xPilot: A Platform-Based Behavioral Synthesis System

Deming Chen, Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, Zhiru Zhang
University of California, Los Angeles
Email: {demingc, cong, fanyp, leohgl, wjiang, zhiruz}@cs.ucla.edu

***Abstract*— Behavioral synthesis is an automated design process that compiles functional and/or algorithmic descriptions into optimized hardware architectures. It has long been identified as one of the critical technologies for enabling the transition to the higher level of abstraction. Unfortunately, it stumbled in its debut on the EDA marketplace during the mid-1990s and so far has had a limited adoption among chip designers.**

With the billion-transistor chip on the horizon, the design complexity of integrated circuit systems is outgrowing the capabilities of current RTL methods. This brought about a renewed interest in behavioral synthesis, and innovations are required to address the new challenges in nanometer IC designs.

In this paper we present the *xPilot* behavioral synthesis system being developed at UCLA. xPilot aims to provide novel platform-based synthesis technologies to optimize logic, interconnects, performance, and power simultaneously at the high level. The ultimate goal is to improve both design productivity and quality of results. Preliminary experiments on FPGAs demonstrate the efficacy of our approach on a wide range of applications and its value in exploring various design tradeoffs.

I. Motivation

The design complexity of integrated circuit systems is outgrowing the capabilities of traditional RTL methods, and it is commonly acknowledged that the ultimate solution is to move to the next level of abstraction beyond RTL. Electronic system-level (ESL) design automation has been identified by Dataquest [1] as the next productivity boost for the semiconductor industry. However, despite some recent success in ESL simulation, the transition to ESL design will not be as well accepted as the transition to RTL without robust and efficient behavioral synthesis (also known as high-level synthesis) technology that automatically compiles functional and/or algorithmic descriptions into optimized hardware architectures.

Unfortunately, although behavioral synthesis has been a topic of research for almost two decades, it stumbled in its debut on the EDA marketplace during the mid-1990s and so far has had a limited adoption among chip designers. There were several reasons for the previous failure of behavioral synthesis. First, one may argue that design complexity was still manageable at the RT level a decade ago, and hardware designs were carried out with traditional HDLs such as VHDL and Verilog in a cycle-accurate manner. Also, in the 1990s the industry was still struggling with timing closure between logic and physical designs. There was no dependable RTL to GDSII flow to support behavior synthesis. More importantly, the results produced by previous behavioral synthesis tools were often inferior to manual designs. This further prevented the market success of an automatic synthesis approach at the high level.

In today's nanometer-scale technologies, it is perfectly feasible to design an System-on-a-Chip (SoC) device with over 500 million transistors [2], and billion-transistor chips are already on the horizon. Even with the availability of a new generation of robust RTL-to-GDSII flows, this extreme complexity can no longer be efficiently handled by the current RTL-based methodologies. Therefore, behavioral synthesis is experiencing renewed interest after a long drought. We believe that behavior-level design and synthesis is becoming an imperative step in EDA design flows as it provides these combined advantages:

(i) **Better complexity management**: Design abstraction is one of the most effective methods for controlling rising complexity and improving design productivity. For example, a recent study from NEC [3] shows that a typical RTL design requires about 300K lines of code, clearly beyond what can be handled by a human designer. However, the code density can be improved by nearly 10× when moved to the behavior level. This results in a human-manageable 40K lines of behavioral description.
In addition to the line-count reduction in design specifications, behavioral synthesis has the added value of allowing efficient reuse of behavioral IPs. As opposed to the RTL IPs which have fixed micro-architectures and interface protocols, a behavioral IP can be synthesized to various implementations (e.g., with different performance throughput) to match different system requirements.

(ii) **Shorter verification/simulation cycle**: Behavioral synthesis allows the designers to start with a specification in a high-level programming language (HLL) such as C or SystemC [4] that is directly executable and simulatable with high speed. According to the same study from NEC in [3], the simulation speed at behavior level is up to 100× faster than the one at RT level. To be more concrete, this means that an RTL design which requires dozens of simulation hours can be simulated in minutes at behavior level.
More importantly, behavioral synthesis automatically compiles the input descriptions into RTL code through a series of formal constructive optimizations and transformations. This avoids the slow error-prone manual process

where misinterpretations, syntax errors, and logical mistakes are easily introduced. Thus the design verification and debugging effort will be greatly simplified.

(iii) **Rapid system exploration**: Given the high design complexity and tight schedule, designers currently prioritize time-to-market over design optimization. For hardware design, they tend to commit to one specific micro-architecture implementation during the early stage even if more optimized alternatives are potentially available. On the other hand, behavioral synthesis tools excel at generating multiple RTL implementations from one functional specification by varying the design constraints. This can help hardware designers efficiently evaluate tradeoffs in micro-architectures and algorithms.

Furthermore, designing a modern embedded system is a much larger problem than just building the hardware gates. With the coexistence of micro-processors, DSPs, memories and custom logic on a single chip, more software elements are involved in the design process. One of the fundamental challenges of system-level design is the hardware/software partitioning, a task that is too complex to be feasible at the RT level.

HLL-based design methodologies (especially C-based designs) offer a promising solution to this problem. Unlike HDLs, HLLs are originally designed for software programming. With the aid of behavioral synthesis, they can also be used to specify functionality in hardware. In this flow, designers can quickly experiment with different hardware/software boundaries by co-simulating the HLL descriptions and the automatically synthesized RTLs.

(iv) **Higher quality of results**: VLSI designs in current semiconductor technologies are already limited by interconnect, and the interconnect delay and power are predicted to have an even larger impact as technology advances. However, at the RT level, it is extremely difficult for designers to accurately estimate the interconnects which are determined by downstream physical design tools. To achieve timing and power closure, designers have to adjust the initial RTL in an adhoc manner and iterate over the time-consuming synthesis and layout process.

We believe that the full consideration of physical reality during behavioral synthesis will lead to higher quality of results. By integrating automatic high-level optimizations together with physical planning, logic and interconnects can be optimized simltaneously.

In this paper we present the *xPilot* behavioral synthesis system being developed at UCLA. The goal of xPilot is to provide novel platform-based behavior synthesis technologies to optimize logic, interconnects, performance, and power simultaneously (which becomes much more difficult for human designers), so that we can improve both design productivity and quality of results.

The reminder of this paper is organized as follows: Section II presents an overview of the xPilot infrastructure and the main features of current xPilot implementation. Section III and Section IV briefly discuss the system front end and the core synthesis engine. The preliminary experimental results are shown in Section V.

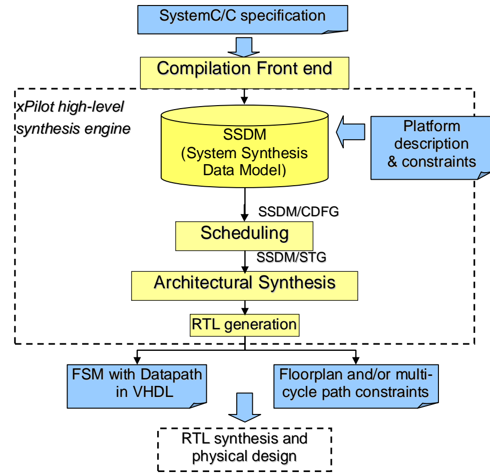

Fig. 1. xPilot behavioral synthesis framework

II. xPilot System Overview

The overall design flow of the xPilot system is shown in Figure 1. xPilot accepts synthesizable C or SystemC as input. The behavioral description is first parsed and optimized by the UIUC LLVM compiler infrastructure [5]. A system-level synthesis data model (*SSDM*) is then constructed from the LLVM's internal representation. The basic building blocks in SSDM are processes and channels. A *process* describes the behavior of one module, and each process uses a control data flow graph (CDFG) to capture its behavior. Each process interacts with other processes through *ports* and *channels*. Each channel implements some interface to implement certain communication protocols. Altogether, an SSDM defines a process network to model the concurrent behavior of a complex system. On top of SSDM, xPilot performs platform-based synthesis and physical-aware optimizations during scheduling and resource binding; these construct an optimized state transition diagram (STG) and an associated datapath model. At the back end, xPilot generates RTL implementations together with constraint files (e.g., multi-cycle path constraints, physical location constraints, etc.) to leverage the existing logic synthesis and physical design toolset.

In its current stage, xPilot exhibits the following features:

- **Applicable to a wide range of application domains**: xPilot can efficiently support various types of behavioral descriptions with different characteristics, such as computation-intensive (e.g., DSP kernels), data/memory-intensive (e.g., multi-media applications), control-intensive (e.g., controllers), etc. Moreover, xPilot accepts either untimed or partially timed behavioral specification. Cycle-accurate I/O protocols can be specified for interfacing with the surrounding modules.
- **Amenable to a rich set of synthesis constraints**: xPilot honors a variety of synthesis constraints that can be either implicitly derived from the input descriptions

(e.g., dependency constraints) and target platform (e.g., resource limits), or explicitly specified by the users, such as frequency constraint, latency constraints, relative I/O timing constraints (cycle-fixed mode, superstate-fixed mode, free-floating mode), etc.
- **Platform-based synthesis and optimization**: xPilot takes advantage of the target plaform information to carry out more informed synthesis and optimization. The platform specification shall characterize the delay, area, and power for each type of available resource (e.g., functional units, memories, steering logic, etc.) under different input/output count and bitwidth configurations.
- **Extensible to consider physical information**: In xPilot, every module is readily extensible to consider layout information. The ongoing research aims at simultaneously integrating synthesis techniques, such as scheduling and resource binding, with the physical planning engine to optimize the communications together with the computations.

III. xPilot Front End

xPilot accepts synthesizable C or SystemC as input. C language is effective for describing sequential behavior within one single module of the entire system. SystemC [4], on the other hand, provides the capability to capture many hardware-specific features such as process-level parallelism and the communication/synchronization among the concurrent modules.

Our front end complier translates design descriptions written in C or SystemC into SSDM — the internal data model of xPilot. Currently, we use the UIUC LLVM compiler [5] to parse in C/SystemC code. LLVM consists of a GCC-based C/C++ front end, a virtual instruction set, a link-time optimization framework, and various back ends for common target machines. We leverage the GCC-based front end compiler to obtain an LLVM intermediate representation (IR). On top of this IR, we first recover certain high-level programming constructs from the low-level virtual instruction set. We then perform elaboration to extract the processes, ports, channels and their interconnection topologies, and construct our SSDM accordingly based on this information.

Another major task at the front end is platform characterization. Specifically, we characterize the delay, area, and power for each type of available resource (e.g., functional units, memories, steering logic, etc.) under different input/output count and bitwidth configurations. We also capture the layout information of the target platform to facilitate our physical-aware synthesis. The heterogeneous resources distribution map and the interconnect delay/power lookup tables will be also collected.

IV. Synthesis Engine

In this section we will highlight xPilot synthesis engine, which includes scheduling and resource binding.

A. Scheduling

One of the major drawbacks to previous scheduling techniques in behavioral synthesis is the limited applicability to a specific class of algorithms and lack of efficient support of various design constraints. To address this problem, we propose a unified performance-driven scheduling algorithm which provides efficient solution for a wide range of application domains (e.g., computation-intensive applications, data/memory-intensive applications, control-intensive applications, etc.) and honors a rich set of real-life design constraints (e.g., clock frequency constraint, latency constraint, relative timing constraint, resource constraint, IO constraint, etc.).

Specifically, we represent all the scheduling constraints as a system of pairwise difference equations. Using this formulation, the feasibility check of the constraint system can be carried out efficiently by solving a single-source shortest path problem. We can also express the performance objective as a linear function so that the global optimization can be performed by any linear programming (LP) solver. Since the matrix of pairwise difference constraints is totally unimodular, the solutions from the LP solver are guaranteed to be integers.

Under this novel and unified mathematical framework, we can easily incorporate existing list-scheduling heuristics to optimize the data-flow-intensive designs. In the meantime, the aggressive instruction-level parallelism techniques from compiler domains such as code motions, speculations, and loop pipelining can be naturally integrated to optimize the control-intensive applications. Our scheduler can also be extended support incremental scheduling and interactive scheduling.

B. Simultaneous Resource Binding

The resource allocation and binding process is performed after scheduling. It determines the numbers of functional units and registers, and the sharing among compatible operations and data transfers. These optimization steps have a dramatic impact on the final design quality as they determines the interconnection network with wires and steering logic. It is well recognized that the traditional techniques of minimizing a single objective number, e.g., interconnection, functional unit, or register count, cannot guarantee a satisfiable design quality. More importantly, the impact of the same binding solution to final design quality is different on different technologies. We have to incorporate platform information tightly into the whole exploration procedure.

Our resource binding algorithm in xPilot is based on a solid performance and cost estimation model. Specifically, we compute the optimization objectives using the realistic platform-based measurements. The design cost could be design area, power consumption, or their combination. Performance mainly refers to clock frequency (or cycle time) in the scope of resource binding as the overall latency is determined by scheduling.

To explore the huge design space, we form, propagate and prune synthesis solution points guided by the area and delay estimation model. Eventually, we obtain a cost/performance

tradeoff curve, which provides a sound guidance for achieving different design objectives.

V. Experimental results

The xPilot system is implemented based on a C++/Linux environment. In this paper we report the results targeting the Altera Stratix FPGA platform [6], using Quartus II v4.2 as the downstream RTL synthesis and physical design tool.

A. Test Examples

We have tested xPilot through several real-life designs which are from different application domains. These benchmarks are listed in Table I and their characteristics are described as follows:

- PR and MCM are two DSP kernels with pure additions/subtractions and multiplications.
- $CACHE$ is a cache controller implementation which is a pure control-intensive design with cycle-accurate I/O operations.
- $MOTION$ performs the motion compensation algorithm for the MPEG-1 decoder. This design has multiple branches and a modest amount of computations.
- $IDCT$ implements the inverse discrete cosine transform algorithm used in the JPEG standard, and DWT implements the discrete wavelet transform algorithm adopted in the JPEG2000 standard. These two benchmarks contain a large amount of computations and memory accesses.
- $EDGELOOP$ design is extracted from the H.264 decoder. It features a mix of computation, control branches, loops and memory accesses.

TABLE I
C vs. RTL VHDL code sizes

Design	C lines	VHDL lines	LE	Fmax(MHz)
PR	90	600	1349	178.7
MCM	161	1260	2402	152.6
CACHE	295	1277	371	161.6
MOTION	130	1200	888	161.2
IDCT	236	7388	9351	162.9
DWT	180	1371	1862	147.3
EDGELOOP	329	7296	7440	100.1

B. Advantage of Behavioral Synthesis: Code Size Reduction

In Table I the second and third columns report the comparison on code sizes before and after behavioral synthesis for the seven test cases. The area and frequency results reported by Quartus II are also shown in the last two columns.

On average, the code size of the synthesized RTL designs is about one order of magnitude larger than the corresponding C code. If we assume design complexity is proportional to the code line count, we can expect an over 10× reduction in design effort by raising to the behavioral level and applying our behavioral synthesis tool.

TABLE II
Design tradeoff in xPilot

Target Cycle time	State	Fmax MHz	Cycle	Latency (ns)	LE
9ns	34	123.56	4830	39.1	1777
7ns	36	147.28	5211	35.4	1862
5.5ns	51	183.62	6926	37.8	1926

C. Advantage of Behavioral Synthesis: Design Tradeoffs

One of the advantages offered by behavioral synthesis tools is their ability to explore design tradeoffs among several design metrics, such as latency, area, and frequency. Currently, xPilot accepts user-specified assignments of target frequency and optimization preference (speed or area). Table II shows a set of design points that xPilot generates for the DWT design. When we decrease the target cycle time from $9ns$ to $7ns$ to $5.5ns$, as expected, the resulting state numbers, execution cycle counts, and LE counts increase accordingly. In this case, the optimal latency appears on the second setting.

D. Comparison with SPARK

We further make comparisons with *SPARK* [7], a state-of-the-art academic high-level synthesis system.

TABLE III
xPilot Scheduling results compared with Spark

Benchmark	Spark		xPilot		Latency Improvement
	#States	LP	#States	LP	
MPEG2-dpframe	32	424	53	375	11.6%
GIMP-tiler	27	2234	42	1977	11.5%
ADPCM-decoder	15	327	12	251	23.2%
ADPCM-encoder	16	133	14	122	8.3%
Average					13.6%

Table III shows a comparison of scheduling results on four SPARK bechmarks. In this experiment a $10ns$ target cycle time is set for both xPilot and SPARK, and two-cycle multipliers and five-cycle dividers are used. On average, xPilot is 13.6% better in worst-case execution length (LP in the table).

Compared to SPARK's binding results, xPilot is more than 2× better on average in terms of frequency with comparable circuit area. The full consideration and detailed characterization of the target platform partly contribute to the significant improvement.

VI. xPilot for System-Level Synthesis

In this section we show that our xPilot system does not limit itself to the scope of pure hardware behavioral synthesis. In fact, xPilot can be adapted to accommodate both software and hardware elements, and its second focus is to provide efficiently platform-based system-level synthesis, especially for Field Programmable System-on-a-chip (FPSoC) platforms.

The recently emerged Field Programmable System-on-a-chip offers a promising platform for system-level design. Several FPGA manufactures [6] [8] have announced their FPSoC platforms which combines programmable fabric with one or several processors. These processors could be either soft

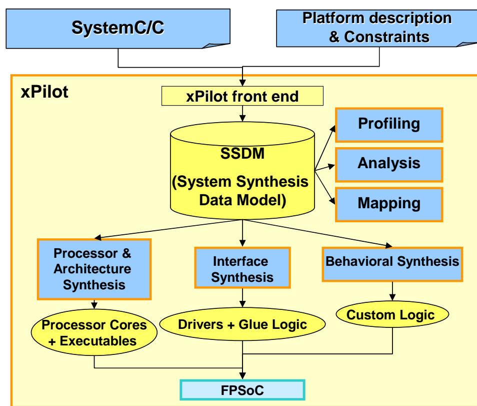

Fig. 2. xPilot system-level synthesis framework.

processors (e.g., Nios/NiosII from Altera and MicroBlaze from Xilinx) or hard core processors (e.g., PowerPC processors). The soft core processors have the capability to extend the base instruction set with a set of customized instructions. These instructions are implemented by the programmable fabrics as extra application specific function units in the data path. These augmented function units are used to exploit the instruction-level parallelism within the specific applications or application domains. Programmable fabric can be also loosely coupled with soft core processors as coprocessors. Users can offload more complicated tasks to the coprocessors to boost performance.

Figure 2 shows that the xPilot design flow as a complete system-level synthesis system — from the system-level specification to hardware implementation on an FPSoC platform. As mentioned in Section III, xPilot can handle system-level specification languages such as C or SystemC. After front end processing, the application specification is parsed into our SSDM, which is a concurrent process network. SSDM is able to accommodate both hardware and software components. More importantly, it can explicitly describe the process-level parallelism and inter-process communications. On top of this powerful data model, various analysis, simulation and profiling passes can be performed. We currently reply on the designers to manually partition the application into software and hardware based on the performance analysis.

Once the hardware/software partitioning is available, we will invoke the xPilot behavioral synthesis engine to compile the hardware portion of the design to the custom logics. Since the original design specification is written in a C-based language, it is relatively easy to generate the software code for embedded processors. To integrate the micro-processors and custom logics together, an interface synthesis is being developed to generate the software drivers and glue logics.

During the software code generation step, we also generate the application-specific instruction set for the extensible soft core processors. This is a difficult task to be managed by manual designs for large programs, and is further complicated by various micro-architectural constraints, such as the clock period, available chip area, etc. In our xPilot framework, a template generation, matching, and covering algorithm has been developed [9] to automatically identify and generate the custom instructions. In template generation, we generate all of the candidate instruction sets satisfying the architectural input/output constraints. Then we select a subset of candidates to maximize the potential speedup while satisfying the resource constraint. Finally it maps the application to the extended instruction set so that the total execution time is minimized. Application of our techniques on Altera NIOS processor has demonstrated an encouraging performance speedup (up to 3× in total latency).

REFERENCES

[1] D. Nadamuni, "ES Level Design: the View in 2004," June 2004, Dataquest's annual briefing.

[2] *International Technology Roadmap for Semiconductors*, http://public.itrs.net, 2003.

[3] K. Wakabayashi, "C-Based Behavioral Synthesis and Verification Analysis on Industrial Design Examples," in *Proceedings of the Asian and South Pacific Design Automation Conference*, January 2004, pp. 344–348.

[4] *SystemC Website*, http://www.systemc.org.

[5] *The LLVM Compiler Infrastructure*, http://llvm.cs.uiuc.edu.

[6] *Altera Website*, http://www.altera.com.

[7] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau, *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Springer, May 2004.

[8] *Xilinx Website*, http://www.xilinx.com. [Online]. Available: http://www.xilinx.com

[9] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-Specific Instruction Generation for Configurable Processor Architectures," in *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*, 2004, pp. 183–189.