Visions for Application Development on Hybrid Computing Systems

Roger D. Chamberlain, Joseph Lancaster, Ron K. Cytron
Dept. of Computer Science and Engineering
Washington University in St. Louis

Abstract

Hybrid computing systems (incorporating FPGAs, GPUs, etc.) have received considerable attention recently as an approach to significant performance gains in many problem domains. Deploying applications on these systems, however, has proven to be difficult and very labor intensive. In this paper we present our vision of the application development languages and tools that we believe would greatly benefit the process of designing, implementing, and deploying applications on hybrid systems.

1 Introduction

Recent years have seen a slowing of the rate of increase in clock speed for individual general-purpose processors, yet the demand for ever-improving computational performance has continued unabated. One emerging approach to satisfying this performance demand is the use of hybrid systems—systems that employ more than one type of computing engine to perform application tasks—rather than exclusively relying on greater numbers of traditional processors. We use the term hybrid system to imply the collection of a disparate set of constituent computing engines, which we call components. Hybrid systems can be constructed using any number of computing components, including traditional general-purpose processors (GPPs), homogeneous or heterogeneous chip multiprocessors (CMPs), field programmable gate arrays (FPGAs), field programmable object arrays (FPOAs), graphics processing units (GPUs), digital signal processors (DSPs), application specific instruction processors (ASIPs), etc.

Each of the above computing components has its merits, including gaining significant performance for the application domain that motivated its development; however, true general-purpose utilization of these components has been somewhat elusive. Each component has its own unique characteristics, both in terms of architecture and application development process. In addition, deploying portions of applications across multiple computing components further requires that the different components coordinate their efforts. This imposes additional difficulties on the application developer, as these disparate subsystems are not typically well integrated into a coherent whole.

In this paper, we articulate our vision for the future state of application development on hybrid platforms. This vision is driven by our experience building hybrid systems [5] as well as our experience developing applications for hybrid systems [13]. In some cases, we make concrete proposals as to how, we believe, developers should be building applications for hybrid systems. In other cases, we describe open problems we consider to be important for the future in this area. We start with a discussion of what it means to be a hybrid system and how hybrid systems are constructed. We follow that with a discussion of programming languages used to express applications for execution on the system and finish with a discussion of tools that are appropriate for the application development task.

2 Hybrid Computing Systems

We define a hybrid computing system as a number of heterogeneous computing components connected together to create a larger computational resource, typically (but not necessarily) packaged within a single enclosure. Hybrid systems are frequently used in high-performance embedded computing (HPEC) applications (e.g., medical instrumentation, military signal processing) where there are often stringent size, weight, and/or power constraints on the system. Also, as is common with traditional general-purpose processors, large collections of hybrid "nodes" can be networked together to form a cluster, thereby obtaining significant advantages in the high-performance computing (HPC) arena. For many compute-intensive applications, a hybrid system can perform significantly faster than a cluster of similar size that is constructed exclusively with general-purpose processors. Another approach to utilizing hybrid computing systems is to reduce the size of

the cluster required to compute a given task, thereby reaping the benefits of greatly reduced power and maintainability obligations. Recent large-scale HPC clusters have demonstrated a trend towards hybrid computing systems as with *Roadrunner*[1], which is a Cell and Opteron hybrid architecture. We will now survey the types of computing components used in hybrid systems.

2.1 Traditional Components

General-purpose processors (GPPs) are still the most common computing components used in both the HPEC and HPC worlds, and virtually all hybrid systems still maintain a healthy quantity of GPPs as computing components. For a given generation of these processors, improved performance is provided via coarse-grained parallelism, and massively parallel HPC clusters have been constructed consisting of network-connected collections of single-processor nodes. As symmetric multiprocessor (SMP) systems became available, cluster nodes were expanded to include additional processors (scaling up until the local interconnect, typically a bus, was saturated). From one generation to the next, clock frequency increases and the resulting per processor performance gains had a dramatic impact sustaining this general approach to achieving high performance. Rather than invest in alternative architectural approaches, one could realize dramatic computational capacity increases simply by buying and installing the latest generation of GPPs. Recently, performance gains due to increased ILP and clock frequency have diminished and architects are exploring alternate paths to increased performance. Figure 1 shows the clock frequency trends of recent Intel Xeon server processors.

HPC clusters still take similar forms today. With the arrival of homogeneous chip multiprocessors (CMPs), the core density within an SMP node is rapidly increasing. While GPPs offer many advantages in terms of flexibility and programmability, increases in per-core performance are dwindling while power requirements are soaring. The trend is therefore toward more cores on a chip, typically running at a slower clock frequency than single core processors.

While considerable research has been undertaken into how best to construct these chips (e.g., determining an appropriate cache organization) as well as program these chips (current practice is to view them as similar to SMP systems and use memory-based synchronization), there is no clear consensus yet as

[1]See `http://www-03.ibm.com/press/us/en/pressrelease/20210.wss`.

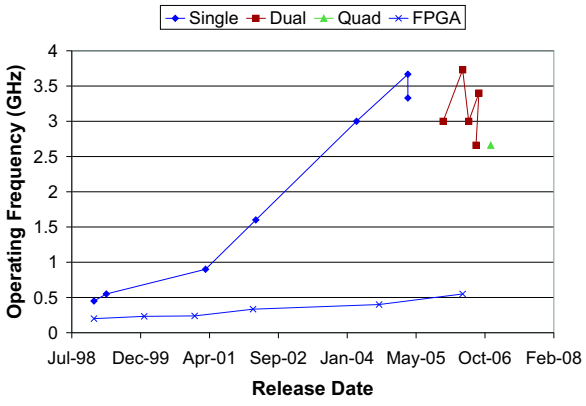

Figure 1: Operating frequencies of Intel Xeon processors and Xilinx Virtex series FPGAs over time. FPGA performance is for a 16-bit addition on the Virtex through Virtex-5.

to appropriate solutions to these issues. The issues are complicated further by the advent of heterogeneous CMPs (e.g., the Cell). Here, not only does one have to re-address the scheduling problem due to the heterogeneous nature of the chip, but in addition the memory model for the different processors within the chip is distinct.

2.2 Alternative Components

Since the improvement in performance from one generation to the next with GPP cores has slowed, further performance gains must come from alternative approaches. A number of approaches to exploiting silicon for computation have been designed, often originally for some specialized application purpose, and the hybrid system community is attempting to utilize these alternative computing components for general purposes.

Reconfigurable logic, in the form of field-programmable gate arrays (FPGAs), is becoming increasingly popular as a computing component. Historically relegated to the task of providing simple "glue logic" in custom hardware designs, FPGAs have grown in size tremendously in recent years to the point that entire applications can effectively deployed on them. Manufacturers are also tailoring FPGAs to the needs of different application areas, including optional embedded processor cores, multiply-accumulate units, specialized I/O functionality, etc. FPGAs can achieve better performance than that of a GPP by exploiting both fine- and coarse-grained parallelism present in an application. To exploit this parallelism, an FPGA provides a large quantity of

configurable logic with which calculations can be performed. By controlling this logic precisely, many calculations can be performed concurrently. In addition, FPGAs have extremely large amounts of on-chip memory bandwidth available to support the data needs inherent to this degree of parallelism. Carefully architecting an FPGA circuit can lead to a large speedup over a GPP in many non-trivial applications [11, 13]. Even though FPGAs tend to run at much lower clock frequencies than GPPs, the clock frequency gap between the two is actually decreasing for the first time as modern GPPs scale back clock frequency in favor of more cores on a chip (see Figure 1). Unfortunately, not all applications are suitable for deployment to an FPGA. FPGAs generally run at lower clock speeds than GPPs, do not have native support for floating point units, rely on board designers to incorporate external components (external I/O, memory), and require significantly larger design cycles than that of software. In spite of these drawbacks, FPGAs are increasingly being used to accelerate computations in performance critical applications.

Partially to address the clock frequency limitations of FPGAs, field-programmable object arrays (FPOAs) have been proposed as an alternative to FPGAs [3]. Similar in nature to FPGAs, FPOAs allow arbitrary interconnection between on-chip function units. They differ from FPGAs in that the functional units are fixed, and therefore can be designed to operate at higher frequencies. Mathstar offers a commercial FPOA that operates at speeds of up to 1 GHz.

Graphics processing units (GPUs) are also a recent addition to the general computing component toolbox [2, 10]. Both major GPU manufacturers have provided the ability to utilize at least a subset of the processing elements for computations other than graphics applications. Most recently, there has been a move toward a unified stream processor architecture, greatly increasing their worth as a general computation resource. Offering many programmable units running in parallel at high speeds, GPUs are well suited for deployment of data-parallel, floating-point applications.

Digital signal processors (DSPs) have long been, and still remain, the most popular choice for many signal processing applications. DSPs operate much like GPPs with the notable exception that their ISA and resulting microarchitecure have been carefully tailored to perform tasks in the signal processing domain (e.g., multiply-accumulate instructions, parallel address register computations).

Application specific instruction processors (ASIPs) are also a candidate for hybrid system deployment. Commercial examples of these systems, such as the PhysX from Ageia, custom instruction-set processors from Tensilica, and the floating-point processor from ClearSpeed, promise performance and power benefits if the application can effectively exploit their capabilities.

2.3 The Viability of Hybrid Systems

There are a number of hybrid systems in existence, both in the research community and commercially available. The simplest to construct machines are based upon commercially available motherboards containing GPPs that are then expanded with one or more hybrid computing components. Several manufacturers make FPGA boards, including I/O bus-based boards from Annapolis Microsystems and Nallatech; and HyperTransport boards from Celoxica, DRCComputers, and XtremeData. GPUs are traditionally connected via PCI-e slots on the motherboard. ASIPs from both ClearSpeed and Ageia can also attach via an I/O bus.

To avoid the limitations of standard motherboards, a number of companies have built hybrid systems that include higher performance interconnects between the computing components. Examples here include the SGI Altix line (which uses their proprietary NUMAlink interconnect); a whole family of machines from Mercury (which support GPPs, FPGAs, DSPs, Cells, etc.); and SRC systems. Of these companies, SGI is focused more toward the HPC community while more of Mercury's and SRC's business is in the HPEC community.

Unfortunately, hybrid computing systems are not without drawbacks. Developing and deploying an application on a hybrid system is more challenging than traditional GPP clusters due to the heterogeneous nature of the system. The model of the computer taken when developing applications for GPP systems is often resource agnostic. Many applications are developed with a mostly unrestricted view of memory, which creates problems porting code to components with a restricted working set. The interconnect between the GPP and the non-traditional component(s) may be high latency or low bandwidth, which may cause bottlenecks not present on the original cluster. The number of computational units sharing an interconnect is greatly increased in some cases, which can also lead to link saturation. Finally, some computing components, such as FPGAs, require more explicit description of the fine- and coarse-grain parallelism.

All of these issues transform a complex development task into a formidable Gordian knot of deployment.

Many of the issues described above can be addressed by altering the state of practice for application development on hybrid systems. There is a need for changing the level of abstraction at which some of the components are programmed. More importantly, the lack of a common communication model and associated data delivery infrastructure is hindering the ability to both develop and reuse application code. Little portability between different components also requires a rewrite of applications for deployment on different hardware. For FPGAs, developing languages explicitly designed for synthesis will help software engineers understand how to utilize these powerful devices. In the following two sections, we describe our vision of open problems and partial solutions to these issues.

3 Languages

There is considerable latitude in terms of languages, compilers, and tools that can target hybrid systems. At one extreme, developers could author code in the language of their choice, leaving the details of mapping their application to a hybrid system in the hands of a compiler. While compilers can substantially affect performance of an application, experience has shown that compilers cannot completely overcome gross mismatches between an application and a target platform. In this section, we discuss languages commonly used for programming hybrid systems, pointing out their deficiencies and envisioning how they could evolve to increase developer productivity. We defer discussion of compilers and development tools to Section 4.

3.1 Coordination Languages

In the current state of affairs, portions of an application must be coded explicitly for each of the computing components in a hybrid system individually and the developer is responsible for all of the coordination (i.e., data delivery, synchronization, etc.) between components. Going from bad to worse, this coordination activity usually entails significant interaction with the specific low-level implementation details of the development platform (e.g., developing DMA transfer engines, OS drivers, etc.), all of which are both difficult to construct and are rarely portable to any other platform. Requiring an application developer to implement the low-level drivers required to move data between a graphics processor and a digital signal processor is guaranteed to result in few applications actually ever being developed. The current state of affairs is clearly unacceptable.

While a single unified language that can be targeted (via an appropriate compiler) to any computing component is desirable (and will be discussed below in Section 3.3), significant benefits can be realized by ongoing support for languages that are computing-component specific. Typically there is a community of users that is familiar with both the language and the associated development tools (compilers, debuggers, performance analyzers, etc.). If well designed, they represent a good compromise between the developers' needs (high abstraction level, clarity of exposition) and the requirements of the computing platform (explicit representation of parallelism, etc.).

With the near-term necessity of supporting a set of component-specific languages and a long-term desire to continue this support, what is significantly lacking is the expression of the interactions between components required by the application. Essentially what is needed is a coordination language, a language that is used to manage the interactions between the individual languages used for algorithm development on individual computing components.

Lee [15] has argued that coordination languages represent a better mechanism for reasoning about concurrency than traditional thread-based approaches. Franklin et al. [6] have proposed the *X* language specifically as a coordination language for hybrid systems. Common to both of the above is the use of dataflow semantics between "kernels" or "blocks," undecomposable computations that are to be mapped to an individual computing component.

3.2 Component-Specific Languages

In many cases, component-specific languages in common use do not readily enable the application developer to take maximum advantage of the underlying computing component. In other cases, the ability to exploit the benefits of the component are present, but the programmer efficiency is poor, e.g., it is cumbersome to express the kinds of things one wishes to express. In this section, we will illustrate this point with two examples, one on low-level languages used to implement designs in reconfigurable logic and the other on common needs of signal-processing systems that are not effectively met in the language.

3.2.1 Hardware Description Languages

Reconfigurable logic is most frequently programmed using hardware description languages such as VHDL

or Verilog. While these languages are sufficiently expressive to enable the application developer to fully exploit the inherent capabilities of the physical system, they have a (well deserved) reputation for being difficult to learn and use.

This difficulty stems from two distinct sources, which can (and should) be separated. The first source of difficulty is the level of abstraction one is using to both think about and then specify the solution to some computational problem. In hardware description languages, the developer is describing the computation at the level of individual clock cycles, characterizing the parallelism explicitly, and keeping track of available functional (i.e., logic) and memory resources. It is design at this low level that enables the developer to fully exploit the capabilities of the reconfigurable logic computing component. While there has been research ongoing for a decade into the idea of specifying hardware at a higher level of abstraction [18, 19, 21], to date many of the resulting implementations have had limited performance [23]. While we encourage work in this area, our vision is along an alternate path.

The second source of difficulty in learning and using hardware description languages is that the two most commonly used languages (VHDL and Verilog) were both originally developed for a different purpose. In their original purpose, both were used as modeling languages to drive discrete-event simulations of digital systems with the intent of verifying logical properties of a design prior to its instantiation in physical hardware. As the capabilities of hardware synthesis algorithms increased, the typical use case transitioned from simulation followed by manual implementation to simulation followed by automatic synthesis. Clearly, the simulation languages already in existence had market penetration and a knowledgeable user community which institutionalized the adoption of these languages for synthesis purposes.

While inertia might be the primary reason VHDL and Verilog are the most widely used hardware description languages, there is ample room for improvement. One significant improvement that could readily be incorporated into hardware description languages is to promote state to a first-class object. In the current languages, the synthesizer infers whether or not state is implied within a block of code. As a result, it is easy (especially for a novice developer) to unintentionally imply state in a code block that the developer intended to be combinational, with dramatic implications on the functional correctness and resource usage of the resulting system.

In addition, it is necessary for the developer to author code that explicitly instantiates the state when desired. Rather than simply declaring a name to be registered (i.e., saved as state), a construction such as that illustrated in Figure 2 is commonly used. In the figure, the intent is for the registered signals (`regCount` and `regDir`) to be saved as state on the rising edge of the clock signal `clk`, drawing their values from the combinational signals `nxtCount` and `nxtDir`, respectively. The code segment also indicates that the values of these registers are 0 on reset.

```
always @(posedge clk) begin
    if (reset) begin
        regCount <= 0;
        regDir <= 0;
    end
    else begin
        regCount <= nxtCount;
        regDir <= nxtDir;
    end
end
```

Figure 2: Instantiation of state in Verilog.

There are several problems with this approach. First, if the developer intends `nxtCount` and `nxtDir` to be combinational (as is typically the case), care must be taken to not accidentally infer state in their assignment. Second, a separate name is used for the combinational signal that is input to the register and the registered signal itself, even though one name could suffice. Third, 10 lines of code were used to (in effect) indicate that the names `Count` and `Dir` are registered and have reset value 0. It is the developer's responsibility to keep straight the appropriate usage of the `nxt`-variant vs. the `reg`-variant of the names; and mistakes are often still legal expressions in the language, limiting the compiler's ability to check for errors in developer intent.

If, on the other hand, state was a first class object in the hardware description language, the 10 lines of code above could likely be reduced to only 2. Given that this construction is quite frequent, the overall code savings could be quite significant. More importantly, the code that describes the "next" values of `Count` and `Dir` could have combinational semantics enforced. The end result would be more compact code that is less prone to errors, an important consideration in any application development environment.

3.2.2 Custom Numerical Representations

On general-purpose processors, the native representation for fixed-point numbers is an integer representation, typically a 2's complement integer which ranges from $-(2^{n-1})$ to $+(2^{n-1}-1)$ for an n-bit variable. The (implied) radix point is immediately to the right of the least-significant bit, yielding a numerical resolution of 1. IEEE Std. 754 provides a representation for floating-point numbers that is natively supported on many platforms.

The above conventions, however, are overly constraining in many applications. For example, many fixed-point DSP chips have a native representation in which the implied radix point is to the right of the most-significant bit. A variety of fixed-point and floating-point [1] representations are commonly used in FPGAs, which are also flexible enough that logarithmic representations have received interest [7]. While some languages allow these non-standard data types to be directly manipulated, most do not.

We propose that language conventions be developed that allow the programmer to explicitly express the nature of the numerical representation(s) to be used for each variable and (optionally) for each expression. For fixed-point representations, the commonly used Q notation would be appropriate. In Q notation, a Q$m.n$ number contains m bits of integer, n bits of fraction, and an additional sign bit. For example, the 16-bit fractional representation common in DSPs would be expressed as being in Q0.15 format. Annotating variables (and expressions) with Q notation enables the compiler to more directly map the desired intent onto the available computing component, independent of the computing component involved (DSP, microcontroller, FPGA, etc.).

3.3 General Languages

The above sections make the assumption that a different source language is appropriate for each distinct computing component. While this approach has the advantage that the language can be tailored to the particular needs of the component, there are clear benefits to be gained if a single common language could be effectively used for all components. While we consider the design of such a language an open problem, we next describe our vision of a few of the properties we believe it should have.

3.3.1 Expression of Concurrency

First and foremost, a language that hopes to be common across most types and styles of computing component must have effective mechanisms to express the allowable concurrency present in the underlying computation. Current approaches to this are woefully inadequate. At the thread level, synchronization via shared memory mechanisms such as locks and/or semaphores is much too heavyweight an approach for reconfigurable logic computing components. A similar comment can be made about explicit message-passing systems as well (e.g., MPI, CORBA). Streaming languages (e.g., Streams-C [9], StreamIt [14], Brook [2]) are an improvement, as they are reasonably effective at expressing pipelined concurrency. It is not clear, however, how truly general purpose is the streaming data model. At the other end of the spectrum, hardware description languages are excellent at expressing concurrency and allow very low overhead synchronization between concurrently operating agents, but they require way too much detailed time management on the part of the developer (i.e., explicit scheduling of operations to individual clock cycles).

What is needed are language mechanisms that:

- allow the programmer to express the concurrency that is potentially present in the application, and
- are at a sufficiently high level of abstraction that the programmer can focus his or her reasoning on the operation of the algorithms.

The expression of concurrency must handle both fine-grained concurrency (that can be exploited within an individual computing component) and course-grained concurrency (for exploitation across multiple computing components within an individual hybrid system or across multiple nodes in a parallel clustered environment).

An important piece in any expression of concurrency is the need to coordinate data dependencies across the system. In Section 4.3 we address approaches to discovering producer-consumer data dependencies in existing code. What is also required are mechanisms to efficiently describe these relationships when they are known to the author.

3.3.2 Memory Resource Management

While the clear, efficient expression of concurrency is crucial to a universal language for hybrid systems, that is not all that is required. In spite of the varied architectures present in hybrid systems, the memory wall is still with us and is not likely to depart any time in the foreseeable future. By the memory wall, we are referring to the extremely high relative cost of

accessing one or more data elements from an off-chip remote memory vs. an on-chip local memory.

Table 1 shows the on-chip memory available for a number of recently produced chips. While there is some variation, partially due to the silicon process used and the die size, the major conclusion is the fact that these memory capacities are all relatively close to one another. Independent of whether one is architecting a GPP, an FPGA, a DSP, or any other chip, the physical constraints of fabrication limit the amount of memory that will fit on the chip. As the number of independent functional units on the chip increases, this memory must be divided up into ever smaller units, fundamentally limiting the quantity of on-chip memory that can be allocated to each functional unit on the chip.

Chip	Silicon Process (nm)	Memory (MB)
Intel Xeon (Potomac)	90	9
Intel Xeon (Tulsa)	65	16
Xilinx Virtex-5	65	1.4
TI TMS320C DSP	90	2

Table 1: Available on-chip memory capacity.

While general-purpose systems have employed highly sophisticated caching mechanisms to mitigate the performance impact of the memory wall, the performance gains achievable with many hybrid computing components are often directly attributable to their use of memory in ways that are inconsistent with traditional caching. DSPs regularly provide direct programmer control over a fraction of the address space that is on-chip. FPGAs are even more flexible, often having several independent memory subsystems, each having dedicated ports (independent address and data lines for each memory) and each separately tasked by the application developer.

An important implication is that to achieve high performance, developers must be given more direct control over the use and management of physical memory, whether it be local on-chip or remote off-chip. There is already a need for this type of control, and it is frequently gerrymandered when not explicitly available. As an example, consider the automatic tuning of scientific libraries to match the available memory subsystem [4]. Here, applications have their implementations altered (often in significant ways) to exploit the particular properties of the cache on which the code is executing. The Cell processor is another example of memory control, where the local memory associated with the eight compute engines is explicitly managed by the programmer. Loading and unloading of data is via DMA invocation.

While we do not have a specific approach to propose (we consider this to be an open problem), future languages that hope to be applicable to a wide variety of hybrid computing components must provide mechanisms to reason about locality (physical locality) of memory. The convenient fiction of an extremely large, flat address space is a luxury that is simply not viable.

4 Tools

Compilers and other tools play an important role in allowing developers to program in the language of their choice while obtaining reasonable performance on a hybrid system.

For example, the StreamIt project [14] recognized the value of reifying streams in an application; detecting streams statically in programs that manipulate general-purpose arrays and pointers is generally intractable. In their solution, the StreamIt project forces declaration of stream containers as filters; the number of tokens consumed, inspected, and forwarded must be declared *a priori* for each filter. However, the internal computation within each filter can be realized using a general-purpose programming language. Using this simple and relatively unburdensome approach allowed relatively sophisticated optimization of an application, including translation between time and frequency domains where such translation improved the application's performance.

At a higher level, advances in model driven architectures (MDAs) [20] have enabled developers to specify and configure applications using software-component specifications. Given a set of extant implementations for each component, MDA tools select and configure the components to satisfy primarily nonfunctional requirements and goals, such as application performance, footprint, or power.

Our vision is to leverage the approaches of the above communities to fashion tools that deploy code efficiently on hybrid architectures. Moreover, our approach includes developer annotations on existing libraries and middleware to allow legacy code bases to be used on hybrid systems to greater advantage. Such annotations can allow creation of pipeline structures and improve resource allocation on hybrid systems.

4.1 Compilers

Much work has already been invested in developing compilers for each component of a hybrid system.

While there is probably room for compiler improvements within each component, the open issues for the hybrid system exist primarily between the system's components, at the "coordination" level described in Section 3:

- How should the hybrid system's components be allocated to the kernels (distributable pieces) of an application?
- For components offering reconfigurability, how should they be configured to improve the application's performance?
- How can concurrency among the components be exploited to improve performance?

For the first problem, consider a function f that offers an API of interest to a developer. In principle, each component i of the hybrid system offers an implementation f_i with performance cost K_i; if no such implementation is available or possible for some i, then $K_i = \infty$. Given unlimited resources, the choice of implementation is basically the *shapes* problem [16], for which dynamic programming can reason about the cost of implementation choices and the cost of transforming data into and out of them.

While there is a best assignment for any given kernel of an application, the global optimization problem is not so simple, due to the limited resources available on the hybrid system. We have successfully applied nonlinear binary and integer programming methods to optimize a smaller and lower-level instance of this problem [17], and we envision applying such techniques to component allocation.

As described in Section 3, languages commonly in use for programming the components of a hybrid system are not amenable to expressing fine-grained concurrency structures such as pipelines. On a hybrid system, arrangements of the system's components into one or more pipeline structures can dramatically improve an application's performance.

We envision implementation libraries for the various components of a hybrid system, sufficiently annotated to allow compilation of pipeline structures at the coordination level. As an example of success using this approach, consider StreamIt [14] with its declarations concerning how far into a stream a filter must peek to do its job.

4.2 Debuggers

On traditional systems, debugging is usually accomplished using standard debugging tools (e.g., `gdb`). Debuggers are invaluable tools to software developers since they offer fast and complete visibility into the user's program state without perturbation. However, primarily due to the advent of multi-core processors, the current software trend is toward multi-threaded code. While this allows expression of coarse-grained parallelism, expression of concurrency as threads adds layers of complexity to the debugging process. Fortunately, debugging tools have made progress toward supporting multi-threaded applications running on traditional computing resources. While debuggers may not be able to provide perfect visibility into the multi-threaded application, due to synchronization effects and other issues, debuggers still provide a reasonably uniform view of the application executing on a multi-core traditional computer.

Hybrid systems, on the other hand, generally have no coherent view of the different computing elements or the application as a whole. Kernels of an application are distributed on components that are generally difficult to control or inspect when compared to CPUs. Furthermore, streaming approaches to deployment have been shown to be a fruitful approach for these systems, and here the latency of individual kernels can vary dramatically. This state of affairs is unacceptable if these systems are to become more widely adopted. For brevity, the discussion in this section will be limited to hybrid systems which contain CPUs and FPGAs.

The debugging effort can be categorized into two levels of abstraction: system level and kernel level. A typical debugging strategy is to isolate the system component that is producing incorrect output, and then focus the debugging efforts on the kernel(s) deployed on the misbehaving component. As non-traditional computing components grow in capacity, more of the application may be deployed on them. In this situation, the first step becomes easier while the second becomes much harder since discovering the incorrect kernel on an FPGA with limited visibility can be a time-consuming task.

For kernels deployed on FPGAs, simulation is still the most common debugging approach. Simulation, especially at the behavioral level, is an invaluable tool for checking the correctness of the implementation. Its drawback is that it is too slow for full evaluation of today's large FPGA designs, often taking days to complete a thorough simulation. As an alternative, a designer must either manually build custom debugging logic into the HDL description or utilize commercial tools to insert an embedded logic analyzer (ELA) into the design. These approaches are labor-intensive and are best suited for a seasoned hardware designer. Recently, research has been reported where the scan-

chain of newer devices can be utilized to read back the state of parts of the chip, especially in combination with LUT-based custom ELAs [22]. Using these approaches, we envision instantiating automatically-generated ELAs at the kernel boundaries to provide sufficient visibility into the FPGA for debugging while minimizing the area overhead on the FPGA. Adding special control to the clock circuitry can allow for step-by-step or multi-step debugging of the FPGA kernels. ELAs can also be added inside kernels to provide visibility of internal kernel traces. A hybrid debugging system will be a challenge to synchronize across the disparate computing components, but will afford developers a more coherent view of an applications behavior. We envision this as a vital step towards allowing debugging to proceed using familiar debugging interfaces.

4.3 Performance Monitoring

Performance is a primary goal and concern when deploying an application on a hybrid system. Developers require feedback concerning performance to understand the implications of a given allocation of application kernels to computing components. Since these computing components may have dramatically difference latencies both for computation and I/O, the standard blocking API call programming model is not very effective. Instead, authoring applications in a streaming model provides bounded memory accesses and parallelism through pipelining. Unfortunately, most legacy code is not authored in this manner, leading to the issue of library re-development.

To address this, we consider a novel form of performance monitoring to help establish producer–consumer relations among the data of a computation, with the goal of creating pipelined computations. As discussed above, language annotations can allow creation of pipeline structures, but most legacy code will lack such annotations.

Given an ordinary sequential implementation of a program, we can instrument and then observe the program's use of data to detect the presence of streams. Once found, these streams allow creation of pipeline structures by either utilizing a sophisticated compiler or though manual methods. This approach could operate as follows. With the program deployed on an ISA, all communication between components occurs through shared memory. Each memory cell is annotated by the component that assigned it. When a memory cell is referenced, the producer–consumer relation is recorded and analyzed.

If a pattern is found such that producers and consumers occur in a given direction through the program's kernels, then the associated components of the hybrid system can be arranged in a pipeline. This form of dynamic instrumentation is technically insufficient to *prove* pipeline behavior in all executions of an application, but it provides strong evidence of such behavior and the developer can make an informed decision.

Once a pipelined application is formed, either through porting legacy library calls or authoring new libraries from scratch, the developer must now monitor and analyze the performance of the hybrid application. This task shares many similarities with debugging. At the system level, it is usually easy to determine which computing component is the bottleneck by monitoring the I/O channels. However, at the individual kernel level, the same issue exists as with debugging: as more kernels are deployed on non-traditional computing components, determining which kernel is the bottleneck becomes increasingly challenging. To monitor individual kernels within non-traditional components the key is observability, and many of the computing components we are concerned with do not have convenient tools that support observability on the part of the application developer. Incorporating performance monitoring tools in non-traditional computing components is imperative for efficient application development. A recent chip that has made progress in this area is the Cell, with its built-in performance monitoring infrastructure [8].

Specifically for FPGAs, Hough et al. [12] have described a non-intrusive performance monitoring capability for soft-core processors on FPGAs that can easily be adapted for more general FPGA designs. We envision significant extensions to this work by automatically instantiating non-intrusive (or minimally intrusive) performance monitoring blocks (PMBs) to monitor kernels deployed within an FPGA. PMBs can be used to estimate kernel performance and facilitate development of robust performance understanding for the complete application. Data collected from the PMBs can be used to both calibrate and verify the applicability of analytic model.

In some circumstances, each kernel may not need a PMB. If the system is programmed using synchronous dataflow semantics, composability properties can be leveraged to reduce the instrumentation required to monitor performance. In other words, one may only need a PMB for a subset of kernels on the FPGA for sufficient observability to determine performance. Our vision is to extend the theory to a wider class of programming models, including pipelines of filters, and apply the same or similar techniques to reduce in-

strumentation requirements. Just like the techniques for debugging described above, the scan chain provides a convienent, non-competing data path for collecting the results from the PMBs in real-time or at the end of a run.

5 Conclusions

While physical construction of hybrid systems is relatively straightforward, and individual examples exist of significant performance gains on specific applications, application development is currently sufficiently difficult that it is a serious impediment to the ultimate success of this type of system. In this paper, we have articulated our collective vision of both what needs to be done to correct this and what open research problems need to be addressed. Our recommendations range from the straightforward to the complex, requiring reasonable engineering effort in some cases and new fundamental insights in others.

In spite of the need for further research, it appears clear that hybrid systems have the ability to provide dramatic benefits in the ever-present push for greater computational performance, and their use is likely to increase significantly as time progresses.

References

[1] P. Belanovic and M. Leeser. A library of parameterized floating point modules and their use. In *Proc. of 12th Int'l Conf. on Field Programmable Logic and Application*, September 2002.

[2] I. Buck et al. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, August 2004.

[3] P. Chiang and S. Riley. Using a field programmable object array (FPOA) to accelerate image processing. In *Real-Time Image Processing, Proc. of SPIE – Volume 6063*, February 2006.

[4] J. Dongarra and D. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, June 1995.

[5] M. Franklin et al. An architecture for fast processing of large unstructured data sets. In *Proc. of Int'l Conf. on Computer Design*, pages 280–287, 2004.

[6] M. Franklin et al. Auto-pipe and the X language: A pipeline design tool and description language. In *Proc. of Int'l Parallel and Distributed Processing Symp.*, April 2006.

[7] H. Fu et al. Optimizing logarithmic arithmetic on FPGAs. In *Proc. of Symp. on Field-Programmable Custom Computing Machines*, April 2007.

[8] M. Genden et al. Real-time performance monitoring and debug features of the first generation Cell processor. In *Proc. of 1st Workshop on Tools and Compilers for Hardware Acceleration*, September 2006.

[9] M. Gokhale et al. Stream-oriented FPGA computing in the Streams-C high level language. In *Proc. of IEEE Int'l Symp. on FPGAs for Custom Computing Machines*, pages 49–58, 2000.

[10] N. Govindaraju et al. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proc. of SIGMOD Int'l Conf. on Management of Data*, pages 325–336, 2006.

[11] Z. Guo et al. A quantitative analysis of the speedup factors of FPGAs over processors. In *Int'l Symp. on Field Programmable Gate Arrays*, pages 162–170, February 2004.

[12] R. Hough et al. Cycle-accurate microarchitecture performance evaluation. In *Proc. of Workshop on Introspective Architecture*, February 2006.

[13] A. Jacob et al. FPGA-accelerated seed generation in Mercury BLASTP. In *Proc. of Symp. on Field-Programmable Custom Computing Machines*, 2007.

[14] A. Lamb et al. Linear analysis and optimization of stream programs. In *Proc. of ACM Conf. on Programming Language Design and Implementation*, pages 12–25, 2003.

[15] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, May 2006.

[16] M. E. Mace and R. A. Wagner. Globally optimum selection of memory storage patterns. *Proc. Int'l Conference on Parallel Processing*, 1985.

[17] S. Padmanabhan et al. Extracting and improving microarchitecture performance on reconfigurable architectures. *Int'l Journal of Parallel Programming*, 33(2–3):115–136, June 2005.

[18] I. Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.

[19] D. Pellerin and S. Thibault. *Practical FPGA Programming in C.* Prentice Hall, 2005.

[20] D. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2), 2006.

[21] C. Sullivan et al. Using C based logic synthesis to bridge the productivity gap. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 349–354, January 2004.

[22] A. Tiwari and K. A. Tomko. Scan-chain based watchpoints for efficient run-time debugging and verification of FPGA designs. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 705–711, 2003.

[23] R. Wain et al. An overview of FPGAs and FPGA programming; Initial experiences at Daresbury. Technical report, Computational Science and Engineering Dept., CCLRC Caresbury Laboratory, November 2006.