

A Java Virtual Machine for Runtime Reconfigurable Computing

Brian Greskamp Ron Sass

Parallel Architecture Research Lab
Holcombe Department of Electrical
& Computer Engineering
Clemson University
105 Riggs Hall
Clemson, SC 29634-0915

E-mail: {bgreska, rsass}@parl.clemson.edu

Abstract

Reconfigurable Computing (RC) is a technology that makes use of programmable logic (FPGAs) in conjunction with a traditional microprocessor to accelerate general-purpose computations. RC machines have demonstrated impressive speedup on a variety of applications. Unfortunately, they are often difficult to program. This paper presents an experimental new RC platform, the RTR-JVM, which executes ordinary Java programs and makes use of online algorithms to select customized hardware at runtime. The RTR-JVM is thus a means of automating reconfigurable computing.

1. Introduction

It is well known¹ that the computational power of general-purpose computers is growing exponentially. Nevertheless, demand for computational power is growing even faster. This deficit has driven research in new computer architectures which might overcome some limitations of current microprocessors. To date, most performance improvements have stemmed from incremental (though by no means trivial) enhancements of the theoretical von Neumann Architecture. All of these designs, including the most recent superscalar CPUs, still execute a sequenced stream of instructions taken from a fixed *instruction set*. The instruction set is a list of all of the operations the processor can perform, and it is fixed at the time of chip design. In contrast, it is interesting to explore architectures that do not have this fixed instruction set limitation — architectures that are *reconfigurable*. This is the focus of a field known as *reconfigurable computing* (RC).

¹though not rigorously established

RC architectures are of interest because they have been shown to speed up a wide range of applications from image processing to gene sequence matching [8]. Their distinguishing feature is that they contain a limited amount of programmable logic in which arbitrary circuits can be realized. Simply speaking, algorithms implemented with circuitry in the programmable logic execute very fast. Indeed, many algorithms such as DES encryption have stunningly efficient hardware implementations. However, another class of algorithms does not map well to hardware. The following description of a real-world RC implementation shows how a reconfigurable computer handles computations of both classes.

For the remainder of this paper, we can consider an RC system to consist of a traditional von Neumann processor (“processor” for short) augmented with a limited quantity of programmable logic resources. Almost always, FPGAs provide these resources. An FPGA, or Field Programmable Gate array, is a device which can realize arbitrary sequential logic circuits by programming a routing network to connect hard-wired logic blocks in a specified manner (a more detailed explanation appears in [subsection 2.1.2](#)). The two components can be integrated in various ways, but for the ensuing discussion, the arrangement shown in [Figure 1](#) will be assumed.

An important feature is that the FPGAs may be reprogrammed at any time while the computer runs. It is therefore possible to program the logic with specific circuits as a program is loaded or even as it executes (“runtime reconfiguration”). The program then executes most of its computation on the host processor, but transfers control to the RC logic to perform the specialized calculations. Some oversimplifications have been made here, but this is the basic theory of a reconfigurable architecture.

It is now necessary to formally define some terms. Firstly, we have already mentioned that the program will be

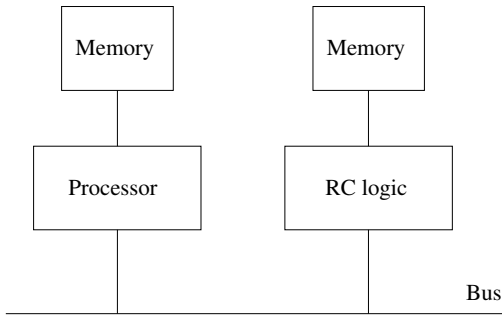


Figure 1. A simplified RC system

partitioned between the processor and RC logic. What are the entities to be partitioned? They are contiguous segments of the algorithm called *features*. More formally, a feature is the smallest portion of a program which may move from hardware to software and back. No theoretical limit on feature size exists; it is solely a function of the RC system. For example, a feature might be as large as a Java class or method, or as small as a basic block or smaller. The set of features resident in hardware at any given time is called the *feature set*. All non-resident features execute in software on the traditional processor.

The remainder of this paper will proceed as follows. Some fundamental challenges of RC will be laid out in [section 2](#) and solutions based on automation and online algorithms will be suggested. As to implementing these solutions, [section 3](#) contains details about the RTR-JVM, an automated RC system. Next, [section 4](#) presents some preliminary performance results from the prototype system, while [section 5](#) suggests future work based on these results. Finally, [section 6](#) places these results in perspective and argues the case for further work on systems similar to the RTR-JVM.

2. Background

The main problem with current RC systems is that they are difficult to program. To solve a problem on a reconfigurable computer, programmers are often forced to design software and hardware components separately. Design methodologies for hardware (*i.e.* using HDLs or schematic capture) are vastly different in concept from those of software design. Additionally, low-level interfacing and partitioning issues are often exposed to the programmer (*i.e.* making the software and hardware components communicate). One way to solve this problem is to increase automation. Ideally, an RC system should be capable of generating the required hardware and software features from a single high-level source specification. The system should also perform partitioning and interfacing without programmer inter-

vention. These are the goals for the RC platform described later in this paper.

It has already been stated that the platform must be programmable in a high-level language capable of generating both hardware and software implementations (one for the processor and the other for the RC logic). Fortunately, it has recently become possible to translate high-level software programs into hardware specifications. For example, Transmogripher [7] and Handel-C [1] are two translators that operate on C language programs. Forge [6], recently released by Xilinx, translates Java class files to Verilog HDL. The Verilog can then be synthesized to hardware using commonly available tools. Although research into HLL translation continues, the above tools are in our opinion adequate, and the translation problem will not be discussed further.

The remaining problem is automation of design partitioning. The challenge here is to select, at any given point during program execution, the set of features that should be resident in hardware. Recall that this is called the *feature set*. It is true that there always exists an *optimal feature set*: that set of features that will contribute the greatest overall speedup to the application. Also notice that the limited capacity of the RC logic usually forbids the trivial solution of having all features resident. Furthermore, note that the optimal feature set is a function of both time and the program's input data. In other words, the optimal feature set will vary continuously during program execution, requiring features to migrate back and forth between software and hardware.

A class of architectures known as *online architectures* can solve the feature selection problem more efficiently than existing approaches. In existing RC systems, feature selection is most often performed at compile time by searching the program for hot spots and replacing them with hardware implementations [10]. Alternatively, feature selection is not performed at all; the entire application is converted to hardware [1]. Clearly the latter approach is unsuitable for large applications, and the former is limited for two main reasons: (1) The compiled programs are generally not portable because different machines have varying types and quantities of RC resources, and (2) Selecting features for hardware implementation at compile time can be difficult and inefficient, especially when control flow is complex. Alternatively, systems in which the programmer makes the partitioning decisions can perform well with regard to 2, but demand a great deal of hardware expertise from their programmers [14]. Online RC systems can overcome these limitations because they are capable of using additional information gained *during program execution* to select, synthesize, and instantiate hardware features.

The process by which the feature set is constructed and continually updated to approximate the optimal feature set is called the *feature selection algorithm* and it is an *online problem*. Online problems have the property that inputs are

revealed one step at a time, and future inputs must be predicted based only on past ones. The page replacement algorithms used in virtual memory systems are a prime example. Decisions about which pages to swap out must be made without knowing what future access patterns will be. The feature selection algorithm must likewise predict future feature invocations based on past patterns.

The online approach proposed here is analogous to the one used by the Sun HotSpot Java Virtual Machine wherein runtime profiling data are gathered during execution. In the Sun JVM, this information is used to determine when to translate a method's bytecode into native code. Others have proposed [9] that the same profiling information might also be used to decide when to instantiate features in hardware. The research presented here differs from prior work because it represents a system capable of running pure, unmodified, high-level Java programs. This work also suggests additional uses of runtime profiling (other than feature selection) within the online RC context.

2.1. Technology Primer

Before proceeding to the description of the RTR-JVM, a little more background is required. This section introduces technology and terminology that will be used extensively in the ensuing discussion.

2.1.1. The Java Virtual Machine

Java is a popular high-level, object-oriented, buzzword-compliant software programming language. The language itself is not particularly relevant to the following discussion, but the way in which Java programs are executed is. Unlike most compilers which typically target a hardware architecture such as SPARC or x86, Java compilers target a *virtual machine*. Although this virtual machine does not correspond to any actual microprocessor architecture, any computer can execute compiled Java programs by emulating this virtual machine in software. The piece of software that performs this emulation and executes Java programs is known as the "Java Virtual Machine" (JVM). JVMs come in two major types. The simplest, the interpretive JVM, simply fetches each VM instruction one at a time and performs a sequence of actual machine instructions for each one. The more complex "Just in Time" (JIT) JVM compiles the VM code to native code before executing it, yielding a performance increase.

2.1.2. FPGAs

An FPGA contains an array of identical logic units called Combinational Logic Blocks (CLBs), shown in Figure 2(a). Modern FPGAs contain thousands or even tens of thousands of CLBs. Arbitrary circuits are realized by programming

the routing network to connect these CLBs in the desired way. The routing network, shown in Figure 2(b), is a two-dimensional mesh of switch elements. The control inputs of each switch are connected to a configuration RAM cell. Thus by writing different data into the configuration RAM, the switches are reprogrammed and the device is reconfigured. The data pattern used to program the configuration RAM is called the configuration bitstream.

Generating a configuration bitstream from a high-level functional description of a circuit is a job for *synthesis* software. This task is more complicated than it might first appear. A program specified in a hardware description language (HDL) must be synthesized into a netlist which describes the circuit to be implemented in terms of interconnected components. This process can take in the range of minutes to hours. The next phase, "place and route", maps the netlist to a specific FPGA architecture. The place and route phase is even more computationally intensive than synthesis, commonly taking many hours. When it completes, the configuration bitstream is available. Transferring the bitstream to the FPGA takes on the order of microseconds. Although current tools are discouragingly slow, ongoing work [12] is showing dramatic progress in reducing place and route times.

3. The RTR-JVM

In order to implement the proposed feature selection techniques as well as other optimizations to be discussed later, a prototype online RC platform was developed. The system is called the "Runtime Reconfigurable Java Virtual Machine" (RTR-JVM). An overview of the RTR-JVM system is shown in Figure 3. All method calls are intercepted by a profiler module before being dispatched. This profiler facilitates automatic reconfiguration by collecting performance data which can be used to compare the prospective speedup of different configurations. The feature replacement thread runs continuously, using data from the profiler to decide when to move features in and out of hardware. The dispatcher is responsible for marshalling arguments to and from the hardware in the case of a hardware invocation. The diagram also shows the pool of candidate features and the set of instantiated features. Importantly, the pool of available features is read-only (*i.e.* no new features are synthesized at runtime).

3.1. Limitations and Assumptions

The RTR-JVM is based on Kaffe v.1.0.7, a JVM chosen for its open license and stability on the Linux platform. For ease of implementation, the interpretive engine, as opposed to JIT, was chosen. The Forge Java-to-Verilog synthesis tool

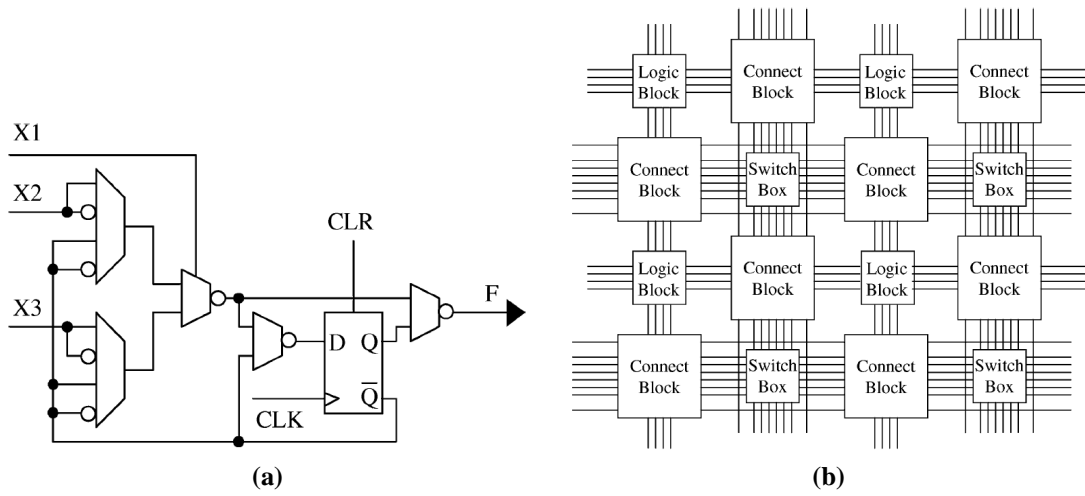


Figure 2. FPGA internal structure

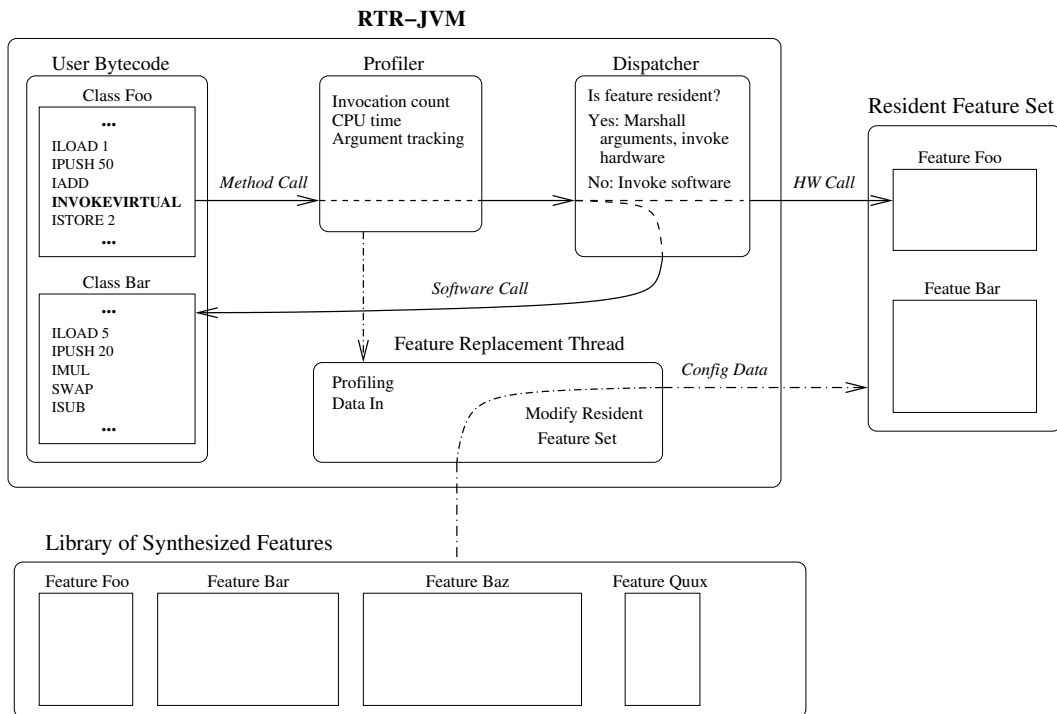


Figure 3. RTR-JVM block diagram

from Xilinx is used to generate Verilog HDL implementations from Java `.class` files. It introduces several restrictions on the Java source. For example, classes may instantiate objects and arrays, but all references must be resolvable in the class constructor. Additionally, floating-point variables and operations are not currently permitted. Finally, at the time this project started, Forge synthesized complete classes only². Consequently, features in the current RTR-JVM comprise complete classes and each must be a leaf in the call graph – it must not call methods outside of itself.

The Synopsys synthesis tool chain is used to link custom VHDL “glue logic” components which allow the host to communicate with the Forge-generated cores and the resultant design is synthesized with XST, the Xilinx Synthesis Tools. The synthesis output is a configuration bitstream suitable for programming one FPGA device. Although the long runtime of the commercial synthesis process currently forbids runtime feature synthesis, we expect to overcome this impediment in the future. Presently, all instantiable features are pre-generated before JVM startup.

3.2. Feature Selection Algorithm

The goal of the profiler is to determine \mathbf{H} , the optimal feature set, at any given timeslice. Since each class is a feature, it does this by calculating a metric M for each class. Before defining this metric exactly, it is useful to have a qualitative understanding of the traits that make a feature a strong candidate for hardware implementation. Intuition suggests that it is desirable to select features that:

- (a) use a lot of processor time
- (b) do enough computational work to offset communication overhead (argument passing)
- (c) have a fast hardware implementation
- (d) have a hardware implementation that doesn't use many resources

From these last two items, it is clear that the resource requirements and throughput of the hardware component must be known in advance. Resource requirements for each feature are expressed in terms of *slots*, where the slot is the smallest allocable RC logic unit. In the case of the prototype, each slot comprises an entire FPGA. Hereafter, these statistics are referred to as $slots(f)$ and $T_{hw}(f)$, respectively. In the JVM, they are read at class load time from a file called $X.hwspec$ where X is the name of the class. Of course this file is only available if a pre-synthesized copy of the class exists. Recall that one of the simplifying assumptions is that all synthesizable classes *are* pre-synthesized.

²Forge now has a method invocation interface

It follows that some profiling data must be gathered for software-resident classes as well so that hardware implementations can be compared with their software counterparts. For that purpose, these statistics are gathered on a per-class basis and continuously updated at each time slice. Each is maintained as a sliding-window average of constant width:

$rate(f)$: Number of invocations per second incurred by the class

$T_{sw}(f)$: CPU time expended per software invocation of the class

At this point, all of the data necessary for feature selection has been defined. Using this information, the goal is to construct a set of hardware features \mathbf{H} that will provide maximal speedup. Intuitively, the prospective hardware speedup for any given feature can be expressed as the product of the percentage of CPU time that the feature would consume executing *in software* and the per-call speedup afforded by the hardware. Alternatively, consider the product of the feature invocation frequency and the per-call time saved when the feature is in hardware. The latter viewpoint gives the expected speedup $S(f)$ as follows. The factor d is necessary when one or more features are resident in hardware and reflects the reduction in invocation rate that would occur if all features were in software.

$$d = 1 + \sum_{f \in \mathbf{H}} rate(f) \times (T_{sw}(f) - T_{hw}(f))$$

$$S(f) = \frac{rate(f) \times (T_{sw}(f) - T_{hw}(f))}{d}$$

Fortunately, it is not necessary to know $S(f)$ exactly; only a basis for comparing the relative merits of each feature is needed. Therefore, the simpler $S'(f)$, which is *proportional* to the theoretical speedup will suffice:

$$S'(f) = rate(f) \times (T_{sw}(f) - T_{hw}(f))$$

Normalizing the pseudo-speedup $S'(f)$ with respect to the number of slots required by each implementation provides a measure of performance per cost, the desired figure of merit. Finally, to prevent thrashing when two classes of similar merit exist, the metric for each class that is currently hardware-resident is inflated by a constant multiple β .

$$M(f) = \begin{cases} \frac{S'(f)}{slots(f)} \times \beta & : f \in \mathbf{H} \\ \frac{S'(f)}{slots(f)} & : \text{otherwise} \end{cases}$$

The final metric M has units of $\frac{\text{speedup}}{\text{slot}}$, but the critical reader might note that since the prototype system allows only one feature per FPGA, the division by slot count is superfluous. This will not be true in future implementations. In either case, after obtaining metrics for each instantiable class, the new feature set can be determined according to these steps:

1. Create a temporary structure to hold the new feature set.
2. Create a table of classes sorted in order of decreasing M .
3. Traversing the table from top to bottom and proceeding until all RC resources have been exhausted, add classes to the new feature set.
4. Synchronize the new feature set to the hardware, reverting expired features to their software implementations.

To illustrate this process, assume that **Table 1** represents the current state of the profiler. Classes 1 and 3 are in hardware, having obtained the highest scores on the preceding timeslice. Therefore, their metrics are inflated by a factor of β – in this case 1.20. When hardware selection is again performed in the current time slice, the ordering changes. Based on the rightmost column, the system would try first to instantiate class 1, then 4, then 3, and finally 2, traversing the sorted list until all RC resources are exhausted. Note that it is generally ³ not satisfactory to stop once a class has been found which will not fit because lower-ranked classes may still exist which *will* fit. Since it is known that all classes in the table will contribute a speedup ($T_{hw} < T_{sw}$), it is usually better to instantiate these lower-ranked classes than to let RC resources remain idle.

Class #	rate Hz	T_{sw} mS	T_{hw} mS	slots	Resident bool	M 1/slots
1	42	15	4	7	Yes	0.0792
2	17	5	2	3	No	0.0170
3	3	45	9	5	Yes	0.0259
4	15	10	5	2	No	0.0375

Table 1. Example merit calculation.

Now that the basic selection process has been outlined, a few caveats should be mentioned. First, when a class is resident in hardware, its T_{sw} variable can not be updated. Instead, the value from the previous time slice is used. Second, there is an enforced lockout period at interpreter startup

³It is acceptable to stop at this point in the prototype system because all classes are of the same size (a full 4085 FPGA)

during which no reconfigurable hardware may be synthesized or instantiated. This lockout period enables the runtime statistics to stabilize before any costly decisions are based upon them. Third, even though statistics collection occurs at every timeslice (ie. every $10mS$), hardware swap events are only allowed to occur every $100mS$. This ensures that the cost of copying the configuration bitstream to the FPGAs (not accounted for in M) remains negligible.

3.3. Communicating with Hardware

The task of efficiently transferring data (operands and state information) to and from the RC resources is a challenging one indeed, especially when these transfers must traverse a slow bus such as PCI. Past efforts [11] have used a stream-based paradigm, grouping operands, results, and configuration data into packets and taking advantage of DMA. Although efficient for large packet sizes, this approach does not work well for transferring small amounts of data. Since many features manipulate only a few bytes of data, it could be helpful to have an intelligent transfer mechanism capable of choosing the appropriate transfer mode for each transaction. Work on such a mechanism is underway, but in the meantime a very flexible temporary solution has been adopted. Associated with each instantiable feature is a shared object (.so) library, which exports stub [5] functions that interface with the hardware-resident class. These functions perform control tasks such as feature invocation and state exchange. Since each class has a separate stub library, data transfer methods can potentially be optimized on a per-class basis. Each library defines the following symbols:

enter hardware: Called to load the bit file into an FPGA and to perform initial configuration of the hardware. Any additional system resources required by the hardware can also be mapped at this time.

leave hardware: Called when a class leaves hardware. State information is retrieved from the hardware and written back into the software object data structures. Any resources held by the hardware are released.

call_X: Called whenever a hardware-resident feature X is invoked. The appropriate state (corresponding to the object being referenced) is loaded into the hardware, operands are transmitted, and the result is returned.

Above, *state information* refers to data that is associated with a particular feature instance. To clarify this definition in terms of the JVM, consider that although exactly one hardware entity exists per instantiated class, each class might correspond to multiple software objects (ie. many instances of the class). Each object has its own instance

variables, comprising a state that must be saved and restored when the hardware operates on a different object. Currently, this state is stored only in the software structures of the JVM and must be synchronized with the hardware each time a method is applied to a new object. There are more efficient solutions, such as mirroring the data in fast memory near the RC logic such that hardware features can directly access the appropriate state given only an instance ID.

All of the above interface library stubs must currently be hand-coded on a per-class basis, but there is nothing to preclude automation. It is conceivable that the Java compiler toolchain (ie. Forge) could be modified to generate stubs for each class automatically, selecting the communication modes which are most appropriate for each class. The compiler might also determine where to store state information for a given class. If there are few instances, they might be mirrored in the SRAM connected to the RC resources, or if there are many they may be stored exclusively in host memory.

4. Results

Post-publication addendum (02-11-06): *The following evaluation does not show a useful speedup. The JVM we used (Kaffe without JIT) is extremely slow. If compared against a modern high-performance JVM like SUN's hotspot, even the hardware-accelerated platform would look slow. However, with greater development effort, the techniques of this paper could be applied to a better JVM, and a real speedup might result.*

Previously described was the RTR-JVM software, which is largely independent of the target platform. The results presented here are tightly coupled to the hardware platform, and unfortunately, the available hardware is extremely unsuitable for this particular application. The hardware does not support partial reconfiguration and contains fewer CLBs than currently available devices. Nevertheless, it functions as a proof-of-concept, showing the selection algorithm in action.

4.1. Prototype Hardware Platform

For these experiments, an ACE2 Reconfigurable Computing card is installed in an x86-based Linux PC. The ACE2 card, manufactured by TSI-Telsys, carries two Xilinx 4085 FPGAs for a total of 6200 CLBs. Although unused in these experiments, the card also features a μ SPARC CPU, FIFOs, SRAM, and DRAM. Also unused is the Gigabit Ethernet controller. For reference, a block diagram of the card is shown in Figure 4. The Linux device driver, which was originally optimized for high-bandwidth applications, was modified to do memory-mapped transactions (instead

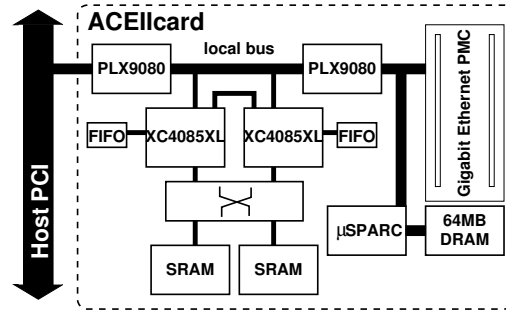


Figure 4. ACE2 card architecture.

of DMA). Word-by-word transfers with memory mapping are more appropriate for transferring the small amounts of data involved in feature invocation. Even so, every transaction has to cross two peripheral busses, which is a serious bottleneck for latency-sensitive applications such as the RTR-JVM.

4.2. Experiments

The RC-JVM is still in an extremely early state of development and no practical applications have yet been run. Tests to date have focused on validating the feature selection and invocation methods and on improving I/O performance. The following sections describe the results of these efforts and lay out plans for more practical demonstrations.

Bandwidth Tests In order for the RTR-JVM to be successful at all, software and RC resources must communicate with a minimum of overhead. Three communication methods were considered: DMA, PIO, and memory-mapping. DMA makes optimal use of bus bandwidth by transferring data across the bus in blocks, but each transfer requires many cycles to prepare. Involvement of the operating system kernel (a system call) is also required. In other words, it suffers from high *latency*. In the PIO mode, each word of data is across the bus separately, yielding lower bandwidth and also lower latency. A system call is still required. Memory mapping is a technique that gives an application direct access to memory so that transfers can be performed without a system call. As in PIO mode, bandwidth is sub-optimal, but latency is minimal. Figure 5 shows the relative speeds of each method when transferring three words of data.

Profiler Overhead On a Pentium-III 550Mhz benchmark system with the profiling system enabled in dry-run mode (no features are actually instantiated), the modified JVM scored 1.19 on the Scimark2 [2] benchmark, while the un-modified Kaffe-1.0.7 interpretive JVM scored 1.17

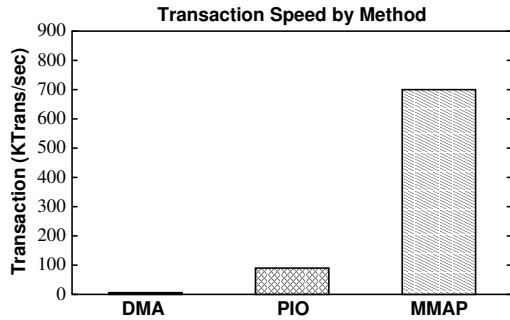


Figure 5. Small packet transfer rate by method

(higher is better). Clearly, the profiling overhead is negligible in this case. This is especially encouraging given the fact that the current profiler implementation is rather naïve and could be re-written much more efficiently.

Trial Run with Trivial Bitfile One trivial application that was used throughout testing contained a single synthesizable feature – an `Adder` class that simply adds two integers and returns the result. The communication-to-computation ratio is very high, so this feature will not provide any speedup, but it surprisingly doesn’t *slow* execution as much as expected. When forced to use the hardware implementation, the JVM ran a loop of six million `add()` calls in 31.8 seconds as opposed to 29.0 seconds using software. This simple test demonstrates that, with the interpretive JVM, speedups on practical applications with larger feature sizes should be demonstrable.

4.3. A Non-Trivial Test Kernel

It may seem that due to limitations in both software and hardware, the prototype system is crippled when it comes to executing practical applications. This is not the case. The following test demonstrated the RC-JVM’s ability to execute non-trivial features in hardware Figure 6. This simple test kernel models the common task of securely and reliably exchanging data over a network. It consists of four features, all of which are synthesizable: a DES encryptor, a DES decryptor, a Hamming code generator, and a Hamming code verifier. A simple test driver generates random data and feeds it to both the transmit and receive paths, with $RX\%$ of the generated packets traversing the receive path and the remainder following the transmit path.

In this application, all four features experience a speedup when implemented in hardware. The speedup factors are approximately 40 for the encryption features, 2 for the Hamming code generator, and 3 for the Hamming code verifier. The cryptography features take much longer to execute than the ECC features, a fact that explains the RTR-JVM’s pref-

erence for instantiating them. Figure 7 shows the time to process eight million bytes of data for both the RTR-JVM and the unmodified Kaffe interpretive JVM. As expected, the encrypt and decrypt features are in hardware for most values of $RX\%$, and total speedup is approximately 40. When very few packets are being received ($RX < 0.4$), the encrypt and Hamming generator functions come to dominate and speedup increases to 50. A similar condition prevails when $RX > 99.6$. In all cases, the RTR-JVM’s actions result in near-optimal feature assignment and substantial speedup.

4.4. Performance Limitations

Limited communication bandwidth will negate speedup in many situations. For example, the 8×8 IDCT used in the JPEG and MPEG codecs can execute in only 77 cycles on an SSE-enabled x86 processor [3]. Of course any Java implementation, whether JIT or not, will be significantly slower than that. In any case, it would be trivial to beat software IDCT performance with synthesized hardware if only data were readily accessible to the RC fabric. The connecting bus is clearly a major limitation in current designs, but new platforms might move RC resources close enough to the processor that this is not a concern. The Xilinx Virtex-II Pro, for example, embeds one or more PowerPC processors directly in the FPGA fabric. Other projects, such as GARP [10], have demonstrated that it is feasible to allow reconfigurable logic to access memory through the processor cache, allowing the CPU and RC logic to access memory symmetrically. Both approaches ameliorate the bandwidth problem.

5. Future Work

Here an attempt is made to redress the oversimplifications of section 3. Thus far, only a small portion of the RTR-JVM’s potential has been illuminated. Each of the following short sections introduces a topic for future research that follows as a direct consequence of the current work.

Runtime Feature Synthesis So far, the assumption has been that all of the program’s features have been synthesized at compile-time. This approach is time-consuming and impairs portability. Run-time feature synthesis addresses both problems. The time cost can be reduced by synthesizing at runtime only those features that prove “interesting” based on runtime profiling metrics. Further, the cost can be amortized over several program runs, since features may persist on disk indefinitely once synthesized. Still, synthesis tools must be improved significantly before runtime synthesis becomes practical.

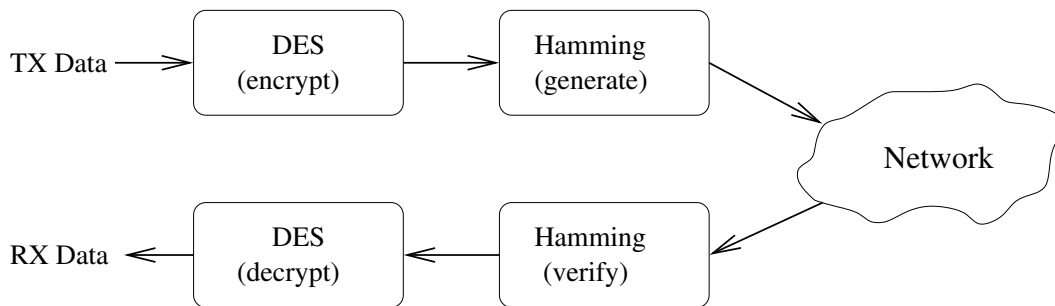


Figure 6. Test kernel with four instantiable features

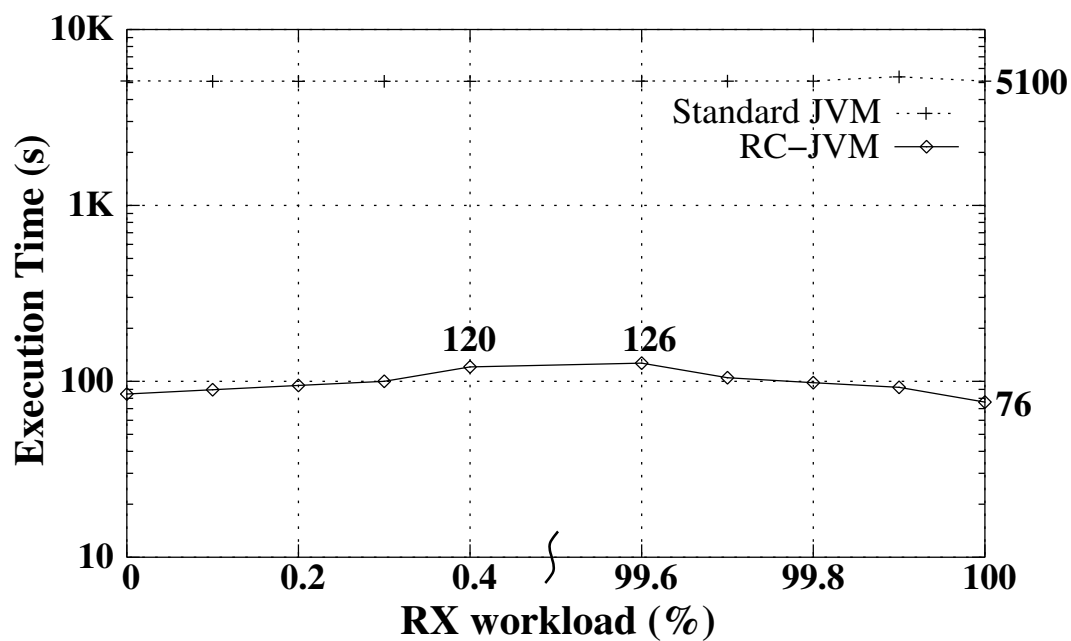


Figure 7. Network kernel benchmark execution times.

Memory Access from Hardware Features Many features behave as functions. They take in a fixed number of arguments and return a result without affecting the system memory. In these cases, the JVM sends all data needed by the feature (the arguments to the function) at each invocation. There are cases, however, when it is better for the hardware to be given *direct* access to the data in system memory. Image processing applications are one example, since features make random access to the image data which resides as an array in main memory. To facilitate direct memory access from hardware, FORGE could be modified to insert memory access states which would generate memory requests and stall feature execution until completion of the request.

Non-Leaf Features Throughout this paper it has been assumed that the classes implemented in hardware make no method calls outside of the class. Clearly, this is an oversimplification. Hardware features should be able to invoke other features residing in either hardware or software. Taking this into account, determination of the optimal feature set becomes more difficult. A particular feature might produce significant speedup if implemented in hardware only when the supporting features it invokes are also in hardware. Thus the new set calculation must take into account all method calls which a candidate feature makes, and the probability with which they occur.

JIT Compilation Since it is based on an interpretive JVM, the current implementation is much slower than other JVMs which perform “Just in Time” (JIT) code translation. The interpretive route was chosen for ease of implementation, but adding similar capabilities to a JIT JVM presents no major challenges. It simply requires the insertion of profiling callbacks into the compiled code and the Kaffe JVM already generates some such hooks to support xProf in its JIT engine. Running under JIT *will, however, decrease software execution time and therefore decrease the speedup* delivered by the RC hardware. For example, the SciMark benchmark assigns the Kaffe interpretive JVM a score of 1.17, whereas the Kaffe JIT JVM attains 15.04 SciMarks. Therefore, a hardware class scoring a respectable $S = 10$ under the interpretive JVM would result in an approximate *slow-down* of $S = 0.78$ under the JIT JVM. Note, however, that in embedded environments (eg. Virtex-II Pro), even a JIT JVM will run relatively slowly, while RC bandwidth will be much greater than in the prototype.

Partial Reconfiguration The prototype system can instantiate only one feature per FPGA. Newer FPGAs such as the Xilinx Virtex-II series allow portions of the FPGA to be reprogrammed independent of other portions (so-called *partial reconfigurability*). With these chips, it becomes practi-

cal to instantiate multiple features in a single FPGA. We have already acquired GRIP-2 [4] cards which possess this capability, and they will be used in the construction of a next-generation RTR-JVM platform. Extending the RTR-JVM to exploit this capability is a priority.

Constant Value Propagation By invoking a profiler callback for each feature invocation, a history of arguments passed to each feature could be maintained. Analysis of this history could reveal arguments that remain constant across many calls, especially constant arrays (eg. in convolution filtering). This information can be passed to the synthesis phase to create a special version of the feature with particular constant arguments. Such specialized features often execute faster and require less space because the synthesizer is able to eliminate extraneous logic and registers [13].

6. Conclusions

An RC machine augmented with runtime profiling capabilities can provide both ease of programming and considerable speedup for a wide range of applications. Many of the problems encountered during implementation have already been solved. Others are being actively researched. The development of hierarchical place-and-route tools may soon make runtime synthesis practical. Meanwhile, the integration of CPUs and reconfigurable fabrics on a single chip will eliminate the communication barrier and open RC to a wide range of bandwidth-intensive applications (eg. multimedia).

Online algorithms are ideally suited to RC application because they can help maximize the use of limited hardware resources and eliminates the guesswork associated with static feature assignment. They can also assist in other RC-related optimizations. These advantages come at very low cost; in the demonstration system, the overhead of the profiler is unmeasurable. With future work planned to develop a JIT-based machine and improved data transfer mechanism, the RTR-JVM may soon be able to accelerate a wide range of practical applications.

Acknowledgements

Thank go to Dr. Keith Underwood of Sandia National Labs for his continual contributions to the project, including the Linux device drivers for the ACE2 card and the next-generation GRIP2 hardware. Also contributing were Krishna Muriki, designer of the glue logic, and Srinivas Beeravolu and Ranjesh Jaganathan, who both provided valuable consultation.

References

- [1] Handel-c design tool. <http://www.celoxica.com>.

- [2] Scimark 2.0 java benchmark. <http://math.nist.gov/scimark2>.
- [3] Using streaming SIMD extensions in a fast DCT algorithm for MPEG encoding. http://www.intel.com/software/products/college/ia32/strmsimd/appnotes/ap817/fast_dct.pdf.
- [4] P. Bellows, J. Flidr, T. Lehman, B. Schott, and K. D. Underwood. Grip: A reconfigurable architecture for host-based gigabit-rate packet processing. In *IEEE Symposium on FPGAs for Custom Computing Machines*, Los Alamitos, CA, April 2002. IEEE Computer Society Press.
- [5] M. Budiu, M. Mishra, A. R. Bharambe, and S. C. Goldstein. Peer-to-peer hardware-software interfaces for reconfigurable fabrics. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 200–208, Los Alamitos, CA, 1999. IEEE Computer Society Press.
- [6] D. Davis. Forge: High performance hardware from high-level software. <http://www.xilinx.com/ise/advanced/forge.htm>, September 2002.
- [7] D. Galloway. The transmogrifier C hardware description language and compiler for FPGAs. In P. Athanas and K. L. Pocek, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 136–144, Los Alamitos, CA, 1995. IEEE Computer Society Press.
- [8] S. A. Guccione and E. Keller. Gene matching using JBits. In *12th International Field-Programmable Logic and Applications Conference*, September 2002.
- [9] Y. Ha, R. Hipik, S. Vernalde, D. Verkest, M. Engels, R. Lauwereins, and H. De Man. Building a virtual framework for networked reconfigurable hardware and software objects. *The Journal of Supercomputing*, pages 131–144, 2002.
- [10] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, Los Alamitos, CA, 1997. IEEE Computer Society Press.
- [11] R. Laufer, R. R. Taylor, and H. Schmit. PCI-PipeRench and the SwordAPI: A system for stream-based reconfigurable computing. In *IEEE Symposium on FPGAs for Custom Computing Machines*, Los Alamitos, CA, April 2002. IEEE Computer Society Press.
- [12] J. Ma and P. Athanas. Incremental design ide for multi-million gate fpgas. In *Proceedings of the Engineering Design and Automation Conference*, Hawaii, USA, 2002.
- [13] N. McKay and S. Singh. Dynamic specialisation of XC6200 FPGAs by partial evaluation. *Lecture Notes in Computer Science*, 1482:298–??, 1998.
- [14] M. J. Wirthlin and B. L. Hutchings. DISC: The dynamic instruction set computer. In *5th International Workshop on Field Programmable Logic and Applications*, pages 352–361, August 1995.