably create a better, highly optimized hardware design.

For example in the work of MARC[LPL$^+$11], a multi-core processor-like architecture was used as an intermediate compilation target. In this project, user applications are expressed as OpenCL programs. To implement these applications, they are first compiled as an application-specific multi-core processor, which is subsequently implemented on the physical FPGA in a separate process. With this set up, users no longer need to understand the detailed implementation of their algorithm on FPGAs. Instead, they write essentially standard OpenCL programs, and focuses exclusively on writing the best code for the accelerated architecture assumed. The task to map their OpenCL code into custom core on the target overlay, and to implement that overlay on the FPGA fabric, was left to a separate team.

In their case studies, the resulting application achieves about one third of the speedup when compared to fully custom designs. Yet with the 2-layer approach using OpenCL, the design effort with MARC is significantly lower when compared to making a custom design of the same application.

1.3 Types of Overlays

Over the years, quite a few works on FPGA overlays have been developed. In this section, we will take a quick walk through some of them and also look at various types of FPGA overlays that have been explored in the past decade.

1.3.1 Virtual FPGAs

To begin, one of the most easiest to understand categories of overlay are virtual FPGAs[LMVV05, BL12, GWL11, CS10, KBL13]. They are built either virtually or physically on top of off-the-shelf FPGA devices. These overlays have different configuration granularity but typically feature coarser configuration granularity than a typical FPGA device. Similar to virtual machines running on a typical computer, such virtual FPGA provides an additional layer that improves application portability and compatibility. Furthermore, because of the coarser-grained configurable fabric, implementing designs on such overlay is relatively easier than on a fine-grained device. However, the additional layer imposes restrictions on the underlying fabrics' capability and usually results in moderate hardware overhead and timing degradation.

In one of the earlier works, Lysecky et al. developed a relatively fine-grained virtual FPGA as firm cores expressed as structural VHDL [LMVV05]. The virtual layer provides effective portability yet incurs relatively high performance and hardware overhead. In [GWL11], Grant et al. proposed a time-multiplexed virtual FPGA CAD framework MALIBU. The virtual FPGA used in MALIBU has both fine-grain and coarse-grain processing elements integrated into each logic cluster and can be used to reduce the compilation time significantly with moderate timing penalty. Around the same time, Coole and Stitt also proposed another island-style coarse-grained overlay called Intermediate Fabric [CS10]. It uses coarse-grained operators such as adders instead of logic clusters and routes data through 8 to 32 bit buses achieving both portability and fast compilation. Finally, Koch et al. developed a fine-grained FPGA overlay in [KBL13] to implement customized instructions on FPGAs from different vendors providing a portable application consisting of a program binary and an overlay configuration in a completely heterogeneous environment.

1.3.2 Coarse-Grained Reconfigurable Arrays

Another category of overlay architecture commonly employed is in the form of coarse-grained reconfigurable arrays (CGRAs) [KHKT06, FVM$^+$11, SBB06, LS12, CA13, JFM15]. The use of CGRAs provides an efficient tradeoff between flexibility of software and performance of hardware especially for compute intensive applications as demonstrated by numerous earlier works [TB01, CH02].

In one of the earlier works in the area, Kissler et al. developed WPPA (weakly programmable processor array), a VLIW architecture based parameterizable CGRA overlay [KHKT06]. It featured an interconnection wrapper unit for each processing element (PE) that could be used for dynamic CGRAs topology customization. Around the same time in [SBB06], a customized CGRA overlay called QUKU was developed for DSP algorithms. It had a two-level configuration capability, while the high-speed configuration was used for operator reuse within an application and low-speed reconfiguration was used for optimization between different applications. In [FVM$^+$11], Ferreira et al. proposed a heterogeneous CGRA overlay with a global multi-stage interconnection on FPGA. Compiling applications onto the overlay took only milliseconds for smaller DFGs. In [LS12], Lin and

So also proposed a soft CGRA overlay for rapid compilation. In addition, they demonstrated that by customizing the overlay connection between PEs on a per-application basis, improvement in energy-efficiency could be obtained in the expense of longer tool run time. The authors in [CA13] built a generic high speed mesh CGRA overlay using the elastic pipeline technique to achieve the maximum throughput. It adopted a data-driven execution flow and was suitable for smaller pipelined DFG execution. Recently in [JFM15], Jain et al. also proposed an overlay that is constructed around the primitive FPGA DSP blocks to achieve high-frequency implementation and high throughput result. Also, in [CS15], Coole and Stitt proposed to provide the overlay with limited flexibility instead of full configurability specifically to a group of design. With this customization, the area overhead was reduced significantly.

1.3.3 Processor-Like Overlays

A third category of overlay moves away from the traditional FPGA architectures and instead explores using processor-like designs as an intermediate layer. The main concern for works in this category are usually compatibility and usability of the overlay from a user's perspective. To provide the necessary performance, these overlay architectures usually feature a high degree of control and provide ample of data parallelism to make them suitable for FPGA accelerations. As an early attempt, Yiannacouras et al. explored the use of a fine-grained scalable vector processor for code acceleration in embedded systems [YSR09]. Later in [SL12], Severance and Lemieux proposed a soft vector processor named VENICE to allow easy FPGA application development. It accepts simple C program as input and execute the code on the highly optimized vector processor on the FPGA for performance. In the work of MARC, Lebedev et al. explored the use of a many-core processor template as an intermediate compilation target [LCD$^+$10]. In that work, they have demonstrated improved usability with the model while also highlighting the need for customizing computational cores for sake of performance. To explore the integration between processor and FPGA accelerators, a portable machine model with multiple computing architectures called MURAC was explored in [HIS14]. Finally, a GPU-like overlay was proposed in [KS11] that demonstrate good performance while maintaining a compatible programming model for the users.

1.4 Case Study – QuickDough

As a case study to illustrate how an FPGA overlay works in practice, the design and implementation of the research project QuickDough will be examined in this section.

In short, QuickDough is a nested loop accelerator generation framework that is able to produce hardware accelerators rapidly [LS12, LS15]. Given a user-designated loop for acceleration, QuickDough automatically generates and optimizes the corresponding hardware accelerator and its associated data I/O facilities with the host software (Figure 1.2).

The overall design goal of QuickDough is to enhance designer's productivity by greatly reducing the hardware generation time and by providing automatic optimization of the data I/O between the host software and the accelerator. Instead of spending hours on conventional hardware implementation tools, QuickDough is capable of producing the targeted

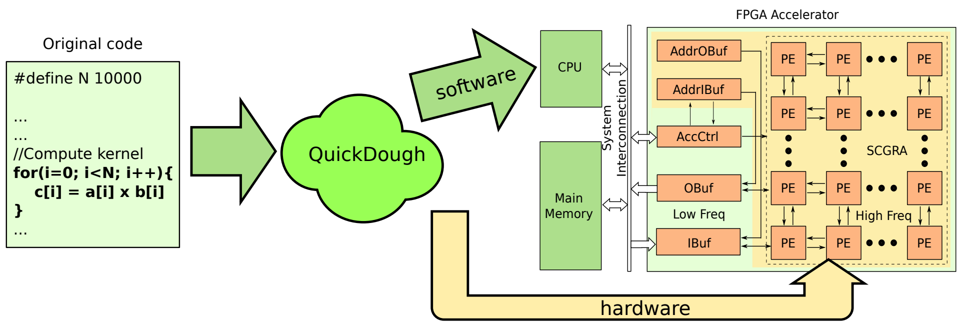


Figure 1.2: QuickDough takes a user-designated loop as input and generate the corresponding hardware accelerator system using a soft coarse-grained reconfigurable array overlay.

hardware-software system in the order of seconds. By doing so, it provides a rapid development experience that is compatible with that expected by most software programmers.

To achieve this compilation speed, while maintaining a reasonable accelerator performance, QuickDough avoids the creation of custom hardware directly for each application. Instead, the compute kernel loop bodies are scheduled to execute on a CGRA overlay, which is selected from a library of pre-implemented hardware library. By sidestepping the time-consuming low-level hardware implementation tool flow, the time to implementing an accelerator in QuickDough is reduced to essentially just the time spent on overlay selection and scheduling compute operations on the resulting overlay. In addition, taking advantage of the overlay's softness and regularity, QuickDough allows users to perform tradeoff between compilation time and performance by selecting and customizing the overlay on a per application basis. The result is a unified design framework that seamlessly produces the entire hardware-software infrastructure with a design experience similar to developing conventional software.

Through these facilities and through carefully partitioning the design process, QuickDough strives to improve design productivity of software programmers utilizing FPGA accelerators in 3 aspects:

1. It automates most of the hardware accelerator generation process, requiring only minimum input from the application designer;
2. It produces functional hardware designs at software compilation speed (order of seconds), greatly increasing the number of debug-edit-implement cycles per day achievable;
3. It allows software programmers to progressively improve performance of the generated accelerator through subsequent optimization phases, essentially separating the functional verification and optimization process of application development.

In the following subsections, an overview of QuickDough is presented to illustrate how it achieves the above goals. For details, please refer to [LS12, LS15].

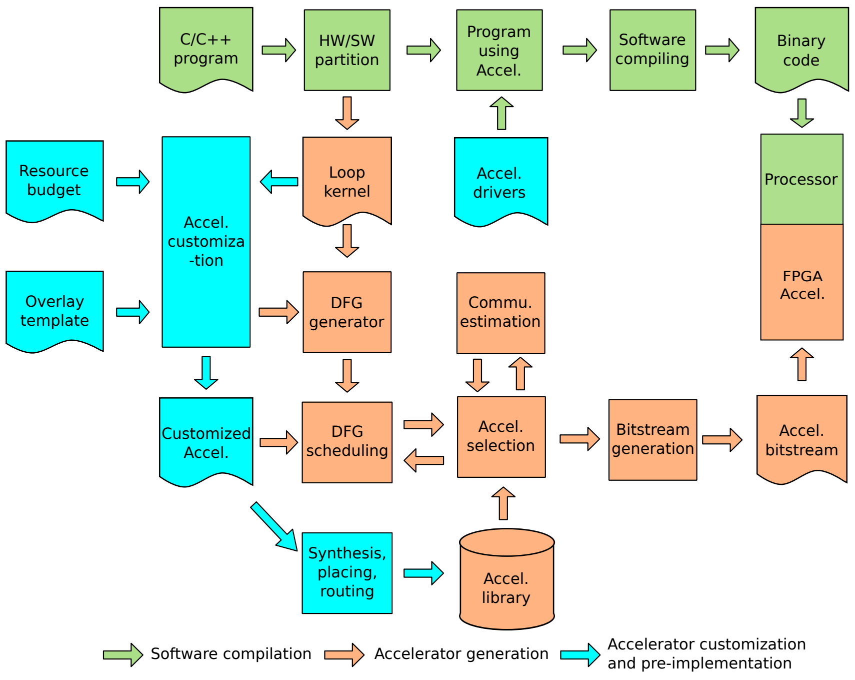


Figure 1.3: QuickDough: FPGA loop accelerator design framework using SCGRA overlay. The compute intensive loop kernel of an application is compiled to the SCGRA overlay based FPGA accelerator while the rest is compiled to the host processor.

1.4.1 Generation Framework

Figure 1.3 shows an overview of the QuickDough compilation flow. The key to rapid accelerator generation in QuickDough is to partition the complex hardware-software compilation flow into a fast and a slow path.

The fast path of QuickDough consists of the steps necessary to generate a *functional* loop accelerator, which are shown in orange for hardware and in green for software generation in Figure 1.3. To begin generating a loop accelerator, QuickDough first partially unroll the compute kernel loop and extract the loop body into its corresponding data flow graph (DFG). Subsequently, a suitable overlay configuration is selected from the pre-built implementation library on which the DFG is scheduled to execute. Optionally, the user may choose to iteratively refine the selection by feeding back scheduling performance and estimated communication cost at each iteration. The selected pre-built overlay implementation is then updated with the corresponding scheduling result to create the final FPGA configuration bitstream. Finally, this updated bitstream is combined with the rest of the software to form the final application that will be executed on the target CPU-FPGA system.

While there may seem to be a lot of steps involved in this fast path, all of them run relatively quickly. Furthermore, the only loop in this compilation flow is the accelerator selection process, which as explained is an optional step that can be bypassed if the user opts for speed over quality. The result is that the run time of this fast path can be kept within the order of tens of seconds. It allows users to perform rapid design iterations, which is particularly important during early application development phases.

On the other hand, the slow path of QuickDough consists of all the remaining time-consuming steps in the flow from Figure 1.3. These steps are responsible for implementing the overlay in hardware, optimizing the CGRA to the user application, and updating the overlay library as needed. Although these steps are slow, they are not necessarily by run for every design compilation. For example, running through the low-level FPGA implementation is a slow process, however, they are needed only when a new overlay configuration is required. If a compatible overlay configuration already exists in the pre-built overlay library, then the user may choose to reuse the existing implementation during early developments to facilitate fast design turn-around. When the user decides that she is ready to spend time optimizing the overlay configuration, she may then instruct QuickDough to execute these slow steps. With the slower steps, QuickDough will then be able to analyze the user application requirement, customize the overlay accordingly, and finally generate the FPGA configuration bitstream. Once this bitstream is stored in the overlay library, the user will not need to go through these slow process again.

Throughout the entire application development cycle, most of the time the user will be able to execute only the fast steps, and run through the slow steps only occasionally. As a result, the overlay compilation speed remains orders of magnitude better than traditional hardware design flow on average.

1.4.2 The QuickDough Overlay

Figure 1.4 shows an overview of the QuickDough overlay together with its data I/O infrastructure. The QuickDough overlay as shown in the right hand side consists of an array of simple processing elements (PEs) connected by a direct network. Each PE computes and forwards data to their neighbors synchronously according to a static schedule. This schedule is stored in the instruction ROM associated with each PE and controls the action of each PE's action in every cycle. Finally, each PE contains a scratchpad data memory for run-time data that may be reused in the same PE or be forwarded in subsequent steps.

Communication between the accelerator and the host processor is carried through a pair of input/output buffers. Like with the rest of the PE array, accesses to these I/O buffers from the array also take place in lock step with the rest of the system. The location to access in each cycle is controlled by a pair of address buffers, which contains address information generated from the QuickDough compiler.

Figure 1.4 also shows the connection within a processing element. Each PE is constructed surrounding an ALU, with multiplexors connecting the input/output of the ALU to either the internal memory or the neighboring PEs. In this particular version, the optional load/store path is also shown, which is presented only at dedicated PE connected to the I/O buffer. Within the PE, the ALU is supported by a multi-port data memory and an instruction memory. Three of the data memory's read ports are connected to the ALU as inputs, while the remaining ports are sent to the output multiplexors for connection to neighboring PEs and the optional store path to output buffer (`OBuf`) external to the PE. At the same time, this data memory takes input from the ALU output, data arriving

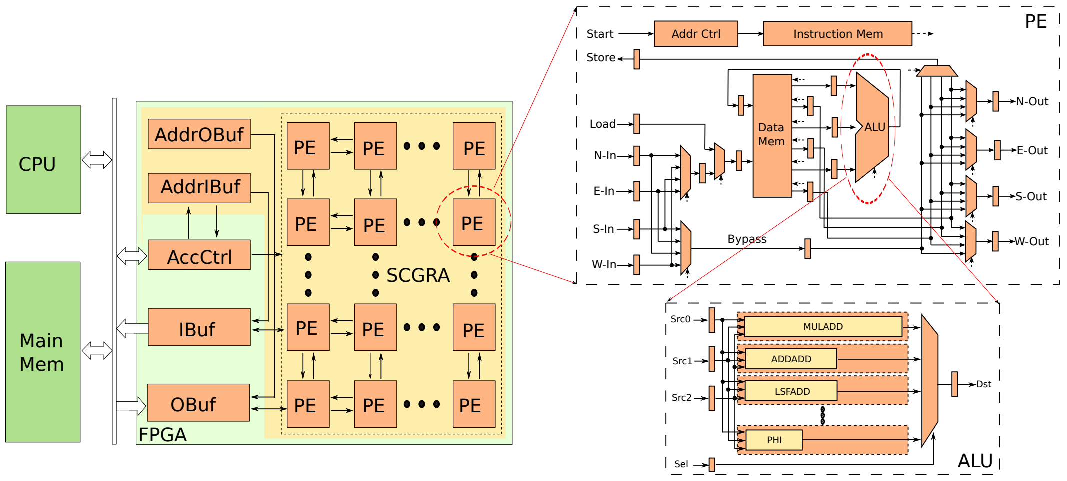


Figure 1.4: Overview of the QuickDough generated system. Implemented on the FPGA is the QuickDough overlay, which is an array of PEs connected with a direct network, and the communication infrastructure with the host.

from neighboring PEs, as well as from the optional loading path from the data input buffer (`IBuf`).

The ALU itself has a simple design that supports up to 16 fully pipelined operations. It is constructed also as a template and may be customized to support any user-defined 3-input operations. Regardless of its function, each operation must have a deterministic pipeline depth. With that, the QuickDough scheduler will ensure there is never any output conflict and allow the operations to execute in parallel as needed.

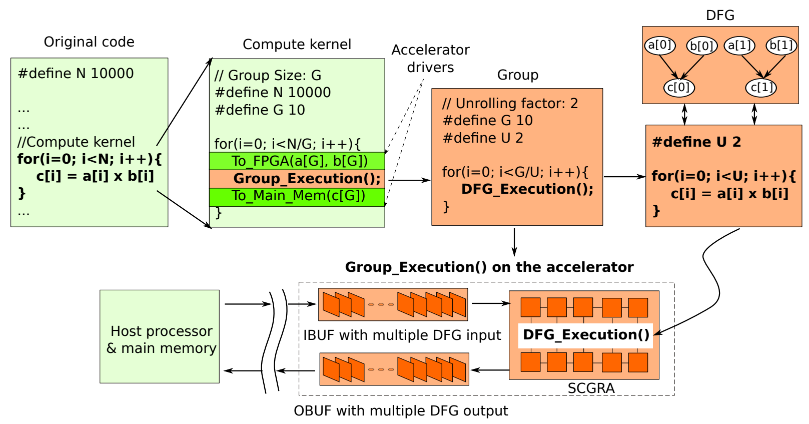
Finally, note that the overlay is designed as a soft template. Many parameters of the overlay, including the SCGRA array size, I/O buffer size, as well as the operations supported by the ALU, are configurable. They can be optimized for a particular group of application upon user's request as part of the framework's customization steps.

1.4.3 Loop Accelerator Generation

As mentioned, an important goal of QuickDough is to generate high-performance FPGA accelerator systems in software compilation speed. In this subsection, we will walk through some of the major steps involved and explore the details on how they help generate hardware accelerators very efficiently.

DFG Generation

The top-level inputs to QuickDough are loops that the user has designated for acceleration. To accelerate a loop, the straightforward way is to treat each loop iteration as an individual acceleration target and rely on the host processor for loop control. However, it is not ideal because (i) the overhead for data and control transfer between the host and FPGA will be


Original code
Compute kernel
Accelerator drivers
Group
DFG
a[0]
b[0]
a[1]
b[1]
c[0]
c[1]
Group_Execution() on the accelerator
Host processor & main memory
IBUF with multiple DFG input
DFG_Execution()
SCGRA
OBUF with multiple DFG output

```
#define N 10000

...
...
//Compute kernel
for(i=0; i<N; i++){
    c[i] = a[i] x b[i]
}
...
```

```
// Group Size: G
#define N 10000
#define G 10

for(i=0; i<N/G; i++){
    To_FPGA(a[G], b[G])
    Group_Execution();
    To_Main_Mem(c[G])
}
```

```
// Unrolling factor: 2
#define G 10
#define U 2

for(i=0; i<G/U; i++){
    DFG_Execution();
}
```

```
#define U 2

for(i=0; i<U; i++){
    c[i] = a[i] x b[i]
}
```

Figure 1.5: Loop execution on an SCGRA overlay based FPGA accelerator

too high, (ii) the amount of parallelism encapsulated in a single iteration of the loop body is likely to be too low for acceleration, and (iii) there will be no data reuse between loop iterations to amortize the transfer overhead between the host and the FPGA.

Therefore, just like most other acceleration frameworks, QuickDough begins the accelerator generation process by partially unrolling the input loop U times to increase the amount of parallelism available. The body of this partially unrolled loop subsequently forms the basic unit B for acceleration in later steps. This is the unit that is implemented on the FPGA accelerator using the SCGRA overlay. With a single copy of B implemented in the accelerator, the original loop is completed by executing the accelerator N/U times, where N is the original loop bound.

Furthermore, to amortize the communication cost between the host processor and the accelerator, input/output data are transferred in groups for G invocations unit B as shown in Figure 1.5. By buffering I/O and intermediate data on the accelerator, this grouping strategy also allow reusing data between loop iterations, further enhancing performance.

In our current implementation, the unrolling factor U is specified by the user, while the grouping factor G is determined by the framework automatically. G is determined based the on-chip buffer size

At the end of this process, the unrolled loop body B will form the core of the accelerator and the corresponding data flow graph (DFG) will be extracted, which will drive the rest of the generation flow.

Accelerator Selection

As an accelerator generation framework, an important task of QuickDough is without doubt to generate the physical implementation of the accelerator on the FPGA. With its SCGRA serving as an intermediate layer, this translates into physically implementing the overlay on the FPGA. Unfortunately, no matter how simple the overlay is, running through the low-level hardware implementation tools is going to be a lengthy process, contradicting the original goal of employing the overlay in the first place. In order to sidestep this lengthy hardware implementation process, QuickDough instead transform it into a selection process from a library of pre-implemented overlay.

When compared to directly implementing the overlay using standard hardware implementation tools, this accelerator selection process is much faster. In return, the performance of the resulting overlay configuration may not be optimal for the given user application. Despite the suboptimal performance, the selected overlay implementation is functionally correct and should give a good sense of the overall hardware-software codesign process to the software programmer, which is necessary for rapid early development. The user may subsequently opt for the slower optimization and customization steps explained in the next subsection to progressively improve performance.

As such, the goal of this process is to select the *best* and *functional* overlay configuration from the pre-implemented library *rapidly*. While functionality is easy to determine, QuickDough must also be able to estimate the resulting performance swiftly for this selection process. Now, the performance of the accelerator depends mainly on two factors: (i) computational latency, and (ii) communication latency. As the SCGRA overlay is regular and computes with a deterministic schedule, its exact computational latency of carrying out the user DFG can readily be obtained from the scheduler. Of all the configurable parameters of the overlay, the SCGRA array size is the dominant factor affecting this computational latency. On the other hand, communication latency depends not only on the scheduling results, but also other factors such as communication pattern, grouping factor, I/O buffer sizes, as well as DMA transfer latency, etc. These parameters interact with each other to form a large design space. For rapid estimation, instead of performing a full design space exploration, QuickDough is able to rely on simple analytical model for estimating communication latency based on the overlay SCGRA array size.

Finally, QuickDough leaves this choice of effort in finding the best accelerator configuration to the user as 3 optimization effort levels:

- Level 0 (`O0`) – No optimization. QuickDough selects a feasible accelerator configuration with the smallest SCGRA size.
- Level 1 (`O1`) – QuickDough estimates performances of 3 accelerators with different array sizes and select the one that results in the best performance.
- Level 2 (`O2`) – QuickDough performs an exhaustive search on all accelerators in the library and searches for the best accelerator configuration.

Obviously, the more effort is being put into the selection process, the longer the process will take, and the resulting performance is improved most of the time. The choice on how much effort to pay is up to the user.

DFG Scheduling

This is the step where the DFG extracted from the unrolled loop body is scheduled to execute on the selected SCGRA overlay. For each operation in the DFG, the scheduler must decide the cycle and the PE in which the operation should be carried out. The scheduler also determines the routing of data between the producing and consuming PEs, as well as to/from the I/O buffer.

As QuickDough tends to target DFGs with close to thousands of node, to keep the scheduling time short, a classical list scheduling algorithm was adopted [Sch96]. A scheduling metric proposed in [LS12] that considers both load balancing and communication cost was used in the scheduler.

At the end of this scheduling step, a schedule with operation of each PE and data I/O buffer in every cycle is produced. This schedule essentially turns the generic overlay implementation into the specific accelerator for the user application.

Accelerator Bitstream Generation

The final step of the accelerator generation process is to produce the instructions for each PE and the address sequences for the I/O buffers according to the scheduler's result.

As a way to reduce area overhead of the overlay, both the instruction memory of each PE as well as the I/O address buffers are implemented as read-only memories (ROMs) on the physical FPGA. To update the content of these memories, the QuickDough framework must update the FPGA implementation bitstream directly. To do that, the memory organization and placement information of the target overlay implementation is obtained from an `XDL` file corresponding to the overlay implementation [BKT11]. The resulting memory organization information is encoded in the corresponding `BMM` file, which is combined with the scheduler results to update the overlay implementation bitstream with the the `data2mem` tool from Xilinx [Xil11].

While it may sound complicated, the whole process is automated and consumes only a few seconds of tools run time. The result of this process is an updated bitstream with all application-specific instructions embedded. This updated bitstream is the final configuration file used to program the physical FPGA, and it concludes the QuickDough flow.

1.4.4 Customization & Optimization

The true power of utilizing an FPGA overlay rests on that fact that the overlay is virtual, soft, and easily customizable for the need of the application. In this part, we will examine various ways QuickDough enables user to customize and optimize the overlay to improve power-performance of the generated accelerators. Unlike the fast generation path explained above, these customization steps are much more time consuming. As such, these customization steps are considered part of the slow path in QuickDough's design philosophy, and are expected to execute only occasionally as needed.

There are a number of reasons why a user may want to spend the time on customization despite the slow process. To begin, the user may want to construct a prebuilt implementation library specific to the target application domain. It is useful as the result can be memoized in the implementation library and be reused by all other target applications, amortizing the initial implementation effort. Once the project development has gone through the initial debugging phase, a user may also opt for creating a customized overlay for the particular accelerated application. This per-application optimization will help improve the performance of the accelerator but the process will undoubtedly be slower than to select a prebuilt implementation. Luckily, the result of this per-application customization can be stored in the prebuilt library such that it can be reused on subsequent compilations.

What can be customized?

For the purpose of customization, the QuickDough overlay was designed from the beginning to be a template from which a family of overlay instances could be generated. The SCGRA overlay consists of a regular array of simple PEs connected with direct connections. Depending on the application requirements, a range of design parameters of this overlay can then be customized, including the array size, on-chip scratchpad data memory capacity, instruction memory capacity and I/O buffer capacity. On top of that, as a hardware-software system generator for loop acceleration, QuickDough may also optimizes the communication between the accelerator and the host software by varying the *grouping factor*, which determines the amount of data transferred between the two in each transaction. Finally, it may also optionally optimizes the *loop unrolling factor* on behalf of the user depending on the computational capability of the overlay as a function of its array size.

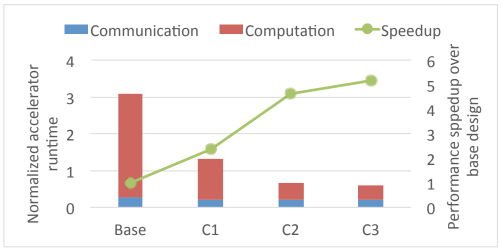
Obviously, a major challenge when optimizing this set of parameters is that they tend to interact with one another in fairly complex ways. For example, increasing the array size increases the compute capability of the array, potentially improving the accelerator performance. However, the increased size is beneficial only if the input DFG has enough parallelism presented to take advantage of the increased capability. To increase the amount of available computation, QuickDough may optionally increase the amount of loop unrolling in the user-supplied kernel. However, that also increases the amount of data I/O required, as well as the on-chip instruction and data buffer requirement, which in turn limits the size of the array in the first place. A full discussion of the detailed customization and optimization process is beyond the scope of this chapter. Yet it is worthwhile to realize the potential of customization here.

As an illustration, Figure 1.6 shows the effect of customization on the generated accelerators from QuickDough. In this example, accelerator for a 49-tap FIR filter was generated on the Zedboard using QuickDough. Three different customized overlay configurations were generated. Beginning with the specified array size, the rest of the configuration parameters were determined automatically.

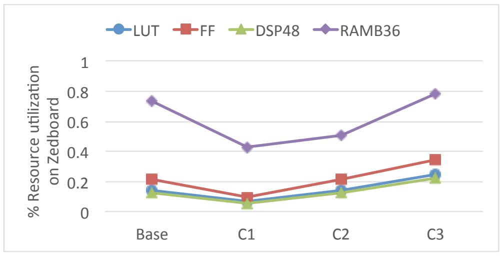
A few interesting observations can be made from the figures. In terms of performance, comparing to the sub-optimal baseline configuration, the best configuration (C3) results in more than 5 times improvement in performance. Furthermore, comparing the baseline and C2 that feature the same array size (3×3), $4.7\times$ improvement in performance can be obtained by optimizing the unrolling and grouping factor in combination with the on-chip memory configuration. Interestingly, looking at the resource consumptions, C2 in fact consumes 31% less on-chip memory and the same resource otherwise as the baseline.

1.4.5 Summary

Using QuickDough as a case study, we have illustrated how the use of an FPGA overlay has enabled a very different experience for software programmers when designing hardware-software systems. The 2-layer approach to hardware design allows for a very rapid design experience that sharply contrasts the lengthy low-level hardware tool flow. Furthermore, the softness of the overlay provides opportunity for further customization and optimization as the development effort progresses. Together, it creates for novice users a “software-like” way



(a) Performance



(b) Resource Consumptions

	base	C1	C2	C3
Loop unrolling factor	10×50	50×50	50×50	50×50
Grouping factor	100×50	2500×50	1250×50	5000×50
SCGRA size	3×3	2×2	3×3	4×4
Inst Mem Depth	4k	4k	2k	1k
IO Buffer Depth	1k	4k	2k	8k

(c) Overlay configuration

Figure 1.6: Effect of overlay customization on performance and resource consumption. Example here shows the result of an FIR filter using QuickDough with 3 different overlay configurations over a base design.

to designing complex accelerator systems while maintaining considerable overall acceleration performance promised by FPGAs.

1.5 Research Challenges & Opportunities

While the early results of using FPGA overlay are encouraging, there remain many challenges that must be overcome before its full potential can be unleashed. In particular, for FPGA overlays to be useful, future research must be able to address two important challenges: reduce overhead and enhance customization.

1.5.1 Reducing Overhead

By introducing an additional architectural layer between a user application and the physical fabric, an FPGA overlay inevitably incurs additional performance and area overhead to the system. It is especially important when the FPGA is used as an accelerator in a processor-based system. If the overhead is so large that the FPGA is no longer offering any significant performance enhancement, then software programmers will have little incentive to devote effort into using them.

From a hardware implementation's point of view, the additional layer must also incur additional area overhead. Such overhead usually manifests as additional usage of configurable fabric and on-chip memory. In some cases where spare resources are available, the effect of such area overhead may not be apparent. In fact, as illustrated earlier, researchers have deduced ways to make sure of such spare resources for debugging purposes [HW13]. However, in other cases where the area of FPGA overlay results in a reduction of resources available to implement a user's design, then it is likely that this area overhead will translate into performance overhead. On a circuit level, such area overhead may also manifest as additional delays impose on the critical path, resulting in overall performance loss. Therefore in short, area overhead, if not well controlled, may easily translate into performance overhead.

Luckily, as mentioned in some early works, despite such overhead, the use of an overlay may still be worthwhile from a performance perspective. For example in [CA13], Capalija and Abdelrahman has demonstrated that by taking advantage of the regularity of its overlay and with the help of detailed floor planning, they were able to control the overhead while maintaining reasonable throughput performance when compared to a simple push-button synthesis flow. The very use of overlay allows amortization of the optimization effort in the long run as the overlay structure may be reused many times.

1.5.2 Enhancing Customization

Another major challenge faced by overlay designers is the very notion of using FPGA overlay in the first place: "If by overlaying a different architecture over an FPGA may provide all sort of nice properties, then why not implement the overlay architecture directly on silicon to enjoy all the benefits instead?" For example, if a GPU overlay provides good performance and good programmability, then maybe the design should be implemented on an actual GPU instead of a GPU overlaying on top of an FPGA.

Indeed, if the so called "overlay" in question is simply a fixed reimplementation of another architecture on an FPGA, then the only incentive to so are probably to provide compatibility through virtualization, or simply to save cost on silicon implementation. In the latter case, this intermediate architecture may hardly be called an "overlay" any more.

However, the true power of FPGA overlays is that they can be adapted to the application through *customization*. As a virtual architecture, many aspects of an overlay architecture can be customized to the target application to improve power-performance. In [LCD$^+$10], for example, the computational core in the multi-core overlay may be customized to the specific targeted application. By customizing the core to their targeted application, almost 10x improvement in the overlay performance can be observed, bring the overlay performance to be within factor of 3 of the reference custom FPGA design. In [LS12], the direct connection topology among the process elements in coarse-grained reconfigurable array overlay was customized against the input application. When compared to the best predefined topology, the application-specific interconnect provides up to 28% improvement in resulting energy-delay product.

The improved results should come at no surprise: the extreme case of an overlay customization is simply a full-custom design of the application on the target FPGA, which is supposed to have the best possible performance if designed correctly. What is challenging is therefore to ability to fine-tune the tradeoff among design productivity, virtualization and performance of the resulting system. In other word, the research question to ask in the future should therefore be: "How to improve performance of an overlaying system through customization without significantly sacrificing the benefits of using an overlay?" The answer to this question is going to open up a wide field of exciting research in FPGA-based reconfigurable systems.

1.5.3 Closing Thoughts

FPGA overlay is without doubt going to be an important part of future FPGA-based reconfigurable systems. The potential is plentiful and we anticipate that overlaying technology will benefit all aspects of future reconfigurable systems, especially on the important aspect of design productivity. With silicon technologies continue to evolve, the amount of available on-chip configurable resource is going to increase. The abundance of high-performance configurable resources on FPGA will enable a new generation of systems that relies heavily on advanced overlay architectures for virtualization and to improve design productivity.

Bibliography

[BKT11] C. Beckhoff, D. Koch, and J. Torresen. The Xilinx design language (XDL): Tutorial and use cases. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, pages 1–8. IEEE, 2011.

[BL12] A. Brant and G.G.F. Lemieux. ZUMA: An open FPGA overlay architecture. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 93–96, 2012.

[BTA93] J. Babb, R. Tessier, and A. Agarwal. Virtual wires: overcoming pin limitations in FPGA-based logic emulators. In *FPGAs for Custom Computing Machines, 1993. Proceedings. IEEE Workshop on*, pages 142–151, Apr 1993.

[CA13] D. Capalija and T.S. Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, Sept 2013.

[CH02] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, 34(2):171–210, 2002.

[CS10] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 13–22, Oct 2010.

[CS15] James Coole and Greg Stitt. Adjustable-cost overlays for runtime compilation. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 21–24, May 2015.

[DMC$^+$06] André DeHon, Yury Markovsky, Eylon Caspi, Michael Chu, Randy Huang, Stylianos Perissakis, Laura Pozzi, Joseph Yeh, and John Wawrzynek. Stream computations organized for reconfigurable execution. *Microprocessors and Microsystems*, 30:334–354, 9 2006.

[FC05] W. Fu and K. Compton. An execution environment for reconfigurable computing. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 149 – 158, april 2005.

[FVM$^+$11] R. Ferreira, J.G. Vendramini, L. Mucida, M.M. Pereira, and L. Carro. An FPGA-based heterogeneous coarse-grained dynamically reconfigurable architecture. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 195–204. ACM, 2011.

[GWL11] David Grant, Chris Wang, and Guy G.F. Lemieux. A CAD framework for Malibu: An FPGA with time-multiplexed coarse-grained elements. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 123–132, New York, NY, USA, 2011. ACM.

[HIS14] B.K. Hamilton, M. Inggs, and H.K.-H. So. Mixed-architecture process scheduling on tightly coupled reconfigurable computers. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–4, Sept 2014.

[HW13] Eddie Hung and Steven J.E. Wilton. Towards simulator-like observability for FPGAs: A virtual overlay network for trace-buffers. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 19–28, New York, NY, USA, 2013. ACM.

[HW14] E. Hung and S.J.E. Wilton. Incremental trace-buffer insertion for FPGA debug. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 22(4):850–863, April 2014.

[JFM15] Abhishek Kumar Jain, Suhaib A. Fahmy, and Douglas L. Maskell. Efficient overlay architecture based on dsp blocks. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 25–28, May 2015.

[KBL13] D. Koch, C. Beckhoff, and G.G.F. Lemieux. An efficient FPGA overlay for portable custom instruction set extensions. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, Sept 2013.

[KHKT06] Dmitrij Kissler, Frank Hannig, Alexey Kupriyanov, and Jürgen Teich. A dynamically reconfigurable weakly programmable processor array architecture template. In *ReCoSoC*, pages 31–37, 2006.

[KS11] Jeffrey Kingyens and J. Gregory Steffan. The potential for a GPU-Like overlay architecture for FPGAs. *Int. J. Reconfig. Comp.*, 2011, 2011.

[LCD$^+$10] I. Lebedev, Shaoyi Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, Mingjie Lin, and J. Wawrzynek. MARC: A many-core approach to reconfigurable computing. In *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pages 7 –12, dec. 2010.

[LMVV05] Roman Lysecky, Kris Miller, Frank Vahid, and Kees Vissers. Firm-core virtual FPGA for just-in-time FPGA compilation. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, FPGA '05, pages 271–271, New York, NY, USA, 2005. ACM.

[LP09] Enno Lübbers and Marco Platzner. ReconOS: Multithreaded programming for reconfigurable computers. *ACM Trans. Embed. Comput. Syst.*, 9(1):8:1–8:33, October 2009.

[LPL$^+$11] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 117 –124, may 2011.

[LS12] Colin Yu Lin and Hayden Kwok-Hay So. Energy-efficient dataflow computations on FPGAs using application-specific coarse-grain architecture synthesis. *SIGARCH Comput. Archit. News*, 40(5):58–63, March 2012.

[LS15] Cheng Liu and Hayden Kwok-Hay So. Automatic nested loop acceleration on FPGAs using soft CGRA overlay. In *FPGAs for Software Programmers (FSP), Second International Workshop on*, Sept 2015.

[SB08] Hayden Kwok-Hay So and Robert Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *Transactions on Embedded Computing Systems*, 7(2):1–28, 2008.

[SBB06] S. Shukla, N.W. Bergmann, and J. Becker. QUKU: a two-level reconfigurable architecture. In *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, March 2006.

[Sch96] JMJ Schutten. List scheduling revisited. *Operations Research Letters*, 18(4):167–170, 1996.

[SL12] A. Severance and G. Lemieux. VENICE: A compact vector processor for FPGA applications. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 261–268, Dec 2012.

[TB01] Russell Tessier and Wayne Burleson. Reconfigurable computing for digital signal processing: A survey. *Journal of VLSI Signal Processing*, 28(1):7–27, June 2001.

[TCJW97] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed FPGA. In *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, page 22, Washington, DC, USA, 1997. IEEE Computer Society.

[Xil11] Xilinx. *Data2MEM User Guide*, ug658 (v13.3) edition, Oct 2011. [Online; accessed 19-September-2012].

[YSR09] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. Fine-grain performance scaling of soft vector processors. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 97–106, New York, NY, USA, 2009. ACM.