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
Carnegie Mellon University

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 [1]. 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 of 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 [2]. 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. But we think that the choice of a good interface is extremely important for unleashing the full potential of a new computing paradigm. Witness the success of such interfaces as libraries, system calls, remote procedure calls, sockets, etc.

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, and not as forming a master-slave relationship.
- 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 are mainly focusing 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 adaptable 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 [3] and Mediabench [4] benchmark suites to evaluate the effectiveness of the implementation we describe. While these applications are implemented in C, the interface we describe 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:

- 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 traditionally.
- A call should appear to be invoked in the same way, independent of where the implementation actually resides. The way a software program invokes a computation on the RH should be the exact same way it invokes a computation on the CPU.

Figure 1 displays a legal invocation sequence under our proposal.

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

2.1 Stubs

The way hardware-independent service invocation is accomplished is similar to the technique used in implementing remote procedure calls: instead of calling the remote procedure, a *stub* procedure is called on the same side, with the correct arguments, and using the local calling conventions. The stub procedure is hardware-dependent and takes care of all low-level communication, by marshaling the arguments and invoking the remote service.

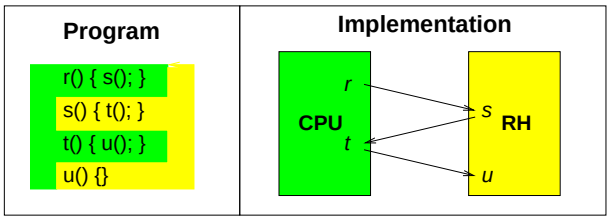

Figure 1: *A sample partitioned program and a legal implementation and invocation sequence.*

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.

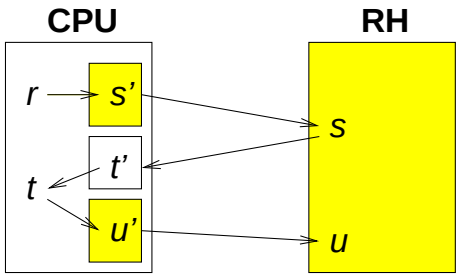

Figure 2: *Implementation of the example in Figure 1. The primed boxes are stubs for the respective procedures, i.e. s' is a stub for s. Stubs mediate the low-level communication but otherwise look like ordinary procedures.*

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');

- The hardware must provide a method to select the procedure to be invoked on the RH, because multiple procedures may reside simultaneously on the RH (e.g., is s or u called by r?);

- The CPU must have a method to obtain from the RH the address of a procedure for calls originating on the RH (e.g., does s call r or t?);

In Section 3 we describe precisely our 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 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 of mapping only self-contained code on the RH, which has no external procedure calls, 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 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 which way to steer the actual execution.

- Finally, a lot of the tedious work for interfacing the CPU and RH can be automated. As we will show in this paper, the generation of the low-level stub interfaces can be automatically done by a compiler once the partition of the program is known.

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 configurable-instruction model, which has been explored in prior work, e.g. [5, 6]: 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 configurable-instruction model however is applicable only 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 generality will only increase with time. We present here a set of assumptions which led us to design this interface:

- Technology advances will make Moore's law 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 applications will be executed on the RH. The processor will continue to handle "odd jobs", such as the operating system, virtual memory, resource management and arbitration, and RH configuration management.

- Moving just small pieces of code on 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 to the application-specific customizations they enable. In order to obtain important speed-ups one must be able to exploit the parallelism and customization on large code fragments. The attainable speed-up is proportional to the parallelized code coverage and inversely proportional to the overhead of CPU–RH crossings. Small code fragments on the RH imply either low coverage or many 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. While some data may be co-located on the RH with the code accessing it, in general the RH should be able to access any data dynamically.

- Finally, a very important motivation for our proposal is the observation that in production-quality software there are very few leaf functions. Most program functions call library functions, either for basic operations, or, very importantly, for error handling. If we restrict the selection for RH to functions doing pure computations there will be very few choices. The RH has to be able to invoke services from the processor if we want to move large parts of the computation on that side.

  The importance of these last two observation has been noticed before by researchers in the GARP project [7, 8]; 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 A Peer-to-peer CPU-RH Architecture

In this section we describe an example implementation of our proposed hardware/software interface on a simulated computer architecture comprising a superscalar processor and a tightly-coupled reconfigurable hardware unit.

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 have to be 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; same comments as above apply.

`rh_start R:` starts the execution of the R-th procedure loaded on the RH.

`rh_load R:` "loads" the binary configuration describing the R-th procedure into the RH.

`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. Excepting the interface to the load-store queue, in our implementation there is no other architectural feature of the processor visible from the RH; the processor has no other control over the RH

[1] The current implementation doesn't support passing floating-point inputs to a RH procedure.

[2] An alternative choice would have contained a procedure identifier in the `input` instruction. We have preferred this encoding due to its increased compactness when supporting procedures with multiple arguments.

except through the indicated instructions. We are considering adding also a second non-coherent memory interface to the RH, in the style of GARP [7], which can for instance be used for decoupled execution [9]. Decoupled execution has been proven to be extremely beneficial to streaming-data applications (see e.g. [7, 10]).

The way configurations are represented 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[3]; 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 ammended.

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.

```
        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);
        rh_input(r);             call CPU proc T
        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 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

[3]Consecutive instructions accessing the RH depend on each other, and will be serialized by the reorder buffer of the processor.

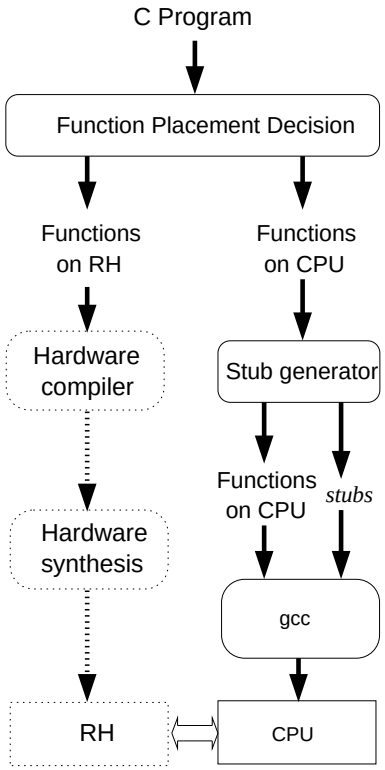

Figure 4: *Toolflow for compilation of applications on mixed hardware/software systems. The dotted-line components are not implemented. The hardware-compiler is in development.*

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:

- First we evaluate how restrictions on the RH computational capabilities reflect on the amount of program computation which can be mapped to the RH.
- Secondly, we estimate the overhead introduced by the stubs using micro-benchmarks.

4.1 Program Coverage

Here we examine how the CPU–RH interface we propose increases the program coverage, i.e., how much of an application's code can be put on the RH. We do this by

analyzing programs from the SpecInt 95 [3] and Mediabench [4] benchmark suites. We express coverage in percent of the running time. We obtain the coverage percentage for a procedure by profiling the program; the coverage of a set of procedure is the sum of their individual running times. Based on the capabilities of the hardware this coverage depends on a number of orthogonal dimensions:

- Whether RH can implement 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.
- Whether the RH can only implement leaf procedures: previous approaches could map only leaves to the RH.
- Whether the RH can call CPU procedures: can RH make calls to other RH procedures, but not to code on the CPU?
- 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.
- Unrestricted: the RH is able to handle any application procedure.
- Local variables accessibility: can a procedure on the RH pass the address of a local variable to other procedures? If not, the RH local variables can 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 dynamic execution time, using profiling.
- Information about the presence or absence of floating point operations in each procedure.
- A statically built, conservatively approximated call-graph.
- Per-procedure information about whether it passes pointers to local variables to 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 (arithmetic operations, condition evaluations, memory dereferencing etc.) 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 getting an estimate of how much of an application can be mapped to RH given certain size restrictions. We also do not make use yet of compiler analyses such as BitValue [11] which can be used to reduce the size of the computational units.

Finally, we estimated how much of a benchmark's dynamically executing code could be mapped to the RH by setting the RH's size to different limits, allowing and disallowing floating point operations on the RH, allowing and disallowing RH access to CPU memory, and allowing different kinds of procedures to be mapped to the RH: leaf only, procedures calling other RH procedures, non-recursive procedures and all procedures.

Figures 5, 6 and 7 show the coverage as function of the various restrictions. The bottom part of the bars represents the coverage when RH local variables are allocated to registers, while the top part is the coverage when locals are allocated in memory (and thus their address can be passed between procedures).

4.1.1 Discussion

Figure 5 presents the results of our analysis for the SpecInt 95 programs with an unlimited RH size. Several interesting observations can be made from these graphs about the power of our interface, and about the capabilities required in the RH to achieve significant program coverage.

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;
- Many benchmarks had a significant proportion of their execution time attributable to library routines (e.g., 20% in mesa), which we do not analyze.

Except for three (very small) Mediabench programs, all others spend less than 50% of their execution time in leaf procedures, and for most this proportion is less than 20%. This confirms 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 (second bar), the coverage goes up significantly for many benchmarks, but remains low for others.

Limiting the size of the reconfigurable fabric does not cause a significant change to the coverage figures; there were only two benchmarks that exceeded 1 million bit-operations in total size (mesa and go), and even for these,

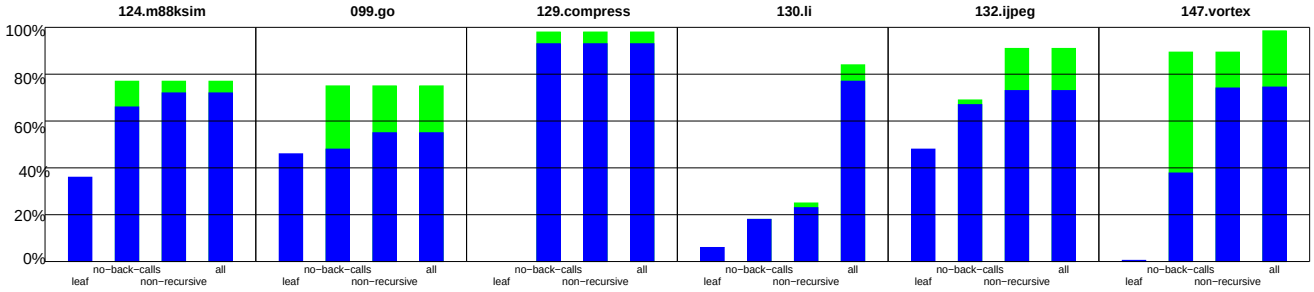

Figure 5: *Program coverage as a function of constraints for SpecInt programs. RH implements FP operations.*

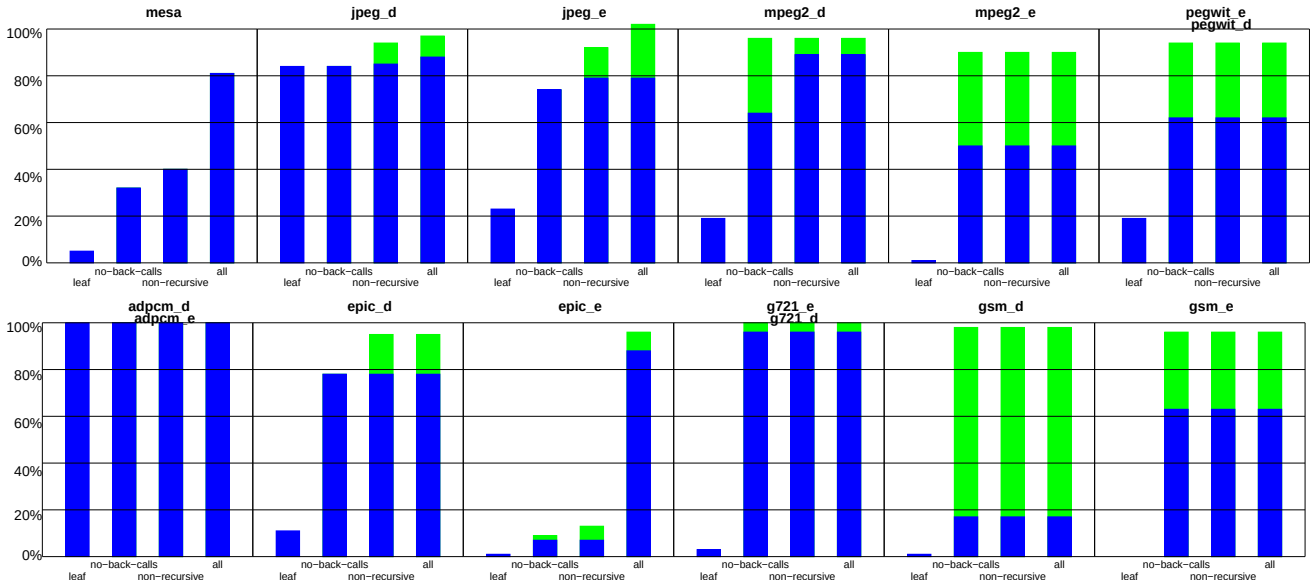

Figure 6: *Program coverage as a function of constraints for Mediabench programs. RH implements FP operations.*

all the compute intensive routines fit within one million bit-operations.

Disallowing floating point implementation on the RH significantly reduces the coverage for two benchmarks (epic_e and mesa), and causes moderate changes to two others (compare Figures 6 and 7). However, it was not the size of the RH which imposed limitations: coverage figures with floating-point implementation do not decrease significantly if we restrict the size of the RH to 1 million bit-operations. The decision whether to implement floating point computations on the RH depends thus not on the size of the RH, but on the performance achievable by these operations on the RH versus on the CPU.

The top part of each bar is the difference in coverage obtained when procedure-local variables are moved from registers to a global memory space, addressable by all procedures. The first type of implementation is likely to exhibit much better performance. The size of the top bar varies widely for different benchmarks; we conclude that such a decision is application-dependent, and has also to factor the performance difference of the two models.

The difference between the third and fourth bars shows the advantage achievable by allowing recursive procedures on the RH, which require the RH to have its own stack. This difference is significant for only three benchmarks (li, mesa and epic_e); we conclude that for many cases significant coverage is achievable without building a stack for the RH.

4.2 Stub generation and Overheads

To validate the effectiveness of the stub structure and to measure the overheads that stubs introduce, we implemented a stub-generator. We then modified the SimpleScalar 3.0 [12] sim-outorder simulator to simulate both CPU and RH components: implementations of the new instructions described in Section 2 were added to the simulator and the RH procedures are invoked through the

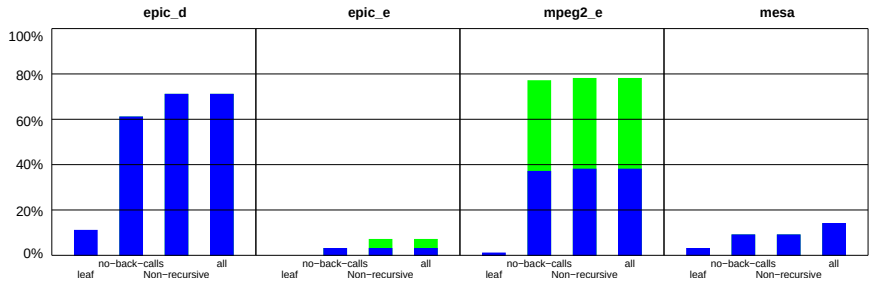

Figure 7: *Program coverage as a function of constraints for Mediabench programs. RH cannot implement FP operations.*

`rh_start` instruction. The new machine instructions were each assigned a latency of 1 cycle.

The performance of the stubs was observed by running micro-benchmarks using this infrastructure and carefully evaluating the running time of our stubs. This implementation does not give us the overhead of the RH-side interface routines (which is hardware-dependent) but does allow us to gain a reasonably accurate picture of the CPU-side overheads.

We have also analytically derived the expected cost incurred by the stubs, which is presented in Table 1. The analytic formulas match the measured overhead closely.

- **Overheads for CPU to RH calls:** a stub executes the following instructions: 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. This process is more lightweight than building the stack frame for a regular procedure call.

  Our results are summarized in Table 1; they show that our interface is actually more efficient than a software procedure call. However, the savings in cycles (compared to a regular procedure call) are smaller than the savings in dynamic instructions executed because the RH instructions cannot be issued in parallel, as they depend on each other.

- **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, 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 [6], Chimaera [5, 13] and T1000 [14] 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 the computation. While 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 [7, 15], OneChip [5], RaPiD [16], Morphosys [17]. 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 [8] 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 [18]. 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 can-

	Dynamic instructions executed	Extra simulator cycles
CPU to RH calls	$-2-2$*(#params $-$ #`rh_input`)	$1-1$*(#params $-$ #`rh_input`)
RH to CPU calls	4 + #`rh_output`	6 + #`rh_output`

Table 1: *A summary of CPU-side overheads associated with our stubs for handling CPU-RH communication. Note that an* `rh_input` *instruction is executed for every four data values sent from CPU to RH, and an* `rh_output` *instruction is executed for every three data values received by the CPU from RH.*

not invoke services on the CPU, and can implement only leaf functions of the call graph.

- Another class of coarse-grain systems reconfigurable consists of NAPA1000 [19], RAW [20], Smart Memories [21]. 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 [2]; 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 more for structuring the control-flow between multiple computational units than for data transmission.

Finally, let us note strong similarities between our stubs and the inlets from the Threaded Abstract Machine [22]; the way the stub dispatches procedure invocations from the RH is similar to Active Messages [23]. 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 have presented a proposal for a high-level hardware-software interface between processors and reconfigurable hardware devices. In this proposal 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 have also evaluated 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 have estimated how much of the computation can be offloaded from the processor. We have noticed that even the computational resources available in current-generation devices are sufficient to implement large portions of each program or even entire applications.

References

[1] A. DeHon, "The density advantage of configurable computing," *Computer*, vol. 33, pp. 41–49, Apr. 2000.

[2] B. J. Nelson, "Remote Procedure Call," Tech. Rep. CSL-81-9, Xerox Palo Alto Research Center, Palo Alto, California, 1981.

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

[4] 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*, pp. 330–335, 1997.

[5] 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*, pp. 87–96, 1997.

[6] 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)*, pp. 172–180, Nov. 1994.

[7] J. R. Hauser and J. Wawrzynek, "GARP: A MIPS processor with a reconfigurable coprocessor," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (J. Arnold and K. L. Pocek, eds.), (Napa, CA), pp. 12–21, Apr. 1997.

[8] T. J. Callahan and J. Wawrzynek, "Instruction Level Parallelism for Reconfigurable Computing," in *FPL'98, Field-Programmable Logic and Applications, 8th International Workshop, Tallinn, Estonia* (Hartenstein and Keevallik, eds.), vol. 1482 of *Lecture Notes in Computer Science*, Springer-Verlag, September 1998.

[9] J. E. Smith, S. Weiss, and N. Pang, "A Simulation Study of Decoupled Architecture Computers," in *IEEE Computer*, vol. 35 (8), pp. 692–702, August 1986.

[10] 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.

[11] M. Budiu, M. Sakr, K. Walker, and S. C. Goldstein, "BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations," in *Proceedings of the 2000 Europar Conference*, vol. 1900 of *Lecture Notes in Computer Science*, Springer Verlag, 2000.

[12] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," in *Computer Architecture News*, vol. 25 (3), pp. 13–25, ACM SIGARCH, June 1997.

[13] 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.

[14] X. Zhou and M. Martonosi, "Augmenting Modern Superscalar Architectures with Configurable Extended Instructions," in *Proceedings of the Reconfigurable Architectures Workshop RAW*, 2000.

[15] 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.

[16] D. Cronquist, P. Franklin, S. Berg, and C. Ebeling, "Specifying and compiling applications for RaPiD," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (K. Pocek and J. Arnold, eds.), (Napa, CA), pp. 116–127, IEEE Computer Society, IEEE Computer Society Press, Apr. 1998.

[17] 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.

[18] T. J. Bauer, "The Design of an Efficient Hardware Subroutine Protocol for FPGAs," Master's thesis, MIT, 1994.

[19] 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.

[20] 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," Tech. Rep. TR-709, MIT/LCS, March 1997.

[21] 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.

[22] 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.

[23] 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*, (Gold Coast, Australia), pp. 256–266, 1992.