An Independent Evaluation of:

The AutoESL AutoPilot High-Level Synthesis Tool

By the staff of
Berkeley Design Technology, Inc.

OVERVIEW

Interest in high-level synthesis tools for FPGAs is intensifying as FPGAs and their applications grow larger and more complex. Prospective users want to understand how well high-level synthesis tools work, both in terms of usability and quality of results. To meet this need, BDTI launched the BDTI High-Level Synthesis Tool Certification Program™. This program evaluates high-level synthesis tools used to implement demanding digital signal processing applications on FPGAs.

This white paper presents detailed results of BDTI's analysis of the AutoESL AutoPilot high-level synthesis tool used in conjunction with Xilinx RTL tools to target a Xilinx Spartan-3A DSP 3400 FPGA. We discuss the ease of use, productivity, and quality of results obtained using AutoPilot compared to implementing the same application on a DSP processor. We also compare AutoPilot quality of results vs. traditional RTL FPGA design.

Our findings will be surprising to many—and may indicate a major shift on the horizon for FPGA and DSP processor users.

Contents

Introduction

AutoESL Design Technologies, Inc., ("AutoESL") is a high-level synthesis tool company founded in 2006 and headquartered in Cupertino, California. The company unveiled its AutoPilot high-level synthesis tool at the Design Automation Conference in 2009. AutoPilot is based on tools and techniques developed at UCLA and licensed exclusively to AutoESL.

AutoPilot takes as its input a C, C++ or SystemC description of functionality at a high level of abstraction. It then generates a device-specific Verilog or VHDL register-transfer-level (RTL) description of a hardware implementation targeting an FPGA (for example, from Altera or Xilinx) or ASIC. This eliminates the time-consuming and error-prone step of manually creating the RTL implementation. According to AutoESL, AutoPilot also generates a cycle-accurate SystemC simulation model for the synthesized results.

High-level synthesis tools, such as AutoPilot, that target FPGAs are typically of interest to two classes of prospective users: current FPGA users who want to improve their productivity and gain design portability, and processor users who are considering switching to an FPGA to achieve superior performance or cost/performance for computationally demanding applications. There are good reasons for considering such a switch—BDTI's report, *FPGAs for*

DSP, includes benchmark results showing that FPGAs can achieve 100X higher performance and 30X better cost-performance than DSP processors in some highly parallel DSP applications. Yet FPGAs are not widely used as processing engines in these applications, primarily due to development challenges—using traditional design techniques, it takes much longer to develop an application on a FPGA than on a DSP processor. In addition, RTL FPGA design requires different skills than those required for DSP processor software development, and most DSP engineers don't have the necessary RTL hardware design expertise.

High-level synthesis tools have the potential to overcome these challenges and bring the performance of FPGAs to a much wider range of users, but such tools are often met with skepticism. In part, this skepticism is due to the common notion that software cannot produce results that are as good as what a skilled engineer can produce. And in part, the skepticism is due to the fact that high-level synthesis tools have been around for a long time but have never achieved widespread use.

Opposing this skepticism is a growing body of anecdotal evidence suggesting that some modern high-level synthesis tools, such as AutoPilot, are very effective, both in terms of usability and quality of results. Given this conflicting information, how is a prospective user to judge whether a high-level synthesis tool is worth considering?

In 2009, BDTI created the BDTI High-Level Synthesis Tool Certification Program to fill this gap, with the goal of providing objective, credible data and analysis to enable potential users of high-level synthesis tools for FPGAs to quickly understand the capabilities and limitations of these tools.

BDTI has used this methodology to evaluate AutoESL's AutoPilot tool used in conjunction with Xilinx's ISE and EDK tool chain targeting a Xilinx Spartan-3A DSP 3400 FPGA. This white paper summarizes the results of BDTI's evaluation, and is based on BDTI's in-depth, independent assessment of AutoPilot augmented with the results of detailed interviews of AutoPilot users conducted by BDTI. It includes several results that will be surprising to many, and that may indicate that the time for high-level synthesis tools for FPGAs has finally arrived.

About BDTI

Berkeley Design Technology, Inc. (BDTI) is an independent technology analysis, consulting, and engineering services company headquartered in Oakland, California. Founded in 1991, BDTI is widely known for its highly regarded, independent performance analysis of processing platforms for embedded applications. Its six suites of chip benchmarks have been licensed for use with nearly 100 processing platforms, from MCUs to FPGAs. Further information about the company and benchmark results are available at www.BDTI.com.

Design Using AutoPilot and Xilinx ISE

Figure 1 shows the design flow used in BDTI's evaluation of the AutoESL AutoPilot high-level synthesis tool. AutoPilot is used to compile C code and

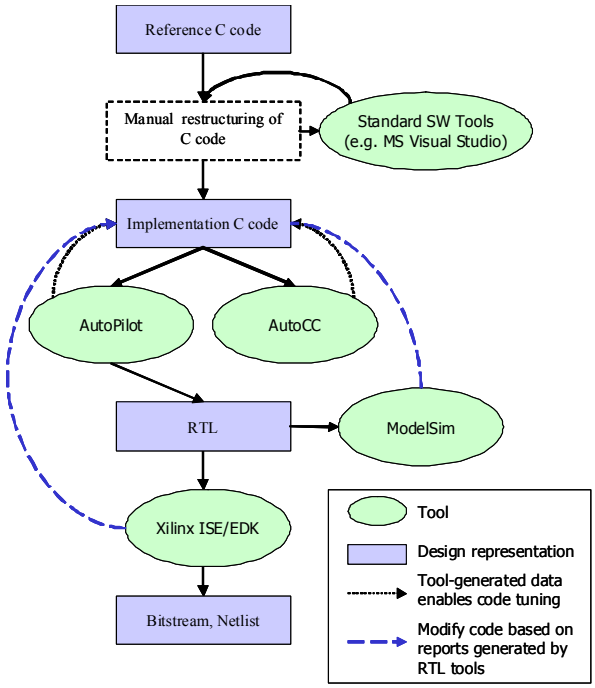

FIGURE 1. Design Flow Using the AutoPilot High-Level Synthesis Tool with Xilinx "ISE" RTL Tools

generate an RTL implementation, and Xilinx's RTL tools (which include the ISE tool chain and the Embedded Developer's Kit, or EDK) are used to transform that RTL implementation into a complete FPGA implementation in the form of a bitstream for programming a specific FPGA. AutoPilot offers a significant advantage over hand-coded RTL in terms of retargetability; the tool enables users to migrate from one FPGA to another (or even to an ASIC design) without manually rewriting their RTL.

For this project, BDTI could have limited the evaluation to the use of AutoPilot alone, ignoring the RTL-to-bitstream portion of the design flow. We believe, however, that it is important for potential users to understand how difficult (or easy) it is to get

from the high-level application description all the way to an FPGA implementation—which requires the Xilinx RTL tools in addition to the high-level synthesis tool. For this reason, BDTI evaluated the *entire* implementation process shown in Figure 1—not just the C-to-RTL portion, but also the Xilinx RTL tool chain.

When starting with a C language description of required functionality, the first step in implementing an application on any hardware target is often to restructure the input reference C code to represent an architecture that is suitable for hardware implementation. On a DSP, for example, it may be appropriate to rearrange the application's control flow so that intermediate data always fits in cache. On an FPGA, restructuring often provides a more parallel, FPGA-friendly representation of the application.

Restructuring can sometimes enable improvements of several orders of magnitude in speed and/or resource utilization. If the original C source code is not appropriate for the hardware target, restructuring is required in order to obtain an efficient implementation. For example, a video application may pass entire video frames of intermediate data between algorithm blocks. A straightforward C implementation of the application may require large frame buffers that do not fit in an FPGA, but typically the code can be restructured to enable much smaller buffers. By appropriately streaming the dataflow between algorithm blocks, the application may be able to buffer just a few scan lines—or even just a few pixels—rather than buffering entire frames.

In some cases, applications may require the use of external memory (for example, if a video application must store entire frames and can't be restructured to use scan lines or pixels). In this case, the restructuring process requires the addition of code that supports the use of an external memory interface.

The effort required to restructure the application depends on the specifics of the application and the hardware being targeted, as well as the capabilities of the tools used.

AutoPilot, like other current high-level synthesis tools, does not handle restructuring automatically. Instead, the restructuring is typically done by hand. In fact, the restructuring can be done entirely independently of AutoPilot (in our evaluation, for example, we used Microsoft Visual Studio for restructuring and verifying the C code). Compared to hand-written RTL, where restructuring and language translation are performed as a single combined step, restructuring entirely in C is easier and less error-prone.

The reference C code may also need to be modified to take advantage of the FPGA's ability to implement arbitrary-precision data paths and other features. AutoPilot supports arbitrary-width data types and also automates synthesis of a variety of interfaces between modules, including interfaces between various buses and third-party intellectual property modules such as the Xilinx multi-port memory controller (MPMC).

The AutoPilot tool chain includes AutoCC, which compiles the C code for execution on the workstation and handles FPGA-oriented modifications such as arbitrary-width data types. AutoCC doesn't generate RTL; instead, it provides a quick check of the functionality of the FPGA-oriented C code. This intermediate verification capability provides a significant advantage over verifying hand-written RTL, since RTL simulation is orders of magnitude slower. In our evaluation, for example, the C simulation typically required 15-30 seconds, while RTL simulation time varied from hours to days, depending on the workload under consideration. However, C simulation does not simulate the full hardware features (such as pipelining and scheduling), so RTL simulation is still needed for hardware verification.

The functionally verified C code can then be optimized for better performance or efficiency. This step is primarily accomplished using AutoPilot synthesis directives that can be either embedded in the C code as pragmas or placed in AutoPilot scripts as commands. Because AutoPilot generates RTL very quickly, the user can explore a variety of design strategies that would be prohibitively time-consuming to explore using RTL.

AutoPilot compiles the optimized C code and generates an RTL implementation by converting each function call in the C code into an RTL module. For the moderately complex workloads used in this project, that process typically took less than 30 seconds.

In addition to generating RTL, AutoPilot also generates reports that estimate the FPGA resource utilization, latency, and throughput of the RTL implementation. The reports include a breakdown by individual functions and loops in the C source code, allowing users to pinpoint areas for improvement and fine-tune synthesis directives or C code in response.

Once the user is satisfied with the performance and efficiency estimated by AutoPilot, the process switches over to more traditional RTL tools. An

RTL simulator such as Mentor Graphics' ModelSim can be used to functionally verify the AutoPilot-generated RTL. Alternatively, AutoPilot can generate a cycle-accurate SystemC RTL model with all the necessary test bench related files and scripts based on the C language reference test bench which then can be used for a faster functional verification than that achieved with ModelSim.

Finally, Xilinx's RTL tools (ISE and EDK) are used to take the RTL generated by AutoPilot as input, perform synthesis and place-and-route tasks, report the exact resource utilization of the implementation and alert the user to any timing closure issues. The user may then further tune the C source code and repeat the RTL generation if needed.

The transition point from AutoPilot to the Xilinx tools is smoothed, to some extent, by AutoPilot's ability to automatically generate scripts and constraints for running synthesis and place-and-route steps, which would be quite time-consuming if done manually. AutoPilot does not, however, provide a unified "cockpit" from which the entire C-to-FPGA implementation process can be executed. For example, AutoPilot does not generate scripts to transfer RTL and netlist outputs to appropriate folders for the Xilinx EDK tools; this must be done manually. It's a trivial process once the user understands exactly what needs to be done—but understanding what needs to be done requires a high level of familiarity with the Xilinx tools, including their complex directory structure requirements. AutoPilot also does not provide a script to download the bitstream onto the FPGA; this requires working with the Xilinx tools. In other words, while AutoPilot eliminates the labor-intensive (and error-prone) process of hand-coding in RTL, it does not entirely insulate the user from having to learn and use the RTL tools.

The BDTI High-Level Synthesis Tool Certification Program™

BDTI evaluates high-level synthesis tools (including the underlying RTL tools) using two well-defined sample applications, or "workloads" (described briefly below, and in more detail in Appendix A). These applications (described in the next section) are representative of typical digital signal processing applications for FPGAs. The two applications are implemented using several approaches. First, a given workload is implemented on the target FPGA using the high-level synthesis tool in conjunction with the Xilinx RTL tools. The same workload is then implemented on the same FPGA using a traditional RTL-based approach (for the BDTI DQPSK Receiver Workload), or on a DSP processor using its associated development tools (for the BDTI Optical Flow Workload). In this manner, BDTI is able to compare the quality of results and productivity levels associated with using various design flows.

The two workloads have been chosen to be broadly representative of the types of embedded computing applications that electronic system designers implement using FPGAs, and as such, they are inherently well suited for FPGAs. This is an important point to keep in mind. There are many important applications (such as high-definition audio codecs) that don't require the computational performance levels or data rates of the workloads used here, and that require much more complex algorithms. Such applications may yield very different results than those reported in this analysis.

Evaluation Workloads

The two workloads used in BDTI's evaluation are the BDTI Optical Flow Workload™ and the BDTI DQPSK Receiver Workload™.

The term "optical flow" (or "optic flow") refers to a class of video processing algorithms that analyze the motion of objects and object features (such as edges) within a scene. The BDTI Optical Flow Workload operates on a 720p resolution (1280×720 progressive scan) input video sequence and produces a series of two-dimensional matrices characterizing the apparent vertical and horizontal motion within the sequence. In designing this workload, BDTI has increased the control complexity relative to what is often used in this class of optical flow algorithms in order to ensure that the workload provides a sufficiently challenging test case for the tools. More specifically, BDTI incorporated dynamic, data-dependent decision making and array indexing into the optical flow application.

There are two Operating Points associated with the BDTI Optical Flow Workload, each of which uses the same algorithm but is optimized for a different metric.

- Operating Point 1 is a fixed workload defined as processing video with 720p resolution (1280×720 progressive scan) at 60 frames per second. The objective for Operating Point 1 is to achieve the required throughput while *minimizing resource utilization*. Resource utilization refers to the percentage of total processing engine resources required to implement the workload.

- Operating Point 2 is defined as the maximum throughput capability of the workload implementation on the target device (measured in frames per second) for 720p resolution (1280×720 progressive scan). The objective for Operating Point 2 is to *maximize the throughput* (measured in frames per second) using all available device resources.

The secondary workload is the BDTI DQPSK Receiver Workload. This workload is a wireless communications receiver baseband application that includes classical communications blocks found in many types of wireless receivers. It is a fixed workload with a single Operating Point defined as processing an input stream of complex, modulated data at 18.75 Msamples/second with the receiver chain clocked at 75 MHz. The receiver produces a demodulated output bitstream of 4.6875 Mbits/second. The objective for this workload is to *minimize the FPGA resource utilization* needed to achieve the specified throughput.

Additional details on the two workloads are provided in Appendix A.

Description of Metrics

Two categories of metric are used in this evaluation:

- **Quality of results metrics** assess the performance and efficiency of the workload implementation. For the BDTI Optical Flow Workload, quality of results metrics are reported for the AutoESL-Xilinx implementation and for the DSP processor implementation. For the BDTI DQPSK Receiver Workload, quality of results metrics are reported for the AutoESL-Xilinx flow and for a traditional FPGA implementation using a hand-written RTL design.
- **Usability metrics** assess the productivity and ease of use associated with the AutoESL-Xilinx design flow, and are based on the BDTI Optical Flow Workload. These metrics compare the productivity and ease of use associated with using the AutoPilot and Xilinx tools targeting an FPGA relative to using a DSP processor with its associated software development tool chain.

  Usability metrics are evaluated qualitatively based on nine aspects of tool use, including out-of-the-box experience, ease of use, completeness of tool capabilities, efficiency of overall design methodology, and quality of documentation and support.

Description of Platforms

For this evaluation, the target FPGA was the Xilinx Spartan-3A DSP 3400 (XC3SD3400A). For the BDTI Optical Flow Workload, the Xilinx XtremeDSP Video Starter Kit (which is based on the XC3SD3400A) was used. Spartan-3A DSPs are based on Xilinx's low-cost Spartan-3A family, but have a number of enhancements to accelerate digital signal processing. For example, Spartan-3A DSP chips have double the block RAM ("BRAM") memory of other Spartan devices and incorporate hard-wired DSP data paths, called "DSP48A slices." Each DSP48A slice contains an 18×18 multiplier with pre-adders and an accumulator, among other features. The XC3SD3400A includes 126 DSP48A slices that can be clocked at up to 250 MHz, and roughly 54,000 logic cells. Xilinx RTL tools, including the ISE and EDK tool suites, were used with AutoPilot. (ISE and EDK version 10.1.03 (lin64)).

The target DSP processor was the Texas Instruments TMS320DM6437. The TMS320DM6437 is a video-oriented processor that includes a 600 MHz Texas Instruments TMS320C64x+ DSP core along with video hardware accelerators. (The hardware accelerators do not support the BDTI Optical Flow Workload, and therefore were not used in the DSP processor implementation of the BDTI Optical Flow Workload.) The evaluation used the Texas Instruments DM6437 Digital Video Development Environment combined with the Texas Instruments Code Composer Studio tools suite (version V3.3.82.13, Code Generation Tools version 6.1.9).

We should note here that, as is typical for high-level synthesis tools, AutoPilot costs considerably more than DSP processor software development tools. Tool cost is not reflected in the cost-performance results presented later in this paper. A fundamental question is whether the higher tool cost is justified by higher quality of results and/or higher productivity enabled by AutoPilot. We believe that the results and analysis presented in this paper will prove valuable to prospective AutoPilot users in answering that question.

Implementation, Certification Process

The work of implementing the two workloads on the two chips was distributed between AutoESL, Xilinx, and BDTI based on the chip and tool chain used. AutoESL implemented both workloads using the AutoPilot and Xilinx tools and submitted resource utilization results to BDTI for independent verification and certification. In parallel, BDTI's engineers

independently implemented portions of the BDTI Optical Flow Workload using the AutoPilot and Xilinx tools to gain first-hand insight into the usability of the tool chain. BDTI implemented the BDTI Optical Flow Workload on the DSP processor, and Xilinx implemented the hand-coded RTL FPGA version of the BDTI DQPSK Receiver workload (which was then verified and certified by BDTI).

In addition, BDTI interviewed a number of AutoPilot customers about their experiences in using the tool and the quality of results they have attained. These interviews were used to augment (and sanity-check) the results obtained using the workload implementations.

Quality of Results Metrics

In this section we present the Quality of Results findings of our evaluation. The first results we present are for the BDTI Optical Flow Workload. Table 1 shows results for Operating Point 1, which evaluates the percentage of on-chip hardware resources consumed by a 60 fps Optical Flow Workload at 720p resolution.

As shown in Table 1, the FPGA implementation using the AutoESL-Xilinx tool chain utilized 39% of the FPGA resources to implement the workload. The DSP processor was unable to implement this workload; a minimum of 12 'DM6437 chips would be needed to achieve 60 fps operation on the BDTI Optical Flow Workload.

TABLE 1. Quality of Results for BDTI Optical Flow Workload Operating Point 1: Fixed Throughput (1280×720 Progressive Scan, 60 fps)

Platform	Chip Unit Cost (Qty. 10K)	Chip Resource Utilization
AutoESL AutoPilot plus Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA	$26.65	39%
Texas Instruments software development tools targeting the TMS320DM6437 DSP processor	$21.25	N/A (a minimum of 12 DSPs would be required to meet this operating point)

These results illustrate the difference in processing horsepower available on the Spartan-3A DSP versus the TI 'DM6437 chip—for a 25% higher chip price, the FPGA used with AutoPilot provides more than an order of magnitude higher computational power. For applications that can make good use of this horsepower (such as the BDTI Optical Flow Workload), the FPGA's performance advantage is compelling.

In contrast to Operating Point 1 of the BDTI Optical Flow Workload, which specifies a fixed throughput requirement, the goal of Operating Point 2 is to achieve the highest throughput possible, using all of the available chip resources. Table 2 presents these results in terms of the maximum frames per second supported by each single chip and the associated cost per frame per second.

TABLE 2. Quality of Results for BDTI Optical Flow Workload Operating Point 2: Maximum Throughput (1280×720 Progressive Scan)

Platform	Chip Unit Cost (Qty. 10K)	Maximum Frames per Second (FPS)	Cost per FPS (Lower is Better)
AutoESL AutoPilot plus Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA	$26.65	183 fps	$0.14
Texas Instruments software development tools targeting the TMS320DM6437 DSP processor	$21.25	5.1 fps	$4.20

Here again, the FPGA implementation created with AutoPilot achieves much higher performance than the DSP processor, and also much better cost-performance.

Prospective users are often interested in understanding the capacity of synthesis tools. The Optical Flow Operating Point 2 represents the largest design evaluated by BDTI, and used 76% of the Xilinx XC3SD3400A FPGA. AutoPilot had no trouble handling a design of this size. For this workload, the reference C code contained 559 functional lines of code. The restructured and optimized reference C code contained 1,604 lines of code, which then generated 38,222 lines of Verilog RTL or 35,849 lines of VHDL RTL. AutoPilot synthesized the C description and generated the RTL in less than 30 seconds.

The next question our evaluation attempted to answer was: How efficient is an AutoPilot-based FPGA implementation relative to an implementation

created using hand-written RTL code? For this part of the evaluation we used the BDTI DQPSK Receiver Workload (the Optical Flow workload would have been prohibitively time-consuming to hand code in RTL).

Table 3 shows the efficiency of the implementation using the AutoESL-Xilinx tool chain versus the hand-written RTL implementation.

TABLE 3. Quality of Results for DQPSK Receiver Workload: Fixed Throughput (18.75 Msamples/ Second Input Data with a 75 MHz Clock Speed)

Platform	Chip Resource Utilization (Lower is Better)
AutoESL AutoPilot plus Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA	5.6%
Hand-written RTL code using Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA	5.9%

In this case, the hand-written RTL implementation was created by an experienced FPGA designer, and made use of Xilinx Coregen IP blocks where applicable. As shown in Table 3, AutoPilot was able to achieve a level of efficiency comparable to that of hand-coded RTL on this workload. Achieving this result required modifications to the C language description of the workload; however, the extent of these modifications was modest.

The similarity of AutoPilot and hand-written RTL results is probably not accidental; AutoESL was provided with the resource utilization for the hand-coded RTL result at the outset of the implementation process, and likely used this as a target in optimizing its implementation. It is important to note, however, that such information is not required for effective use of AutoPilot, and that AutoESL was not provided with the hand-coded design. The results shown in Table 3 are consistent with results reported by AutoPilot customers interviewed by BDTI. These users generally reported that the tool produced results that were similar in efficiency to hand-coded RTL. Furthermore, they said that using a high-level synthesis tool enabled them to easily explore alternative architectures, which often led to more efficient implementations.

Historically, poor quality of results has been one of the biggest pitfalls of high-level synthesis tools. From the results presented here and the user interviews conducted by BDTI, it is clear that AutoESL has done an excellent job in overcoming this problem, at least for the class of workloads used in this analysis.

Usability Metrics

In this section we present the usability metrics for the BDTI High-Level Synthesis Tool Certification Program. The usability metrics, presented in Table 4 and Table 5, provide an assessment of the productivity and ease of use of the high-level synthesis tool flow compared with the DSP processor tool chain. For each usability metric described below, BDTI assigns a score of Excellent, Very Good, Good, Fair, or Poor. For the AutoESL-Xilinx flow, the first score listed is the score for the complete flow. Beneath those scores, in parenthesis, separate scores are provided for AutoPilot and for the Xilinx RTL tool chain.

In assigning these scores, BDTI considers the overall design process for a complete project—starting with a C language application specification and ending with a real-time implementation on the target processing platform (either an FPGA or DSP processor). Detailed descriptions of each usability metric are provided in Appendix B.

In general, AutoPilot was quite straightforward to install and use. In contrast, the Xilinx RTL tools were difficult to install and complicated to use, particularly for a novice FPGA user. The net result is that, as shown in Table 4 and Table 5, the AutoPilot plus Xilinx tool chain has productivity ratings similar to those of the DSP processor flow.

Although we did not directly compare the ease of use of AutoPilot vs. hand-written RTL code, the AutoPilot customers we interviewed estimated that using AutoPilot cut their design times by roughly 50% compared with using hand-written RTL code. In addition some saw a more significant reduction in time required for verification. For example, some customers said that because much of their simulation could be done at the C level rather than at the RTL level, it was much faster and easier. And some said that they expected to be able to reduce total design time even further.

The quality of documentation is a weak spot for AutoPilot; there is very little of it. This is likely attributable to the newness of the tool.

TABLE 4. Usability metrics (1 of 2)

	Out-of-Box Experience	Ease of Use	Completeness of Capabilities	Quality of Documentation and Support
Combined AutoESL AutoPilot plus Xilinx RTL tools rating[1] (AutoESL AutoPilot rating / Xilinx rating)	**Fair** (Very Good/Poor)	**Good** (Very Good/Fair)	**Good** (Good/Good)	**Good** (Fair/Very Good)
Texas Instruments software development tools rating[2]	**Good**	**Very Good**	**Very Good**	**Very Good**

[1.] AutoESL AutoPilot plus Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA
[2.] Texas Instruments software development tools targeting the TMS320DM6437 DSP processor

TABLE 5. Usability metrics (2 of 2)

	Efficiency of Design Methodology				Extent of Modifications Required to Reference Code
	Learning to Use the Tool	Design and Implementation (First Compiling Version)	Design and Implementation (Final Optimized Version)	Platform Infrastructure Development	
Combined AutoESL AutoPilot plus Xilinx RTL tools rating[1] (AutoESL AutoPilot rating / Xilinx rating)	**Very Good** (Very Good/NA[2])	**Very Good** (Very Good/NA)	**Good** (Good/Good)	**Good** (Good/Good)	**Good** (Good/NA)
Texas Instruments software development tools rating[3]	**NA (assuming already familiar)**	**Excellent**	**Good**	**Good**	**Fair**

[1.] AutoESL AutoPilot plus Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA
[2.] "NA" = not applicable
[3.] Texas Instruments software development tools targeting theTMS320DM6437 DSP processor

The process of developing the platform infrastructure was similar for the FPGA and DSP in this project. In general, DSP processors come well equipped with peripherals for implementing DSP applications, but provide limited flexibility in peripherals and interfaces. FPGAs, in contrast, provide excellent flexibility in peripherals and interfaces, but may not, by default, include needed peripheral blocks.

As mentioned earlier, obtaining an efficient implementation of the BDTI Optical Flow Workload using AutoPilot required modifications to the C code. This was also true for the DSP processor. The extent of modifications required to the C code was moderate for the AutoESL-Xilinx flow and high for the DSP processor flow. Optimizing the BDTI Optical Flow Workload for the DSP processor required extensive code reorganization relative to the reference code. This reorganization process is not only time-consuming but also requires a very high level of expertise.

Part of the reason the DSP processor required so much reorganization is that the BDTI Optical Flow Workload used in this analysis is well suited for implementation on FPGAs and not well suited for implementation on DSP processors with caches (such as the TMS320DM6437 chip used in this analysis). The amount of reorganizing required will vary by application.

Overall, considering the entire design flow we expect that, in general, the level of effort for implementing many kinds of applications on a Xilinx FPGA using AutoPilot and the Xilinx RTL tools will be similar to that of implementing the same applications on a DSP processor using software development tools.

As shown in Table 6, two different skill sets are required for the FPGA implementation: skills associated with AutoPilot, and skills associated with the Xilinx RTL tools. In comparison, for DSP processor application development, an engineer with specialized DSP algorithm and programming skills is required. A typical DSP software engineer with an awareness of hardware architecture fundamentals (e.g., pipelining, latency) can learn to effectively use AutoPilot. Although a learning curve on the order of several weeks to a few months is required to become proficient using AutoPilot (depending on the background of the engineer), no other specialized expertise is required. Since BDTI engineers learned to use AutoPilot from the ground up, we include a *Learning to Use the Tool* usability metric. However, because the DSP processor and FPGA tools were used by engineers already familiar with them, BDTI did not assign a score for learning to use these tools.

Perhaps the most significant usability weakness of the AutoPilot-based tool flow is not AutoPilot itself—it's the Xilinx RTL tools. We had assumed that AutoPilot would largely abstract the user from the process of going from RTL to the FPGA bitstream, and, as mentioned earlier, that turned out to be a false assumption. Generating the final FPGA implementation requires the user to create scripts that run some parts of the Xilinx tools, and more importantly, can require the user to manually integrate and run various RTL blocks together.

Though it is relatively straightforward to get good results from AutoPilot, the remaining task of getting from RTL to bitstream using the Xilinx RTL tool chain was time-consuming and not a process that most DSP software engineers would be equipped to handle without assistance from an FPGA expert. It's clear that, while AutoPilot can produce efficient results and significant improvements in productivity, it does not eliminate the requirement for an FPGA expert as part of the team. Along these same lines, for current FPGA users, AutoPilot accelerates several significant portions of the design process, but does not address all important aspects of the process.

Conclusions

BDTI's earlier benchmarking of FPGAs and DSP processors showed large performance and cost-performance advantages for FPGAs on some applications when the FPGA implementations were created using traditional RTL design techniques. The new analysis presented here shows that FPGAs can achieve similar performance and cost performance advantages when used with the AutoESL AutoPilot high-level synthesis tool. In addition, we found that AutoPilot can achieve quality of results equivalent to hand-written RTL code. We were surprised and impressed by the quality of results that AutoPilot was able to produce, given that this has been a historic weakness for high-level synthesis tools in general.

FPGA designs done using traditional hand-written RTL coding typically take much more effort than the equivalent application implemented in software on a DSP processor. Therefore, perhaps the most surprising outcome of this project was that it took roughly the same effort to implement the evaluation workload on the FPGA using the AutoPilot plus Xilinx RTL tools as it took on the DSP processor. This is a significant breakthrough, and one with the poten-

TABLE 6. BDTI High-Level Synthesis Tool Certification Program Results Skills Required

Platform	Required Skill Set
AutoESL AutoPilot high-level synthesis tool	• Application expertise • C programming • Hardware architecture fundamentals • Algorithmic optimization and restructuring
Xilinx RTL tools	• FPGA architecture details • Basic RTL knowledge • RTL tools • Devices and interfaces
TI DSP tools	• Application expertise • C and assembly programming • Processor chip architecture details • Algorithmic optimization and restructuring • Devices and drivers

tial to have a major impact on the design of high-performance embedded computing applications. Given the FPGA's advantages in speed and cost-performance, we expect that the availability of AutoPilot will significantly change the trade-offs between DSPs and FPGAs for certain types of applications.

Overall, our analysis indicates that the AutoESL-Xilinx tool chain used with the Spartan-3A DSP 3400 FPGA yields much better performance and cost/performance than the TI DSP processor on the workload we considered, while offering similar development effort. And when compared with hand-written RTL, AutoPilot was able to deliver equivalent results. In exchange for these advantages, users will pay much more for the tool chain and will need FPGA expertise on the development team.

We expect that many system designers will be happy to make that trade-off.

APPENDIX A: Workload Details

The BDTI Optical Flow Workload™

A block diagram of the BDTI Optical Flow Workload is shown in Figure 2.

The BDTI Optical Flow Workload is a video processing application suitable for implementation on an FPGA or high-performance processor. The term "optical flow" (or "optic flow") refers to a class of video processing algorithms that analyze the motion of objects and object features (such as edges) within a scene.

The BDTI Optical Flow workload operates on a 720p resolution (1280×720 progressive scan) input video sequence and produces a series of two-dimensional matrices characterizing the apparent vertical and horizontal motion within the sequence. As mentioned in the main text of this white paper, there are two Operating Points associated with the BDTI Optical Flow Workload, each of which uses the same algorithm:

- Operating Point 1 is a fixed workload defined as processing video with 720p resolution (1280×720 progressive scan) at 60 frames per second. The objective for Operating Point 1 is to achieve the required throughput while minimizing resource utilization. Resource utilization refers to the percentage of total processing engine resources required to implement the workload.
- Operating Point 2 is defined as the maximum throughput capability of the workload implementation on the target device (measured in frames per second) for 720p resolution (1280×720 progressive scan). The objective for Operating Point 2 is to maximize the throughput (measured in frames per second) using all available device resources.

Quality of results scores for Operating Points 1 and 2 are based on results obtained when using a BDTI-proprietary video clip.

The block diagram shown in Figure 2 characterizes the implementation of the BDTI Optical Flow Workload in the C language reference implementation included with the BDTI specification package. However, it is not required that final optimized implementations of the workload maintain this structure. Blocks may be merged or restructured to improve the efficiency of the final implementation, provided all acceptance criteria are met (e.g., all test vectors pass).

Note that, at the level of this block diagram, the workload is characterized by a feed-forward chain of signal processing blocks. The feed-forward nature of the application enables straightforward pipelining of workload implementations to increase throughput. The recommended data widths at the input and output of each block are specified by BDTI.

A brief description of each of the five main functional blocks in the workload follows.

Gradient Calculation

The gradient calculation block computes luminance gradients of the incoming video sequence in the horizontal, vertical, and time dimensions. It is implemented as a one dimensional FIR filter applied to the video sequence in each of the respective dimensions to produce three 1280×720 gradient matrices.

Gradient Weighting

The gradient weighting block smoothes the gradient matrices computed in the gradient calculation block using a separable two-dimensional FIR filter. The separable filter is implemented in the reference code using one-dimensional horizontal and vertical FIR filters applied to each of the three gradient matrices.

Outer Product

The outer product calculation is implemented as an element-wise product of each of the weighted gradient matrices as shown below:

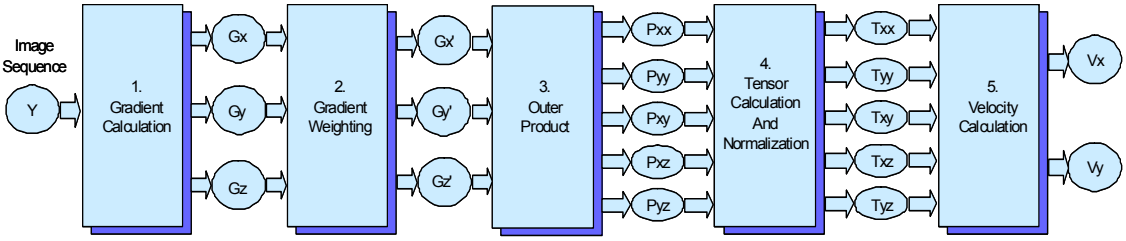

FIGURE 2. Block Diagram of the BDTI Optical Flow Workload

Pxx = Gx' * Gx'
Pyy = Gy' * Gy'
Pxy = Gx' * Gy'
Pxz = Gx' * Gz'
Pyz = Gy' * Gz'

Where Gx', Gy', and Gz' are each 1280×720 gradient matrices.

Tensor Calculation and Normalization

The tensor calculation is a two-dimensional, separable FIR filter. It is implemented in the reference code using one-dimensional horizontal and vertical FIR filters, applied to each outer product.

Tensor normalization manages data word widths by scaling the five tensors at each pixel position. The tensors at each pixel position are normalized based on the largest tensor magnitude within a small neighborhood of pixels. The size of the neighborhood is data dependent, but this block is designed to have bounded computational requirements, guaranteeing that a real-time implementation is feasible.

Velocity Calculation

The velocity calculation computes the horizontal and vertical components of velocity at each pixel position, and is implemented as follows:

velocityX = (Tyz*Txy-Txz*Tyy) / (Txx*Tyy-Txy*Txy)

velocityY = (Txz*Txy-Tyz*Txx) / (Txx*Tyy-Txy*Txy)

Where each T in the above equations is a 1280×720 matrix output from the tensor calculation.

The BDTI DQPSK Receiver Workload™

A block diagram of the BDTI DQPSK Receiver Workload is shown in Figure 3. The BDTI DQPSK Receiver Workload is a wireless communications application suitable for implementation on an FPGA or a high performance digital signal processor. The workload includes classic communications blocks that can be found in many wireless receivers in various forms and complexities.

The BDTI DQPSK Receiver Workload is a fixed workload with a single Operating Point defined as processing an input stream of complex modulated data at 18.75 Msamples/second with the receiver chain clocked at 75 MHz. The corresponding DQPSK demodulated output bitstream is 4.6875 Mbits/second. The objective for this workload is to minimize the FPGA resource utilization needed to achieve the specified throughput. Resource utilization refers to the percentage of total processing engine resources required to implement the workload.

The high-level block diagram shown in Figure 3 characterizes the implementation of the workload in the C language reference implementation included with BDTI's specification package. Blocks may be merged or restructured to improve the efficiency of the final implementation, provided all acceptance criteria are met (e.g., all test vectors pass). The recommended data widths at the input and output of each block are specified by BDTI.

A brief description of each of the five main functional blocks in the workload follows.

Matched Filter

The matched filter block implements a square root raised cosine (SQRC) FIR filter with selectable roll-off factor. The input to the block is a complex DQPSK modulated signal at four times the symbol rate. The output of the block is a complex filtered signal at four times the symbol rate.

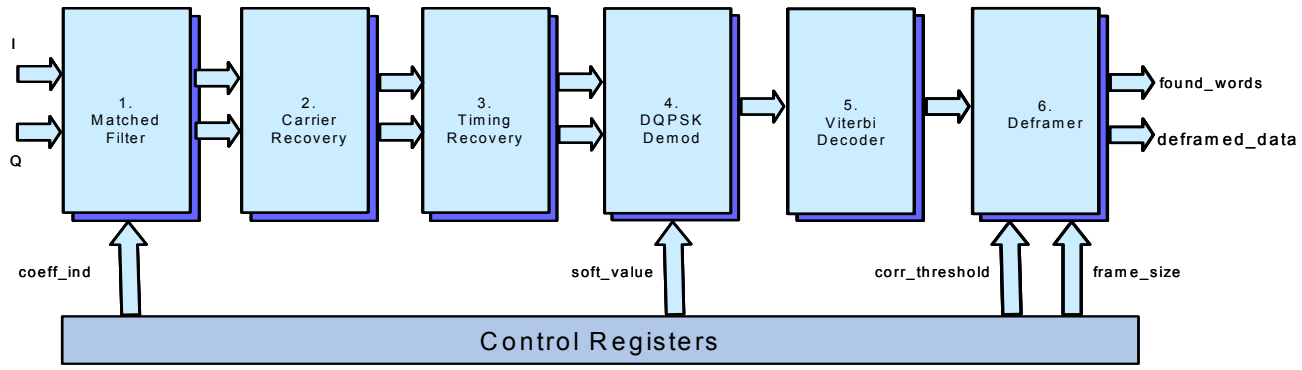

FIGURE 3. Block Diagram of the BDTI DQPSK Receiver Workload

Carrier Recovery

The carrier recovery block implements a modified Costas Loop to estimate, correct, and track errors between the transmitter and receiver carrier frequencies.

The input to this block is the complex output of the SQRC at four times the symbol rate. The output is a complex sample corrected by the estimated phase error at four times the symbol rate.

Timing Recovery

The timing recovery block implements a modified Mueller and Mueller algorithm for symbol timing recovery. The timing error is calculated using the current received and detected (sliced) symbols, and previously received and detected symbols. The block also includes an interpolator that interpolates the received data value at the estimated sampling time using the newly received data and the previously sampled data.

The input to this block is a phase corrected complex sample at four times the symbol rate from the carrier recovery block and the output is a complex sample at the estimated sampling time at symbol rate.

DQPSK Demodulator

The DQPSK demodulator accepts a complex symbol and differentially decodes its phase relative to the previously received complex symbol. It then slices this phase into corresponding decoded bits.

The input to the DQPSK demodulator is a complex value and the output consists of two soft values corresponding to the two decoded bits.

Viterbi Decoder

The Viterbi decoder accepts a continuous stream of soft decisions from the slicer and outputs a stream of decoded binary bits.

Deframer

The deframer is a correlator that operates on every output bit searching for an "access code." The access code is a predefined sequence of bits indicating the start of the payload.

APPENDIX B: Usability Metrics Details

Usability Metrics

The usability metrics shown in Table 4 are defined as follows:

Required Skills

In addition to assigning scores for the usability metrics listed below, BDTI also identifies the skills required to effectively work with each tool chain. No score is provided for this metric.

Out-of-the-Box Experience

This includes all activities from unpacking the box to getting everything set up and installed so that the user can start the design process. The assessment includes items such as: clarity of documentation, smoothness of the installation process, the time required to perform the installation, and the helpfulness of tutorials or demo applications. Note: for the Xilinx tools, the Out-of-the-Box Experience was assessed by DSP software engineers rather than an experienced FPGA designer.

Ease of Use

This is an assessment of how easy it is to use the features provided by the tool chain. It is not meant to identify missing features (this is addressed as part of the "completeness of capabilities" metrics). The assessment includes items such as the intuitiveness and user-friendliness of the user interface, responsiveness (i.e., whether the tool chain was slow to complete actions), reliability (did the tools crash or hang?) and clarity of on-line help.

Completeness of Capabilities

This is an assessment of the extent to which the tool chain includes all capabilities necessary to enable a user to efficiently complete the implementation of the workloads.

Quality of Documentation and Support

Throughout the certification process BDTI assesses the documentation supplied with the tool chain. This includes the quality of the getting started guide, tutorials, and the ease of finding answers to specific questions (including technical support).

Efficiency of the Overall Design Methodology

This is primarily an assessment of user productivity. It includes assessing the extent to which a user of

the high-level tool is abstracted from the underlying architecture (RTL for FPGA implementations and the DSP processor core and chip architecture for DSP processor implementations). Since this is a broad category, it is broken up into the following sub-categories:

Learning to Use the Tool:

- For FPGA designs: Learning to efficiently use the high-level synthesis tool. As mentioned above, no score is provided for learning the Xilinx RTL tools because an experienced BDTI FPGA engineer worked with them.
- For DSP designs: As mentioned above, no score is provided for learning the DSP tools because experienced BDTI DSP engineers worked with them.

First Compiling Version:

- The effort required to create an initial functional implementation of the application. For the FPGA this includes only the HLS tool—use of the RTL tools to support integration into the FPGA is not included. The first compiling version is not expected to be optimized in terms of performance or resource utilization, but rather an initial implementation based on which the optimization process can begin.

Final Optimized Version:

- The effort required to take the application from the first compiling version to a final optimized implementation (not including completion of interfaces required to run on an actual chip). For the FPGA, this does not include final integration with the video and memory interfaces, but does include adding C language code required to interact with external memory. For the DSP processor, this includes testing via file I/O, but not integration with external video ports.

Platform Infrastructure Development:

- This category includes integration of platform components that must be incorporated into a design so that it will run on a physical chip (i.e., a DSP processor or FPGA). This includes, but is not limited to:
  - For FPGA implementations: Complete integration of the memory controller, external memory and video I/O.
  - For DSP processor implementations: Installation and configuration of drivers and libraries, and interfacing to video I/O.

Extent of Modification to the Original Reference Code

This is an assessment of how closely the code used to generate the final optimized BDTI Optical Flow Workload implementation matches the original C language reference code provided by BDTI. This is not a simple count of the number of lines of code that have been changed, but rather reflects the complexity and effort involved in making the necessary changes. Typically changes are made for one of the following reasons:

- Structural changes: Changes to the overall data flow and code structure required to map the application the underlying device architecture
- Changes required because of tool restrictions or limitations
- Timing and resource optimizations on individual blocks
- Interfacing to peripherals and other external modules

BDTI considers factors such as: the number of changes, the extent to which the tools automate implementing these changes, the level of difficulty for the developer in incorporating the changes and the level of difficulty in debugging and testing the changes.