Platform-Based Behavior-Level and System-Level Synthesis

[Preprint of the invited paper for 2006 IEEE SOC Conference]

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

***Abstract*— With the rapid increase of complexity in System-on-a-Chip (SoC) design, the electronic design automation (EDA) community is moving from RTL (Register Transfer Level) synthesis to behavioral-level and system-level synthesis. The needs of system-level verification and software/hardware co-design also prefer behavior-level executable specifications, such as C or SystemC. In this paper we present the platform-based synthesis system, named xPilot, being developed at UCLA. The first objective of xPilot is to provide novel behavioral synthesis capability for automatically generating efficient RTL code from a C or SystemC description for a given system platform and optimizing the logic, interconnects, performance, and power simultaneously. The second objective of xPilot is to provide a platform-based system-level synthesis capability, including both synthesis for application-specific configurable processors and heterogeneous multi-core systems. 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 relentless tracking of Moore's curve by the entire semiconductor industry has showcased the exponential scaling of the transistor feature size by a factor of 0.7 reduction every three years. This leads to exponentially increasing transistor counts and results in an explosive growth in functionality and the amount of computing power available on a single chip. Today it is perfectly feasible to design a System-on-a-Chip (SoC) with one billion transistors [1], and it is generally believed that industry will continue to overcome technical hurdles to sustain this trend for another decade. However, the cost of developing these chips and providing production facilities is also growing at a very fast pace. For instance, the total development cost of a single complex, high-density SoC at today's 90-*nm* technology can easily be in the $20 to $30 million range. The ITRS 2005 edition [1] has also emphasized that the cost of design remains the greatest threat to continuation of the semiconductor roadmap.

Unfortunately, the progress of design technologies lags behind that of process manufacturing technologies. The constantly improving CAD tools can help to mitigate the problem by delivering faster simulation, higher capacity formal verification, and better logic synthesis coupled with place-and-route. However, these improvements fail to close the design productivity gap, i.e., the number of available transistors grows faster than the ability to meaningfully design them.

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 [2] 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 behavior-level and system-level synthesis technologies that automatically synthesize high-level functional descriptions into optimized software/hardware implementations. We believe that behavior-level and system-level synthesis and optimizations are becoming imperative steps in EDA design flows. They provide the following combined advantages:

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 the code density (in terms of line counts) can be improved by nearly 10X when moved to the behavior level. In addition, behavior-level and system-level synthesis have the added value of allowing efficient reuse of soft functional/behavioral IPs, which are technology-independent and can be synthesized for different requirements.

Shorter verification/simulation cycle: System-level synthesis and optimizations allow the designers to start with a specification in a high-level programming language (HPL) such as C or SystemC [4] that is directly executable and simulatable with high speed (up to 1000X faster than RT-level simulation according to [3]). More importantly, behavioral synthesis automatically compiles the input descriptions into RTL code through a series of formal constructive transformations. This avoids the slow and error-prone manual process and simplifies the design verification and debugging effort.

Rapid system exploration: With the coexistence of microprocessors, DSPs, memories and custom logic on a single chip, more software elements are involved in the process of designing a modern embedded system. 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. HPL-based design methodologies (especially C-based designs) offer a promising solution to this problem. With the aid of behavior-level synthesis, the software programming languages 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 HPL descriptions and the automatically synthesized HDLs.

Higher quality of results: VLSI designs in current semi-

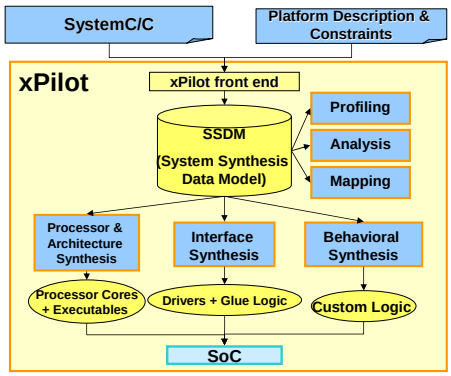

Fig. 1. xPilot system-level synthesis framework.

conductor technologies are limited by interconnect in both delay and power. However, since the interconnects are determined by downstream physical design tools, it is very difficult for designers to make accurate estimation at the RT level. To achieve timing and power closure, designers have to adjust the initial RTL in an ad hoc manner and iterate over the time-consuming synthesis and layout process. We believe that by integrating automatic high-level optimizations together with physical planning, we can optimize the logic and interconnects simultaneously and achieve higher quality of results (QoR).

In this paper we present the *xPilot* synthesis system being developed at UCLA. The goal of xPilot is to provide novel platform-based synthesis technologies to simultaneously optimize the logic, interconnects, performance, and power (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 highlight the behavioral synthesis engine, respectively. The preliminary experimental results are reported in Section V.

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 our front end complier. 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 compiler front end 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 system-level synthesis data model (SSDM)—the internal data model of xPilot.

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 one or more interfaces to capture certain communication protocols. Altogether, an SSDM defines a process network to model the concurrent behavior of a complex system. On top of this powerful data model, various analysis, simulation and profiling passes can be performed.

We currently rely 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 invoke the xPilot behavioral synthesis engine to compile the hardware portion of the design to the custom logics. Specifically, xPilot performs platform-based synthesis and 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 RT-level VHDL together with the associated constraint files (e.g., multicycle path constraints, physical location constraints, etc.) to leverage the existing logic synthesis and physical design toolset. Additionally, xPilot also generates RT-level SystemC code for fast co-simulation with other software/hardware modules specified in SystemC.

Since the original design specification is written in a C-based language, it is relatively straightforward to generate the software code for embedded processors. During the software code generation step, we also generate the application-specific instruction set for the extensible microprocessors. A pattern generation, selection, and covering algorithm has been developed [6] to automatically identify and generate the custom instructions. We are also developing an microprocessor-network synthesis and mapping algorithm for homogeneous and/or heterogeneous multiprocessor systems to exploit task-level parallelism and further speed up the software implementation.

In order to integrate the microprocessors and custom logics together, an interface synthesis module is being developed to generate the software drivers and glue logics.

III. xPilot Front End

xPilot accepts synthesizable C or SystemC as input. C language is effective for describing sequential behavior within one single process 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 concurrent modules.

Our front end compiler 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. On top of the LLVM intermediate representation, we first recover certain high-level programming constructs from the low-level virtual instruction set, then perform elaboration to extract the processes, ports, channels and their interconnection topologies, and construct our SSDM accordingly based on this information.

Several analysis and optimization passes will be performed during this stage, including traditional compilation techniques

such as dead code elimination, and hardware-specific passes such as bitwidth analysis. We observe that the bitwidth analysis can be very beneficial for many designs, since the operations and variables can use the function units and the registers with the minimal widths.

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 operand 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 are also collected.

IV. Behavioral Synthesis Engine

In this section we will highlight the xPilot behavioral synthesis engine, including scheduling and resource binding.

A. Scheduling

Scheduling, which exploits parallelism in the behavior-level design and determines the time at which different computations and communications are performed, is recognized as one of the most important problems in behavioral synthesis. However, the existing scheduling techniques either have limited efficiency in a specific class of application or lack general support of various design constraints. To address the above deficiencies, We have developed a new scheduling algorithm, which consists of three main steps.

First, we convert a rich set of scheduling constraints into a system of difference constraints. In particular, for dependency constraints, we can exactly model data dependencies and control dependencies. For timing constraints, we can precisely transform the frequency constraint, relative I/O timing constraints, and latency constraints. Since resource-constrained scheduling problem is intractable in general, we heuristically convert the resource constraints into difference constraints. Using this formulation, the consistency of the constraint system can be checked efficiently by solving a single-source shortest path problem.

Second, we express the scheduling objective as a linear function so that the global optimization can be performed by linear programming. In addition, the matrix formed by the constraint equations has a special property that guarantees integral solution. Under this unified mathematical framework, we can apply a variety of powerful optimizations by reformulating the objective function. Specifically, we can generalize As-Soon-As-Possible and As-Late-As-Possible schedules, and optimize the worst-case longest-path latency, the expected average-case latency, and the overall slack distribution.

Third, after we obtain the integer results by the LP solver, we can directly translate this solution into an actual schedule in STG representation.

Experiments on a set of real-life behavioral designs show that our scheduler achieves 15% shorter latency on average when compared to the scheduler of SPARK [7], a state-of-the-art academic high-level synthesis system. The technical details of our scheduling algorithm are discussed in [8].

B. Resource Binding

In the resource binding stage, we focus on the optimization of interconnects and multiplexors to improve the quality of results using a distributed register-file microarchitecture.

At the RT level, large number of discrete registers may result in wide multiplexors and a complicated interconnect structure. To address this problem, register files can be used, which replace the multiplexors with the dedicated decoding logic. However, due to the limited number of read and write ports, a centralized register file may not provide sufficient bandwidth for the applications with multiple concurrent data reads and writes. Distributed register files are required for such applications. The usage of distributed register files is further encouraged for modern FPGA and Structured ASIC platforms, which have rich on-chip distributed memory IP blocks.

We propose a distributed register-file microarchitecture (DRFM), which consists of multiple islands. Each island contains a local register file, a functional unit pool, and data-routing logic. In DRFM, each register file allows a variable number of read ports but only a fixed number (typically one) of write ports. The DRFM-based resource binding tries to minimize the inter-island connections. This will simplify the data-routing logic in each island and reduce the overall complexity of the resulting datapath. Our heuristic approach uses a weighted bipartite matching algorithm to gradually bind operation set to the target DRFM. Each binding step handles a set of compatible operations and binds them onto different islands, with the objective to minimize the amount of newly introduced inter-island connections.

Our DRFM study on the Virtex-II FPGAs shows more than 2X logic area reduction when compared to the traditional discrete-register-based approach [7], with a faster clock period (27% improvement on average). The technical details of our resource binding algorithm are available in [9].

V. Experimental results

The xPilot system is implemented in a C++/Linux environment. In this paper we report the results of targeting the field-programmable SoC platforms (e.g., Altera [10] and Xilinx [11] FPGAs).

A. Test Examples

We tested xPilot hardware synthesis through several real-life behavioral designs from different application domains. As listed in Table I, *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. The *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 xPilot 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. In this experiment we target the Altera Stratix FPGAs [10] using Quartus II v4.2 as the downstream RTL synthesis and physical design tool.

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 more than a $10\times$ reduction in design effort by rising to the behavioral level and applying our behavioral synthesis tool.

C. Advantage of xPilot Synthesis: Design Tradeoffs

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

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 ($7ns$).

D. Advantage of xPilot Synthesis: Shorter Simulation Cycle

We also performed experiments on the *DWT* design to show the efficiency of high-level verification/simulation. In fact, running this module in C specification for 100,000 rounds takes only one second, while the Modelsim v6.0 [12] simulation on the RTL VHDL generated by xPilot using the same input stimulus takes about 2300 seconds.

E. Advantage of xPilot Synthesis: Faster Design Exploration

As one case study on system-level design, we have implemented a Motion-JPEG encoder on a Xilinx XUP Virtex-II Pro development board [11]. The original C design is obtained from the UC Berkeley Metropolis group [13].

In this experiment we evaluated the performance of two alternative design implementations. The first model uses five MicroBlaze soft-core processors to implement the five major modules of Motion-JPEG encoder, i.e., preprocessing, DCT, quantization, Huffman encoding, and table modification. These MicroBlazes communicate through FIFOs and execute on the incoming image blocks in a pipelined fashion. In the second model, we use xPilot to synthesize the software DCT in C into hardware VHDLs and replace the corresponding MicroBlaze with our own custom hardware DCT block. After the hardware synthesis process, which only takes a few seconds, we are able to obtain a 38% faster implementation using the second model. In fact, the hardware DCT module itself is at least 6X faster than the software counterpart, but the overall speedup is limited by the I/O bandwidth.

F. Advantage of xPilot Synthesis: Better QoR

We made comparisons with *SPARK* [7] on our benchmark suite targeting Xilinx Virtex-II FPGAs. After scheduling and resource binding, xPilot achieves nearly 40% better performance (in final latency) with $2\times$ area reduction on average. The full consideration and detailed characterization of the target platform partly contribute to the significant improvement.

ACKNOWLEDGMENTS

This research is supported by Semiconductor Research Corporation, Gigascale Silicon Research Center, National Science Foundation, and grants from Altera Corporation, Magma Design Automation, Inc., and Xilinx, Inc. under the California MICRO program.

REFERENCES

[1] Semiconductor Industry Association, "International Technology Roadmap for Semiconductors, 2005 Edition."
[2] D. Nadamuni, "ES Level Design: the View in 2004," Jun. 2004, Dataquest's annual briefing.
[3] K. Wakabayashi, "C-Based Behavioral Synthesis and Verification Analysis on Industrial Design Examples," in *Proc. ASPDAC*, Jan. 2004, pp. 344–348.
[4] *SystemC Website*, http://www.systemc.org.
[5] *The LLVM Compiler Infrastructure*, http://llvm.cs.uiuc.edu.
[6] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-Specific Instruction Generation for Configurable Processor Architectures," in *Proc. International Symposium on FPGAs*, 2004, pp. 183–189.
[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] J. Cong and Z. Zhang, "An Efficient and Versatile Scheduling Algorithm Based On SDC Formulation," 2006, under review.
[9] J. Cong and Y. Fan, "Resource Binding on a Distributed Register-File Microarchitecture," 2006, under review.
[10] *Altera Website*, http://www.altera.com.
[11] *Xilinx Website*, http://www.xilinx.com.
[12] *Mentor Graphics Website*, http://www.mentor.com.
[13] *Metropolis Website*, http://www.gigascale.org/metropolis.