International Parallel and Distributed Processing Symposium (IPDPS):
Workshop on Next Generation Software (NSF-NGS);
Santa Fe, NM, April 26, 2004, Paper: W10-NSFNGS-13.

Liquid Architecture [*]

Phillip Jones, Shobana Padmanabhan, Daniel Rymarz, John Maschmeyer
David V. Schuehler, John W. Lockwood, and Ron K. Cytron

Department of Computer Science and Engineering
Washington University, St. Louis
Contact email: cytron@acm.org

January 22, 2004

Abstract

We present an implementation of a liquid-architecture system that supports efficient development, prototyping, and performance evaluation of custom architectures. The implementation integrates the LEON soft-core, SPARC-compatible processor into the Field-programmable Port Extender (FPX). The resulting platform can be instantiated, configured, and executed via the Internet.

1 Introduction

For many years, research in the programming systems community has examined how software tools—especially the compiler—can and should adapt to advances and evolution in computer architecture. Thanks to advances in fabrication technology, the chip-area occupied by a standard processor has decreased dramatically. As a result, architects are faced with the exciting challenges of putting newly available chip area to beneficial use.

One possibility that has surfaced is the notion of *liquid architecture*, through which the instruction set, the co-processors, and the supporting structures such as cache, pipelines, and memory controllers can be dynamically reconfigured to increase the performance of a given application. With this paradigm shift, future research in architecture and software systems must extend beyond adapting applications to a rigid architecture; instead, research must address how best to design and configure liquid architecture elements so as to improve program performance.

[*]Sponsored by National Science Foundation under grant ITR–0313203.

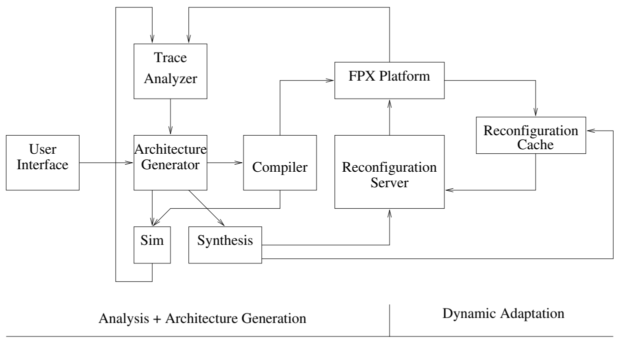


Figure 1: Application reconfigurability environment [1]. To the left is shown a loop by which the architecture is improved in response to trace information pulled off the FPX. To the right is shown automatic reconfiguration at runtime, where components that have been pre-generated are selected from the reconfiguration cache.

We are currently experimenting with an approach based on precompiled FPGA images for many points in a configuration space. At runtime, an application can be dynamically optimized by reconfiguring the FPGA to use a different precompiled image. For example, there are many cache configurations, and some are better suited to a particular application than others. The application's performance can be improved by reconfiguring the hardware to use a cache scheme or alternative memory structure (such as a prefetch unit) better tailored to the application.

Figure 1 shows our view of an environment that supports application migration to reconfigurable platforms.

Trace Analyzer: Execution traces are analyzed to identify candidate portions of an application whose performance could be improved through reconfigurability.

Architecture Generator: The applications developer explores reconfigurability options. Reconfiguration then consists of the following two components:

- Architecture reconfiguration is specified by a VHDL-like program that causes customized structures to be deployed in reconfigurable logic, such as FPGA devices.
- A recipe for rewriting the application is specified, so that the application can take advantage of the reconfigured architecture. In Figure 1, that recipe is provided to the compiler so that the application's instructions can be tailored for the architecture.

Compiler: The recipe provided for rewriting a program's instructions is provided to the compiler, which can then generate code appropriate for the reconfigured architecture.

Sim: Based on the reconfigured architecture and the automatically rewritten application, simulation can provide additional instruction traces to assist the developer in evaluating the effectiveness of the current configuration.

Synthesis: This component is responsible for realizing the reconfigured architecture in an FPGA (or more generally, in any device that can be the target of hardware synthesis, such as ASIC). For our purposes, this component processes VHDL and emits FPGA layouts of the liquid architecture.

FPX Platform and Reconfiguration Server: We have already developed a prototype of a reconfigurable node. We call this prototype the "FPX" and it is shown in Figure 2.

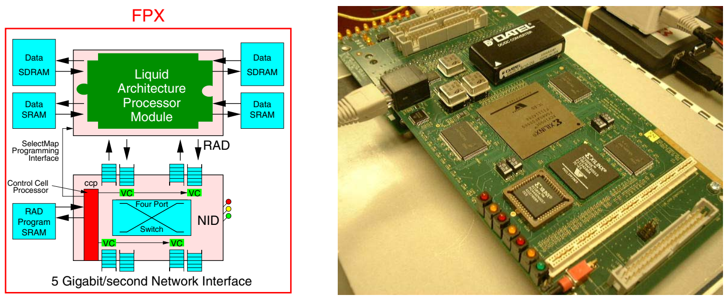
Each FPX node offers connection to a high-speed network as well as reasonable amounts of high-speed and high-density storage. The high-speed network facilitates the installation of FPGA descriptions as well as the streaming of instrumented traces to the Trace Analyzer. The Reconfiguration Server controls access to the FPX Platform, sequencing the loading and execution of applications.

Reconfiguration Cache: As features are identified for reconfiguration, instances of those features are pregenerated in the user- or application-defined parameter space. Each such instance requires $\sim$1 hour to synthesize, and the results are captured in the reconfiguration cache. At runtime, an application can switch between these pre-generated modules to improve performance [2].

In this paper, we report on the current status of our work: the creation of a hardware and software environment for compiling and deploying a suite of FPGA images that vary in some design dimension. The LEON2 processor core, developed by the ESA (European Space Agency) is a synthesizable SPARCV8 compliant microprocessor implemented in VHDL93 [3]. This core is capable of FPGA and ASIC synthesis and is designed for numerous SoC (System on a Chip) applications. This project implements a testbed to be used for the prototyping and evaluation of extensions to the LEON soft-core. Some examples of what these extensions could be are:

- Modifiable pipeline depth
- Variable instruction/data cache size
- Specialized hardware to accelerate frequently used instructions or instruction sequences
- New instructions to the SPARC base instruction set

To do this, we ported the LEON processor to operate inside of the RAD (Reconfigurable Application Device) of the WUSTL (Washington University at St. Louis) FPX (Field-programmable Port eXtender) [4]. The LEON



(a) (b)

Figure 2: Reconfigurable node: (a) logical view; (b) photo of prototype

processor includes several internal modules for integration including simple serial controllers, an interrupt controller, discrete output ports and two internal main buses: an AMBA AHB high-performance bus for memory and processor cache interface and an AMA APB low bandwidth I/O bus for serial, timer and other devices. Figure 3 illustrates the layout of the LEON embedded system.

The Field Programmable Port Extender platform was used to test the Liquid processor system. The FPX is an open hardware platform that includes two multi-gigabit-per-second network interfaces and dynamically reconfigurable hardware that can be reprogrammed over a network [5]. To implement this, a bitfile for the circuit was uploaded into the Virtex XCV2000E on the FPX for in-system testing [6].

2 Approach

2.1 Liquid Processor System

Figure 3 shows the high level architecture of the Liquid processor system. The Layered Protocol Wrappers were developed by Washington University's Reconfigurable Network Group [7]. The entity is responsible for correctly formatting incoming/outgoing network traffic. The Control Packet Processor (CPP) is responsible for routing internet traffic that contains LEON specific packets (command codes) to the LEON controller (leon_ctrl). The leon_ctrl entity uses these command codes to direct the LEON processor (Restart, Execute), and read and write the contents of the external memory that the LEON

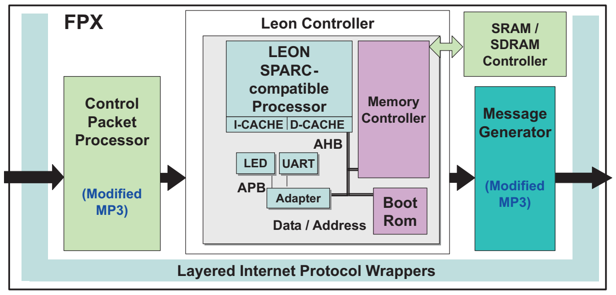


Figure 3: High-level architecture of the Liquid processor system

processor uses for instruction and data storage. Finally, the Packet Generator is used to send IP packets in response to receiving a subset of the command codes (e.g. Read Memory, LEON status).

This system leverages the underlying FPX architecture to allow future research in evaluating extensions of the LEON processor and remote control of experiments via the internet.

2.2 System Requirements

The two major goals of the Liquid processor are to (1) provide the ability to reconfigure features of the processor hardware, and (2) allow remote interaction with the memory and control of the CPU. The following were required to develop the Liquid processor system:

- FPX platform
- LEON processor base system
- Memory Interface (currently using SRAM)
- Cross compiler environment
- Control software

2.3 LEON Processor System

Figure 3 illustrates some of the main components of the LEON processor base system [8]. As can be seen in Figure 3, the LEON processor system provides fairly sophisticated computer architecture model. It has many features, such as instruction and data caches, the full SPARC V8 instruction set, and separate buses for high speed memory access and low speed peripheral control.

For inclusion in the Liquid processor system, it was necessary to modify portions of the LEON processor system to interface with the FPX platform. The interface modifications to the standard LEON processor include:

- Memory controller was modified to interface with the FPX SDRAM controller.
- Boot ROM was changed to have the LEON processor being execution of user code out of the FPX platform s RAM.

All other components necessary to implement the interface between the LEON processor system, RAM, and User were implemented outside of the base LEON processor system. They are discussed in Section 3.

2.4 SDRAM Interface

The LEON processor comes packaged with a complete memory interface [3] . This includes a programmable ROM, SRAM, SDRAM, and memory-mapped IO devices. To support this, we use the FPX SDRAM controller [9]. This controller has several benefits as compared to the SDRAM controller bundled with the LEON processor. First, it has been designed and tested for use with the FPX. The FPX SDRAM controller provides an arbitrated interface with support for up to three modules. This arbitration allows simultaneous use by both the LEON processor and the network control components on the FPX. The FPX SDRAM controller also provides high performance accesses to SDRAM. Support is provided fir sequential bursts, for both read and write operations, with burst lengths up to 256 64-bit words.

The LEON processor uses the Advanced Microcontroller Bus Architecture (AMBA) to connect the processor core to its peripheral devices [10]. The AMBA Advanced Peripheral Bus (APB) is used to connect to low-power peripherals. The AMBA High-Performance Bus (AHB) is used as the backbone bus, connecting the processor, the memory system, and a bridge to the APB bus. In order to allow for high performing memory, a direct connection to this AMBA AHB is needed.

Through simulation, it can be seen that the LEON processor does not make full use of the AMBA AHB protocol. An AHB slave must be able to implement 8 separate bursting modes in data sizes of up to 1024 bits. The LEON processor, however, only makes use of the two modes of bursting: simple mode and incrementing. In addition, all data sizes are less than 32 bits. Split transfers are also unused. These features allow for some simplifications in the design of an AHB slave for use in the LEON processor. The AHB protocol also allows each slave to delay is results indefinitely. This allows the device to process each request sequentially without performance concerns. To implement a high performing memory adapter, however, it is necessary to make use of burst transfers.

2.5 Cross Compiler

We used the ”LECCS-1.1.5.3, Windows/cygwin” distribution from Gaisler Research [8] which is based on gcc-2.9.5.2. A memory map is extracted from the design of our supervisory state machine, which controls reset and memory decoding for LEON. The memory map information can be included either in link command files fed to LD (the loader) or on the LD command line, which is what we do in the batch file.

2.6 Control Software

The web-based control software provides an interface to load compiled instructions over the internet into LEON’s memory.

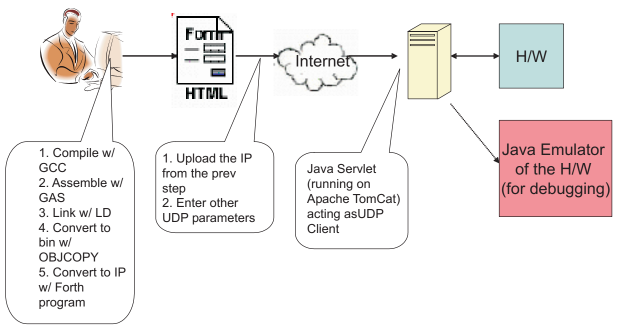

Form
HTML
Internet
H/W
Java Emulator of the H/W (for debugging)
1. Compile w/ GCC
2. Assemble w/ GAS
3. Link w/ LD
4. Convert to bin w/ OBJCOPY
5. Convert to IP w/ Forth program
1. Upload the IP from the prev step
2. Enter other UDP parameters
Java Servlet (running on Apache TomCat) acting asUDP Client

Figure 4: Components of the control software

The different components of the software system are shown in Figure 4.

A web interface is provided for the user to submit a request. This request is received by a Java servlet running on an Apache TomCat server. The servlet creates UDP (IP) control packets and sends them to the FPGA at a specified destination IP and port. A dedicated Java program running in a different thread on the control software server listens continuously for UDP packets transmitted by FPGA and displays them on the console as they arrive. A Java program was used to emulate the hardware while the hardware was being developed.

Communication with the FPGA is in the form of UDP control packets. All control packets carry an IP header, UDP header and a payload specific to the command. The different commands supported currently are:

- LEON status - to check if LEON has started up.
- Load program - to load a program into LEON.
- Start LEON - to instruct LEON to execute the program that was loaded in the previous step.
- Read memory - to read the result that was generated by LEON in step 3.

In order to identify these commands uniquely and efficiently in the VHDL state machine, a unique code, called command code, is assigned to each command listed above. In addition to the command code, following commands also have a payload associated with them:

- Load program:
  - Total data length (1B)
  - Packet sequence length (2B)
  - Memory address (4B) where the program needs to be loaded
  - Assembly instructions or compiled C or C++ program, in binary format. If the binary does not fit in 1 packet, they can be sent as multiple packets and the packet sequence number listed above will need to used to mark the order (as UDP protocol does not guarantee order of delivery). If the program is shorter than the UDP packet length (which will be identified using the length mentioned above), the remaining bytes would be ignored.
- Read memory:
  - Memory address (4B) where the result is expected.

3 Implementation

3.1 LEON Processor implementation on FPX

In integrating the LEON processor into the FPX, two main issues needed to be addressed:

- Defining the location in main memory where the LEON processor should begin execution of user code
- Arbitrating access to memory between the LEON processor and the ”User”.

The location in memory of the LEON processor was defined by modifying the LEON processors Boot ROM. In our implementation, programs are sent to the FPX via UDP packets, then written directly to main memory (SRAM).

In order to indicate to the LEON processor that a valid program is in its main memory, where this program is located, and when to start execution, the following was done:

- Modify the default Boot ROM, instead of waiting for an UART event, poll a specified main memory location (0x4000_000). See Figure 5.

- Have an external circuit between the LEON processor system and main memory that can disconnect the LEON processor from main memory (i.e. always drive 0s on the LEON processor's data bus). See Figure 6 .

The above ROM modification and external circuitry work together as follows in order to appropriately direct the LEON processor. The external circuitry monitors the LEON processor's progress through its Boot ROM, by probing LEON's address and data bus. Once it detects that the LEON processor is about to enter its main memory polling routine, it disconnects LEON from main memory until it finds a non-zero value at address location 0x4000_0000. This address location is used to give the LEON processor the starting address of a program loaded by the user. While the LEON processor is disconnected from main memory, the external circuitry allows the user to load a program and sets address location 0x4000_0000 to the starting address of the user's program. Finally the external circuitry reconnects the LEON processor to main memory. Once the polling routine reads the non-zero value at location 0x4000_000, it jumps to this memory location and begins execution of the user's program. The last instruction in the user program instructs the LEON processor to jump back to its polling loop. On detection of the LEON processor returning to its polling routine, the external circuitry disconnects the LEON processor from main memory until a new program is loaded to main memory or the user sends a command to re-execute a program already loaded in main memory.

3.2 SDRAM

To connect the FPX SDRAM controller directly to the AMBA AHB bus, the existing memory controller found in the LEON processor was replaced. The adapter used to replace the LEON memory controller bridged together the desired response of an AMBA AHB busslave and the handshaking of the FPX SDRAM controller. A finite state machine was used to implement both of these.

The major hurdles in the implementation are the difference in bus size between the AMBA AHB bus and the FPX SDRAM controller, and the handling of high performance features. The AMBA AHB bus uses a word size of 32 bits while the FPX SDRAM controller uses 64. Read

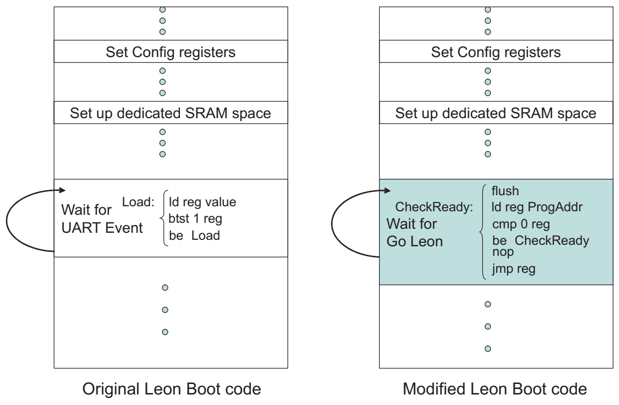


Figure 5: Original and modified LEON Boot code

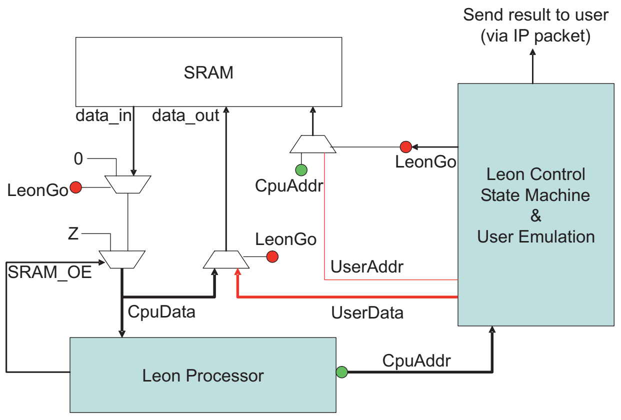


Figure 6: External circuitry for disconnecting LEON for main memory

requests from the LEON processor are handled buy selecting the appropriate 32-bit word from the 64 bits of data. This does, however, waste a significant amount of memory bandwidth. Write requests cause a more significant problem. To maximize the efficiency or memory. two 32-bit words must be stored at each 64-bit address. Since 64 bits must be written at a time, the controller must first read the entire contents of the memory address. modify the appropriate 32 bits, and then rewrite the data. This requires two separate handshakes for each write request, significantly impairing performance.

In order to gain performance, the memory controller must support burst transfers. Since the FPX SDRAM controller cannot support non-sequential burst transfers, a separate handshake must be performed for each operation using single mode bursting. In sequential mode, however, a performance boost can be gained. The FPX SDRAM controller required the burst length before the transfer takes place. On the other hand, the AMBA AHB bus supports burst transfers of an unspecified length. This incompatibility limits the effectiveness of bursting. Simulations have shown that the LEON processor uses burst transfers of length less than or equal to 4 words.To gain a good deal of performance, the controller was designed to always use a short burst when reading to guarantee a sequential burst of up to 4 32-bit words. Only a couple of cycles are wasted when the burst length is shorter, but a significant amount of time is gained buys avoiding additional handshakes for 4-word bursts. Sequential bursts that require more than 4 32-bit words will required at least one additional handshake. Since burst lengths are unknown ahead of time, write burst are not allowed. This will keep memory integrity intact.

4 Results

One of the many benefits of a liquid architecture is the ability to customize cache configuration according to application performance requirements. To demonstrate this, we changed the data cache size between 1KB and 16KB while keeping the cache line size constant at 32B and the instruction cache size constant at 1KB. A simple C program was developed to access a 4KB array under these cache configurations. This software configuration was chosen to generate cache misses when data cache size is below 4KB. A hardware state machine counts and returns the number of clock cycles to run this program. The code snippet corresponding to the array access is shown in Figure 7.

```
_start() {
  for (i=0; i < 100000; i=i+32)
  {
    address = i % 1024;
    x = count[address];
  }
}
```

Figure 7: Array access code snippet

Data Cache Size	Number of clock cycles
1KB	140897
2KB	140902
4KB	116158
16KB	116198

Figure 8: Array access running time

The average running time under different data cache sizes is summarized in Figure 8.

Figure 9 illustrates the data shown in Figure 8 in graphical form.

This clearly shows that there are no cache misses (excluding the initial loading of the cache) once the cache size reaches 4KB.

4.1 Testing on the FPX Platform

Hardware debug was accomplished by inserting error states into the state machine that implemented the external circuitry discussed in Section 3.1. If the circuit entered one of these error states, then, an output IP packet containing an error message would be transmitted to the network. Figure 2 shows the FPX platform that was used for implementation.

4.2 Device Utilization

The results after place and route for the synthesized Liquid Processor System in the Xilinx Virtex XCV2000E are

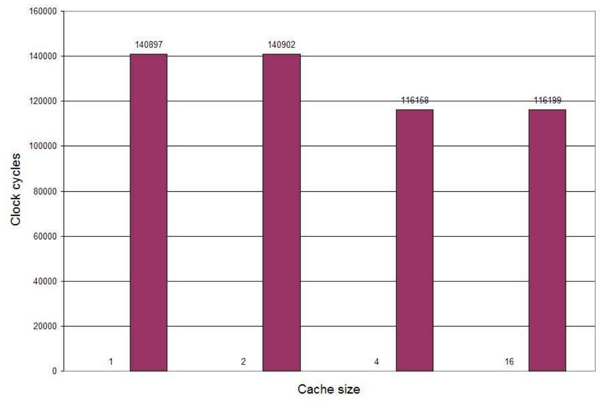


Figure 9: Average running time under different cache sizes

Resources	Device Utilization	Utilization %
Logic Slices	7900 of 19200	41%
BlockRAMs	54 of 160	54%
External IOBs	309 of 512	60%
Frequency	30 MHz	NA

Figure 10: Liquid Processor System Statistics

listed in Figure 10. The core logic occupied 41% of the logic and 54% of the block RAMs. The system has been synthesized at 30 MHz.

5 Conclusion and Future Work

A base system for performing research in prototyping extensions to the LEON processor system in an interactive remote manner was implemented. A flow has been set up to convert C programs into a specific format to be loaded into a UDP packet payload. Control software has been developed to interact with the leon_ctrl hardware via the internet. Control hardware has been developed to direct the actions of the LEON processor system. Finally, a SDRAM interface is in development that will aid in loading an OS, such as Linux, into the Liquid processor platform.

References

[1] J. W. Lockwood, "Evolvable Internet hardware platforms," in *The Third NASA/DoD Workshop on Evolvable Hardware (EH'2001)*, pp. 271–279, July 2001.

[2] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour, "Dynamic hardware plugins in an FPGA with partial run-time reconfiguration," in *Design Automation Conference (DAC)*, (New Orleans, LA), June 2002.

[3] "Leon specification." `http://www.gaisler.com-/doc/leon2-1.0.21-xst.pdf`, 2003.

[4] "Field Programmable Port Extender Homepage." Online `http://www.arl.wustl.edu/arl/projects-/fpx/`, Aug. 2000.

[5] T. Sproull, J. W. Lockwood, and D. E. Taylor, "Control and configuration software for a reconfigurable networking hardware platform," in *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, (Napa, CA), Apr. 2002.

[6] J. W. Lockwood, J. S. Turner, and D. E. Taylor, "Field programmable port extender (FPX) for distributed routing and queuing," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000)*, (Monterey, CA, USA), pp. 137–144, Feb. 2000.

[7] F. Braun, J. Lockwood, and M. Waldvogel, "Protocol wrappers for layered network packet processing in reconfigurable hardware," *IEEE Micro*, vol. 22, pp. 66–74, Jan. 2002.

[8] "Gaisler research." `http://www.gaisler.com`.

[9] S. Dharmapurikar and J. Lockwood, "Synthesizable design of a multi-module memory controller." Washington University, Department of Computer Science, Technical Report WUCS-01-26, Oct. 2001.

[10] "Amba specification." `http://www.gaisler.com-/doc/amba.pdf`, 2003.