

*Review of:*  
Peer-to-peer Hardware-software Interfaces for  
Reconfigurable Fabrics

- Paper by:
  - Mihai Badiu, Mahim Mishra, Ashwin R. Bharambe, and Seth Copen Goldstein (Carnegie Mellon University)
- Published in:
  - IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)
  - April 2002
- Review by:
  - James Moscola

## Introduction

- Motivation for the work
  - Problem: Reconfigurable logic devices aren't used on a wide scale
    - Reason: because they are difficult to integrate into a system
  - Solution: Create an interface between the hardware and software to make integration easier.
    - the proposed interface will follow similar methods of Remote Procedure Calls (RPCs)
      - hardware-independent
      - software-independent

## A Hardware-Software Interface

- computation is mapped to the CPU or the RH at the procedural granularity
- code is called by either the CPU or the RH by using regular procedure calls
- the CPU and the RH should both be able to request services from each other (no master-slave relationship)
- calls should be invoked in the same manner independent of where they are actually implemented

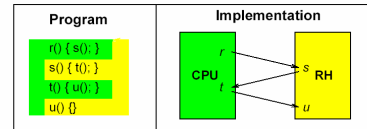


Figure 1. Sample program and a legal partitioning.

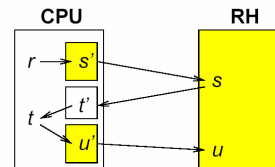


Figure 2. Implementation of the example in Figure 1. The primed boxes are stubs for the respective procedures, i.e.  $s'$  is a stub for  $s$ . Stubs mediate the low-level communication but otherwise look like ordinary procedures.

## What is a stub?

- a local procedure call that takes the same arguments as the remote procedure it represents
- a hardware-dependent procedure that takes care of all the low-level communication over the CPU-RH interface
- stubs require the following to maintain successful communication over the interface
  - a mechanism to send data from the CPU to the RH
    - sends procedure arguments when calling RH functions
    - returns values to RH callers
  - a mechanism to retrieve data from the RH
    - returns values from RH procedures
    - receive arguments from procedures invoked on the RH
  - a mechanism to select which procedure to invoke on the RH
  - a mechanism to select which procedure is being invoked by the RH on the CPU

## Advantages of the Proposed Interface

- treating the RH as a peer to the CPU, as opposed to a slave, increases the percentage of code that can be mapped to the RH
- the interface is simple and clean; all the low-level details of the interface are left unspecified
- it decouples the development of the two parts of the application in a precise way making it easy for them to be developed independently
- the interface offers portability of the software among various RH architectures
- program partitioning algorithms search at a procedure level granularity, dramatically reducing the search-space
- the interface can be dynamically changed during run-time based on performance (assuming a procedure has both implementations)
- much of the tedious work for interfacing the CPU and RH can be automated

## Example CPU-RH Architecture

- built on top of a simulated computer architecture
  - 4-wide issue superscalar processor using the MIPS instruction set architecture (ISA)
  - ISA was extended to add the following RH-specific instructions
    - rh\_input R1, R2, R3, R4: sends four integer-register values to the RH inputs
    - rh\_output R1, R2, R3: reads into integer registers three values from the RH output
    - rh\_start R: starts execution of the k<sup>th</sup> procedure loaded on the RH, k is the content of R
    - rh\_load R: loads the binary configuration for the k<sup>th</sup> procedure into the RH, k is the content of R
    - rh\_cont: reads an address from RH and branches to it

## Example of CPU-RH Architecture (cont...)

- Three example stubs from the example implementation

```

rh_input(...)
rh_start(s)
repeat:                                call RH proc S
    rh_cont;
    goto repeat;

done_cont:
    o = rh_output;    return continuation;
    return o;

call_T_cont:
    Targ = rh_output(...);
    r = call t(Targ);    call CPU proc T
    rh_input(r);
    rh_start(ret_from_CPU);
    goto repeat;
    
```

Figure 3. Implementation of three stubs on our system: a stub for calling RH procedures from the CPU, a stub returning control from the CPU to an RH caller, and a stub calling a procedure on the CPU from the RH.

- The toolflow for the CPU-RH architecture

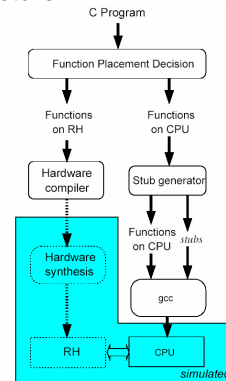


Figure 4. Toolflow for compilation of applications on mixed hardware/software systems. The dotted-line components are not implemented. The hardware-compiler is in development.

## Program Coverage for the CPU-RH Architecture

- An analysis was done on the architecture using SpecInt95 and MediaBench to determine the percentage of application code that can be placed on the RH
- the following graphs show the following
  - the bottom part of the bars represents the coverage when all local variables on RH must be allocated to registers (i.e. there are no stack frames on RH)
  - the middle part of the bars represents the coverage when RH procedures use statically allocated stack frames (i.e. does not support recursion but can pass addresses of locals to other procedures)
  - the top part of the bars represents the coverage when using arbitrary stack frames for RH procedures
  - L, R, U
    - L: only leaf procedures can be placed on the RH
    - R: RH procedures can call other RH procedures, but not CPU procedures
    - U: any procedure can be mapped to RH

## Program Coverage: SpecInt95

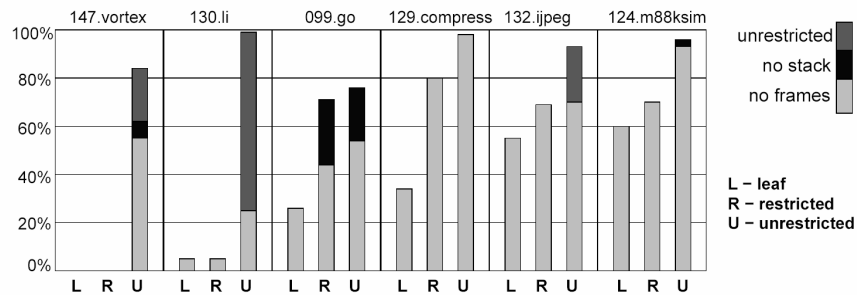


Figure 5. Dynamic program coverage as a function of the RH/CPU interface and RH architectural constraints for SpecInt programs, assuming RH can implement FP operations. Bars marked L are coverage with only leaf functions on the RH, those marked R are with RH functions that can call other RH functions, and those marked U are with RH functions that can invoke CPU routines.

## Program Coverage: MediaBench

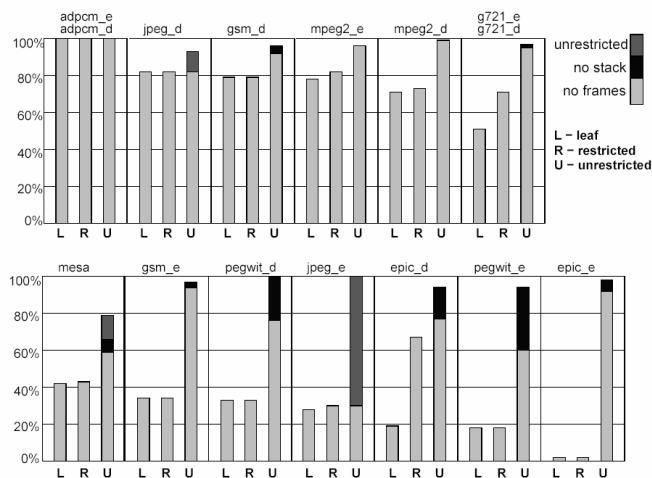


Figure 6. Dynamic program coverage as a function of the RH/CPU interface and RH architectural constraints for MediaBench programs, assuming RH can implement FP operations. The bars have the same meaning as in Figure 5

## Program Coverage: Analysis

- Less than half of the graphs spend more than 50% of their execution time in leaf procedures
  - therefore, if RH was only capable of executing procedures it would not obtain substantial speed-ups
- if RH is capable of executing RH and CPU procedures, the greatest gain can be obtained, hence showing how the peer-to-peer relationship is beneficial

## Stub Generation Overhead

- Overhead was estimated by counting the number of instructions executed when performing a remote invocation
  - the cost of a remote invocation includes constructing the stack frame, transferring control and returning
  - the cost of a stub includes moving arguments into registers, making the appropriate call, and returning the result

## Stub Generation Overhead (cont ...)

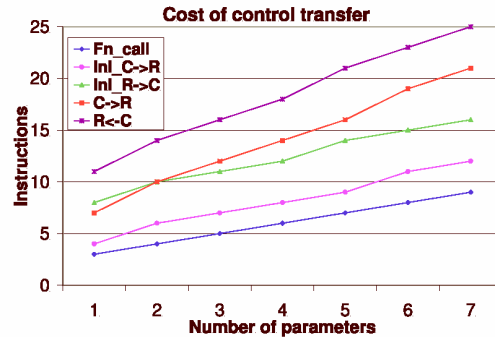


Figure 8. Cost of various control-transfer methods, as function of number of arguments. In order, from bottom to top: CPU→CPU call, CPU→RH with inlined stub, RH→CPU with inlined stub, CPU→RH, RH→CPU. The cost is measured in instructions and is only for the CPU side.

## Conclusion

- slick implementation of a hardware/software interface
- good analysis
- would like to see it in a real system, not just in simulation
- how many people would actually develop programs that are based partially in software and partially in hardware?