Partial Run-Time Reconfiguration Using JRTR

Scott McMillan and Steven A. Guccione

Xilinx Inc.
2100 Logic Drive
San Jose, California 95124
{Scott.McMillan, Steven.Guccione}@xilinx.com

Abstract. Much has been written about the design and performance advantages of partial Run-Time Reconfiguration (RTR) over the last decade. While the results have been promising, commercial support for partial RTR has lagged. Until the introduction of the Xilinx Virtex$^{(tm)}$ family of devices, no mainstream, commercial FPGA has provided support for this capability. In this paper we describe *JRTR*, a software package which provides direct support for partial run-time reconfiguration. Using a cache-based model, this implementation provides fast, simple support for partial run-time reconfiguration. While the current implementation is on the Xilinx Virtex family of devices using the *JBits* tool suite, this approach may be applied to any SRAM-based FPGA that provides basic support for RTR.

1 Introduction

Perhaps the greatest advantage to using SRAM based FPGAs is the ability to modify the configured circuit at any time. In the majority of cases, however, this capability is used exclusively in the design phase of a project. Once a design is tested and verified, it is seldom if ever changed. Over the last decade, researchers have explored using this ability to modify SRAM-based FPGAs to reduce circuit complexity, increase performance and simplify system design [1],[2],[3],[4]. In spite of these demonstrated advantages, these techniques have not found widespread acceptance. Much of the reason for this can be attributed to lack of both hardware and software support for partial reconfiguration.

With the introduction of the Xilinx Virtex family of devices [5] , hardware support for partial run-time reconfiguration is available for the first time in a large, mainstream commercial device. In addition, the Xilinx *JBits* tool suite [6] provides the software support necessary to make use of all of the features of the Virtex architecture. While run-time reconfiguration has always been supported in *JBits*, simple and direct support for partial run-time reconfiguration has not. This paper describes *JRTR*, a model and software implementation that addresses this deficiency.

2 Run-Time Reconfiguration Hardware and Software

While all SRAM-based FPGA devices are capable of having their circuit configuration modified at any time, software tools, and even the information necessary to perform run-time reconfiguration has been largely unavailable commercially. This situation changed

with the introduction of the Xilinx XC6200 family of devices [7]. This device family featured an open architecture, with all circuit configuration data exposed to the users. Unfortunately, with the exception of some small research tools such as *JERC6K* [8], *RAGE* [3] and *ConfDiff* [9], no commercially available software for the XC6200 device supported this run-time reconfiguration feature.

At Xilinx, the *JERC6K* work was transferred to the older and somewhat more limited *XC4000* family and renamed the *Xilinx Bitstream Interface* or *XBI* [6]. This software was later renamed *JBits* and supported direct construction and modification of circuits. Unfortunately, the *XC4000* family of devices only supports bulk configuration and contains no support for partial reconfiguration. Any changes to the device circuit configuration resulted in a halting of the device and a relatively slow reloading of the entire configuration bitstream. This was unacceptable for most applications.

More recently, the *JBits* software has been ported to the Xilinx *Virtex* family of devices. As with the *XC4000* family version, circuit configurations could be built and modified off-line, but reconfiguring the entire device resulted in the same sort of interruption as in the 4K version.

The recent addition of the *JRTR* software to the Virtex version of the *JBits* tool suite has resulted in direct support for partial reconfiguration. This support makes use of combined hardware and software techniques to permit arbitrarily small changes to be made directly to Virtex device circuit configuration data quickly and without interruption of operation.

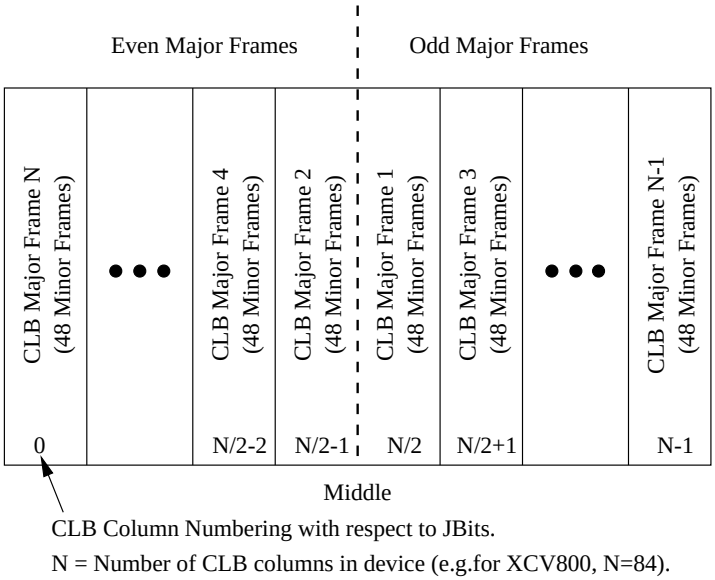

Fig. 1. The Virtex device configuration data organization.

3 Partial Reconfiguration in Virtex Device

As with most other FPGAs, the Virtex device can be viewed as a two-dimensional tiled array of *Configurable Logic Blocks* or *CLBs*. Associated with each CLB tile is some amount of logic and routing. In older device families such as the XC4000, the entire device would be configured using a single block of configuration data commonly referred to as a *bitstream*. This bitstream contained all of the data necessary to program the device into the desired configuration. If any piece of the configuration was to be changed, the entire configuration would have to be re-loaded.

While these earlier Xilinx device architectures were programmed using static data, the Virtex device configuration bitstream takes a packet-based approach. Here, the configuration bitstream consists of a sequence of command / data pairs of varying lengths that are used to read and write internal registers and configuration and state data. These commands operate on individual *frames* or column slices of the configuration memory. These frames are the smallest addressable units in the Virtex device configuration bitstream and may be accessed independently.

Figure 1 shows the Virtex device configuration data addressing scheme which uses major and minor frames. The major frame address refers to an entire CLB column containing a group of 48 minor frames. A minor frame refers to the individual columns of data used to program the device and represents the smallest grain of reconfigurability. To modify data within a CLB, affected data must be masked into frame(s) and re-loaded into the device. Reconfiguring an entire CLB requires this operation to be done on 48 contiguous minor frames within the device.

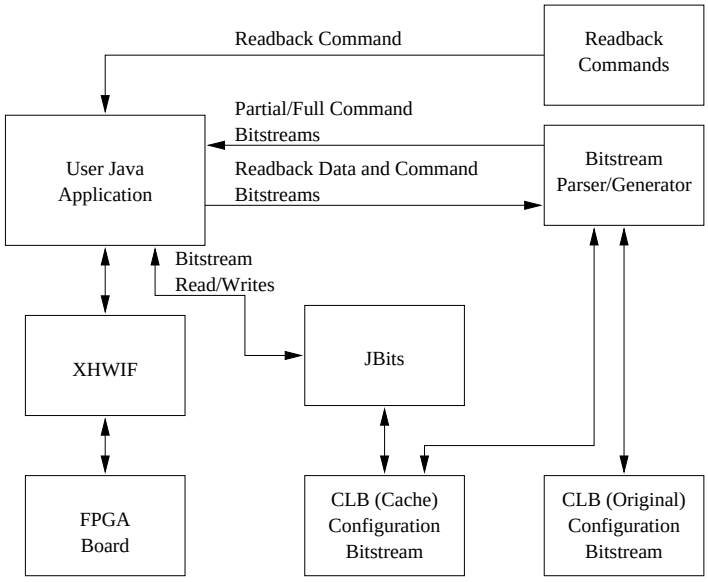

Fig. 2. An overview of the JRTR system.

4 The JRTR Interface

The *JBits* configuration bitstream Application Programming Interface (API) provides a set of Java classes to access and manipulate Virtex device configuration data. Currently, this data is manipulated in some piece of host memory, and later downloaded to hardware. Not coincidentally, this approach is nearly identical to the bulk download of the older XC4000 device family. This is because the current *JBits* for Virtex device software was essentially ported from the earlier XC4000 device version.

In order to better take advantage of the hardware support for partial run-time reconfiguration in Virtex device, the *JBits* API has been extended with the *JRTR* API. This interface provides a caching model where changes to configuration data are tracked and only the necessary data is written to or read back from the device.

Figure 2 shows the high level block diagram for the *JRTR* code. The existing *JBits* interface is still used to read and write bitstream files to and from disk and other external devices. But the *JRTR Bitstream Parser / Generator* is used to analyze the bitstream configuration data and to maintain both the data image and the access information.

The *JRTR* API is described in more detail in Table 1. The model for this code resembles a writeback cache in a traditional microprocessor system. Changes to the configuration bitstream are tracked each time the `get()` function call is used. This list of modified frames is eventually used to produce a set of packets to perform the necessary partial reconfiguration.

Table 1. The JRTR Application Program Interface.

Function	Description
parse()	Parses write and readback bitstream packet commands and overlays them onto the CLB and Block RAM (BRAM) configuration memories.
get()	Generates full or partial CLB and BRAM configuration packet streams.
clearPartial()	Clears the partial reconfiguration flag and forces a full reconfiguration only on the next get().
clearFull()	Clears the partial and full configuration flags and puts the object into an initial state.
writeClbCache()	Forces a write of the cache to the original CLB configuration.
getClbCache()	Returns a pointer to the CLB configuration stream. This will be used to synch up with the JBits object after parsing.

Figure 3 gives a typical code example of how *JRTR* is used. The current *JBits* interface is used to load in the configuration bitstream data from a file and then the data is parsed into a *JRTR* object with the `parse()` function call. The initial synchronization of *JBits* CLB configuration memory with the *JRTR* cache is somewhat more subtle, however. These two objects must reference the same memory location in order for the modifications made to the CLB configuration with *JBits* to be marked in the *JRTR* cache.

```
/* Parse <inputfile> bitstream */
JBits jBits = new JBits(deviceType);
jBits.read(inputfile);

JRTR jrtr = new JRTR(deviceType);
jrtr.parse(jBits.getAllPackets());

/* Sync JBits and parser cache */
jBits.setClbConfig(jrtr.getClbCache().get());

/* Download the full bitstream to device */
board.setConfiguration(jrtr.get());

/* Modify the CLB Cache with JBits. */
jBits.set(....)
   .
   .
   .
jBits.set(....)

/* Download partial configuration data to device */
board.setConfiguration(jrtr.get());
```

Fig. 3. *JRTR* code to perform partial reconfiguration.

Once the configuration data is downloaded to the hardware, any number of *JBits* `set()` calls, including those in cores or other classes and subroutines, can be made. Once the new configuration is set to the desired state, the *JRTR* `get()` call is used to produce the minimum number of packets necessary to perform the partial reconfiguration of the device. These packets can then be directly downloaded to the hardware.

Currently the API provides fairly simple, but complete, control of the configuration caching. The user can produce partial configurations at any time, and load them into hardware. While this level of control is desirable in most applications, this API also provides the tools necessary to experiment with other more transparent models of partial run-time reconfiguration.

Figure 4, shows a representation of a typical *JRTR* cache after some configuration data has been modified. In this case, data in five different frames has been modified. When the partial reconfiguration packets are produced by *JRTR*, they will actually contain only four distinct packets. The two frames near the center of the device are contiguous in the Virtex device address space and will be grouped into a single packet, minimizing the reconfiguration overhead.

Table 1 describes each of the function calls in the *JRTR* API. While most of the API provides calls for housekeeping functions, the `parse()` and `get()` functions provide most of the needed functionality. The `parse()` function loads in a bitstream containing either full or partial configuration data and uses this to modify the *JRTR* internal cache. The `get()` function call returns the partial packets generated by *JRTR*

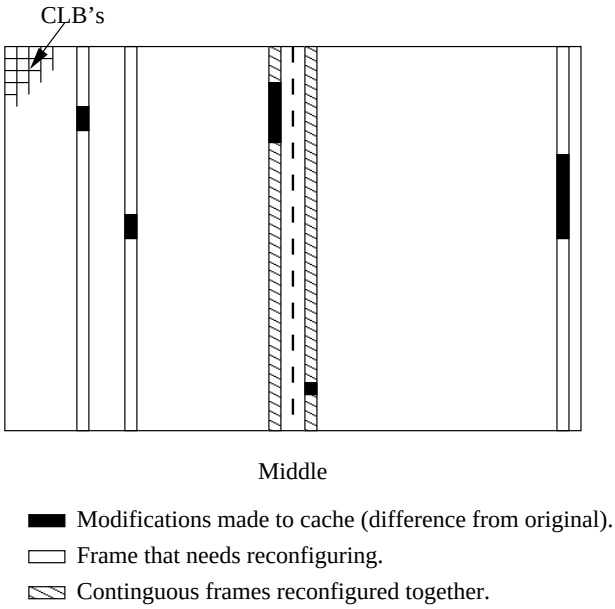

Fig. 4. Partially reconfigured SRAM cells and their associated frames.

to provide the Virtex device bitstream packets necessary to partially reconfigure the device. Note that each `get()` function also clears all of the "written" markers in the cache. Only subsequent modifications to the device configuration will be tracked and returned by future `get()` calls. Explicit calls do exist, however, that will set or reset the cache, for applications that may require this level of control.

5 Partial Readback

As with configuration, the *JBits* API currently assumes a bulk model for readback. All configuration bits are read back in a single operation and accessed in the host memory. As with the configuration model, this is primarily for historical reasons. And as with configuration, *JRTR* also provides a model for partial readback. Because of the nature of readback, this model is somewhat different from that of configuration and is necessarily not cache-based. Explicit requests for frame data must be made to the hardware, and the requests are done sequentially, not as a single block of Virtex device packets.

The partial readback example below shows how the *JRTR Bitstream Parser / Generator* object can be used in conjunction with the *JRTR Readback Commands* to perform partial readback. As in the case of partial reconfiguration, the CLB cache state is maintained and kept in synchronization with the configured hardware.

Unlike reconfiguration, the readback process is somewhat iterative. A readback request packet is written to the hardware, then the data is read back. For multiple partial readbacks, this process of write / read must be repeated. While there is some overhead

```
/* Readback CLB columns 4 thru 10. */
byte readbackCommand[] =
   ReadbackCommand.getCols(DEVICETYPE, 4, 10);

/* Get the readback data length */
int readbackSize = ReadbackCommand.getReadLength();

/* Send readback command to hardware */
board.setConfiguration(readbackCommand);

/* Read back the data */
byte readbackBytes[] =
   board.getConfiguration(DEVICENUMBER, readbackSize*4);

/* Load the readback data */
jrtr.parse(readbackCommand, readbackBytes);

/* Synchronize the JBits and readback data */
jBits.setClbConfig(jrtr.getClbCache().get());
```

Fig. 5. A partial readback example.

associated with this process, it has still been demonstrated to provide speedups over the bulk readback approach.

Figure 5 gives an example of some partial readback code. Note that frame columns are explicitly requested. While hardcoded values are used in the example for clarity, these constants could also have been taken from *JBits* resource constants, requesting, for instance, the frame containing the first bit of the F LUT in Slice 0.

Again, the nature of readback is somewhat different and is typically used to read back flip flop state information, or occasionally embedded RAM. This means that for a flip flop, which is found in a single frame of the 48 per CLB, the savings in data read back from the hardware is a factor of 48. While the amount of data is reduced by more than an order of magnitude, actual wall clock performance will depend on the speed of the interface and various software overheads. Networked debug applications, for instance, have demonstrated as much as a factor of three increase in performance using *JRTR*. Clearly this effect is more pronounced for slower data links, where the amount of data transferred is the major component of overall system performance.

6 Implementation Issues

The implementation of frame addressing in Virtex device is somewhat unusual. Frame address zero is at the center of the device, with addresses growing from the center outward, alternating right, left. This results in an interleaving of the data in the configuration bitstream. If, for instance, two physically contiguous CLBs are read as a block, the data will actually be returned interleaved with the associated frames from the other half of the device. Figure 6 illustrates this graphically. In order to read the “even” frames in

a single operation, an equal number of “odd” frames are also returned. This hardware feature of the Virtex device architecture makes grouping of contiguous frames difficult, and typically results in a factor of two penalty for blocks of data written to or read back from the device.

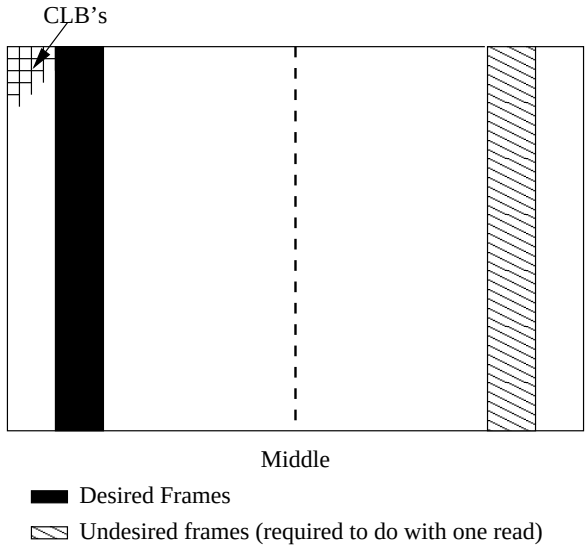

Fig. 6. Frame addressing in Virtex Device.

In addition to the odd / even addressing scheme, some other modifications to the Virtex device support for run time reconfiguration would be welcome. First, placing dynamic components such as flip flop outputs and embedded RAM in a distinct, contiguous address space would be useful. Since these are the only parts of the configuration bitstream capable of changing in normal operation, grouping these would simplify and enhance the performance of debug applications such as *BoardScope*. In addition, applications which only modify state, and not routing, would benefit.

State initialization is yet another issue. Being able to specify state of a flip flop when performing run time reconfiguration is crucial. Depending on a reset line of some sort is impractical in most cases. Lastly, design for partial reconfiguration at the silicon level presents some new challenges. Currently the sequence in which frames are reconfigured can result in intermediate states which are illegal or produce illegal results. Atomic operation of changes for any reconfigurable resource should be supported. In general, this means that all configurable MUXes and other such resources should reside in a single frame.

7 Conclusions

JRTR provides a simple and effective model and implementation to support partial run-time reconfiguration. Using a cache-based approach, partial run-time reconfiguration and readback has been implemented for the Xilinx Virtex family of devices and integrated into the *JBits* tool suite. It is hoped that *JRTR* will provide a platform not just for run-time reconfiguration application development, but for research and development into new software tools and techniques for using these capabilities.

Much work also remains on the hardware side. While the Xilinx Virtex device architecture provides adequate support for *RTR*, several other features are desirable. The ability to guarantee safe transitions when reconfiguring is crucial and simpler addressing which better reflects the actual device architecture would benefit software. In addition, the ability to set device state via reconfiguration is very desirable. That said, we also believe that the frame-based approach in the Virtex device provides a good balance of hardware and software support for run-time reconfiguration. Providing smaller grained addressing, perhaps at the CLB or programmable interconnect point level is likely to be overkill and may increase neither the performance nor the functionality.

References

1. Patrick Lysaght and John Dunlop. Dynamic reconfiguration of FPGAs. In Will Moore and Wayne Luk, editors, *More FPGAs*, pages 82–94. Abingdon EE&CS Books, Abingdon, England, 1993.
2. Herman Schmit. Incremental reconfiguration for pipelined applications. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 47–55, Los Alamitos, CA, April 1997. IEEE Computer Society Press.
3. Jim Burns, Adam Donlin, Jonathan Hogg, Satnam Singh, and Mark de Wit. A dynamic reconfiguration run-time system. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 66–75, Los Alamitos, CA, April 1997. IEEE Computer Society Press.
4. Steven A. Guccione and Delon Levi. Design advantages of run-time reconfiguration. In John Schewel, editor, *Reconfigurable Technology: FPGAs for Computing and Applications, Proc. SPIE 3844*, pages 87–92, Bellingham, WA, September 1999. SPIE – The International Society for Optical Engineering.
5. Steve Kelem. Virtex configuration architecture advanced users' guide. Xilinx Application Note XAPP151, version 1.1, Xilinx, Inc., July 1999.
6. Steven A. Guccione and Delon Levi. XBI: A java-based interface to FPGA hardware. In John Schewel, editor, *Configurable Computing: Technology and Applications, Proc. SPIE 3526*, pages 97–102, Bellingham, WA, November 1998. SPIE – The International Society for Optical Engineering.
7. Xilinx, Inc. *XC6200 Development System Datasheet*, 1997.
8. Eric Lechner and Steven A. Guccione. The Java environment for reconfigurable computing. In Wayne Luk and Peter Y. K. Cheung, editors, *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, FPL 1997. Lecture Notes in Computer Science 1304*, pages 284–293. Springer-Verlag, Berlin, September 1997.
9. Wayne Luk, Nabeel Shirazi, and Peter Y. K. Cheung. Compilation tools for run-time reconfigurable designs. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 56–65, Los Alamitos, CA, April 1997. IEEE Computer Society Press.