1 Introduction

We find ourselves caught in the snare of ever-increasing computational complexity, where the demands of modern computation can bog down a general-purpose CPU to the point of impracticality[1]. Hardware accelerators are mechanisms used to mitigate such situations by offloading specific computational tasks to hardware. Developing optimized hardware solutions to replace software comes with the trade-off of increased development time due to the introduction of hardware's inherent lower abstraction levels. This increased overhead can often be prohibitive, thus it has been a dream of hardware designers since the 1970s to create tools that synthesize optimized hardware using traditional software development techniques[2]. This dream tool would complete a process known as High Level Synthesis (HLS).

In order to consider the dream realized, HLS must create optimized hardware using an abstract, algorithmic level description of a circuit as the primary input specification[3]. Unfortunately, due to reasons such as standardization[4], reliability[5] and portability[6], the two most ubiquitous languages for constructing hardware descriptions (VHDL and Verilog) do not operate natively at the algorithmic level [7]. This presents a problem: the most oft-used, best supported, languages for specifying synthesizable hardware do not behave like that of traditional compiled software. Instead, these two languages operate at a level of abstraction known as the Register-Transfer Level (RTL), which is one level of abstraction below that of the algorithmic level [8], and HLS seeks to bridge that gap.

The development of HLS tools is not the focus of Homsirikamol and Gaj's study, rather they seek to validate the state of the HLS dream by comparing the performance of hardware generated using an RTL circuit description to that of an algorithmic, HLS, specification. This report will, first, explain the methodology used to conduct the case study and present its findings. Next, an evaluation of their conclusions will be presented alongside alternative methods of comparison. This report concludes with an analysis of what to expect from HLS in the future, ultimately presenting a more complete picture of whether HLS is, or ever will be, a dream-come-true.

2 Summary of Selected Paper

This section will briefly summarize the claims presented by Homsirikamol and Gaj on the state of HLS.

2.1 Background

HLS is the process of converting an algorithmic level description to RTL. Therefore, in this context, the term synthesis can be defined as a method for traversing abstraction levels in hardware design. This is carried out by implementing a description found at higher levels of abstraction using only methods found in the lower [6]. Specifically, HLS translates descriptions found at the algorithmic level, often written in a relatively high-level language, in this study that language is C, to a functionally equivalent representation in an RTL language like VHDL or Verilog.

2.2 Problem

Measuring the quality of HLS tools is a problem that requires a high degree of specificity in order to be useful. Quality has often been measured by simply investigating whether a complex circuit synthesized through HLS can achieve a desired functionality[9][10][11][12][13]. Other studies [14][15] seek to measure the quality of HLS by comparing the performance of an HLS circuit to its software implementation. The authors contend that neither of the above metrics for quality (functionality or software comparison) paint a sufficient picture of the state of HLS, rather they claim that the quality of HLS is better assessed when compared to a circuit synthesized through traditional RTL synthesis. Additionally, since computational challenges vary greatly based on functional domain, Homsirikamol and Gaj claim that comparisons between HLS tools and RTL synthesis tools must be domain-specific, i.e., it is not useful to make blanket statements of a tool's ability without, first, specifying the types of computational tasks each method is expected to synthesize. The selected paper chooses to make an HLS-to-RTL comparison in the cryptographic domain, specifically comparing implementations of the Advanced Encryption Standard.

2.3 Proposal

Ultimately, the authors wish to demonstrate the quality of HLS by comparing the performance of circuits synthesized using a software description to circuits synthesized using an RTL description. Metrics used in the study to define performance included: area, frequency, throughput and efficiency (throuput per unit area). In order to make a fair comparison, the authors need to make a case that the synthesis process alone was responsible for performance changes. This was accomplished by conducting their evaluations across multiple chip manufacturers and chip families, thus removing the target technology as a possible source of performance variability. The RTL language, FPGA tools and synthesis options were all held constant, leaving HLS as the claimed independent variable in the case study.

Using the above methodology, the authors used the AES reference design in ANSI C [16] as a starting point for creating the study's HLS specifications. They, then, created three different implementations of AES, each optimized in a different fashion. The first, referred to as HLSv0, was streamlined by eliminating unnecessary code. The HLSv0 code was then optimized by modifying the methods implementing the actual algorithm; this design is called HLSv1. Finally, HLS synthesizer directives were added to the HLSv1 design, thus creating the last design referred to as HLSv2. The performance of each HLS design was measured against the RTL design.

Three wrappers were then placed around the core AES implementations, thereby creating three distinct cryptographic systems; the performance of these systems underwent the performance analysis defined above. The three cryptographic systems included a twice-unrolled architecture of AES, whereby the AES core was, essentially, duplicated, an implementation of AES in counter mode (AES-CTR) and finally, an implementation of AES-CTR using an I/O-multiplexing communication protocol modified from a previous study[17].

2.4 Evaluation

Four performance analyses were presented. These compared RTL-to-HLS implementations for each of the AES core modules, the twice unrolled architecture, the AES-CTR system and the AES-CTR system using an I/O-multiplexed communication protocol.

2.5 Conclusion

3 Analysis of Selected Paper

3.1 Paper Strengths

3.1.1 Carefully Crafted Caveats in Conclusions

3.1.2 Controlling External Influences

Not allowing proprietary FPGA resources such as BRAM, DSP, etc, although porting Vivado HLS-derived VHDL to Altera tools did not work for HLSv0 and HLSv1. Same synthesis tools. Same synthesis options. Restricting the computational domain.

Synthesis tools vary greatly across manufacturers and device families, therefore comparing implementations of synthesized designs becomes a delicate exercise. The authors present a case that their methodology for comparing synthesis performance is a fair one and this section explores that claim.

3.2 Paper Weaknesses

3.2.1 RTL Optimization

The authors create three different designs for HLS, each optimizing a different aspect of the C code. It is not immediately clear what, if any, optimizations were made to the RTL design to which the performance of the HLS designs were compared. Since the authors do not state otherwise, it can be assumed that no efforts were made to optimize the RTL design in similar fashions to the optimizations made to the HLS designs. This is troubling for the HLS cause as the performance of unoptimized RTL designs still outperformed that of optimized algorithmic level designs in all metrics, save one (a 5% increase in throughput over RTL was measured for the AES-CTR implementation targeting a high-end Xilinx chip).

3.2.2 Parallelization

Failure to address the biggest weakness of HLS.

3.3 Critique

3.4 Future Work

Since the HLS dream has yet to be realized, at least in the cryptographic [18] or matrix-multiplication [1] domains, what should hardware engineers expect from HLS in the future? The answer appears to be not much. Fundamental flaws in the HLS dream seem to erode expectations of HLS ever operating in the manner in which it was intended. Two main issues confronting HLS are performance and the ability of software descriptions to be synthesized.

The missing performance component holding back the dream of creating optimized hardware from a software language intended for sequential execution is automatic parallelization, which has been described as the "holy grail" of parallel computing [19]. As shown in Section 3.2.2, parallelization can significantly improve the performance of a hardware design. The issue HLS is up against is the fact that software languages were originally designed to be performed sequentially on a general-purpose CPU, and thus the HLS-supported C-based languages were not designed with parallelization in mind [20]. It would be useful to develop an automated process to tease out implicit parallelization from this serial software. Unfortunately, since C uses pointers to reference memory [21], the complexity involved with automatically extracting parallelism while guaranteeing that a memory location will not be asked to store two different values at the same time has been shown to be NP-Hard[22]. Even if pointers were carefully managed, or somehow avoided completely, the use of dynamic storage and recursive data structures in software presents the same memory-management dilemma that has been shown to be at least undecidable and at wost uncomputable [23]. If parallelization cannot be inferred automatically, then it must be specified explicitly, thereby lowering the level of abstraction from the algorithmic level back to RTL. If not for raising levels of abstraction what, then, is the case for using HLS as opposed to RTL?

One might contend, as Homsirikamol and Gaj elude to in their conclusions, that raising design abstraction under some conditions is better than not raising it at all. In other words, why not use HLS to reduce development time and sacrifice the performance boosts associated with pipelining? This trade-off appears reasonable until it is put into practice. The original promise of HLS, i.e., creating optimized hardware from a software description [14], seems to imply that one can write code targeting HLS the same way one can write code targeting traditional compilation. Unfortunately, not all language constructs are synthesizable. For example, C++ is supported by the Vivado HLS tool [24] and C++ (targeting compilation) supports object-oriented capabilities such as inheritance [25], but does that mean a hardware designer (targeting HLS) can use all properties of inheritance like virtual subclass destructors? The answer is no [26]. This, then, creates subsets of software languages that are appropriate for synthesis. The issue then becomes knowing the difference between language constructs that are appropriate for synthesis versus those that are not. Currently, the solution is to define standard language constructs suitable for RTL synthesis [27], thus separating the synthesizable subset from the rest of the language. This solution is valid provided that the hardware engineer has deep knowledge of the standard as well as an understanding of the synthesizer's behavior. These requirements seem to defeat the purpose of the original HLS dream and serves to increase development time rather than reduce it.

4 Conclusion

Homsirikamol and Gaj ultimately conclude that HLS can generate results comparable to RTL while significantly lowering development time. They further claim that the time saved using HLS can be used by the designer to hand-optimize problematic or critical modules. For the authors' claim to hold water, HLS should be able to achieve good results while being utilized in the manner for which it was intended: designing hardware at a higher level of abstraction. Their claim hinges so precariously on higher abstraction because this is the only stated means of reducing development time. As shown in Section 3.2, the authors violate the intent

Based on the bleak future presented in Section 3.4, it can be concluded that HLS is, at best, a zero-sum game and, at worst, a costly distraction.

References

[1] S. Skalicky, C. Wood, M. Lukowiak, and M. Ryan, "High level synthesis: Where are we? a case study on matrix multiplication," in *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*, Dec 2013, pp. 1–7.

[2] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 18–25, July 2009.

[3] M. C. McFarland, A. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301–318, Feb 1990.

[4] "IEEE standard for verilog register transfer level synthesis," *IEEE Std 1364.1-2002*, 2002.

[5] S. Tosun, N. Mansouri, E. Arvas, M. Kandemir, and Y. Xie, "Reliability-centric high-level synthesis," in *Design, Automation and Test in Europe, 2005. Proceedings*, March 2005, pp. 1258–1263 Vol. 2.

[6] P. P. Chu, *RTL Hardware Design Using VHDL: Coding For Efficiency, Portability, and Scalability.* John Wiley & Sons, Inc., 2006, pp. 1–22. [Online]. Available: http://dx.doi.org/10.1002/0471786411.fmatter

[7] D. M. Harris and S. L. Harris, *Digital design and computer architecture*, second edition ed. Amsterdam, Boston: Morgan Kaufmann Publishers, Cop., 2013.

[8] F. Vahid, *Digital Design with RTL Design, Verilog and VHDL.* John Wiley & Sons, Inc., 2010, p. 247.

[9] F. Burns, J. Murphy, D. Shang, A. Koelmans, and A. Yakorlev, "Dynamic global security-aware synthesis using SystemC," *Computers Digital Techniques, IET*, vol. 1, no. 4, pp. 405–413, July 2007.

[10] M. Ernst, S. Klupsch, O. Hauck, and S. Huss, "Rapid prototyping for hardware accelerated elliptic curve public-key cryptosystems," in *Rapid System Prototyping, 12th International Workshop on, 2001.*, 2001, pp. 24–29.

[11] S. Morioka, T. Isshiki, S. Obana, Y. Nakamura, and K. Sako, "Flexible architecture optimization and ASIC implementation of group signature algorithm using a customized HLS methodology," in *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, June 2011, pp. 57–62.

[12] S. Ahuja, S. Gurumani, C. Spackman, and S. Shukla, "Hardware Coprocessor Synthesis from an ANSI C Specification," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 58–67, July 2009.

[13] J. Davis, D. Buell, S. Devarkal, and G. Quan, "High-level synthesis for large bit-width multipliers on FPGAs: a case study," in *Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on*, Sept 2005, pp. 213–218.

[14] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in *ASIC (ASICON), 2011 IEEE 9th International Conference on*, Oct 2011, pp. 1102–1105.

[15] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-level Synthesis: Productivity, Performance, and Software Constraints," *JECE*, vol. 2012, pp. 1:1–1:1, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1155/2012/649057

[16] P. Barreto and V. Rijmen, “Reference code in ANSI C v2.2,” http://www.ktana.eu/html/theRijndaeIPage.htm, Mar 2002.

[17] K. Gaj, E. Homsirikamol, and M. Rogawski, “Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs,” in *Cryptographic Hardware and Embedded Systems, CHES 2010*, ser. Lecture Notes in Computer Science, S. Mangard and F.-X. Standaert, Eds. Springer Berlin Heidelberg, 2010, vol. 6225, pp. 264–278. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15031-9_18

[18] E. Homsirikamol and K. Gaj, “Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study,” in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, Dec 2014, pp. 1–8.

[19] J. P. Shen and M. H. Lipasti, *Modern processor design : fundamentals of superscalar processors.* Boston: McGraw-Hill Higher Education, 2005, index. [Online]. Available: http://opac.inria.fr/record=b1129703

[20] D. M. Ritchie, “The development of the c language,” in *The Second ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL-II. New York, NY, USA: ACM, 1993, pp. 201–208. [Online]. Available: http://doi.acm.org/10.1145/154766.155580

[21] R. Reese, *Understanding and Using C pointers.* ” O’Reilly Media, Inc.”, 2013.

[22] S. Horwitz, “Precise flow-insensitive may-alias analysis is np-hard,” *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 1, pp. 1–6, Jan. 1997. [Online]. Available: http://doi.acm.org/10.1145/239912.239913

[23] W. Landi, “Undecidability of static analysis,” *ACM Letters on Programming Languages and Systems*, vol. 1, pp. 323–337, 1992.

[24] T. Feist, “Vivado design suite,” *White Paper*, 2012.

[25] Y.-F. R. Chen, E. R. Gansner, and E. Koutsofios, “A c++ data model supporting reachability analysis and dead code detection,” in *Software EngineeringESEC/FSE’97.* Springer, 1997, pp. 414–431.

[26] D. O’Loughlin, A. Coffey, F. Callaly, D. Lyons, and F. Morgan, “Xilinx vivado high level synthesis: Case studies,” in *Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014). 25th IET.* IET, 2013, pp. 352–356.

[27] “IEEE standard for verilog register transfer level synthesis,” *IEEE Std 1364.1-2002*, 2002.