Peer-to-peer Hardware-software Interfaces for Reconfigurable Fabrics

Mihai Budiu, Mahim Mishra, Ashwin R. Bharambe and Seth Copen Goldstein
{mihaib,mahim,ashu,seth}@cs.cmu.edu
Computer Science Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA-15213

Abstract

In this paper we describe a peer-to-peer interface between processor cores and reconfigurable fabrics. The main advantage of the peer-to-peer model is that it greatly expands the scope of application for reconfigurable computing and hence its potential benefits. The primary extension in our model is that "code" on the reconfigurable hardware unit is allowed to invoke routines both on the reconfigurable unit itself and on the fixed logic processor. We describe the software constructs and compilation mechanisms needed for such an architecture, including a detailed description of the interface between the two parts of the application.

1 Introduction

Reconfigurable hardware (RH) devices have been reported to provide spectacular computational performance on a variety of applications [6]. Despite this and a wealth of other potential advantages, RH devices aren't used on a wide scale, especially in general-purpose computing systems. Several reasons can be cited for the lack of success in their adoption by the industry. Perhaps the major problem with RH devices is the difficulty of integrating them into a system at all levels: for the published and implemented systems, electrical, physical and software interfaces are generally ad-hoc and custom-designed. The lack of interface standardization increases costs, prolongs system development and complicates the task of software development.

This paper proposes a partial solution to the interface problem, addressing the software layer. We argue that RH devices should be integrated in a computing system not as subordinates of the processor, but as equal peers. Moreover, we propose a procedural interface between software on the processor and the RH, in the style of Remote Procedure Calls [13]. Processor-executed programs should be able to invoke code on the RH device in the same way they invoke library functions; RH-based code should also be able to call code on the processor.

Our proposal is not a panacea for solving the problem of hardware-software partitioning: we are proposing here a *mechanism* and not a *policy* for how the two sides of an application should interface. However, we believe that the choice of a good interface is extremely important for unleashing the full potential of a new computing paradigm. Witness the success of interfaces such as libraries, system calls, remote procedure calls and sockets.

1.1 Contributions of this work

The following aspects are novel contributions of this paper:

- We describe (Section 2) a hardware-independent, language-independent hardware-software interface similar to remote procedure calls, which can be used between the code executed on a processor and the code executed on a RH device.
- We propose to treat the processor and RH devices as equal peers in the process of computation, instead of treating the RH as a slave to the processor.
- We describe (Section 3) how a compiler can automatically generate the stubs for interfacing the CPU and the RH device.
- We analyze (Section 4.1) realistic pointer-based high-level language programs and estimate, as a function of architectural constraints, how much of the computation can be assigned to the RH devices when using our interfacing scheme.

2 A Hardware-Software Interface

The computing system used throughout this paper contains both a conventional processor (CPU) and a reconfigurable hardware (RH) device. The RH device is reprogrammable under software control. This paper describes a proposal for a high-level interface between the code running on the processor and on the reconfigurable hardware.

In this paper we mainly focus on single-threaded applications. We do not study parallel applications, which run simultaneously on both computation engines. However, our proposal is not incompatible with multi-threading, and is easily adapted to handle parallel applications.

The application domain under study consists of integer-based desktop and media-processing programs written in high-level languages, containing pointer-intensive code. We analyze programs from the SpecInt95 [18] and MediaBench [10] benchmark suites to determine the effectiveness of the implementation we describe. While these applications are implemented in C, the interface we present is language-independent, and, moreover, can be used even if the two parts of the application are developed using different languages and tools.

Our proposal entails the following aspects:

- The computation is mapped to the CPU or RH at the procedure granularity.
- Code is invoked from either the CPU or RH by using regular procedure calls. When a call crosses the CPU-RH boundary, it is implemented in a way similar to a remote procedure call.
- The CPU and the RH device should both be able to request services from the other side. From this point of view, the two computing devices behave like peers, without a clear master-slave relationship between them. In practice, the actual implementation may have some limitations (for example, the RH device may not be re-entrant), but the RH is assigned a more important role than in traditional architectures.
- A call should appear to be invoked in the same way, independent of where the implementation actually resides; in other words, a software program should invoke a computation on the RH in exactly the same manner it invokes a computation on the CPU.

Figure 1 displays a legal invocation sequence under our proposal.

In the following section we discuss how remote service invocation is implemented on the CPU side. Because our proposal is hardware-independent, we do not describe how procedure calls (local or remote) are implemented on the RH side.

2.1 Stubs

The way hardware-independent service invocation is accomplished is similar to the technique used in implementing

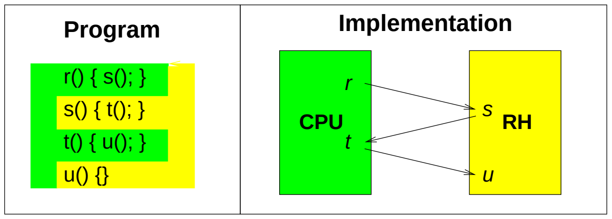


Figure 1. Sample program and a legal partitioning.

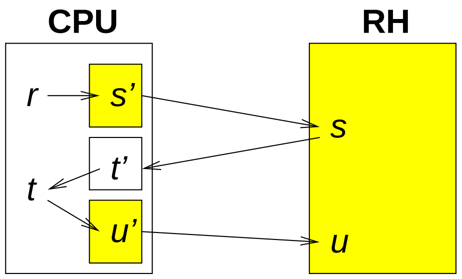


Figure 2. Implementation of the example in Figure 1. The primed boxes are stubs for the respective procedures, i.e. s' is a stub for s. Stubs mediate the low-level communication but otherwise look like ordinary procedures.

remote procedure calls: instead of calling a remote procedure directly, a local *stub* procedure is called, with the same arguments, and using the local calling conventions. The stub procedure implementation is hardware-dependent and takes care of all low-level communication, by marshaling the arguments and invoking the remote service.

Stubs mediate calls crossing the CPU-RH boundary originating from either side. Each procedure residing on the RH which is invoked from the CPU has a stub, and each procedure on the CPU called from the RH has a stub. Figure 2 shows how the example in Figure 1 is implemented.

A stub requires the existence of several simple, low-level, hardware-dependent mechanisms to accomplish its task:

- A mechanism is needed to send data from the CPU to the RH. This mechanism is used to send procedure arguments when calling RH functions (e.g., s' calls s in Figure 2) and to return values when returning to RH callers (e.g., t' returns to s);
- A second mechanism is needed for the CPU to retrieve data from the RH. This mechanism is used to return values from RH procedures (e.g., s returns to s') and to receive arguments for procedures invoked from the

RH (e.g., s calls t');

- There must exist a method to select which procedure to invoke on the RH, because multiple procedures may reside simultaneously on the RH (e.g., is s or u called by r?);
- The RH must be able to indicate the identity of a CPU procedure for implementing calls originating on the RH (e.g., does s call r or t?);

In Section 3 we describe precisely one prototype stub implementation in terms of a particular (simulated) architecture. We quantify the overhead of the stub-based scheme in Section 4.2.

2.2 Discussion

The proposed interface has several advantages over the current state-of-the-art approaches:

- The treatment of RH as an equal peer to the CPU greatly increases the percentage of code which can be mapped to the RH, as we show in Section 4.1. The restriction imposed by many approaches, of mapping only self-contained code having no external procedure calls on the RH, severely restricts the hardware/software partitioning choices.
- The interface is simple and clean, having a well-understood semantics.
- Such an interface decouples the development of the two parts of the application in a precise way: the code executed on the processor and the RH configuration can be independently developed.
- This type of interface offers portability of the software among various RH architectures. The view provided by the RH to the software layer is always the same, independent of the actual details of the hardware implementation and hardware capabilities. Moreover, development of applications is substantially eased: the initial implementation is customarily done entirely in software; when migrating to a mixed CPU+RH, the software side remains completely unchanged, and the required stubs can be automatically generated by a compiler.
- The search-space of the program partitioning algorithm (hardware/software partitioning) is dramatically reduced: procedures are considered as atomic units to be mapped to RH. If more fine-tuning is desired, the programmer (or even an automatic compiler) can control the position of the interface by decomposing the application into procedures in a suitable way.
- The exact details of the low-level interface between the CPU and RH are left unspecified. Our interface is adaptable enough to handle all major paradigms proposed in the literature: memory-based communication, bus-based, coprocessor-style and even datapath-integrated reconfigurable functional units.
- The hardware/software interface can even be dynamically changed during program run-time. The caller of a procedure doesn't have any knowledge whether the actual procedure resides in hardware or software; the calling sequence is always the same. The compiler can generate more complicated stubs which at run-time decide, based for example on performance monitoring, whether to steer the actual execution to a software- or hardware-side implementation of a procedure, if a procedure has both implementations.
- Finally, a lot of the tedious work for interfacing the CPU and RH can be automated. As we show in the rest of this paper, the generation of the low-level stub interfaces can be automatically done by a compiler, once the program partitioning is known.

However, our solution is not universally applicable: we can envision situations where an RPC-like interface is unsuitable, because it is too heavyweight or doesn't match the semantics of the underlying computational model. An important example is the custom-instruction model, which has been explored in prior work, e.g., [8, 14]: under this model, a single instruction simultaneously sends the input data (usually from the CPU registers), starts the computation and collects the result(s). This invocation model is orthogonal to the procedure-call model, and the two can coexist in a single architecture. The custom-instruction model however is mostly applicable to relatively small computations, because the instruction size does not provide enough room to encode many inputs/outputs.

We believe that our proposal has wide enough applicability, and that its usefulness will only increase with time. We present here a set of assumptions which led us to propose this interface:

- Moore's law will hold for at least the next five years, continuing to grow the amount of available hardware resources at an exponential pace. As a consequence, we expect that larger reconfigurable hardware devices will be built, and that multi-million-gate devices will be affordable enough to be included in common computer systems. The computing-system model we have in mind contains one or several general-purpose processors and a large (by today's standards) amount of reconfigurable hardware. Large devices will provide enough hardware resources to migrate whole procedures, if not whole applications into RH.
- RH devices are beneficial mostly on compute-intensive parts of the application. We expect that, with adequate

compiler support, most, if not all, of the compute-intensive kernels of an application will be executed on the RH. The processor will continue to be the sole choice for handling "odd jobs", such as the operating system, virtual memory, resource management and arbitration, and RH configuration management.

- As a direct consequence of Amdahl's law, moving just small pieces of code on the RH enables only modest speed-ups. The much touted high performance of RH devices is due to the massive parallelism (including pipeline parallelism) they provide, and also partly due to the application-specific customizations they enable. To obtain large speed-up, the dynamic coverage of the code has to be high, and the total overhead of the CPU-RH invocations has to be low. Mapping only small code fragments onto the RH implies either low coverage or very frequent CPU-RH crossings.
- An important consequence of the fact that RH devices are expected to execute a substantial portion of the application is that the RH device must have a way to access the CPU-side of the application address-space. Partitioning the code, while a difficult task, is a much simpler task than partitioning the data of the application, especially for pointer-based code. We cannot thus expect to statically separate perfectly the data accessed by the RH from the data accessed by the CPU; thus, run-time mechanisms will be needed to allow the RH to access data on the CPU side.
- Finally, a very important motivation for our proposal is the observation that in production-quality software there are few leaf functions. Most program functions call library functions, either for basic operations, or, very commonly, for error handling. If we restrict the selection for RH to functions doing pure computation there will be very few choices for what can be placed on the RH. The RH has to be able to invoke services from the processor if we want to move large parts of the computation to the RH.

The importance of these last two observations has been noticed before by researchers in the Garp project [9, 3]; these constraints have fundamentally affected their architecture and compiler algorithms. The definition of a formal interface between the hardware and software layers is however missing from their proposals.

3 An Example Peer-to-peer CPU-RH Architecture

One of the merits of the interface we propose is that it is architecture independent, and thus it can be built on top of (almost) any low-level hardware-software interface. To illustrate this, we describe a sample implementation on top of a simulated computer architecture, comprising a superscalar processor and a tightly-coupled reconfigurable hardware unit.

The implementation we describe here is built on top of an extended SimpleScalar [2] out-of-order simulator, which simulates a processor and an associated RH fabric. The CPU is a 4-wide issue superscalar processor using the MIPS instruction set architecture (ISA). We have extended the ISA with the following RH-specific instructions:

`rh_input R1, R2, R3, R4:` sends four integer register[1] values to the RH inputs; if more than 4 values need to be sent (for instance to invoke a procedure with more than four scalar arguments), several `rh_input` instructions are used in sequence. For sending fewer values, the zero-constant `R0` is used. The four values are deposited in a queue inside the RH, from where they are extracted by the configuration that will be executed next[2].

`rh_output R1, R2, R3:` reads into integer registers three values from the RH output; the details are the same as for `rh_input`.

`rh_start R:` starts the execution of the k-th procedure loaded on the RH, where k is the content of register `R`.

`rh_load R:` loads the binary configuration describing the k-th procedure into the RH, where k is the content of register `R`.

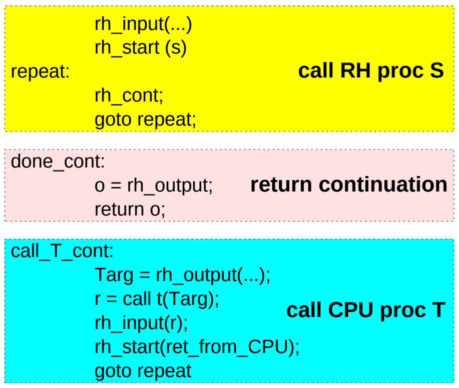
`rh_cont:` reads one address from the RH and branches to it. If the RH hasn't finished execution yet, this instruction behaves like a no-operation. When the RH terminates execution, it sends to the CPU the address of a *continuation* procedure, to which `rh_cont` branches.

The RH can generate virtual addresses in the entire application address space (globals, heap, stack) and can access the corresponding memory locations for reading or writing. The reads and writes of the RH are sent to the CPU, which injects them in the load-store queue used to parallelize memory accesses. In this way memory coherence between CPU and RH is ensured. In our implementation the load-store queue is the only CPU architectural feature accessible from the RH (i.e., no registers can be accessed). Moreover, the above instructions constitute the complete set of mechanisms which the CPU can use to control the RH.[3].

[1]The current implementation does not support passing floating-point inputs to a RH procedure.

[2]An alternative choice would be to encode a procedure identifier in the `rh_input` instruction.

[3]We are considering adding a second non-coherent memory interface to the RH, in the style of Garp [9], which can for instance be used for decoupled execution [17], which has been proven to be extremely beneficial to streaming-data applications (see e.g., [7, 9])



```
         rh_input(...)
         rh_start (s)
repeat:
         rh_cont;
         goto repeat;

done_cont:
         o = rh_output;
         return o;

call_T_cont:
         Targ = rh_output(...);
         r = call t(Targ);
         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.

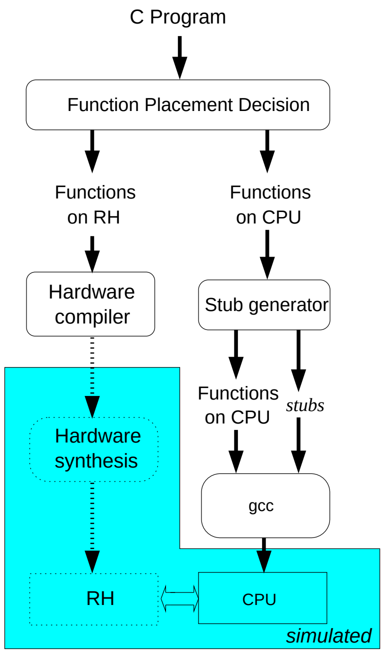


Figure 4. Toolflow for compilation of applications on mixed hardware/software systems. The dotted-line components are not implemented. The hardware-compiler is in development.

The way configurations are encoded and manipulated is not explicitly represented in our simulator. The RH is tightly coupled to the CPU in the sense that the `rh_input`, `rh_output`, and `rh_start` instructions can each be executed in a single clock cycle; also, the RH can inject memory operations into the processor load-store queue in zero clock cycles. As the size of the RH fabric grows, we should expect RH–CPU communication to take longer and longer, so these latency values may have to be revised.

Instructions dealing with the RH are never executed speculatively by the processor; before issuing such an instruction the CPU waits for all preceding branches to be validated. Because the RH instructions depend on each other, two RH operations cannot be executed in parallel; the RH invocations are strictly sequential and non-speculative.

Using these building blocks, a pseudo-assembly-language implementation of sample stub structures is given in Figure 3.

Figure 4 illustrates our toolflow. The input programs are un-annotated C programs. Stub generation is straightforward given information about the function type.

We have used the simulation infrastructure to validate the correctness of our stub generator. In the next section we present data about the effectiveness of our approach in partitioning the application between the CPU and RH. The results in this paper are based mostly on static information; in the future we plan to use the described simulation infrastructure in order to collect performance numbers.

4 Experimental Results

We present two classes of results in this section:

- We evaluate how restrictions on the RH computational capabilities influence the amount of program computation which can be mapped to the RH.
- We quantify the overhead introduced by the stubs.

4.1 Program Coverage

Here we examine how the CPU–RH interface impacts program coverage, that is, the percentage of the application code that can be placed on the RH. We analyze programs from the SpecInt 95 [18] and MediaBench [10] benchmark suites. We obtain the coverage percentage for a procedure by profiling the program; the coverage of a set of procedures is the sum of their individual running times. Based on the capabilities of the hardware, the achievable coverage depends on a number of orthogonal dimensions:

- Implementation of floating point operations: given their complexity, floating point operations take up a large amount of RH resources, and mapping them to the RH could be prohibitively expensive.
- Mapping of non-leaf procedures to the RH: previous approaches could map only leaves to the RH. In our proposal, RH procedures may call other RH procedures or even procedures on the CPU.

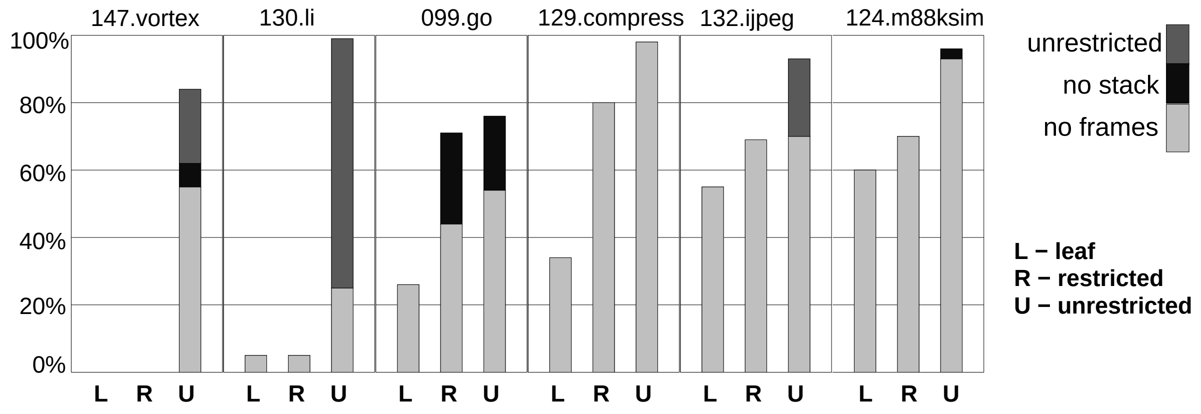


Figure 5. Dynamic program coverage as a function of the RH/CPU interface and RH architectural constraints for SpecInt programs, assuming RH can implement FP operations. Bars marked L are coverage with only leaf functions on the RH, those marked R are with RH functions that can call other RH functions,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

- Recursion: if the RH-mapped procedures are recursive, the RH needs to have a stack for local variables; otherwise the locals can be statically allocated.
- Local variable accessibility: can a procedure on the RH pass the address of a local variable to other procedures? If not, the RH local variables can all be allocated in registers over their entire lifetime.
- Size of the RH: not enough computational elements may be available for the whole computation.

To obtain coverage figures by varying parameters along each of these dimensions, we generated the following information for each benchmark:

- Per-procedure execution time, using profiling.
- Information about the presence or absence of floating point operations in each procedure.
- A statically built, conservatively approximated call-graph[4].
- Per-procedure information about whether it passes pointers to local variables in function calls.
- Estimated size in bit-operations for each procedure, when implemented on the RH. The bit-operation count was generated by counting operations of various types in each procedure and multiplying the count with the estimated size for each of these operations. This is a rough estimate since it does not account for RH interconnects, but is useful in approximating how much of an application can be mapped to RH given size restrictions.

[4]We could not build the call-graph for some benchmarks, which are thus not reported in this section.

Figures 5, 6 and 7 show the coverage as a function of the various restrictions. For each bar we have three potential sections: the bottom part of the bars represents the coverage when all local variables on RH must allocated to registers, (i.e. there are no stack frames for RH); the middle bar represents the case when RH procedures use statically allocated stack frames (i.e, do not support recursion but can pass addresses of locals to other procedures), while the top part allows the use of arbitrary stack frames for implementing RH procedures. The RH size is set to one million bit-operations. The three bars represent the following:

- **L:** only leaf procedures can be placed on the RH
- **R:** RH procedures can call other RH procedures, but not procedures on the CPU.
- **U:** any procedure can be mapped to RH.

4.1.1 Discussion

Figure 5 presents the coverage for SpecInt95 programs, with an unlimited RH size. The rightmost bars do not always reach 100% because of two reasons:

- We included timing information only for procedures which took up more than 1% of the program execution time;
- Some benchmarks had a significant proportion of their execution time attributable to library routines (e.g., 20% in mesa). We assume these procedures can never be placed on the RH.

As mentioned above, we consider three different ways of implementing local variables in RH procedures, corresponding to the three stacked bars for each coverage value.

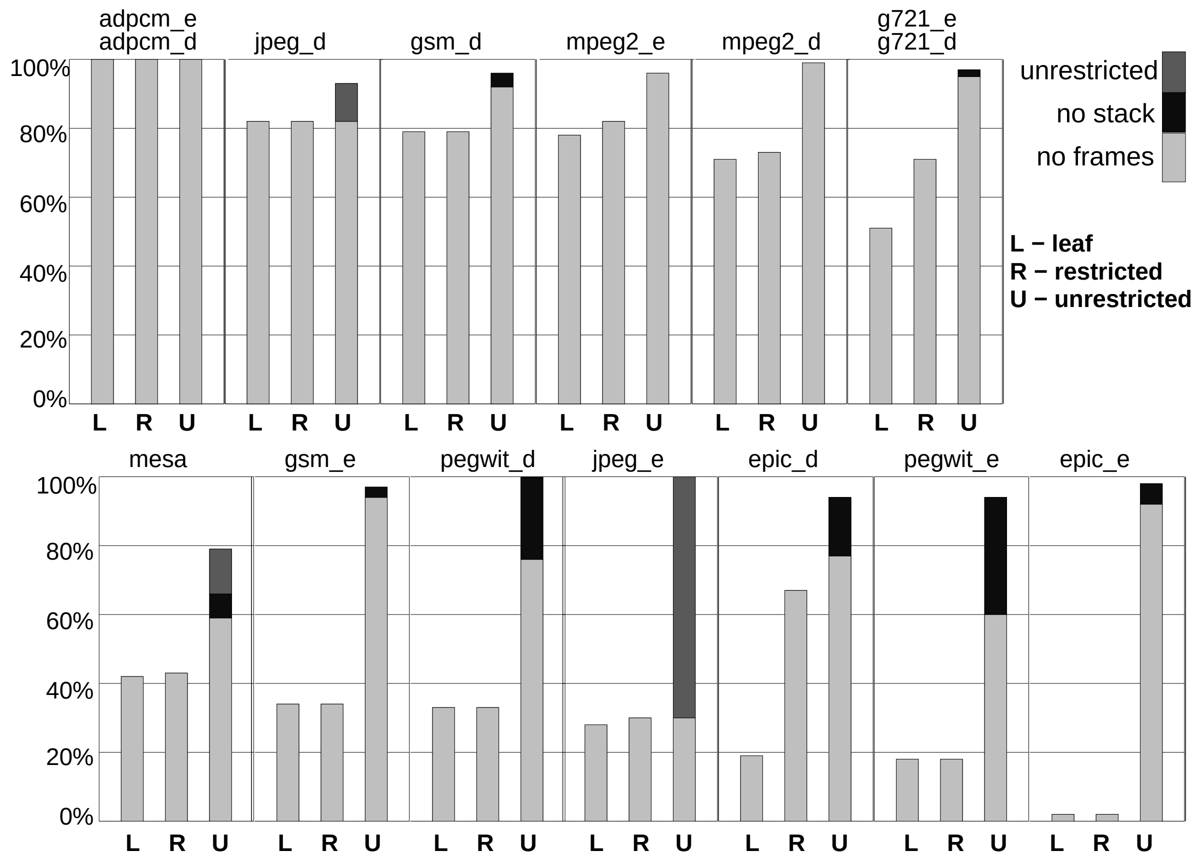


Figure 6. Dynamic program coverage as a function of the RH/CPU interface and RH architectural constraints for MediaBench programs, assuming RH can implement FP operations. The bars have the same meaning as in Figure 5

The bottom bar shows the coverage attainable when the RH procedures' local variables are allocated strictly in registers. This precludes implementing recursive functions and functions with local variables whose address is taken on the RH. The middle part of each bar is the difference in coverage obtained when RH procedures store the local variables in a statically-allocated memory region, thus allowing procedures with locals whose address is taken to be placed on the RH, but not recursive procedures. Finally, the top bar allows RH procedures to use unrestricted, dynamically allocated stack frames.

These graphs help us make several interesting observations about the power of our interface, and about the capabilities required in the RH to achieve significant program coverage.

Less than half the benchmarks (2 out of 6 SPEC programs and 6 out of 13 MediaBench benchmarks) spend more than 50% of their execution time in leaf procedures. We believe this proportion will be even lower for actual production software, because most important functions will have calls to error checking and reporting routines. This reaffirms our belief that, unless RH code is able to call other procedures, on the RH itself and on the CPU, substantial speed-ups cannot be obtained. If the RH is allowed to call other procedures residing on the RH, but not procedures on the CPU, (second bar), the coverage goes up for 4 SPEC and 4 MediaBench benchmarks, but the coverage is still significantly less than the case when the RH can call CPU functions. For the remaining benchmarks, there is practically no difference between bars 1 and 2. When allowing a peer-to-peer relationship between the CPU and RH, the resulting coverage is shown by the third bar. The fact that this bar is in most cases substantially larger than the other ones confirms the importance of a peer-to-peer relationship as proposed in this paper.

Limiting the size of the RH to one million bit-operations does not cause a significant change in the coverage figures; there were only two benchmarks that exceeded one million bit-operations in total size (mesa and go), and even for these, all the compute intensive routines fit within one mil-

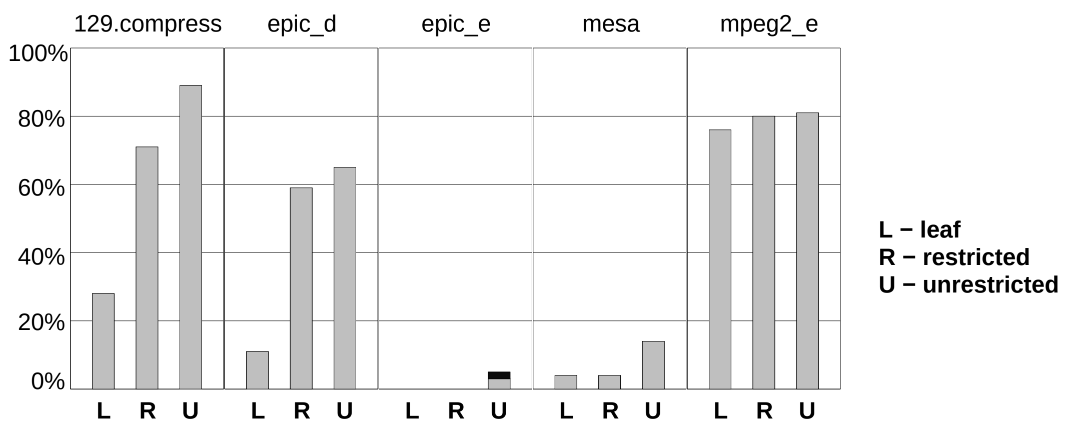


Figure 7. Dynamic program coverage as a function of constraints for assuming RH cannot implement FP operations. Only programs with characteristics different from Figures 5 and 6 are shown. The bars have the same meaning as in Figures 5 and 6.

lion bit-operations. This is true even for benchmarks which contain a substantial amount of floating-point operations (e.g. epic, mesa). Even though we assume a large amount of real-estate for implementing each floating point operation, and we assume that there is no sharing of resources between different operations in the program, a size of one million for the RH is not a limitation!

The decision whether to implement floating point computations on RH should thus depend not on the real-estate they require, but rather on the performance achievable by these operations on the RH versus on the CPU.

Completely disallowing floating point implementation on the RH significantly reduces the coverage for two benchmarks (epic_e and mesa), and causes moderate changes for three other. We depict the impact of this restriction in Figure 7.

4.2 Stub generation and overheads

We have implemented our stub-generating routines as a set of SUIF passes, and modified the SimpleScalar 3.0 sim-outorder simulator to simulate both CPU and RH components [19]: implementations of the new instructions described in Section 2 were added to the simulator and `gcc` was modified to recognize them.

We estimated the overheads introduced by our stubs on the CPU-side by counting the instructions that need to be executed in the process of performing the remote invocation. The overheads on the RH will depend on the actual communication primitives that will be synthesized there.

The costs are presented in Figure 8. The cost a procedure call includes constructing the stack frame, transferring control and returning. The cost of a stub includes marshaling the arguments into registers, making the appropriate call

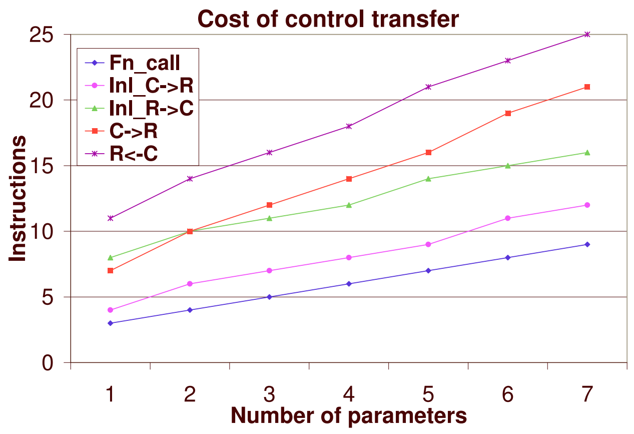


Figure 8. Cost of various control-transfer methods, as function of number of arguments. In order, from bottom to top: CPU→CPU call, CPU→RH with inlined stub, RH→CPU with inlined stub, CPU→RH, RH→CPU. The cost is measured in instructions and is only for the CPU side.

and returning the result (when the stubs are not inlined, they incur the cost of an additional procedure call). We express the cost in instructions rather than cycles, because the actual running time is highly dependent on the state of the processor pipeline at the time of the call.

- **Overheads for CPU to RH calls:** a stub executes the following instructions: instructions to move the procedure parameters into appropriate registers, one or more `rh_input`, one `rh_start`, one `rh_cont` to get the continuation address (in this case the address of the code handling the return), and one `rh_output` to get the return value.

- **Overheads for RH to CPU calls**: each transfer of control in this direction requires execution of the following sequence: an `rh_cont` to obtain the continuation address from the RH, a jump to the appropriate stub, one or more `rh_output` instructions to obtain procedure call parameters, a procedure call, an `rh_input` to pass the return value to the RH, a jump to the location of the `rh_start` and an `rh_start` to re-start the calling RH procedure.

5 Related Work

Several research projects have attacked the problem of partitioning programs between a CPU and a reconfigurable hardware fabric. From the point of view of the interface between the two, we can distinguish several classes of devices:

- Systems such as PRISC [14], Chimaera [8, 22] and T1000 [23] use a custom-instruction style of interface between the CPU and the RFU. A custom instruction is a RISC-like instruction whose opcode indicates an RH configuration that carries out the computation. These very lightweight custom instructions are severely restricted by their small number of inputs and outputs, and thus can only implement small computations.
- Systems using larger granularity RH include Garp [9, 11], OneChip [8], RaPiD [4], Morphosys [16]. In these systems the RH can autonomously access the memory of the system. The invocation of the RH is coprocessor-style. None of these papers proposes a consistent high-level interface, and none assigns an equal status to the RH and CPU (i.e. the RH cannot invoke the CPU in any of these systems).

  The researchers on the Garp project first observed in [3] the need of RH computation to be able to invoke library procedures on the CPU; they dealt with this problem by creating exceptional exits from the RH code. In their proposal the computation is mapped on the RH at the loop granularity; after an exceptional exit the RH computation is resumed at the loop-entry point.
- A proposal for a procedural interface to an RH system is made in Bauer's Master Thesis [1]. He coins the name "hardware subroutine" for the code migrated on the RH, and proposes, like we do, that partitioning should be done at procedure interfaces. In his proposal the RH is still relegated to a slave role, as it cannot invoke services on the CPU, and can implement only leaf functions of the call graph.
- Another class of coarse-grain reconfigurable systems consists of NAPA1000 [15], RAW [21], Smart Memories [12]. All these systems are more related to multiprocessors than to a simple CPU+RH model. In these systems the interface between the multiple computational units is highly specialized; it is not clear how much these systems would benefit from the use of a procedural interface.

The interface we propose is strongly related to the notion of Remote Procedure Call [13]; the idea of compiler-generated stubs derives directly from this work. However, unlike remote procedure calls, the systems that we consider can also communicate using shared memory. In our setting the procedure calls are used mainly for structuring the control-flow between multiple computational units, and less for data transmission.

Finally, let us note strong similarities between our stubs and the inlets from the Threaded Abstract Machine [5]; the way the stub dispatches procedure invocations from the RH is similar to Active Messages [20]. These latter paradigms were developed for dealing with parallel computations; we believe that parallelism can naturally be exploited in the CPU+RH context too, and that our proposed interface naturally extends to handle this case.

6 Conclusions

In this paper we present a proposal for a high-level hardware-software interface between processors and reconfigurable hardware devices. The two computational devices act as equal peers, and can invoke services from one another by using a procedural interface, similar to remote-procedure calls. Such an interface enables the migration of large code fragments to the reconfigurable hardware, simplifies program partitioning, ensures program portability and automates the generation of interface code by using compiler-generated stubs.

We also evaluate the effectiveness of our interface for automating the hardware-software partitioning of complex programs from the MediaBench and SpecInt95 benchmark suites: considering various constraints for the computational capabilities of the reconfigurable hardware device, we estimate how much of the computation can be offloaded from the processor.

We notice that even the computational resources available in current-generation devices are sufficient to implement large portions of each program or even entire applications. We also note that the ability of the reconfigurable hardware device to invoke functions on the processor is necessary and most often also sufficient for enabling complete freedom in software-hardware partitioning: if the hardware cannot invoke processor routines many important functions are not eligible for mapping on the hardware side.

7 Acknowledgements

We would like to thanks Todd Mowry for his guidance during the initial phases of this work. We would also like to thank Suraj Sudhir for his help with the simulator and other members of the Phoenix group and the reviewers of the paper for their helpful comments. This work was supported in part by NSF CAREER award CCR-9876248.

References

[1] T. J. Bauer. The design of an efficient hardware subroutine protocol for FPGAs. Master's thesis, MIT, 1994.

[2] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. In *Computer Architecture News*, volume 25 (3), pages 13–25. ACM SIGARCH, June 1997.

[3] T. J. Callahan and J. Wawrzynek. Instruction level parallelism for reconfigurable computing. In Hartenstein and Keevallik, editors, *FPL'98, Field-Programmable Logic and Applications, 8th International Workshop, Tallinin, Estonia*, volume 1482 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1998.

[4] D. Cronquist, P. Franklin, S. Berg, and C. Ebeling. Specifying and compiling applications for RaPiD. In K. Pocek and J. Arnold, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 116–127, Napa, CA, Apr. 1998. IEEE Computer Society, IEEE Computer Society Press.

[5] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, July 1993.

[6] A. DeHon. The density advantage of configurable computing. *Computer*, 33(4):41–49, Apr. 2000.

[7] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: a coprocessor for streaming multimedia acceleration. In *Published in proceedings of the 26th International Symposium on Computer Architecture ISCA 99*, 1999.

[8] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96, 1997.

[9] J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–21, Napa, CA, Apr. 1997.

[10] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Micro-30, 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, 1997.

[11] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *DAC 2000*, 2000.

[12] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *Proceeding of the International Conference on Computer Architecture 2000*, June 2000.

[13] B. J. Nelson. Remote procedure call. Technical Report CSL-81-9, Xerox Palo Alto Research Center, Palo Alto, California, 1981.

[14] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmed functional units. In *Proceedings of 27th Annual IEEE/ACM Symposium on Microarchitecture (MICRO-27)*, pages 172–180, Nov. 1994.

[15] C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale. The NAPA adaptive processing architecture. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, April 1998.

[16] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and T. Lang. MorphoSys: An integrated re-configurable architecture. In *Proceedings of the NATO Symposium on System Concepts and Integration*, Monterey, CA, April, April 1998.

[17] J. E. Smith, S. Weiss, and N. Pang. A simulation study of decoupled architecture computers. In *IEEE Computer*, volume 35 (8), pages 692–702, August 1986.

[18] Standard Performance Evaluation Corp. *SPEC CPU95 Benchmark Suite*, 1995.

[19] S. Sudhir. Simulating processors with reconfigurable function units. Master's thesis, Carnegie Mellon University, May 2002.

[20] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[21] E. Waingold, M. Taylor, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: The Raw machine. Technical Report TR-709, MIT/LCS, March 1997.

[22] A. Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable unit. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-00)*, ACM Computer Architecture News. ACM PRess, 2000.

[23] X. Zhou and M. Martonosi. Augmenting modern superscalar architectures with configurable extended instructions. In *Proceedings of the Reconfigurable Architectures Workshop RAW*, 2000.