Design Productivity for Configurable Computing*

Brent Nelson[1], Michael Wirthlin[1], Brad Hutchings[1], Peter Athanas[2], Shawn Bohner[2]
[1]Brigham Young University and [2]Virginia Tech
{nelson, wirthlin, hutch}@ee.byu.edu, {athanas, sbohner}@vt.edu

May 13, 2008

Abstract

Abstract goes here.

1 Introduction

Like design productivity for ASIC design, design productivity for FPGA design suffers from the well-known *design productivity gap* [1] — silicon densities continue to double every 1.5 to 2 years while design capabilities are growing at a much slower rate. Just to keep from falling behind in the future, design productivity must improve according to Moore's Law. However, based on the historical pace of design methods and tools for hardware design, this is highly unlikely without significant focused effort by the research community.

Productivity issues related to both software development and hardware development have been studied for many years, yet the solution for addressing the productivity issues for FPGA design has unique characteristics compared to contemporary hardware and software solutions. The focus of this paper is to propose the outlines of a research agenda to help address the challenges associated with FPGA-based configurable computing design productivity. We begin by presenting background material on the possible future of FPGA devices followed by a discussion of FPGA use models. We then provide a historical perspective by comparing and contrasting software and hardware development. Next, we propose a qualitative FPGA design productivity model, the purpose of which is to identify problem areas in configurable computing design productivity and to suggest research approaches to help improve it. We then outline a research agenda for configurable computing design productivity in the form of a proposed set of approaches, suggested by our productivity model, and which promise to significantly improve productivity.

2 Background

This section provides background information and context for the remainder of this paper and sets the stages for the introduction of a productivity model in a later section.

*This work was supported by the Defense Advanced Research Projects Agency - Information Processing Techniques Office (DARPA/IPTO) under contract FA8650-07-C7745 and administered by AFRL/RYDI.

2.1 FPGA Devices

The largest FPGA devices available today are built using 65 nm devices. Table 1 summarizes the largest devices available from Altera and Xilinx. These modern FPGAs contain a tremendous amount of logic, computation, and memory resources and can be used to perform a variety of high-performance computing and embedded computing applications.

Table 1: Current FPGA Architectures

Altera Stratix III EP3SL340		
LUTs	DSP/Mult	Memory
340K	575	17Mb
Xilinx Virtex 5 XCVLX330		
LUTs	DSP/Mult	Memory
331K	192-640	3.4Mb

The growth in density and capability of FPGAs will undoubtedly continue in the future. Table 2 suggests the resources that may become available on future FPGA devices using newer technology nodes. If FPGA density keeps pace with Moore's law, we expect the largest FPGAs in a 22 nm technology to contain almost 3 million look-up tables, several thousand dedicated multiplier/DSP blocks, and up to 100Mb of internal memory.

Table 2: Future FPGA Architectures

Technology	Year	LUTs	DSPs	Memory
65 nm	2007	340K	500	10Mb
45 nm	2010	700K	1,000	21Mb
32 nm	2013	1,400K	2,000	42Mb
22 nm	2016	2,900K	4,300	89Mb

While the density of future FPGAs will certainly increase, it is likely that the architecture of future FPGAs will also continue to evolve. As more transistors become available, it is likely that the logic and computing resources will become coarser grain and more varied to address the needs

of different markets. In addition, we expect that FPGAs will continually adopt the latest and highest speed I/O interfaces available and thus will remain on the forefront in adopting and demonstrating future I/O capability. As such, FPGAs will present a moving target to CAD tools and we believe it will become increasingly difficult to address the gap between FPGA design productivity and FPGA circuit density as a result.

2.2 FPGA Use Models

FPGAs are used in many different ways and a variety of terms (possibly overlapping) have been used to describe those uses. The terms include: ASIC replacement, reconfigurable computing, configurable computing, RC-HPC, RC-HPEC, rapid prototyping, etc. Each of these uses has unique power and performance needs and constraints and the design problems associated with them vary considerably as well.

In this work, our focus is on what we will call *Configurable Computing*. FPGA-based configurable computing is typically used for its superior power and performance compared to a CPU-based implementation of a computation. This is in spite of the increased development time (and therefore longer time to market and increased risk) associated with FPGA-based vs. CPU-based development [2]. An overarching focus in configurable computing is on standard platforms, solutions, and reuse to achieve high design productivity.

A Configurable Computing Machine or CCM is a platform consisting of one or more PC boards containing a collection of standard memories, standard I/O interfaces, processors, and FPGAs onto which a variety of applications can be mapped. These standard platforms and I/O interfaces facilitate reuse, which is important in increasing productivity for FPGA-based design. Further, such CCMs often contain large FPGAs so as to be able to accomodate whatever designs are ultimately mapped onto the CCM.

When mapping a computation onto a CCM the goal is often to simply get the design to fit into the available FPGA(s) rather than to optimize the design down to the last gate. For applications with real-time constraints, the focus further entails getting the design to meet that real-time computational requirement. However, for many CCM-based applications (which are sometimes called *computing* applications), there is no hard real-time requirement, the goal being simply to provide significant acceleration over a software-only solution. It is also important to note that a CCM may be embedded and therefore considered an HPEC (high performance embedded computing) application of FPGA-based computing. Alternatively, it may be a CPU-attached platform such as are found in desktop or HPC (high performance computing) applications. Both scenarios fit our definition of configurable computing

"ASIC Replacement" is a use model that stands in contrast to configurable computing. In this model, the focus is on cost due to volume. As such, the behavior of the FPGA is specified in great detail down to the cycle-by-cycle operation of the circuit to create the smallest custom FPGA design that will meet the performance requirements — the emphasis being on the term *custom*. ASIC Replacement applications typically entail the design of custom PC boards onto which the FPGA is placed, custom I/O interfaces, custom clocking requirements, etc.

Our focus in this paper is on design productivity for the Configurable Computing use model, rather than the ASIC Replacement use model. Although the approaches for increasing productivity detailed later in the paper may provide benefit for ASIC Replacement, that is not the focus of our work.

3 What Can We Learn From Prior Work on Software and Hardware Productivity?

Configurable Computing has mistakenly been viewed as solely a hardware design process. Because of that, much of the prior work on configurable computing tool development has been based on hardware design environments. In fact, many of the processes in contemporary configurable computing have direct counterparts in ASIC design [3]. Configurable computing shares much with software development as well, however, and it can be instructive to look at software productivity to determine how it may or may not provide insight into FPGA design productivity.

Software development environments are generally considered more mature than configurable computing design environments, and thus are good predictors of what is achievable. We believe, in fact, that configurable computing development environments are no more advanced than software practices of the 1960's. Software productivity has progressed dramatically in the past half century, and this history holds important lessons for the configurable computing community.

There have been three notable milestones, or "inflection points" in the course of software evolution that have significantly impacted software productivity. These include: (1) the introduction of standard languages and compilers that promoted platform independence and code reuse, (2) the introduction of standard software interface mechanisms such as stack frames, which in turn led to the preponderance of reusable code libraries and development tools, and (3) the addressing of human factors by providing interactive development environments as opposed to batch mode development environments along with powerful symbolic debugging tools. This last point is particularly important in that it enabled whole new types of development processes and associated tools.

As shown in Figure 1, hardware development environments are far less advanced than software environments in these three areas. Not only are some tools completely lacking on the hardware side, the offerings on the hardware side (where tools are available) are far less well developed, are more costly, and there are far fewer choices to designers.

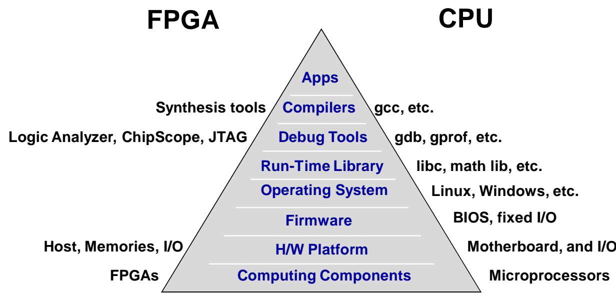

Figure 1: Hardware vs. Software System Support

Since the mid-1980s, software productivity has improved largely by taking a software engineering perspective. Improvements in this timeframe have come in four main categories:

1. *Increased Abstraction* - Moving the representation level up so that reasoning about systems at a higher level could occur resulted in orders of magnitude improvements.

2. *Reusable Artifacts* - Related to abstraction, the very nature of reusing rather than reinventing solutions was a pragmatic advance. Taking a learning perspective, reuse can be thought of as an interim stage in moving the abstraction level up. That is, the reusable components represent patterns learned and incorporated into the next language advancement.

3. *Software Process* - Recognizing that most software development was done in an ad hoc manner, concentrating on effective and efficient software development activities improved productivity by 20-40% in smaller projects to as much as 500% in very large projects.

4. *Automation* - Automating tedious tasks played an important role in software productivity increases. Once software processes were established, tools to automate and integrate the relevant tasks reduced errors and sped up software development by an estimated 30+%. A key effect of automation on the software design process was to enable interactive development environments which were shown to provide as much as a 2× productivity improvement over batch development environments [4].

Reflecting on what this means for configurable computing design productivity, three key observations can be made. First, moving the abstraction level up promises to give multiple orders of magnitude productivity improvements, Second, reuse is a key element. If experience from software can be conveyed to configurable computing development, an order of magnitude improvement in productivity may be available due to reuse. Finally, there are opportunities to improve on configurable computing design processes much like was the case with software in the 1980s. While this produced less than an order of magnitude improvement it was still multifold on large projects and as configurable computing development efforts become large, there is considerable leverage in process improvements. Improved processes will also aid in our ability to create designs which can be maintained over extended periods of time and from which derivative designs can more easily be created. And, as FPGAs are increasingly used in hybrid computing configurations, automation will help with interoperability issues in such systems.

4 A Configurable Computing Design Productivity Model

Design productivity, at its simplest, relates to how quickly a design can be completed. In actuality, there are at least three measures of design time we believe are important:

1. *The time to complete a new design.* This is the conventional definition.

2. *The time to do something new with an existing design.* This relates to the time required to modify or update the functionality of an existing design or retarget an existing design for a new technology.

3. *The rate at which a series of designs can be created.* This relates to the use of configurable computing machines as general platforms for the implementation of a sequence of designs over time.

While the first of these is usually what is considered in most discussions of design productivity, all are important to consider because all are common scenarios.

In the following sections we develop a general model for design productivity for configurable computing. We have at least three reasons for proposing such a model. First, it allows us to precisely define what we mean by design productivity. Second, it provides a vehicle to identify and discuss the most promising approaches which could be developed to improve productivity. Third, it allows us to evaluate such approaches by estimating the magnitude of impact they might have.

Models have limitations and the model we propose is no exception. It is not meant to predict the precise design time required for a given application. Rather, it is more qualitative in nature and points out what we believe to be the first-order contributors to design productivity. It also points out the interactions between the elements of design productivity and, hopefully, suggests approaches for improving productivity. This initial derivation is relatively straightforward and uses constant values for a number of its parameters. The conclusions section of this paper details a series of future experiments to provide more insight into a number of the parameters used in the model, and which will provide

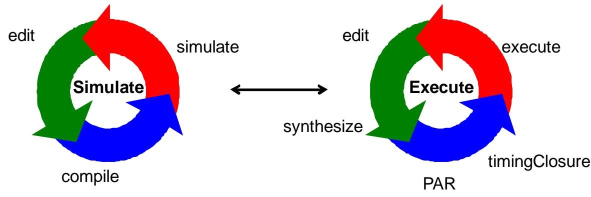

Figure 2: Dual Hardware Verification Cycles

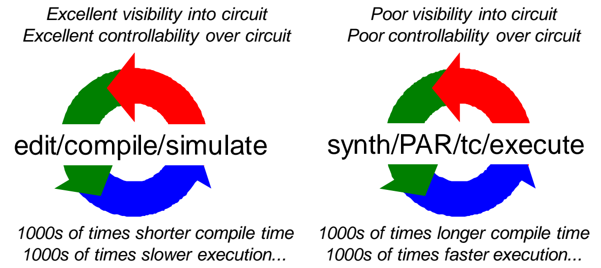

Figure 3: Characteristics of Verification Cycles

more insight into the design process and design productivity for configurable computing.

4.1 A First Model

Our first measure of design productivity (DP) is simply the rate at which hardware is developed:

$$DP = \frac{CC}{DesignTime} \tag{1}$$

Here, CC represents the *circuit complexity* of the final design, as measured in gates, LUTs, transistors, etc. In contrast, a common measure of software productivity has historically been lines of code produced per day (LOC/DAY), but such a measure, may not be the best choice for configurable computing design. The output of hardware design is hardware, a physical artifact that can be measured and that has quantifiable costs in several dimensions (silicon area, power, etc.). We thus base our configurable computing productivity measure on the result or *output* of the implementation process (the final circuit implementation) instead of on the *input* to the implementation process (the input code created by the designer which ultimately resulted in that circuit implementation).

4.1.1 Design Time

The majority of the effort required to complete a hardware design is spent in debug and verification, with values in the 70% range being common. Thus, design time for configurable computing applications depends strongly on the number of design turns required to complete the verification of the design and the ease with which those design turns can be completed.

As shown previously in Figure 1, support tools for reconfigurable computing design are much less well developed than for software. Not only are some tools completely lacking on the hardware side, the offerings on the hardware side (where tools are available) are far less well developed, are more costly, and there are far fewer choices to choose from. In particular, the greatest focus historically has been on design entry tools for configurable computing, with much less attention paid to debug and verification tools.

Further, when debugging configurable computing applications, two different design and verification cycles are required to arrive at a bug-free implementation as shown in Figure 2. The left half of Figure 2 represents simulation-based debug and consist of three steps: *edit-compile-simulate*. In later design stages, a hardware debug cycle is then added (right side of figure), which consists of the following steps: *edit-synthesize-PAR-timingClosure-execute*. In these two cycles, significant user wait times are often experienced in the synthesize-PAR-timingClosure steps. A typical design iteration may consist of one or more times through the first cycle using simulation followed by one or more times through the second cycle using hardware execution. We call a design iteration through one of these cycles a $Turn$.

Figure 3 illustrates that the two cycles have very different characteristics. While the left cycle has some similarities with software development cycles (rapid compilation), the right cycle does not. The *synthesize-PAR-timingClosure* steps in the second cycle can be painstakingly slow, greatly affecting the designer's productivity. Placement and routing of the hardware description often requires several hours to complete (overnight is not uncommon). Thus, engineers can fix, at most, about 1-2 bugs per day, assuming that each fix requires a new place/route run to verify that the bug has been corrected. Contrast this with software compilation which appears to be an instantaneous process by comparison. Further, the *execute* step of Figure 2 often requires either the use of instruments such as logic analyzers, oscilloscopes, and function generators or embedding the design under test into the finished system. As a result, the number of design debug iterations per day (turns per day) for configurable computing design is much lower than for software development, often by more than an order of magnitude.

In view of this, the total time to complete a design can be approximated as:

$$Days = \frac{Turns}{TPD} \tag{2}$$

where, $Turns$ is the total number of design iterations required to complete the design and TPD is "turns per day" (debug iterations per day).

4.1.2 Number of Turns Required to Complete a Design

The number of design turns required to generate a bug-free design ($Turns$) is dependent on the size of the input description as well as the frequency of occurence of bugs embedded in that input description. Design errors can be considered minor bugs (syntax errors, misconnected signals) or major bugs (unconsidered input combinations, misunderstood I/O protocols). Regardless, longer input descriptions present more opportunities for errors of both types. Our assumption is that design bugs are distributed uniformly throughout the design at a particular rate and thus the number of such bugs is a direct function of the complexity of the input (length of the code). We represent $Turns$ as:

$$Turns = ILOC \times \frac{Turns}{ILOC} \quad (3)$$

In this equation, $ILOC$ stands for "Input Lines of Code". While this is a commonly used measure of a program's (or design's) complexity, it should be considered as a proxy for the quantity "complexity of the design source", and could be measured in lines of input code, number of nodes in a graphical description of the circuit, etc. In contrast to CC from Equation (1) above, $ILOC$ is *not* a measure of the finished circuit's complexity, but rather a measure of the complexity of the input description created by the designer.

The term $\frac{Turns}{ILOC}$ in Equation (3) is a measure of how many debug iterations are required per $ILOC$ and is based on a simple assumption. That assumption is that design errors are distributed through the design at a certain rate, and that it will require one design turn to uncover and fix each such error. Thus, in our model $\frac{Turns}{ILOC}$ and $\frac{Bugs}{ILOC}$ are equal.

4.2 A Second Model

Combining Equations (1), (2), and (3) leads to the following design productivity equation:

$$DP = \frac{CC \times TPD}{ILOC \times \frac{Turns}{ILOC}} \quad (4)$$

This equation matches our intuition regarding design productivity in a number of ways. First, productivity is directly proportional to TPD, the number of design iterations a designer can complete in a day. Second, design productivity is proportional to the ratio $\frac{CC}{ILOC}$. One may think of this ratio as an *abstraction factor* — higher levels of design abstraction multiply a designer's efforts by producing more circuitry per line of user input. Third, Equation (4) predicts that productivity is inversely proportional to the average bug rate in code ($\frac{Turns}{ILOC}$).

4.2.1 Effect of Reuse on Design Time

A shortcoming of Equation (4) is that it fails to capture the effect of reuse on design productivity. That is, design productivity improves when the designer is able to reuse preexisting design pieces, requiring less original design. The most common example of reuse is a designer using a library cell for a circuit function instead of creating that function from scratch, but other forms of reuse are important. For example, standard hardware platforms (CCMs) consisting of FPGAs, memories, and I/O interfaces are one such form. To simplify the following discussion, we cast it in terms of library cell reuse, but the concepts are equally applicable to all forms of reuse as will be described in a later section.

Reuse can simply be modelled as reducing the number of lines of code that the designer must write from scratch. However, using a cell from a library often requires circuitry be designed to interface that cell to the rest of the designer's circuit. $ILOC$ (the code the user must create) can thus be modelled as consisting of two parts, the new part of the design that must be created from scratch and the interface code that must be created for the reused portions:

$$ILOC = New_{ILOC} + Interface_{ILOC} \quad (5)$$

It is useful to rewrite this equation into a form where the amount of reuse is explicitly represented, along with the overhead associated with that reuse:

$$ILOC = ILOC_0 \times [(1 - R) + (O \times R)] \quad (6)$$

In this equation, $ILOC_0$ is the amount of code originally required to describe the circuit without the benefit of any reuse (the amount of code required to create it entirely from scratch). R is the *fraction* of the design satisfied by reusing previously-design circuitry. In light of this, the user must only create $ILOC_0 \times (1 - R)$ lines of new design code.

Reuse is not free, however, and O represents the *overhead* of that reuse. It is expressed as a percentage of R and represents lines of new code that the designer must create to interface the reused circuitry to the rest of the design. As a concrete example, consider a design where $ILOC_0 = 100$, $R = 40\%$, and $O = 20\%$. Without the benefit of reuse, this would require the designer to write 100 lines of code. With reuse, the user would save 40 lines of code due to reuse but would have to create 8 additional lines of code to interface that element to the rest of the circuit. The total lines of code created by the user would thus be: $100 \times [0.6 + 0.20 \times 0.4] = 68$, representing a 32% savings over coding the design entirely from scratch. Thus, the reuse overhead 'O' has the effect of lessening the benefit of reuse, and if it is too high there may be no realizable benefit, especially considering the perceived risk of using 3rd party IP.

4.3 A Final Model

Substituting Equation (6) into Equation (4) gives the following final equation for design productivity.

$$DP = \frac{CC \times TPD}{ILOC_0 \times [(1 - R) + (O \times R)] \times \frac{Turns}{ILOC}} \quad (7)$$

Equation (7) contains a number of variables which can be affected by designers and researchers, and thus change a designer's productivity. These relate broadly to the areas of reuse, turns per day, and level of design abstraction used, which are discussed in the remainder of the paper.

5 Reuse

It is well known that reuse of software has been a significant factor in improving software design productivity [5, 6]. Today's software systems are typically created by reusing software libraries, integrating reusable components, and dynamically integrating autonomous executables (COM, CORBA, etc.). Very large and complex software services can be created by exploiting the many reusable software components and service oriented architectures. The successful exploitation of software reuse has led to significant improvements in productivity, higher quality code, fewer bugs, and lower software maintenance costs[7].

While these relatively newer forms of reuse have provided remarkable improvements in productivity, software systems have exploited reuse of system infrastructure for many years. For example, the simplest "Hello World" program written in any language or platform involves a tremendous amount of code reuse. In order to run the program, reusable firmware, operating system, and run-time libraries (i.e. `libc`) are necessary. Without such reusable system software, the creating of a "Hello World" program would be very time consuming and tedious.

Reuse within hardware systems, however, has significantly lagged behind that of software. While there is great interest in exploiting reuse for hardware design, the risk associated with reusing 3rd party circuits and the technical challenges of integrating "reusable" hardware circuits has inhibited the widespread adoption of reuse methods. One study [8] suggested that if the time required to reuse a component was greater than 30% of the time required to design the component from scratch, design reuse would fail (designers would choose not to reuse). The risk and cost of hardware reuse must be reduced before hardware reuse is widely used.

While hardware reuse is difficult, the potential improvements in productivity are significant [9]. For example, if 80% of a hardware design is created by reusing existing hardware (i.e. R=0.8) and the effort to integrate reusable hardware is 15% (i.e. O=0.15) the effort of creating the hardware from scratch then hardware design productivity will increase by a factor of 3 (see Equation 6). Achieving this levels of reuse today and at such a low cost is difficult. However, the improvements in software reuse over the last four decades suggests that significant improvements in hardware reuse can be made with appropriate technology advancement and community cooperation.

Like software, there are many different ways to exploit reuse during the design and deployment of a hardware system. The most obvious form of hardware reuse involves the use of reusable hardware component libraries, but there are many forms of reuse including:

- Library cell reuse - this is what most think of when reuse is proposed and is the use of cells from a standard library which perform a specific function (an FFT, for example).

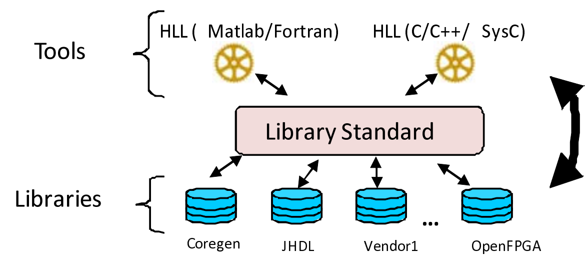

Figure 4: Library Reuse Organization

- Retargeting reuse - the porting of designs between devices from different manufacturers or even between devices from a single manufacturer.
- Design pattern reuse - the reuse of structures such as pipelining or bit-serial arithmetic in the creation of a design [10].
- Architecture reuse - meta-architectures are architectures layered on top of traditional reconfigurable fabrics to facilitate reuse.
- Platform reuse - the use of standard CCM-like platforms with FPGAs, memories, and I/O capabilities.
- Interface reuse - the use of standard I/O connections to alleviate the designer creating custom interconnect for each application.

5.1 Hardware Reuse Approaches

The most common and direct form of hardware reuse is the reuse of hardware components. Predefined hardware circuits (otherwise known as "intellectual property" or IP cores) are created and verified and then later inserted in a larger hardware circuit. While such reuse occurs frequently within an organization, reuse between organizations and third-party developers is limited. In addition, it is difficult to reuse hardware components over time — they become obsolete and reusing today's modules on tomorrow's devices is problematic. One problem is the lack of standards – hardware circuits are developed in a variety of tools and incompatible languages that inhibit the reuse of the circuit in new environments and design flows. As shown in Figure 4, standards for describing and representing reusable hardware will enable a *variety* of high level tools to take advantage of a *variety* of cell libraries developed within different tools [11].

Hardware circuits are often difficult to reuse because of the complexity of the circuit interface. Before a hardware component can be reused, all details about the component interface must be understood. Further, custom interfaces or "wrappers" are frequently created to convert the interface of the circuit into a form that is usable by the new system. The time required to create such wrappers and the errors introduced into a circuit because of such wrappers discourages the benefit of reuse. One way to address this problem is to develop techniques for *synthesizing* these custom in-

terfaces. Automatically synthesizing interfaces for reusable circuit components will simplify the task of integrating a component and reduce the errors and risk associated with third-party circuits.

Another form of reuse that receives less attention is the ability to "retarget" or reuse hardware circuits on multiple FPGAs or CCM platforms. Most design teams develop hardware circuits for a specific vendor or CCM platform. With current tools and methods, it is quite difficult to retarget the design to a device or platform from a different vendor. Vendors provide limited help as there is no incentive for vendors to collaborate and create interoperable tools.

There are a variety of ways to improve the retargetability for CCM platforms. One approach involves a two-level compilation flow [12]. The first step in this flow compiles the behavioral description onto a predefined platform-independent "meta-architecture". The second step converts this platform-independent representation into a platform- and FPGA-specific representation. Retargeting involves a recompilation at the lower-level platform specific compilation step.

5.2 Reuse Design Process

To exploit the benefits of reuse effectively during hardware design, the design process must undergo significant changes. Consider a design process where 80% or more of the circuit involves reuse of existing components. Most of the development time in this environment will involve the integration, interconnction, and verification of existing circuits rather than the development of new circuits. The tools, methods, and processes used to perform these tasks will be very different from the tools and techniques we use today.

For example, an interesting benefit of reusable software components has been the ability of non-programmers to create complex software applications. Traditional programmers create reusable software components and domain specific experts with limited programming skills are able to combine these components into useful and complex sofware applications[13]. With suitable investment in reuse methods, we believe that successful hardware reuse will result in major improvements in productivity and facilitate the development of CCM systems by non-hardware domain experts.

6 Abstraction

Raising the level of abstraction has been a consistent way of improving productivity for both software and hardware. Programming for software systems has undergone a transition between many different levels of abstraction including machine code, assembly language, procedural programming languages, etc. Hardware design has also adopted new abstractions to reduce design detail. Representative hardware abstractions include design with individual transistors, logic gate design with schematics, and register transfer level (RTL) design.

At its core, raising the level of abstraction means reducing the number of details that must be specified by the designer. For example, high level programming languages and compilers eliminated the need for software programmers to do explicit register allocation, instruction scheduling, and stack frame manipulation. This eliminated whole classes of errors from programs. A related aspect of abstraction is the introduction of domain-specific abstractions, which in essence moves the program description closer to the problem domain thereby reducing conceptual gap between the program and the conceptual view of the problem. For example, Matlab is a language specifically for matrix manipulations — its data types, operators, and control constructs are all tailored for matrix computations.

Many new hardware design tools are being introduced for FPGAs with higher level design abstractions. Most of these new abstractions are based on existing programming languages such as C. While these tools certainly provide improvements in productivity for those that learn and use them, they have not significantly improved design productivity for the general design community. One reason for this is that such languages are essentially HDLs — the skill set for the designer with such languages still requires hardware capabilities. They may provide higher levels of abstraction than VHDL or Verilog and thus remove details from the designer's view, but they still require an understanding of clocking, scheduling, pipelining, and other digital systems design concepts. Another reason is that these languages, while clothed in the robes of familiar programming languages such as C, have unique semantics. A familiarity with the base language such as C may actually be a handicap when trying to learn these new semantics. Third, many of these abstractions are based on inherently sequential languages. The sequential nature of these languages limits the ability to specify and to exploit the massive parallelism available in hardware circuits [14].

While these recent tools and languages are a step in the right direction, we believe that they are insufficient for moving hardware design to a significantly new level of design productivity. We advocate a number of ideas for exploiting the benefits of higher-level abstractions.

6.1 Parallel Languages

The recent explosion of interest in multi-core processors is due to a recognition that improvements in uniprocessor performance are ending. Multi-core processors, however, are more difficult to program than traditional uniprocessors. Most programmers are taught to program using sequential languages and compilers struggle to exploit sufficient parallelism from such sequential descriptions. To address this issue, there is tremendous interest in parallel programming languages and compiler tools for targeting multicore architectures.

We believe that we have a unique opportunity to exploit this growing trend. We advocate the adoption of standard, concurrent programming languages for hardware design rather than adopting sequential programming languages and adding non-standard semantic extensions. The use of con-

current programming languages will facilitate the extraction of concurrency for hardware. Further, standard concurrent languages will lead to more platform independent descriptions of algorithms that can be targeted to either hardware or software.

6.2 User Directed Compilation and Synthesis

The feasible design space of all possible implementations for a given algorithm on hardware is immense. One of the great challenges of compilation and synthesis technology is to manage this design space appropriately. While hardware compiler tools can do a good job in identifying potential solutions, the translation of an algorithm into an implementation should not be hidden from all users. Synthesis technologies should provide appropriate feedback on potential implementations and allow a designer to influence the synthesis process. Appropriate designer influence may significantly improve both the quality of result and synthesis time. The challenge is to provide such user influence without requiring the user to understand unnecessary details.

6.3 Two-level Compilation

Synthesizing computing circuits onto arbitrary hardware is much more difficult than compiling a program onto a sequential processor. The tasks must be assigned to resources and scheduled in time. As described in Section 5.1, a two-level compilation strategy may assist the compiler and synthesis tools during this process. With this approach, standard "meta-architectures" are defined that represent more coarse grain architectures than FPGAs and provide a higher level abstraction than low-level LUTs and wires [12]. The compilation and synthesis process can be simplified by compiling to this meta architecture level using higher level abstraction tools and then using low-level device specific tools to generate actual computing circuits. Further, a two-level compilation strategy will lead to greater portability and reusability by more easily allowing computations compiled to a meta-architecture to be retargeted to other low-level device architectures.

6.4 Architectural Support for Compilation

Past and current practice has been to develop architectures and CAD tools as if each was irrelevant to the other. One observation from this is that progress in high level language compilation for FPGAs has been sparse and incremental. While "C to gates" compilers have progressed over the last fifteen years, that progress has been dwarfed by Moore's law increases in circuit density. Another observation is that in light of the market forces which drive FPGA device development, CAD developers have little or no say over architectural developments. This is especially true for configurable computing, which has different needs and goals compared to the ASIC replacement use model as discussed in Section 2.2. While the market has yet to bless any one, emerging alternative configurable computing architectures and their associated programming tools suggest that productivity gains can be obtained when the device and programming model are jointly developed. We believe that we cannot continue to operate as if architecture plays no role in enabling higher levels of abstraction for configurable computing development. Future configurable computing device architectures must consider compilation from high level languages.

7 Turns and Turns Per Day

Turns and turns per day relate to debug and verification tasks in the design process. They are both key components of Equation 7 — the equation predicts that changes in either one will directly impact design productivity.

Current configurable computing debug methods are primitive, and more closely resemble software practices from the 1960's than modern development practices. We rely too much on simulation rather than on target machine execution for much of the development process. When we do move to hardware execution the process is essentially a batch process rather than an interactive one. CAD tool runtimes to prepare bitstreams for test are measured in hours rather than seconds, and hardware debug tools often provide little more than the equivalent of raw data dumps.

The number of turns per day currently achieved for configurable computing is often in the range of $1 - 3$. As noted earlier, TPD for software is at least one order of magnitude higher. A $10\times$ improvement in TPD seems achievable, but to do so will require a change in design methodology, tools, and platform support.

However, there is much more to TPD and design productivity than the equation is able to capture. The long CAD tool run times associated with configurable computing desgn force designers to multi-task while they wait. The result is a much less productive debug environment as the designer continually has to re-focus his efforts, losing concentration and productivity in the process. The difficulty and long wait times associated with doing a design turn further lead the designer to attempt to minimize the number of experiments to perform. While this may seem to be a good thing (the designer is attempting to minimize $Turns$), it can be counterproductive by discouraging "what-if" experiments and small incremental-change runs to better understand the system's incorrect performance. The net effect may be to ultimately increase $Turns$.

Decades ago, the case was made for *interactive* vs. *batch* software development tools, the argument being that a $2\times$ productivity improvement alone may be the result of such a change [4]. We believe the key, then, to increasing design productivity via $Turns$ and TPD is to convert the current batch-oriented development environment to an interactive one.

7.1 Rapid Prototyping for Increasing TPD

One approach for increasing TPD is rapid prototyping. In [9] the author argues that there are three different kinds of workflows which must be addressed for increasing HPC productivity - the researcher workflow, the enterprise workflow, and the production workflow. In the first of these, the

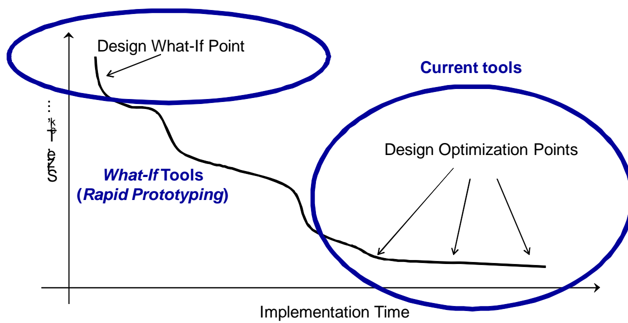

Figure 5: Rapid Prototyping vs. Conventional CAD Tools

researcher workflow, the emphasis is on knowledge discovery and rapid design iterations. Current design environments provide little or no support for this but seem to be optimized for the later workflows (highly optimized results, long batch-oriented design turns).

It may seem strange to consider the use of rapid prototyping tools for FPGA-based design since FPGAs themselves are often used as rapid prototyping vehicles for integrated circuits. Figure 5 shows the reasoning. Typical design tools provide design optimization points far on the right of the curve and give the designer limited flexibility to trade off quality of solution for drastically reduced implementation time as a way of rapidly performing *what-if* experiments (left side of curve). Such a tool would be quite different from the CAD tools currently in use, and would have different constraints which would guide its creation.

The use of rapid prototyping-style tools would not only reduce design iteration times, it would also convert the current batch debug environment to a more interactive one.

7.2 Debug Tools for Increasing TPD

A more interactive development environment without new tools is an improvement over the state of the art but much more can be done with sophisticated debug tools. A wide variety of debug tools have been created over the years. While a full review of them is beyond the scope of this paper, [15] provides some historical context. As discussed in [15], early debugging tools (such as those provided with the Splash2 and Teramac systems) exploited FPGA state readback to provide cycle-by-cycle access into the computation's state which could be displayed in GUIs and back-annotated into the design source. Coupled with clock control, this provided a simple yet powerful debug capability.

Our experience with both systems followed by our work with the JHDL development system [16, 17, 18, 15] indicates the tremendous value of an interactive debug environment, constructed from the start as an integral part of a CAD tool flow combined with a properly designed CCM architecture. When debug is built in from the outset features common for decades in software development processes become easily incorporated into a design flow such as full source-level debug with full visibility into the executing computation (both for state values as well as combinational values) [15], single step capability, breakpoints, checkpointing [19], context switching (sharing of the configurable computing platform) [19], and the ability to alter state values for *what-if* purposes[20, 21]. Further capabilities have been demonstrated for performing on-the-fly configuration modification to add debug capabilities to the design as the debug process unfolds [22, 20, 23] While capabilities for a number of these features have been included in some excellent commercial offerings, there is no standardization of capabilities within either CAD tools or CCM platforms and their inclusion may be uncertain with new platforms.

Because FPGA hardware is reconfigurable, any required circuitry for these debugging tasks can be automatically embedded into a design, much like the use of "-g" for software debugging. Further, any inserted debug circuitry can be removed when the application is ready for deployment.

As long as designers are required to manually modify their designs in order to embed debug circuitry to achieve these capabilities, these powerful debug techniques may go unused. The best way to overcome these problems is to automate the process of synthesizing and embedding debug circuitry into user circuitry, a task best performed directly by the CAD tool environment.

The widespread availability of these capabilities will reduce reliance on external test equipment (scopes, logic analyzers) and will allow the user to spend less time per design turn writing testbenches, capture/conversion/formatting/viewing tools for results, etc. Also, the turns will be more productive as outlined above

8 Conclusions and Future Work

Design productivity for configurable computing suffers from the well-known *design productivity gap*. While sharing characteristics with both hardware and software development, configurable computing has its own unique needs and opportunities that are not adequately addressed by existing FPGA design tools.

We have proposed a productivity model that exposes three key contributors to design productivity. These include: (1) exploiting reuse at multiple levels, (2) exploiting the benefits of higher design abstractions, and (3) providing for a more interactive verification environmnet by increasing turns per day. These three contributors define the essential elements of a configurable computing research agenda. They are inter-related and advances in all of them are necessary to significantly impact design productivity. For example, the use of high level languages instead of HDLs for design would significantly increase simulation speed, thereby increasing turns per day. Similarly, describing designs in a high level language can promote reuse by making designs more portable.

As discussed in Section 3, software has experienced orders of magnitude improvement in productivity over the

past 50 years. Similar productivity improvements are possible for configurable computing provided the research community makes substantive advances in all three of areas of this research agenda.

References

[1] Alan Allan, Don Edenfeld, Jr. William H. Joyner, Andrew B. Kahng, Mike Rodgers, and Yervant Zorian, "2001 technology roadmap for semiconductors," *IEEE Computer*, vol. 35, no. 1, pp. 42–53, 2002.

[2] Scott Hauck and Andre DeHon ed., *Reconfigurable Computing: The Theory and Practce of FPGA-Based Computing*, Morgan Kauffman, 2007, Chapter 21.

[3] B. Hutchings and B. Nelson, "Using General-Purpose Programming Languages for FPGA Design," in *Proceedings of the 37th Design Automation Conference*, June 2000, pp. 561–566.

[4] Harr J., "Programming experience for the number 1 electronic switching system," in *Spring Joint Computer Conference*, 1969, Cited in Chapters 8 and 12 of [24].

[5] Jeffrey S. Poulin, *Measuring Software Reuse: Principles, Practices, and Economic Models*, Addison Wesley, 1997.

[6] Ivar Jacobson, Martin Griss, and Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, ACM Press, 1997.

[7] Will Tracz, *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*, Addison Wesley, 1995.

[8] Emil Girczyc and Steve Carlson, "Increasing design quality and engineering productivity through design reuse," in *Proceedings of the ACM IEEE Design Automation Conference (DAC)*, 1993.

[9] Jeremy Kepner, "HPC Productivity: An Overarching View," *International Journal of High Performance Computing Applications*, vol. 18, no. 4, pp. 393–397, 2004.

[10] DeHon DeHon, Joshua Adams, Michael DeLorimier, Nachiket Kapre, Yuki Matsuda, Helia Naeimi, Michael Vanier, and Michael Wrighton, "Design patterns for reconfigurable computing," in *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, 2004, pp. 13–23, IEEE Computer Society.

[11] M. Wirthlin, D. Poznanovic, P. Sundararajan, A. Coppola, D. Pellerin, W. Najjar, R. Bruce, M. Babst, O. Pritchard, P. Palazzari, and G. Kuzmanov, "Openfpga corelib core library interoperability effort," in *Proceedings of the 2007 Reconfigurable Systems Summer Institute*, 2007.

[12] D. Campbell, D. Cottel, R. Judd, K. MacKenzie, and M. Richards, "The morphware stable interface: A software framework for polymorphous computing architectures," in *Seventh Annual Workshop on High Performance Embedded Computing*, 2003.

[13] Clemens Szyperski, Dominik Gruntz, and Stephan Murer, *Component Software: Beyond Object-Oriented Programming*, ACM PRess, 2002.

[14] Stephen A. Edwards, "The challenges of synthesizing hardware from C-like languages," in *IEEE Design & Test of Computers*. 2006, p. 375, IEEE.

[15] Brent E. Nelson, "The Mythical CCM: In Search of Usable (and Resuable) FPGA-Based General Computing Machines," in *Proceedings of 17th IEEE Conference on Application-Specific Systems, Architectures, and Processors*, September 2006.

[16] P. Bellows and B. L. Hutchings, "JHDL - an HDL for reconfigurable systems," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. M. Arnold and K. L. Pocek, Eds., Napa, CA, Apr. 1998, pp. 175–184.

[17] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting, "A cad suite for high-performance fpga design," in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, K. L. Pocek and J. M. Arnold, Eds., Napa, CA, April 1999, IEEE Computer Society, p. n/a, IEEE.

[18] Scott Hauck and Andre DeHon ed., *Reconfigurable Computing: The Theory and Practce of FPGA-Based Computing*, Morgan Kauffman, 2007, Chapter 12.

[19] Wesley J. Landaker, Michael J. Wirthlin, and Brad L. Hutchings, "Multitasking hardware on the SLAAC1-V reconfigurable computing system," in *Proceedings of the 12th International Workshop on Field Programmable Logic and Applications*. September 2002, vol. 2438 of *Lecture Notes in Computer Science*, p. 806, Springer-Verlag.

[20] P. Graham, B. Nelson, and B. Hutchings, "Instrumenting Bitstreams for Debugging FPGA Circuits," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'2001)*, April 2001.

[21] Paul S. Graham, *Logical Hardware Debuggers for FPGA-Based Systems*, Ph.D. thesis, Brigham Young University, 2001.

[22] P. Graham, B. Hutchings, and B. Nelson, "Improving the FPGA Design Process Through Determining and Applying Logical-to-Physical Design Mappings," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'2000)*, April 2000.

[23] T. Wheeler, P. Graham, B. Nelson, and B. Hutchings, "Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification," in *Proceedings of the 11th International Workshop on Field Programmable Logic and Applications*. August/September 2001, vol. 2147 of *Lecture Notes in Computer Science*, pp. 483–492, Springer-Verlag.

[24] Frederic P. Brooks Jr., *The Mythical Man-Month*, Addison Wesley, 1995.