

AUTONOMOUS COMPUTING SYSTEMS: A PROOF-OF-CONCEPT

Neil Steiner and Peter Athanas

Bradley Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061
email: neil.steiner@vt.edu, athanas@vt.edu

ABSTRACT

This paper describes a proof-of-concept implementation of a basic *autonomous computing system*. The system consists of an XUP Virtex-IIPro board running Linux and a set of software tools on one of the embedded PowerPC processors. A server application accepts mapped EDIF circuit descriptions, which it independently parses, packs, places, routes, configures, connects, and implements within itself, while continuing to run, and without making use of the Xilinx ISE tools. It is also able to remove circuits and/or modify top-level connections at will. The system has complete autonomy in how and where to place and route the circuits: it understands *all* of the logic in the device, and uses true routing and not simply a slot-based architecture.

1. INTRODUCTION

Autonomous computing systems [1] assume a large amount of responsibility for their resources and operation. They absorb much of their own complexity within themselves, and are therefore able to present a simpler interface to their environment. Furthermore they are able to respond to changes in themselves or in their environments with little or no outside intervention.

There are many motivating factors for the development of autonomous systems, but one common theme is the continued increase in complexity of modern systems. That complexity results in significant difficulties, ranging from designability to defect-free manufacturing, but it also provides the basis for some solutions.

Systems are now approaching the point where they can begin to function as active participants rather than passive blocks. With respect to designability, the systems can provide infrastructure and functionality necessary for more abstract *component*-based design, and can handle the implementation details by themselves. With respect to manufacturing defects, they can simply mask out defective areas at

very fine granularities, and avoid those areas when placing and routing components. The systems can also respond dynamically by adjusting to external requests or changing conditions.

Autonomous computing systems are not without drawbacks. They incur area overheads from the autonomous infrastructure, and performance penalties from being implemented on top of configurable logic. They require incremental and embeddable versions of implementation tools. Testing systems that are no longer identical or static likewise becomes enormously more complex. Similar kinds of objections and issues arose with assemblers, compilers, operating systems, and object-oriented languages at one time or another, but those tools are now used extensively, and their benefits are unquestioned.

The proposed shift to autonomy is ambitious, but likely to become a necessity at some sufficiently elevated point on the complexity curve. This work uses a proof-of-concept implementation that supports basic autonomy, to stretch the present limit of technical feasibility.

2. BACKGROUND

The proof-of-concept implementation depends upon an entire system with numerous requirements. The system must generate its own configuration bitstreams, but it must also support full placement and routing. Some of the necessary software tools depend upon the availability of a file system and the ability to run a complete Java Virtual Machine (JVM), and the intended client-server communication with the system makes most sense with ethernet-based TCP/IP support. These various requirements collectively make a strong case for the use of Linux. The following subsections describe the underlying hardware and software components of the system.

2.1. Hardware: XUP Board

The demonstration platform is the XUP Virtex-IIPro board [2], designed by Xilinx Research Labs and available from

This work is graciously supported through a Bradley Fellowship from the Via Foundation.

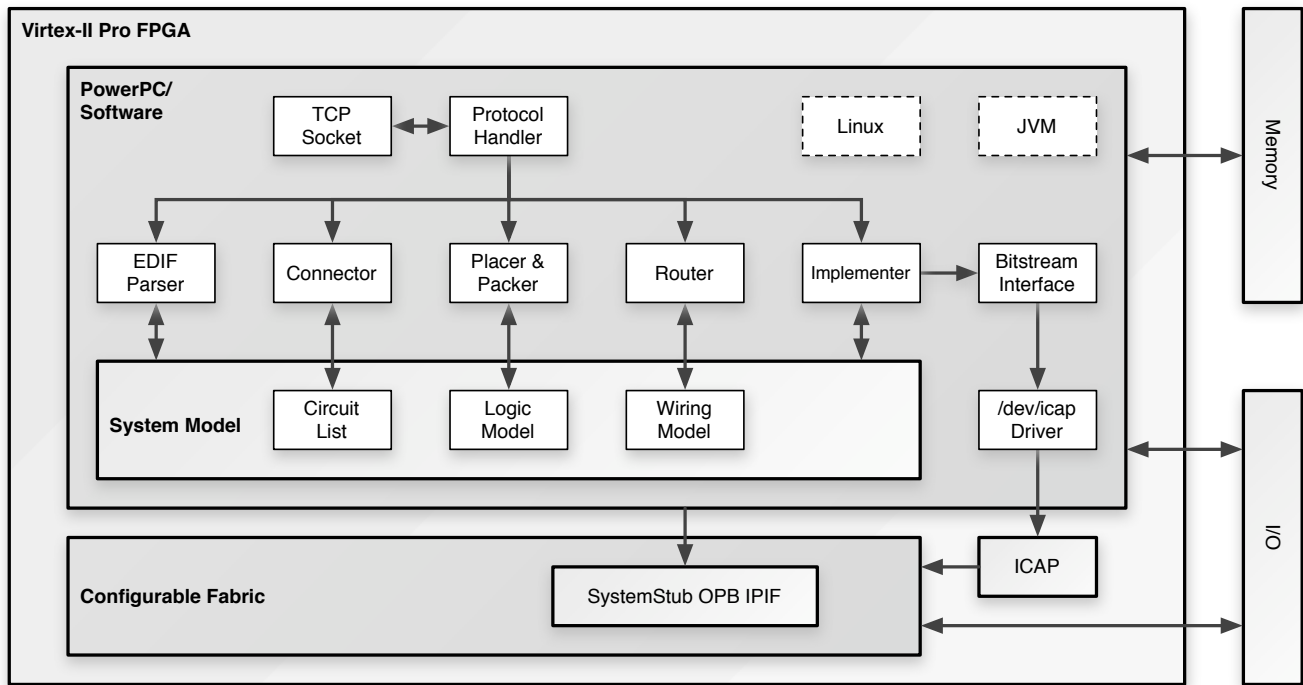


Fig. 1. System block diagram

Digilent. The board includes a number of features that make it suitable for this demonstration, including the proven ability to run Linux.

The XUP board contains a Virtex-II Pro XC2VP30 [3] FPGA, as well as RS-232, ethernet, and CompactFlash ports, and the obligatory push-buttons, DIP switches, and LEDs. This particular board was also populated with a 1 GB Microdrive, and with 512 MB of DDR DRAM, which respectively provided a file system and sufficient memory.

In addition to its configurable fabric, the FPGA also contains an Internal Configuration Access Port (ICAP), that allows it to reconfigure itself while in operation.

2.2. Software: Linux

Linux is a robust, well-known, and freely available operating system. For the purposes of this work, it provides the ability to run complex multi-threaded software, and also provides stable driver, interrupt, and networking support.

This work uses XUP-related Linux resources provided by the EMPART project at the University of Washington (UW) [4], the Linux on FPGA project at Brigham Young University (BYU) [5], and others. The kernel is version 2.4.26. The gcc cross-compiler tools were built with Dan Kegel's *Crosstool*. The root file system was built with Wolfgang Klingauf's *mkrootfs*, relying on *busybox* for the basic utilities. Added into the mix were *OpenSSH* and various other utilities.

2.3. Software: Java

Some of the software tools that compose the system depend upon a Java2 1.4 or later runtime environment. That requirement was somewhat more difficult than expected to satisfy, because embedded JVMs that support the PowerPC 405 are not very common. The one solution that finally worked was Java SE for Embedded [6], available from Sun Microsystems. Although the software was obtained under an evaluation license, Sun kindly agreed to extend the license for the duration of this research, contingent upon non-commercial use.

2.4. Software: ADB

This project depends upon the ability to route real circuits in FPGAs, and it obtains that ability from prior work: ADB is a wire database that provides exhaustive support for Virtex, Virtex-E, Virtex-II, Virtex-II Pro, and other families [7]. It is able to route, trace, or unroute any net in a device, and it relies on an abstract bitstream interface to generate configuration bitstreams.

2.5. Software: Logic and Bitstream API

The ability to manipulate device logic and generate configuration bitstreams is provided by proprietary software, developed elsewhere by the primary author and licensed back for

use under a non-disclosure agreement. This software is conceptually similar to JBits [8], but provides broader family support and a simpler more generalized interface that follows the Xilinx ISE naming and conventions.

3. DESIGN

The proof-of-concept system must instantiate and remove circuits at will, and connect or reconnect or disconnect them, as requested by a remote client. This makes for a highly dynamic system, and probably marks the first instance of a hardware system needing to maintain a detailed non-static model of itself.

A block diagram of the system is provided in Figure 1. The top of the hierarchy depicts the FPGA connected to memory and I/O. The FPGA itself encompasses the PowerPC, the configurable logic, and the ICAP. The server software listens to client requests on a TCP socket, and processes them according to a simple protocol. These requests are handled by appropriate subsystems, which work in close conjunction with the system model. System changes to be implemented in the configurable logic are sent through the bitstream interface and the ICAP, at which point the reconfiguration takes place. The configurable fabric also contains the SystemStub interface that makes it possible for implemented circuitry to talk to the system.

The system hardware design was done with the Xilinx Embedded Development Kit (EDK). Information gleaned from the UW and BYU instructions made it fairly straightforward to develop working hardware and boot Linux. Any deviations consisted mostly of enabling or disabling existing peripherals, and adding custom peripherals.

3.1. Software: Linux ICAP Driver

The ICAP device is connected to the system's On-Chip Peripheral Bus (OPB), and must be memory-mapped into the processor's address space before it can be used. A Linux device driver was written to perform this remapping, and to provide ICAP access to the user. The driver supports file operations—allowing applications or even shell scripts to access the device as `/dev/icap`—and parses bitstream headers before sending the bitstreams to the ICAP device. This driver builds on top of the low-level driver available with the Xilinx EDK [9]. It can be used for both full and partial bitstreams, in active or inactive modes.

3.2. Software: Placer

In spite of significant quantities of research published on FPGA placement, all of the publicly available tools support only simplified abstractions of FPGA devices, and typically assume a uniform 2D grid of logic cells. But newer architectures deviate more and more from that uniform abstraction,

and the Virtex-II Pro architecture with its 27 different types of logic cells, is decidedly not uniform.

In practice these non-uniformities make available placers virtually useless, if one desires to support the full device functionality. Because placement is a necessary but non-central part of this work, the decision was made to implement a simple *simulated annealing* algorithm [10].

3.3. Hardware: SystemStub

Dynamically implemented circuits can be connected to any unused device IOBs, but it is also desirable to be able to interface them with the system. A stub was developed to that end. The stub is based on an OPB IP Interface (IPIF) core with interrupt support, which means that it is accessible through the OPB bus, and exposed to software applications as `/dev/hwstub` by a Linux device driver.

4. OPERATION

The system boots Linux upon startup. When the server software is invoked, it begins by initializing its internal model of the hardware system. This includes loading the wiring and logic databases, which tell it about all of the system resources, and then tracing the current configuration bitstream, which allows it to identify all of the nets and determine which logic cells are in use. Knowledge of the system resources will allow the server to correctly place and route circuits, or to unroute connections when applicable. And knowledge of the resource usage will allow it to modify itself without corrupting itself. When the initialization is complete, the server creates a TCP socket and waits for client connections.

4.1. System Interface

The base system hardware that exists before any other circuits are instantiated, is itself treated as a circuit, and is included within the system's model of itself in order to expose ports that user circuits can connect to: 1) All of the FPGA I/O pins are exposed, making it possible for circuits to interact with the rest of the board. 2) The SystemStub IPIF ports are exposed, making it possible for circuits to tie in to the OPB bus, and potentially interact with user software. 3) All of the global clock buffer outputs are exposed, making it possible for circuits to tie in to any clock signal.

4.2. Protocol Operation

Clients can interact with the system through a simple text-based protocol:

ls: Lists the circuits in the system model, including the system circuit itself. Information about resource usage is included.

connect: Makes arbitrary connections between circuits, including system ports. Supports bus syntax. Transparently extends existing nets when applicable.

disconnect: Removes connections between circuits, including system ports. Supports bus syntax. Transparently trims branches—without removing the entire net—when appropriate.

parse: Parses an EDIF stream from the client. The circuit is tagged with a name supplied by the client and added to the model.

place: Places and packs a named circuit.

route: Routes a named circuit.

configure: Configures the logic settings for a named circuit.

rm: Removes a named circuit and disconnects it from other circuits as appropriate.

reconfigure: Writes all modified configuration frames into a partial bitstream, and reconfigures itself through the ICAP.

exit: Closes the client connection and waits for new connections.

Circuits can be added at any time with a parse command. Any circuit that has been parsed can be placed, although it may be desirable to first specify connections between circuits, as that may influence the placement. Any circuit that has been placed can be routed and/or configured. Any circuit other than the system circuit can be removed. Connections can be added or removed from any circuit ports, including system ports. And pending changes can be applied to the FPGA at any time with a reconfigure command.

4.3. System Connections

Circuit instantiation and connection requests that are sent to the server cause the configurable fabric to be reconfigured appropriately. Figure 2 shows a sample of how some circuits might be connected to each other and to the system circuit ports—IOBs, clocks, and SystemStub OPB IPIF.

5. CONCLUSION

This work provides preliminary proof that *autonomous computing systems* are viable, and demonstrates that basic hardware autonomy can be implemented with current technology. The implementation critically depends upon the authors' prior work, as no other embeddable router or logic and bitstream API exist for the Virtex-II Pro architecture.

The described system can reconfigure itself in response to client requests, by implementing or removing mapped EDIF circuits, and dynamically adding, removing, or changing connections between the circuits. This is believed to be the first instance of non-trivial placement, routing, bitstream generation, and reconfiguration of arbitrary circuits being entirely carried out from within a system while it is running.

The proposed roadmap for autonomy [1] stretches far beyond these initial steps. Many additional tools and capabilities are required in order to achieve full autonomy, and much work will be necessary to boost those tools to levels of commercial maturity and completeness.

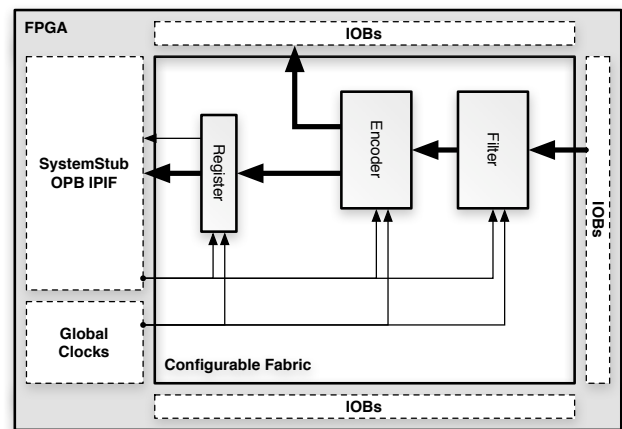


Fig. 2. System interface and sample connections

The authors wish to acknowledge Xilinx, Inc., for their kind support, and Sun Microsystems, Inc., for permitting extended use of their embedded Java software.

6. REFERENCES

- [1] N. Steiner and P. Athanas, "Autonomous computing systems: A proposed roadmap," in *Proceedings of the 2007 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2007.
- [2] *Xilinx University Program Virtex-II Pro Development System: Hardware Reference Manual*, Xilinx, Inc., March 2006, UG069 (v1.0).
- [3] *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, Xilinx, Inc., March 2005, UG012 (v4.0).
- [4] "EMPART," University of Washington, http://www.cs.washington.edu/research/lis/empart/xup_ppc_linux.shtml.
- [5] "Linux on FPGA," Brigham Young University, <http://splish.ee.byu.edu/projects/LinuxFPGA/configuring.htm>.
- [6] "Java SE Embedded Use," Sun Microsystems, Inc., <http://java.sun.com/javase/embedded>.
- [7] N. Steiner and P. Athanas, "An alternate wire database for Xilinx FPGAs," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004.
- [8] S. Guccione, D. Levi, and P. Sundararajan, "JBits: Java based interface for reconfigurable computing," in *Proceedings of the Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999.
- [9] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan, "A self-reconfiguring platform," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL)*, 2003.
- [10] N. Sherwani, *Algorithms for VLSI Physical Design Automation*, 3rd ed. Boston: Kluwer Academic Publishers, 1999.