

Minimizing Partial Reconfiguration Overhead with Fully Streaming DMA Engines and Intelligent ICAP Controller

Shaoshan Liu, Richard Neil Pittman, Alessandro Forin
Microsoft Research

September 2009

Technical Report
MSR-TR-2009- 150

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Minimizing Partial Reconfiguration Overhead with Fully Streaming DMA Engines and Intelligent ICAP Controller

Shaoshan Liu, Richard Neil Pittman, Alessandro Forin
Microsoft Research

ABSTRACT

Configuration overheads seriously limits the usefulness of FPGA partial reconfiguration. In this paper, we propose a combination of two techniques to minimize the overhead. First, we design and implement fully streaming DMA engines to saturate the configuration throughput. Second, we exploit a simple form of data redundancy to compress the configuration bitstreams, and we implement an intelligent ICAP controller to perform decompression at runtime. The results show that our design achieves an effective configuration data transfer throughput of up to 1.2 Gbytes/s, which actually well surpasses the theoretical upper bound of the data transfer throughput, 400 Mbytes/s. Specifically, our fully streaming DMA engines reduce the configuration time from the range of seconds to the range of milliseconds, a more than 1000-fold improvement. In addition, our simple compression scheme achieves up to a 75% reduction in bitstream size and results in a decompression circuit with negligible hardware overhead.

1. INTRODUCTION

The defining characteristic of reconfigurable computing systems is their ability to change computations on demand. In an ideal scenario, we will have reconfigurable accelerators to execute certain parts of each desired program. As the system is used, we can load and unload accelerators to make best use of the reconfigurable resources. However, the configuration process itself incurs some performance overhead, thus it is unclear whether this approach is feasible or not in practice.

The performance overhead incurred by partial reconfiguration can be derived by dividing the bitstream size by the configuration throughput. Hence the key to minimizing this overhead is either to increase the configuration throughput, or to reduce the bitstream size. In this paper, we propose a combination of two techniques to minimize the partial reconfiguration performance overhead. First, we design and implement fully streaming DMA engines to nearly saturate the configuration bandwidth of the device. Second, we exploit a simple form of configuration data redundancy to compress the configuration bitstreams, and implement an intelligent ICAP controller to perform decompression at runtime.

To successfully apply these techniques, we need to have a good understanding of the partial reconfiguration process. Specifically, we would like to find out how a configuration bitstream interacts with the ICAP. While there is limited documentation available, we perform a low-level study to demystify the partial configuration process.

The rest of this paper is organized as follows: in section 2, we review the related work in partial reconfiguration and bitstream compression; in section 3, we introduce the design of the streaming DMA engines; in section 4, we present our study to understand the configuration process; in section 5, we discuss the design of the

intelligent ICAP engine; in section 6 we demonstrate the experimental results and then we conclude in section 7.

2. BACKGROUND

In this section, we discuss the related work in partial reconfiguration, bitstream file compression, as well as introduce the Virtex-4 FPGA and eMIPS platform on which we performed our experiments.

2.1 Fast Partial Reconfiguration

Runtime partial reconfiguration (PR) is a special feature offered by Xilinx FPGAs that allow designers the ability to reconfigure certain portions of the FPGA during runtime without influencing other parts of the design. This feature allows the hardware to be adaptive to a changing environment. First, it allows optimized hardware implementation to accelerate computation. Second, it allows efficient use of chip area such that different hardware modules can be swapped in/out the chip at runtime. Last, it may allow leakage and clock distribution power saving by unloading hardware modules that are not active. One major issue of PR is the configuration speed because the reconfiguration process incurs performance and power overhead. By maximizing the configuration speed, these overheads can be minimized.

In [1], to improve the reconfiguration speed, Liu *et al.* proposed the use of direct memory access (DMA) techniques to directly transfer configuration data to the Internal Configuration Access Port (ICAP). They reported to have achieved 82 Mbytes/s ICAP throughput using this approach. In addition, they placed a block RAM (BRAM) cache next to the ICAP so as to increase the ICAP throughput to 378 Mbytes/s. However, since on-chip storage resources are precious and scarce, putting a large BRAM next to the ICAP is not a practical approach. Similarly, in [2], Claus *et al.* also designed a DMA engine to provide high configuration throughput and they reported to have achieved 295 Mbytes/s on the Virtex-4 chip. In [3], facing the problem that the Xilinx Spartan III FPGA does not have an ICAP for reconfiguration, Paulsson *et al.* proposed and implemented a virtual internal configuration access port, or JCAP, to enable partial self reconfiguration on the Spartan III FPGA. The JCAP was actually an internal hardware interface that directly sent the configuration data to the JTAG to perform reconfiguration. The configuration throughput they approached was 2 Mbits/s.

According to [4], on the Virtex-4 chip the ICAP can run at 100 MHz and in each cycle it is able to receive 4 bytes. Thus the ideal ICAP throughput should be 400 Mbytes/s. In section 3 of this paper, we propose a fully streaming DMA design to approach this ideal throughput.

2.2 Bitstream File Compression

The second approach to reduce configuration time is by reducing the configuration file size through compression techniques. In [5], Li *et al.* studied the redundancy in various bitstream files and applied compression algorithms including Huffman coding, arithmetic coding, and LZ compression on these bitstream files. Their simulation results indicated that a compression ratio of 4:1 could be achieved. However, their study focused on only the compression ratio, it was not clear how much improvement on the actual configuration time their approach would bring. In [6], Dandalis *et al.* proposed a dictionary-based compression approach. Their results demonstrated up to 11~41% savings in memory for configuration bit-streams of several real-world applications. In [7], Pan *et al.* proposed techniques to exploit the redundancy between bitstream files such that certain parts could be reused. They reported that their approach achieved 26.5~75.8% improvement over the DV and LZSS algorithms.

Most bitstream file compression proposals utilized complicated compression algorithms in order to achieve high compression ratios. However, one major problem with this approach is that it requires the implementation of a complicated decompression circuit that may bring excessive area, power, and performance overheads to the design. This issue has been overlooked in many research studies. In most bitstream compression papers, the actual decompression hardware overhead, such as gate count, were not reported. In this paper, we propose an intelligent ICAP controller that can automatically extract the redundancy from the bitstream files. Our approach does not use a complicated compression algorithm, but it achieves a high compression ratio on real circuits and imposes negligible hardware overhead.

2.3 Other Approaches to Improve Configuration Performance

Some other approaches to improve configuration performance include prefetching configuration bitstream files and bitstream file relocation. In [8], Resano *et al.* proposed a prefetch scheduling heuristic to minimize the runtime reconfiguration overhead. Their approach computed the prefetch decision at design time and was able to prevent prediction misses. They reported to have eliminated from 93% to 100% of the configuration overhead. In [9], Li *et al.* provided a performance model for prefetching and proposed hybrid (static and dynamic) prefetching heuristics. They reported 70% reduction on configuration overhead. In [10], Carver *et al.* proposed a bitstream file relocation technique such that if the same hardware extension could be used in different locations of the same chip, then no separate bitstream files needed to be generated. This approach reduced the bitstream file storage overhead as well as the configuration time overhead. And our approach does not do anything to limit the use of these kinds of further enhancements

2.4 The Virtex-4 FPGA and the eMIPS System

The Virtex-4 FPGA consists of two abstract layers. The first layer is the logic and memory layer: it contains the reconfigurable hardware including logic blocks (CLBs), block RAMs (BRAMs), I/O blocks, and configurable wiring resources. The second layer contains the configuration memory as well as additional configuration and control logic that handle the configuration bitstream loading and the configuration data distribution. The smallest piece of reconfiguration information that can be sent to the

FPGA is called a frame. A frame contains the configuration information needed to configure blocks of 16 CLBs. The ICAP allows internal access to read and write the FPGA's configuration memory, thus it allows self-reconfiguration. On the Virtex-4 chip, the ICAP is able to run at 100 MHz and in each cycle it is able to consume 4 bytes of configuration data. Thus the ideal ICAP throughput is 400 Mbytes/s. Also, we store the configuration data in external SRAM which runs at 100 MHz. At its maximum speed, it is able to output 4 bytes per cycle. Thus the SRAM also has a maximum throughput of 400 Mbytes/s.

To test the performance impact of reconfiguration on a real system, we use the eMIPS system as our test platform. The eMIPS system is a dynamically extensible processor [11]. The eMIPS architecture allows additional logic to interface and interact with the basic data path at all stages of the pipeline. The additional logic, called Extensions, can be loaded on-chip dynamically during execution by the processor itself. Thus, the architecture possesses the unique ability to extend its own ISA at runtime. In the eMIPS system, the pipeline stages, general purpose register file, and memory interface match those in the classic MIPS RISC processor. The eMIPS system augments the basic MIPS architecture to include all the facilities for self-extension, including instructions for loading, unloading, disabling, and controlling the unallocated blocks in the microprocessor.

The partially reconfigurable Extensions distinguish the eMIPS architecture from the conventional RISC architecture from which it is derived. Using the partial reconfiguration design flow, the eMIPS system can be partitioned into fixed and reconfigurable regions such that the core architecture is included in the fixed region, whereas the Extensions are included in the reconfigurable regions. In this paper, we implement the fully streaming DMA engines and the intelligent ICAP controller in the eMIPS system to study how fast partial reconfiguration can be achieved as well as its impact on system performance.

3. STREAMING DMA ENGINES FOR THE ICAP PORT

In this section, we design and implement fully streaming direct memory access (DMA) engines to establish a direct transfer link between the external SRAM, where the configuration files are stored, and the ICAP.

3.1 Design of the Streaming DMA Engines

Figure 1 shows our system design for partial reconfiguration. In the original design, the ICAP Controller contains only the ICAP and the ICAP FSM, and the SRAM Controller only contains the SRAM Bridge and the SRAM Interface. Hence, in the original design there is no direct memory access between SRAM and the ICAP and all configuration data transfers are done in software. In this way, the pipeline issues one read instruction to fetch a configuration word from SRAM, and then issues a write instruction to send the word to the ICAP; instructions are also fetched from SRAM and this process repeats until the transfer process completes. This scheme is highly inefficient because for the transfer of one word it requires tens of cycles. This makes the ICAP transfer throughput only 318Kbytes/s. In order to achieve close to ideal ICAP throughput, our streaming DMA design provides three key features: master-slave DMA engines, a FIFO between the two DMA engines, and burst mode to support data streaming.

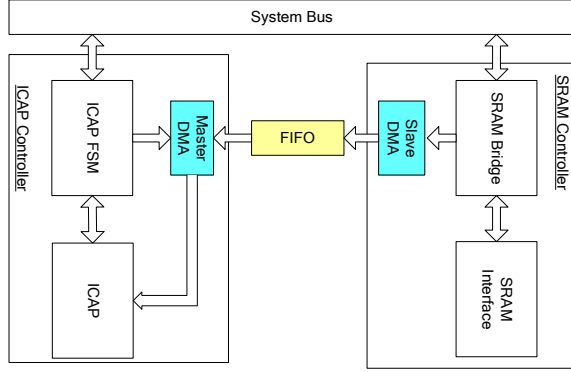


Figure 1: Structure of the Master-Slave DMA for PR

3.1.1 Adding the master-slave DMA engines

First, we implemented the master-slave DMA engines. As shown in Figure 1, the master DMA engine resides in the ICAP controller and interfaces with the ICAP FSM, the ICAP, as well as the slave DMA engine. The slave DMA engine resides in the SRAM Controller, and it interfaces with the SRAM Bridge and the master DMA engine. When a DMA operation starts, the master DMA engine receives the starting address as well as the size of the DMA operation. Then it starts sending control signals (read_enable, address *etc.*) to the slave DMA engine, which then forwards the signals to the SRAM Bridge. After the data is fetched, the slave DMA engine sends the data back to the master DMA engine. Then, the master DMA engine decrements the size counter, increments the address, and repeats the process to fetch the next word. Compared to the baseline design, adding the DMA engines avoids the involvement of the pipeline in the data transfer process and it significantly increases the ICAP throughput to about 50 Mbytes/s.

3.1.2 Adding a FIFO between the DMA engines

Second, we modified the master-slave DMA engines and added a FIFO between the two DMA engines. In this version of the design, when a DMA operation starts, instead of sending control signals to the slave DMA engine, the master DMA engine forwards the starting address and the size of the DMA operation to the slave DMA engine, then it waits for the data to become available in the FIFO. Once data becomes available in the FIFO, the master DMA engine reads the data and decrements its size counter. When the counter hits zero, the DMA operation is complete. On the other side, upon receiving the starting address and size of the DMA operation, the slave DMA engine starts sending control signals to the SRAM Bridge to fetch data one word at the time. Then, once the slave DMA engine receives data from the SRAM Bridge, it writes the word into the FIFO, decrements its size counter, and increments its address register to fetch the next word. In this design, only data is transferred between the master and slave DMA engines and all control operations to SRAM are handled in the slave DMA. This greatly simplifies the handshaking between the ICAP Controller and the SRAM Controller, and it leads to a 100 Mbytes/s ICAP throughput.

3.1.3 Adding burst mode to provide fully streaming

The SRAM embedded in the ML401 FPGA board actually provides burst read mode such that we can read four words at a time instead

of one. Note that burst mode reads are available on DDR memories as well. There is an ADVLD signal to the SRAM device. During a read, if this signal is set, a new address is loaded into the device. Otherwise, the device will output a burst of up to four words, one word per cycle. Therefore, if we can set the ADVLD signal every four cycles, given that the synchronization between control signals and data fetches is correct, then we are able to stream data from the SRAM to the ICAP.

To achieve this, we implemented two independent state machines in the slave DMA engine. One state machine sends control signals as well as addresses to the SRAM in a continuous manner such that every four cycles the address is incremented by four words (16 bytes) and sent to the SRAM device. The other state machine simply waits for the data to become ready at the beginning, and then each cycle receives one word from the SRAM and streams the word to the FIFO until the DMA operation completes. Similarly, the master DMA engine waits for data to become available in the FIFO, and then in each cycle it reads one word from the FIFO and streams the word to the ICAP until the DMA operation completes. This fully streaming DMA design leads to an ICAP throughput that exceeds 395 Mbytes/s, very close to the ideal 400 Mbytes/s number.

3.2 Handshaking between the Master and Slave DMA Engines

Our fully streaming DMA design achieves near perfect, but not perfect, ICAP throughput because in order to initiate a DMA operation there is a handshaking process between the two DMA engines. This process introduces some performance overheads. The waveform shown in Figure 2 illustrates this process. At the beginning, the master DMA engine receives the DMA starting address (ADDR_master) as well as the size (SIZE_master) from the ICAP FSM. After one cycle, the ICAP FSM sets the DMA operation signal (DMA_OP) to notify the master DMA engine to start the DMA operation. Then one cycle later, the master DMA engine forwards the DMA starting address and size to the slave DMA engine (ADDR_slave, SIZE_slave). Upon receiving these signals, the slave DMA engine starts operation by triggering the control state machine to send address and control signals to the SRAM, such that the active-low ADVLD signal (burst_DMA) is set low to stream a new address to SRAM every four cycles. Meanwhile, the data state machine waits for the SRAM data (DATA_SRAM) to become available. After seven cycles, the first SRAM data becomes available and the slave master engine sends one word to FIFO (DATA_FIFO) each cycle until the DMA operation completes. On the other side, the master DMA engine checks whether the FIFO is empty (FIFO_EMPTY), once the FIFO becomes not empty, the master DMA engine starts reads data from the FIFO and sends one word to the ICAP in each cycle until the DMA operation completes. Thus, the handshaking process takes 12 cycles to complete. At the end of the DMA operation, it takes another 5 cycles to re-synchronize and reset the two state machines in the slave DMA engine. Thus, the total control overhead of this design is only 17 cycles. During the rest of the transfer time this design streams configuration data at a full 400 Mbytes/s.

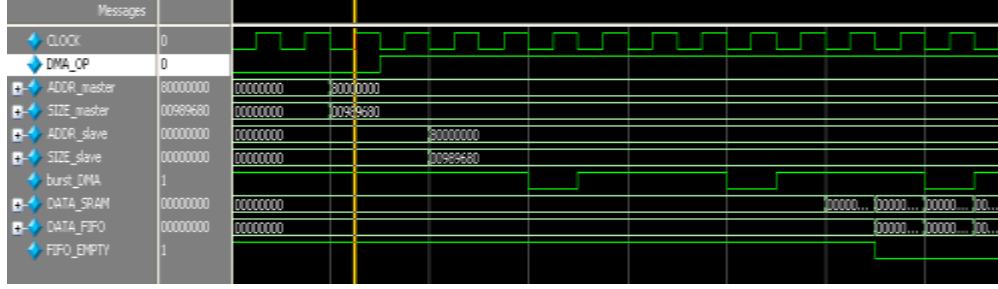


Figure 2: Handshaking of the DMA engines

3.3 Synchronization between the Control and Data State Machines

In the slave DMA engine there are two independent state machines: the control state machine (which sends control signals and addresses to the SRAM interface) and the data state machine (which receives data from the SRAM interface and forwards the data to the FIFO). The synchronization between these two state machines is critical to the correctness of DMA operations. Any mistake in the synchronization between these two state machines may result in missing or redundant data transfer, potentially leading to incorrect configuration.

The state machine diagrams and synchronization mechanisms between are illustrated in Figure 3. These two state machines interact with each other, as well as with two other modules: the master DMA engine and the SRAM interface. At the beginning of a DMA transfer, the master DMA engine sends the start DMA operation signal to both state machines. Upon receiving this signal, both state machines wait for the SRAM interface to become ready, and then they transition to the next state. Up to this point, the two state machines are synchronized.

After verifying that the SRAM interface is ready, the control state machine starts sending control signals and address to the SRAM interface to start burst operation. Each burst takes four cycles to complete. Thus, the control state machine updates the address every four cycles, each time incrementing the address by 16 bytes. As shown in Figure 2, there is a six-cycle delay between the time when the control signals are sent to the SRAM interface and the time when the data returns. Thus, after verifying that the SRAM interface is ready, the data state machine waits on the data ready signal from the SRAM interface. After receiving this signal, the data state machine starts reading data until the DMA operation completes. During this process, the two state machines are not synchronized.

When the DMA operation completes, the data state machine sends out a DMA complete signal to the control state machine and it switches back to the initial state. On the other side, upon receiving this signal the control state machine also transitions to the initial state and waits for the next start DMA operation signal. At this point, the two state machines become synchronized again.

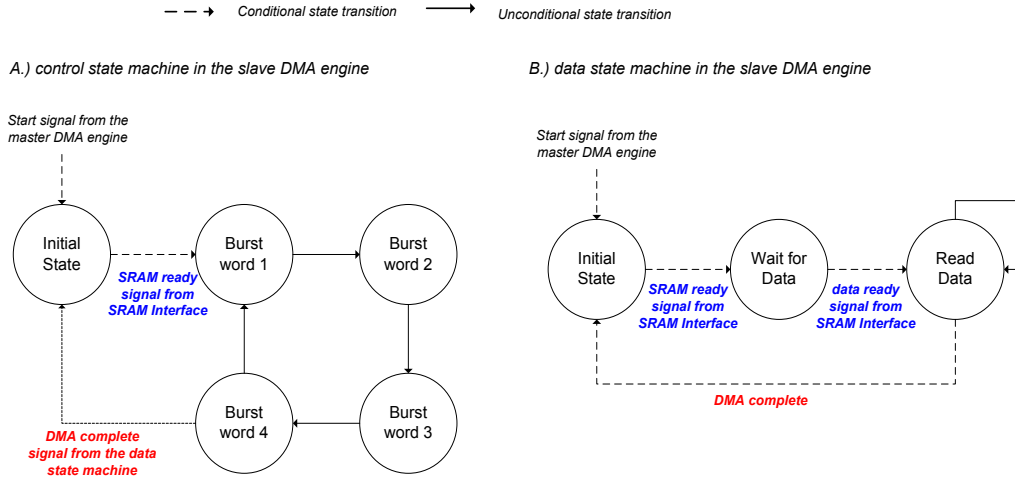


Figure 3: Synchronization between the state machines in the slave DMA engine

4. DE-MYSTIFYING CONFIGURATION BITSTREAM FILES

Although Xilinx provides some documentation [12] to explain the details of the bitstream files, how the contents of the bitstream files would affect the configuration process remains a mystery. In this section, we perform several low-level experiments in order to identify the effect of the bitstream files on the configuration process. To carry out these low-level experiments we implemented

a block RAM (BRAM) next to the ICAP port and stored different bitstreams in the BRAM. We then modified the bitstreams and observed how the modifications affected the behavior of the configuration process.

4.1 Configuration CRC Test

The first question we have is whether we can manually modify the bitstream file. At the end of each bitstream file, there is a cyclic

redundancy check (CRC) command that checks whether the bitstream file has been modified since it is generated. In order to understand how the CRC test would affect the configuration behavior, we performed the following experiments:

1. CRC modification: we modified the CRC word such that we were using a wrong CRC value. Once the CRC error was detected by the ICAP, the ICAP stopped functioning and would not take any more new commands.
2. CRC removal: we removed the CRC command from the bitstream file to test whether it would affect the configuration. The result was that the bitstream was written to ICAP as usual. After the configuration process completed, the design functioned as expected.
3. Disabling CRC using the COR: there is a configuration option register (COR) in the ICAP, we can modify this register to enable or disable CRC; by default CRC is enabled. In this experiment, we modified the COR to disable CRC. Although we used a correct CRC value, the ICAP port still stopped taking any new commands after the CRC command was detected.

The conclusion is that CRC is optional: in case we want to manually modify the bitstream, we can simply remove the CRC command from the bitstream.

4.2 The ICAP Busy Signal

The ICAP interface has an output called BUSY. Without any detailed documentation, we assumed that this signal indicates that the ICAP is performing some function and cannot take commands or data at that time. To test this assumption, we wrote various patterns of commands and data to the ICAP in an attempt to force the ICAP to raise this BUSY signal.

1. We repeated the same register read command 20 times to the ICAP port.
2. We issued a read immediately after a write to the same register, and repeated this process 20 times.
3. We selected several frame addresses within a clock region and wrote configuration data frames to the ICAP port in pseudo random pattern.
4. We wrote the same partial bitstream to the ICAP over and over.
5. We wrote nothing but NOP to the ICAP over and over.
6. We repeated writing the set frame address command followed by the same address.
7. We repeated writing the set frame address command followed by the different addresses.
8. We took a working bitstream and replaced all the configuration frame data with zeros and wrote it to the ICAP.
9. We turned off the ICAP write enable (WE) signal and turned it back on.

Only the last test, cycling the write enable signal, caused the BUSY signal to be set for 6 cycles. In all other cases, the BUSY signal was never set. The conclusion is that the BUSY signal of ICAP would unlikely be set during the configuration process, implying that the ICAP is able to run at full speed during configuration.

4.3 Configuration Time—a Pure Function of the Bitstream Size

In order to find out how configuration time can vary, we performed the following experiments:

1. Location study: we wanted to find out whether the configuration time depends on the location of configuration, thus we implemented a design that contained 256 counters and fit the design into one clock region on the Virtex-4 chip. Then we placed the design in various locations on the chip, some far away from the ICAP port, and others close to the ICAP port. Then we studied whether the location affected the configuration time. The results showed that the ICAP port was able to run at full speed regardless of the reconfiguration location.
2. Command study: we wanted to find out whether the configuration time depends on the configuration command, thus we kept writing NOPs to the ICAP port and verified that the ICAP consumed one command each cycle. Then we repeated the experiments for other commands, including the write configuration data command (WCFG), the multiple-frame writes command (WFWR), *etc.* The result was the same; these commands did not affect the configuration speed.
3. Configuration stress study: to find out whether continuously configuring one clock region of the chip would stress the ICAP, we did an experiment to repeatedly write the bitstream for one clock region to the ICAP. The result showed that the ICAP port was still running at full speed in this case.

During all these tests, we found out that ICAP always ran at full speed such that it was able to consume four bytes of configuration data per cycle, regardless of the semantics of the configuration data. This confirms that configuration time is a pure function of the size of the bitstream file.

4.4 The NOPs

The FPGA tool chain inserts many NOPs throughout the bitstream file. We wanted to understand if these are necessary and if so why. For this purpose, we first examined various bitstream files and identified the following common patterns of NOPs:

1. NOPs are inserted after each write to CMD command has finished
2. NOPs are inserted after each write to FAR command has finished
3. If the write to FAR command is followed by the write to CMD command or by the write to MFWR command, then no NOP is inserted between these commands.
4. A large number of NOPs are inserted at the end of each bitstream file.

Then we performed the following experiments:

1. We removed all NOPs from a working bitstream file and configured the FPGA with the new bitstream file. However, after the configuration process completed, the chip did not function as expected, implying that removing the NOPs changed the behavior of the design.
2. We sent the working bitstream to the ICAP port; however, if a NOP were spotted, we stalled the ICAP for several cycles and jumped to the next command. In this way, we replaced each NOP with several cycles of delay. After the configuration process completed, the design functioned as expected.

The conclusion is that NOPs carry no special meaning to the ICAP. For example, these are not used to flush some buffer internally. The sole purpose of NOP is to insert delay to give the ICAP enough time to finish the current operation.

5. INTELLIGENT ICAP CONTROLLER FOR BITSTREAM SIZE REDUCTION

In the previous section we have verified that we can indeed manually modify the bitstream file in various ways and still maintain ICAP functionality. Based on this property, we propose to exploit a simple form of redundancy. Our approach does not require any complicated compression algorithms; instead it simply scans the bitstream file a word at a time to check how likely it is that the next word is the same as the current word. As an example shown in Figure 4, the same word `0x00000000` repeats itself four times in the middle of the data sequence. We are going to exploit this redundancy pattern to reduce the size of the bitstream files as well as to reduce the configuration data transfer time. The main advantage of this method over others is that it does not require a complicated decompression scheme, thus it minimizes the overhead of the decompression circuit.

```
0x0def8037
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x0539af80
```

Figure 4: sample data sequence

5.1 Redundancy in Bitstream Files

First, we examined the compression ratio achieved by this simple scheme. We scanned through a set of bitstream files with varying sizes and complexities to quantify the simple redundancy. These bitstream files include: *blank* is used to wipe out a design spanning three clock regions; *counters* is a design that consists of 256 counters and spans one clock region; *debug_full* is a full design of the eMIPS base architecture plus a hardware debugger; *debug_partial* is the hardware debugger only; *lr* is a hardware acceleration module for the load-return operation that loads the address of the stack and jumps there; *mmldiv64* is a hardware acceleration module to accelerate 64-bit division; *static_full* is a design of the eMIPS base architecture plus a 64-bit division accelerator; and *timer* is a hardware timer implementation. The results are summarized in table 1: where *new size* is derived by subtracting the redundancy from the *original size*, and *compression ratio* is the ratio of *original size* over *new size*. The results show that even by exploiting this simple redundancy, we can achieve a compression ratio ranging from 1.21 to 3.93, with the average being 1.73.

Table 1: compression ratio of bitstream files

| bitfile | original size (words) | new size (words) | compression ratio |
|---------------|-----------------------|------------------|-------------------|
| blank | 25790 | 17543 | 1.47 |
| counters | 19930 | 13110 | 1.52 |
| debug_full | 244396 | 145470 | 1.68 |
| debug_partial | 31214 | 22267 | 1.40 |
| lr | 28887 | 21048 | 1.37 |
| mmldiv64 | 32487 | 25969 | 1.25 |
| static_full | 244391 | 62231 | 3.93 |
| timer | 32705 | 27054 | 1.21 |

5.2 Design of the Intelligent ICAP Controller

In our scheme, the bitstream files should be pre-compressed and stored in the SRAM. Then after each word is transferred from the SRAM to the ICAP, the ICAP controller examines the word and determines whether decompression is necessary. If so, the ICAP controller performs decompression and sends the decompressed configuration data sequence to the ICAP port; otherwise, the ICAP controller simply forwards the configuration word to the ICAP port.

5.2.1 Encoding and decoding

In order to perform compression and decompression, we need a coding scheme. Our design principle is to keep the decompression circuit as simple as possible. Thus, our coding scheme is very straightforward. As shown in Figure 5, the word `0x00000000` repeats four times in the original bitstream. To encode this in a new bitstream, a new command word `0xecdc0004` is inserted into the bitstream. The upper 16 bits of this word is a special command (`0xecdc`) that signals the decompression circuit to start operation, whereas the lower 16 bits of this word encodes the number of repetitions in the original stream. The word immediately following `0xecdc0004`, in this case `0x00000000`, is the word to be decompressed. Note that if `0xecdcxxxx` existed in the original bitstream, our compression program would detect it and insert a special command to notice the decompression circuit, but we do not go into the details of this mechanism.

```
0x0def8037
0x00000000
0x00000000
0x00000000
0x00000000
0x0539af80
```

`decode`
 \leftarrow
`0x0def8037`
`0xecdc0004`
 \rightarrow
`0x00000000`
`0x0539af80`
`encode`

Figure 5: encoding and decoding

5.2.2 Decompression circuit design

The baseline ICAP controller simply waits for the configuration data and then forwards it to the ICAP port. Our intelligent ICAP controller is able to recognize the decompression command and perform configuration data decompression. In comparison to the baseline ICAP controller, our intelligent controller adds only a small decoding module and a simple state machine to repeatedly send the decompressed configuration word to the ICAP. Due to this simple design, our intelligent ICAP controller introduces little hardware overhead. Table 2 summarizes the hardware resource utilization of the baseline ICAP controller (*baseline*), our intelligent ICAP controller (*new*), and the whole eMIPS design (*eMIPS*). It shows that our intelligent ICAP controller only uses 5% more slices, 12% more slice Flip-Flops, and 9% more 4-input LUTs compared to the baseline ICAP controller. In the context of the whole eMIPS design these overheads are negligible.

Table 2: resource utilization

| | baseline | new | overhead | eMIPS |
|--------------|----------|-----|----------|-------|
| slices | 260 | 274 | 5% | 10416 |
| slice FF | 301 | 336 | 12% | 10398 |
| 4-input LUTs | 336 | 367 | 9% | 19335 |

5.3 Combining the Intelligent ICAP Controller and the Streaming DMA

In order to minimize the configuration data transfer time, we combine the fully streaming DMA engines and the intelligent ICAP controller. To evaluate the performance of our design, we define a metric, the effective transfer throughput. This is equal to the original size of the bitstream divided by the transfer time.

5.3.1 Disruption of the DMA stream

A major problem the intelligent ICAP controller brings is that it may disrupt the DMA operation. For instance, if the ICAP controller receives a decompression command word `0xecdc1000`, the ICAP controller will be busy sending the decompressed configuration data to the ICAP port for the next 4096 cycles. During this time it is not able to fetch new configuration data from the FIFO. On the other hand, the slave DMA engine will continue sending one word per cycle to the FIFO. After the FIFO becomes

full, the incoming configuration data will be dropped. To solve this problem we design a mechanism to pause and resume the DMA operation.

Figure 6 illustrates the extension of the control and data state machines in the slave DMA engine to handle this situation. When decompression starts, the master DMA engine sends a *DMA pause* signal to disrupt the DMA operation in the slave DMA engine. In the control state machine, this signal is masked until it finishes the current burst to prevent the disruption of the burst read. Then the control state machine turns off the read signal to the SRAM interface and transitions to the DMA pause state. On the other hand, the data state machine continues reading data from the SRAM interface until the *data ready* signal from the SRAM interface becomes low. Then the data state machine transitions to the DMA pause state. In the DMA pause state, the control and data state machines become synchronized again, and both state machines wait for the *DMA resume* signal from the master DMA engine to re-start the DMA operation. Note that each DMA pause/resume process introduces a control overhead of 6 cycles.

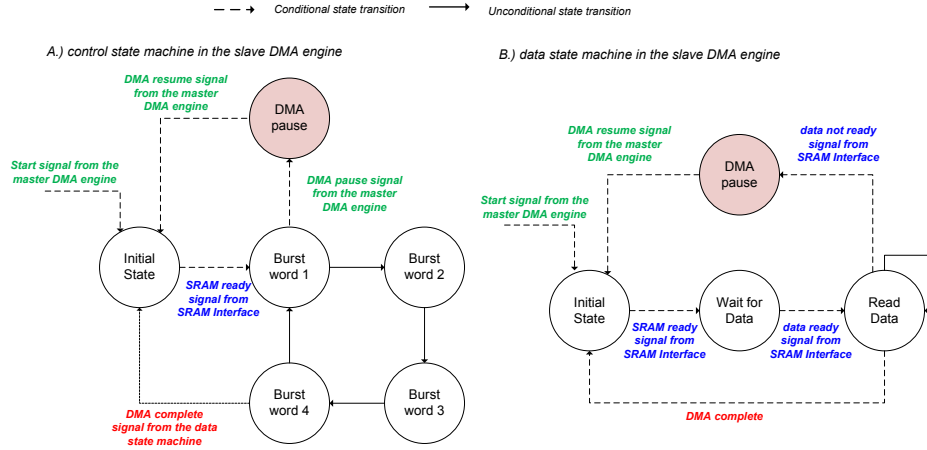


Figure 6: the state machines in the slave DMA engine (with DMA pause and resume capability)

5.3.2 Memory access time sharing

In the eMIPS system, both instruction and data are stored in the SRAM. Thus, during DMA operations the SRAM is not able to service normal memory operations. Nevertheless, as shown in Figure 7, with our intelligent ICAP controller design we can free the SRAM from DMA operations when the ICAP controller is performing decompression. This allows memory access time sharing between the DMA operations and the normal memory operations. In the next section, we show how this technique improves the program execution time compared to the case where only the baseline ICAP controller is used.

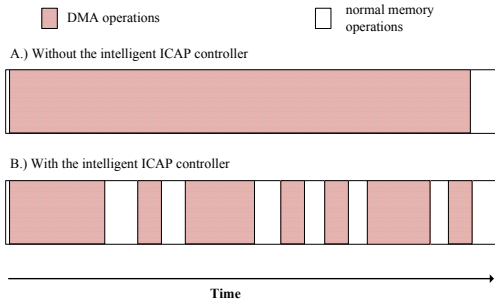


Figure 7: memory access time sharing

6. EXPERIMENTS AND RESULTS

We have performed experiments on the Virtex-4 FPGA to study how the proposed techniques improve FPGA performance. In this section, we show our experimental results on the fully streaming DMA design, the intelligent ICAP controller, and the memory access time sharing scheme. In addition, we present a case study on the 64-bit division accelerator to demonstrate the effectiveness of the combination of these techniques.

6.1 Performance of the Fully Streaming DMA Engines

In order to measure the performance of our fully streaming DMA design, we tested it with the same eight bitstream files as shown in Table 1. Recall that these eight bitstream files have varying sizes and complexities. By inserting hardware cycle counters into the design, we measured the time taken to complete the DMA operation and the results are summarized in Table 3: the second column shows the size of the bitstream file, the third column shows the time taken to complete the DMA operation without the fully streaming DMA engines (in this original design, the ICAP throughput is only 318 Kbytes/s), the fourth column shows the time taken to complete the DMA operation using our fully streaming

DMA engines, and the fifth column shows the throughput of the DMA operation (derived by dividing the size of the bitstream by the time taken to complete the DMA operation). In all eight cases, our fully streaming DMA design achieved an ICAP throughput that was higher than 399 Mbytes/s. This high ICAP throughput reduces the configuration time from the range of seconds to the range of milliseconds, a more than 1000-fold improvement.

Table 3: DMA data transfer throughput

| <i>bitfile</i> | <i>size (word)</i> | <i>original time (seconds)</i> | <i>time (seconds)</i> | <i>TP (Mbytes/s)</i> |
|----------------|------------------------|----------------------------------------|---------------------------|--------------------------|
| blank | 25790 | 0.32 | 0.00026 | 399.71 |
| counters | 19930 | 0.24 | 0.00020 | 399.62 |
| debug_full | 244006 | 3.00 | 0.00244 | 399.33 |
| debug_partial | 31212 | 0.38 | 0.00031 | 399.78 |
| lr | 28886 | 0.35 | 0.00029 | 399.74 |
| mmldiv64 | 32486 | 0.40 | 0.00033 | 399.77 |
| static_full | 244000 | 3.00 | 0.00244 | 399.33 |
| timer | 32140 | 0.39 | 0.00032 | 399.79 |

6.2 Performance of the Intelligent ICAP Controller

Next, we combined the fully streaming DMA engines and the intelligent ICAP controller. We studied the performance of this design with the same set of bitstream files. Recall that there is a 6-cycle overhead associated with each disruption of the DMA operation. Therefore, it is not worthwhile to compress the data sequence that contains less than 6 repetitions of the same configuration word. In our experiments, we define the threshold for compression as 10 repetitions of the same configuration word. In this case, we sacrifice some compression ratio but guarantee that

each compression can contribute to the reduction of data transfer time.

The experimental results are summarized in Table 4. The second, third, and fourth columns show the original bitstream file size, the new bitstream file size after compression, and the compression ratio, respectively. The fifth and sixth columns show the time taken to transfer the bitstream and the time taken for the ICAP to complete configuration, respectively. Finally, the last two columns respectively show the effective data transfer throughput and the ICAP throughput. Note that *effective transfer throughput* is derived by dividing *original size* by *transfer time*, whereas *ICAP throughput* is derived by dividing *new size* by *ICAP time*.

The first observation from Table 4 is that our scheme leads to significant reduction of the bitstream file size, in the case of *static_full*, our scheme reduces the file size by more than 75%. The second observation is that a higher compression ratio leads to a higher effective transfer throughput. In the case of *timer*, the compression ratio is only 1.09 and the *effective transfer throughput* is 434 Mbytes/s. In contrast, in the case of *static_full* the compression ratio reaches 3.15. This leads to a 1203 Mbytes/s *effective transfer throughput*. This is because a higher compression ratio implies a larger reduction of the bitstream file size. Thus it takes less time to transfer the configuration data. The third observation is that *ICAP throughput* ranges from 374 Mbytes/s to 392 Mbytes/s, whereas we show in Table 3 that the throughput should be greater than 399 Mbytes/s. This is because each transition from the normal ICAP operation to decompression operation involves the transition from one state machine to another, thus it incurs a two-cycle overhead. Consequently, the more often the transition occurs, the more overhead it incurs.

The most important message conveyed by Table 4 is that by combining the fully streaming DMA design, which aims at improving configuration data transfer throughput, and the intelligent ICAP controller, which aims at reducing the size of bitstream files, we are able to achieve an effective configuration data transfer throughput that well surpasses the upper bound of data transfer throughput, 400 Mbytes/s.

Table 4: performance of the intelligent ICAP controller

| <i>bitfile</i> | <i>original size (words)</i> | <i>new size (words)</i> | <i>compression ratio</i> | <i>transfer time (seconds)</i> | <i>ICAP time (seconds)</i> | <i>effective transfer throughput (Mbytes/s)</i> | <i>ICAP throughput (Mbytes/s)</i> |
|----------------|--------------------------------------|---------------------------------|------------------------------|----------------------------------------|--------------------------------|-------------------------------------------------------------|-------------------------------------------|
| blank | 25790 | 21474 | 1.20 | 0.000217 | 0.000266 | 474.65 | 387.12 |
| counters | 19930 | 15684 | 1.27 | 0.000160 | 0.000209 | 499.69 | 381.93 |
| debug_full | 244006 | 161640 | 1.51 | 0.001628 | 0.002485 | 599.59 | 392.74 |
| debug_partial | 31212 | 26450 | 1.18 | 0.000268 | 0.000324 | 465.73 | 385.39 |
| lr | 28886 | 24880 | 1.16 | 0.000251 | 0.000298 | 459.69 | 388.27 |
| mmldiv64 | 32486 | 28942 | 1.12 | 0.000292 | 0.000337 | 444.30 | 385.59 |
| static_full | 244000 | 77382 | 3.15 | 0.000811 | 0.002552 | 1203.90 | 382.43 |
| timer | 32704 | 29920 | 1.09 | 0.000301 | 0.000349 | 434.55 | 374.77 |

6.3 Memory Access Time Sharing

We performed an experiment to identify the effectiveness of the memory access time sharing scheme. The test program consists of two parts: the first part performs a DMA configuration file transfer, and the second part is a sequence of numerical computation that takes eMIPS roughly 10 milliseconds to complete. We used this

test program on the eight bitstream files and the results are summarized in Table 5.

The second and third columns, respectively, show the time taken for the baseline ICAP and the intelligent ICAP to finish the configuration process. The fourth and fifth columns, respectively, show the execution time of the original design with the baseline

ICAP and that of the new design with the intelligent ICAP; finally the sixth column shows the speedup, or the ratio of *original execution time* over *new execution time*.

With the intelligent ICAP controller, the transition from the normal ICAP operations to the decompression operations incur some performance overheads, thus *intelligent ICAP time* is higher than *baseline ICAP time*. However, the memory access time sharing scheme allows the memory to service the normal memory operations when the DMA operation is disrupted, thus reducing the

overall execution time. The combined effect of these two tradeoffs leads to an overall program speedup. In this case, it ranges from 1.00 (as in *timer* and *lr*) to 1.17 (as in *static_full*). By comparing the data in Table 5 to that in Table 4, we conclude that a higher compression ratio leads to a higher program speedup because more instructions can be executed during the disruption of DMA operations.

Table 5: speedup through memory access time sharing

| <i>bitfile</i> | <i>baseline ICAP time (seconds)</i> | <i>intelligent ICAP time (seconds)</i> | <i>original execution time (seconds)</i> | <i>new execution time (seconds)</i> | <i>Speedup</i> |
|----------------|-------------------------------------|----------------------------------------|------------------------------------------|-------------------------------------|----------------|
| blank | 0.000258 | 0.000266 | 0.010667 | 0.010580 | 1.01 |
| counters | 0.000199 | 0.000209 | 0.010608 | 0.010543 | 1.01 |
| debug_full | 0.002444 | 0.002485 | 0.012853 | 0.011905 | 1.08 |
| debug_partial | 0.000312 | 0.000324 | 0.010721 | 0.010631 | 1.01 |
| lr | 0.000289 | 0.000298 | 0.010698 | 0.010679 | 1.00 |
| mmldiv64 | 0.000325 | 0.000337 | 0.010734 | 0.010669 | 1.01 |
| static_full | 0.002444 | 0.002552 | 0.012853 | 0.011025 | 1.17 |
| timer | 0.000322 | 0.000349 | 0.010736 | 0.010683 | 1.00 |

6.4 Case Study: 64-bit Division Accelerator

In the ideal scenario, a hardware accelerator can be loaded to accelerate portions of a program. When the accelerator is not active, we can unload the accelerator to either save power or to give room to other accelerators. However, if the performance overhead imposed by the reconfiguration process were overwhelming, this approach would not be feasible. We performed a case study to identify the conditions under which our design would allow this approach to be feasible. In our case study, we use a 64-bit division accelerator and our test program first starts the DMA operation to load the hardware accelerator, and then it performs 64-bit divisions in the rest of the program.

The results of our case study are shown in Figure 8: the x-axis indicates the 64-bit division workload. Each increment of the x-axis represents a sequence of 64-bit divisions, which takes the baseline eMIPS (without extension) about 1 millisecond to complete. The accelerator is able to cut the 64-bit division time by half, thus the same sequence of 64-bit divisions takes the accelerator only about 0.5 millisecond to complete. The y-axis shows the execution time. In this experiment, we compared the execution time of three designs: *no_ext* represents the baseline eMIPS design that does not use accelerator; *ext1* represents the eMIPS with accelerator, but only the fully streaming DMA engines are used to stream the configuration data; and *ext2* represents the same design as in *ext1*, but both the fully streaming DMA engines and the intelligent ICAP controller are used to stream the configuration data.

In this case, configuration only takes 0.2 milliseconds. When the 64-bit division work load is small ($x = 1$), the configuration process imposes a significant overhead of the execution time, thus we do not observe a 2x speedup by using the accelerator. However, as the workload increases the configuration overhead becomes negligible. Note that the 64-bit division work load in this experiment is very small. Even in the case $x = 5$, the total program execution time of *no_ext* is only 5 milliseconds.

Therefore, we show that with our fully streaming DMA using partial reconfiguration to load/unload accelerators becomes beneficial even when the workload to be accelerated is in the millisecond range. This extremely low configuration overhead enables partial reconfiguration to be an effective technique to improve system performance (by loading the accelerators at runtime) and may potentially lead to energy reduction (by unloading the accelerators when acceleration is not necessary).

In addition, the intelligent ICAP controller further reduces the execution time by 1% ($x = 5$) to 5% ($x = 1$) because it allows the SRAM to service both DMA operations and normal memory operations in a time-sharing fashion. Note that the compression ratio of the accelerator bitstream file is only 1.12, a fairly low number. Given a design with higher compression ratio, the improvement brought by the intelligent ICAP controller would be much higher.

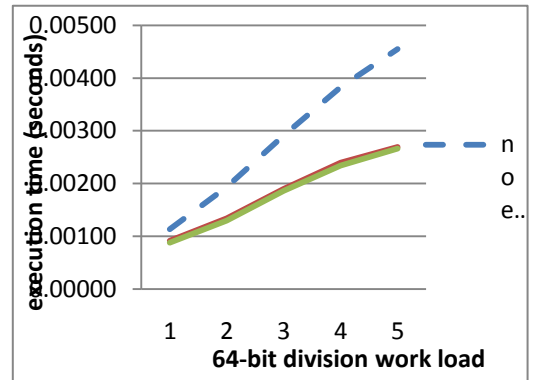


Figure 8: case study with 64-bit division accelerator

7. CONCLUSIONS

In order to minimize the configuration overhead, we proposed a combination of two techniques: one to improve configuration data transfer throughput, and the other to reduce the size of configuration bitstreams. We studied how these designs can improve performance.

First, we designed and implemented fully streaming DMA engines to improve configuration throughput. The experimental results show that our fully streaming DMA engines nearly saturate the throughput of the internal ICAP and reduce the configuration time from the range of seconds to the range of milliseconds, a more than 1000-fold improvement. Second, our low-level study on the configuration process indicates that we can manually modify the bitstream files and the configuration time is a pure function of bitstream size. Third, our compression scheme achieves up to 75% reduction of bitstream size and results in a decompression circuit with negligible hardware overhead.

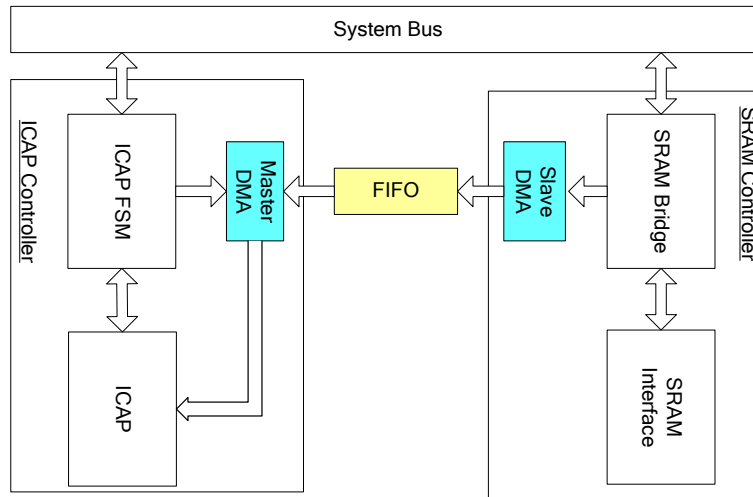
The combination of these two techniques achieves an effective configuration data transfer throughput of up to 1.2 Gbytes/s, which well surpasses the 400 Mbytes/s data transfer throughput upper bound. In addition, our design allows memory access time sharing and results in up to 17% further performance improvement.

REFERENCES

1. M. Liu, W. Kuehn, Z. Lu, and A. Jantsch, "run-time partial reconfiguration speed investigation and architectural design space exploration," in proceedings of IEEE International Conference on Field Programmable Logic and Applications, 2009.
2. C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hubner, and J. Becker, "a multi-platform controller allowing for maximum dynamic partial reconfiguration throughput," in proceedings of IEEE International Conference on Field Programmable Logic and Applications, 2008.
3. K. Paulsson, M. Hubner, G. Auer, M. Dreschmann, L. Chen, and J. Becker. "Implementation of a virtual internal configuration access port (JCAP) for enabling partial self-reconfiguration on Xilinx Spartan II FPGAs," in proceedings of International Conference on Field Programmable Logic and Applications, 2007.
4. Virtex-4 FPGA User Guide:
http://www.xilinx.com/support/documentation/user_guides/u_g070.pdf
5. Z. Li and S. Hauck, "configuration compression for Virtex FPGAs," in proceedings of the IEEE symposium on Field-Programmable Custom Computing Machines, 2001.
6. A. Dandalis and V.K. Prasanna, "configuration compression for FPGA-based embedded systems," in proceedings of ACM/SIGDA Symposium on Field-Programmable Gate Arrays, 2001.
7. L. He, T. Mitra, and W-F. Wong, "configuration bitstream compression for dynamically reconfigurable FPGAs," in proceedings of the IEEE/ACM International Conference on Computer-Aided Design, 2004.
8. J. Resano, D. Mozos, and F. Catthoor, "a hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware," in proceedings of the conference on Design, Automation and Test in Europe, 2005.
9. Z. Li and S. Hauck, "Configuration Prefetching Techniques for Partial Reconfigurable Coprocessor with Relocation and Defragmentation," in proceedings of ACM/SIGDA Symposium on Field-Programmable Gate Arrays, 2002.
10. J. Carver, R.N. Pittman, A. Forin, "Relocation and Automatic Floor-planning of FPGA Partial Configuration Bit-Streams," MSR-TR-2008-111, Microsoft Research, WA, August 2008.
11. D.B. Thomas and W. Luk, "multivariate gaussian random number generation targeting reconfigurable hardware," ACM Transactions on Reconfigurable Technology and Systems, vol. 1, no. 2, June 2008.
12. Virtex-4 FPGA Configuration User Guide:
http://www.xilinx.com/support/documentation/user_guides/u_g071.pdf

APPENDIX 1: DESIGN OF FULLY STREAMING DMA AND THE INTELLIGENT ICAP CONTROLLER

NOTE: ALL CODE CHANGES ARE TAGGED WITH “**/**SL**/**”



In **MEMORY_CONTROLLER**:

1. Modification of the ICAP controller inputs/outputs

```

/**SL**/
*****ICAP Peripheral*****
icap_controller ICAPcntr(
    .ADDR_IN(ADDR_IN),
    .BYTES(BYTES),
    .CADDR(ICAP_ADDR),
    .CLK(MEMCLK),
    .DATA_IN(DATA_AL),
    .DATA_OUT(DATA_OUT_0),
    .DNE(DNE_ICAP),
    //GPIO_IN(GPIO_IN),
    //GPIO_IRQ(GPIO_IRQ),
    //GPIO_OUT(GPIO_OUT),
    //GPIO_TR(GPIO_TR),
    .OE(OE),
    .RESET(RESET & ICAP_EN),
    .SRT(srt_r),
    .WE(WE),

/**SL**// the following I/O ports are for DMA to SRAM
    .DMAF_DOUT(dmaf_dout),
    .DMAF_EMPTY(dmaf_empty),
    .DMAF_RE(dmaf_re),
    .DMAF_DATA_COUNT(dmaf_data_count),
    .DMA_OP_SRAM(DMAOP_ICAP_SRAM),
    .ADDR_OUT_SRAM(ADDR_ICAP_SRAM),
    .SIZE_OUT_SRAM(SIZE_ICAP_SRAM)
);
  
```

2. Addition of a DMA FIFO

```
/**SL**/  
/**** DMA FIFO *****/  
dma_fifo dmaf(  
    .clk(MEMCLK),                //common clock  
    .rd_en(dmaf_re),             //read enable  
    .wr_en(dmaf_we),             //write enable  
    .din(dmaf_din),              //32-bit data input from SRAM  
    .dout(dmaf_dout),            //32-bit data output to ICAP DMA  
    .data_count(dmaf_data_count), //fifo 4-bit data count  
    .empty(dmaf_empty),          //empty status bit  
    .full(dmaf_full),            //full status bit  
);
```

3. Modification of outgoing signals to the SRAM device

```
/* Outgoing Signals */  
BSMUX1_2to1 bsmux0(  
    .a0(NWE_SRAM),  
    .a1(NWE_FLASH),  
    .def(1'b1),  
    .en0(DMA_BUSY | (~DNE_SRAM)), /**SL**/ //en0(~DNE_SRAM),  
    .en1(~DNE_FLASH),  
    .out(NWE_MEM)  
);  
  
BSMUX1_2to1 bsmux1(  
    .a0(NOE_SRAM),  
    .a1(NOE_FLASH),  
    .def(1'b1),  
    .en0(DMA_BUSY | (~DNE_SRAM)), /**SL**/ //en0(~DNE_SRAM),  
    .en1(~DNE_FLASH),  
    .out(NOE_MEM)  
);  
  
BSMUX24_2to1 bsmux2(  
    .a0(ADDR_SRAM),  
    .a1(ADDR_FLASH),  
    .def(24'b0),  
    .en0(DMA_BUSY | (~DNE_SRAM)), /**SL**/ //en0(~DNE_SRAM),  
    .en1(~DNE_FLASH),  
    .out(ADDR_MEM)  
);  
  
BSMUX32_2to1 bsmux3(  
    .a0(SRAMDQ_IN),  
    .a1(FLASHDQ_IN),  
    .def(32'b0),  
    .en0(DMA_BUSY | (~DNE_SRAM)), /**SL**/ //en0(~DNE_SRAM),  
    .en1(~DNE_FLASH),  
    .out(DATA_MEM_IN)  
);  
  
BSMUX32_2to1 bsmux4(  
    .a0(SRAMDQ_TR),  
    .a1(FLASHDQ_TR),  
    .def(32'hfffffff),  
    .en0(DMA_BUSY | (~DNE_SRAM)), /**SL**/ //en0(~DNE_SRAM),  
    .en1(~DNE_FLASH),
```

```
.out(DATA_MEM_TR)
);
```

In **SRAM_CONTROLLER**:

1. Modification to the input/output ports of SRAM_CONTROLLER

```
/**SL**/ //ICAP/SRAM DMA I/O ports
input [31:0] ADDR_ICAP,           //starting address of DMA operation
input [31:0] SIZE_ICAP,           //dma transfer size
input DMA_OP_ICAP,               //dma operation signal
output [31:0] DMAF_DIN,           //dma fifo input from SRAM
output DMAF_WE,                  //dma fifo write enable
input DMAF_FULL,                 //dma fifo full signal
output DMA_BUSY                  //slave DMA busy
```

2. Modification to the inputs/outputs of SRAM_BRIDGE

```
sram_bridge sb(
  .ADDR_IN(ADDR_IN[24:0]),
  .ADDR_OUT(ADDR_OUT_C),
  //BURST_ORDER(control[BURST_ORDER]),
  .BYTES(BYTES),
  .CE2(CE2),
  .CEN(NCEN),
  .CLOCK_MASK(control[CLOCK_MASK]),
  .DATA_IN(DATA_IN),
  .DATA_OUT(DATA_SRAM),
  .DNE(DNE_SRAM),
  .DQ_IN(DQ_IN),
  .DQ_OUT(DQ_OUT),
  .DQ_TR(DQ_TR),
  .DQP_IN(DQP_IN),
  .DQP_OUT(DQP_OUT),
  .DQP_TR(DQP_TR),
  .MODE(MODE),
  .NADVLD(NADVLD),
  .NBW(NBW),
  .NCE1(NCE1),
  .NCE3(NCE3),
  .NOE(NOE0),
  .NWE(NWE0),
  .PR(PR),
  .RESET(RESET & resetcnt[2]),
  .SLEEP(control[SLEEP]),
  .SRAMCLK(SRAMCLK),
  .SRT(SRT & en_reg),
  .WE(WE & en_reg),
  .OE(OE & en_reg),
  .ZZ(ZZ),

  /**SL**/ //DMA operation signals
  .DMA_OP_ICAP(DMA_OP_ICAP),
  .ADDR_ICAP(ADDR_ICAP[24:0]),           //starting address of DMA operation
  .SIZE_ICAP(SIZE_ICAP),                 //dma transfer size
  .DMAF_DIN(DMAF_DIN),                   //dma fifo input from SRAM
  .DMAF_WE(DMAF_WE),                     //dma fifo write enable
  .DMAF_FULL(DMAF_FULL),                 //dma fifo full signal
  .DMA_BUSY(DMA_BUSY)
);
```

In **SRAM_BRIDGE**:

1. Modifications to the SRAM_BRIDGE input/output ports

```

/**SL**/ //DMA operation ports
input DMA_OP_ICAP,
input [24:0] ADDR_ICAP,
input [31:0] SIZE_ICAP,
output [31:0] DMAF_DIN,
output DMAF_WE,
input DMAF_FULL,
output DMA_BUSY

//DMA operation signal
//starting address of DMA operation
//dma transfer size
//dma fifo input from SRAM
//dma fifo write enable
//dma fifo full signal
//slave DMA busy

```

2. New registers/wires for the streaming operations

```

/**SL**/
/***** SL *****/
reg bl; /* Burst Latch */
wire [31:0] data_out_wire; //data_out from SRAM_Interface
wire [3:0] bw_wire; //byte enable wire
wire drdy_wire; //data ready wire

//streaming
reg [3:0] stream_control_state;
reg [3:0] stream_data_state;
reg [31:0] size_reg_dma;
reg [31:0] dmaf_din;
reg dmaf_we;
reg bl_dma, en_dma, cen_r_dma, zz_r_dma, oen_dma, wen_dma;
reg [24:0] addr_in_reg_dma;
reg dne_r_dma;
reg dma_busy;
wire select;
assign select = dma_busy & dne_r; /**SL**/
assign DMAF_DIN = dmaf_din;
assign DMAF_WE = dmaf_we;
assign DATA_OUT = data_out_wire;
assign DMA_BUSY = dma_busy;
/***** SL *****/

```

3. Modifications to the input/output signals to SRAM_INTERFACE

```

sram_interface si(
.ADDR_IN(select ? addr_in_reg_dma : ADDR_IN), //ADDR_IN /**SL**/
.ADDR_OUT(ADDR_OUT),
.ADVLD(NADVLD),
.BUSY(BUSY0),
.BURST_LATCH(select ? bl_dma : bl), /**SL**/
.BURST_ORDER(1'b0), //hardcoded to linear mode /**SL**/
.BW(NBW),
.BYTES(BYTES),
.CE1(NCE1),
.CE2(CE2),
.CE3(NCE3),
.CEN(CEN),
.CLK(SRAMCLK),
.CLOCK_MASK(select ? cen_r_dma : cen_r), /**SL**/
.DATA_IN(DATA_IN),
.DATA_IO_IN(DQ_OUT),
.DATA_IO_OUT(DQ_IN),

```



```

.DATA_OUT(data_out_wire),          /**SL**/
.DATA_PIO_IN(DQP_OUT),
.DATA_PIO_OUT(DQP_IN),
.DIR(DIR),
.DRDY(drdy_wire),                  /**SL**/
.EN(select ? en_dma : en),         /**SL**/
.END(END),
.MODE(MODE),
.OE(NOE),
.OEN(select ? oen_dma : oen),      /**SL**/
.PARE(PR),
.RESET(RESET),
.SLEEP(select ? zz_r_dma : zz_r),  /**SL**/
.WE(NWE),
.WEN(select ? wen_dma : wen),      /**SL**/
.ZZ(ZZ)
);

```

4. Normal memory operation state machine

```

/**SL**/
/*****SL*****/
always@(posedge SRAMCLK) begin
    if(~select) begin
        bl = 1'b0;
        if (RESET == 0) begin
            /* Reset */
            wen = 1'b1;
            oen = 1'b1;
            en = 1'b1;
            zz_r = 1'b0;
            cen_r = 1'b0;
            //bo = 1'b0;
            dne_r = 1'b1;
            bsy = 1'b0;
        end
    end
    else begin
        if (SLEEP) begin
            if (~BUSY) begin
                /* Put SRAM IC to sleep */
                zz_r = 1'b1;
            end
        end
        else begin
            zz_r = 1'b0;
        end

        if (CLOCK_MASK) begin
            if (~BUSY) begin
                /* Disable SRAM Clock */
                cen_r = 1'b1;
            end
        end
        else begin
            cen_r = 1'b0;
        end

        if (zz_r || cen_r) begin
            if (SRT && dne_r) begin
                dne_r = 1'b0;
            end
        end
    end
end

```

```

        else begin
            dne_r = 1'b1;
        end
    end
else begin
    if (SRT && ~BUSY) begin
        /* Recieved Request */
        en = 1'b0;
        dne_r = 1'b0;
    end
    else if (BUSY && END && ~BUSY0) begin
        bsy = 1'b1;
        if (WE) wen = 1'b0;
        if (OE) oen = 1'b0;
    end
    else if (BUSY) begin
        en = 1'b1;
        wen = 1'b1;
        oen = 1'b1;
    end
    else if (~dne_r && ~BUSY && bsy) begin
        /* Interface Done */
        bsy = 1'b0;
        dne_r = 1'b1;
    end
end
end
end
end
end
end

```

5. DMA operation control state machine

```

/***** streaming *****/
always@(posedge SRAMCLK) begin //stream control state machine
    case(stream_control_state)
        4'b0000: begin

            if(DMA_OP_ICAP & dne_r) begin //wait for start
                zz_r_dma = 1'b0;
                cen_r_dma = 1'b0;
                en_dma = 1'b1;
                oen_dma = 1'b1;
                wen_dma = 1'b1;

                addr_in_reg_dma = {ADDR_ICAP[24:4], 4'b0000}; //latch address
                stream_control_state = 4'b0001;
            end
        end
        4'b0001: begin //wait for SRAM to be ready

            if(DMA_OP_ICAP & dne_r) begin
                if(~BUSY) begin
                    en_dma = 1'b0;
                    oen_dma = 1'b1;
                    wen_dma = 1'b1;
                    stream_control_state = 4'b0010;
                end
            end
        end
        else stream_control_state = 4'b1000;
    endcase
end

```

```

end
4'b0010: begin                                     // wait for SRAM to be ready
    if(DMA_OP_ICAP & dne_r) begin
        if (BUSY & END & ~BUSY0) begin
            stream_control_state = 4'b0011;
        end
    end
    end
    else stream_control_state = 4'b1000;
end
4'b0011: begin //burst word 0
    bl_dma = 1'b0;
    en_dma = 1'b0;
    oen_dma = 1'b0;
    stream_control_state = 4'b0100;
end
4'b0100: begin                                     //burst word 1
    bl_dma = 1'b1;
    oen_dma = 1'b0;
    en_dma = 1'b0;
    stream_control_state = 4'b0101;
end
4'b0101: begin                                     //burst word 2
    bl_dma = 1'b1;
    oen_dma = 1'b0;
    en_dma = 1'b0;
    stream_control_state = 4'b0110;
end
4'b0110: begin                                     //burst word 3
    if(dne_r_dma) begin                             //done
        bl_dma = 1'b1;
        oen_dma = 1'b1;
        en_dma = 1'b1;
        wen_dma = 1'b1;
        if(~DMA_OP_ICAP) begin
            stream_control_state = 4'b0000;
        end
    end
    else begin
        bl_dma = 1'b1;
        oen_dma = 1'b0;
        en_dma = 1'b0;
        addr_in_reg_dma = addr_in_reg_dma + 16;      //increment addr to next burst

        if(~DMA_OP_ICAP) begin
            stream_control_state = 4'b0111;
        end
        else begin
            stream_control_state = 4'b0011;
        end
    end
end
4'b0111: begin
    stream_control_state = 4'b1000;
end
4'b1000: begin //pause
    bl_dma = 1'b1;
    oen_dma = 1'b1;
    en_dma = 1'b1;
    wen_dma = 1'b1;

    if(DMA_OP_ICAP & dne_r) begin //wait for resume
        zz_r_dma = 1'b0;
    end
end

```

```

        cen_r_dma = 1'b0;

        stream_control_state = 4'b0001;
    end

    if(dne_r_dma) begin //done
        if(~DMA_OP_ICAP) begin
            stream_control_state = 4'b0000;
        end
    end

end
endcase
end

```

6. DMA operation data state machine

```

always@(posedge SRAMCLK) begin //stream data state machine
    case(stream_data_state)
        4'b0000: begin

            if(DMA_OP_ICAP & dne_r) begin //start streaming operation
                dma_busy = 1'b1;
                dne_r_dma = 1'b0;
                size_reg_dma = SIZE_ICAP; //latch size
                stream_data_state = 4'b0001;
            end
            else begin
                dma_busy = 1'b0;
                dne_r_dma = 1'b1;
            end

        end
        4'b0001: begin //wait for SRAM to be ready

            if(DMA_OP_ICAP & dne_r) begin
                dma_busy = 1'b1;
                if(~BUSY) stream_data_state = 4'b0010;
            end
            else stream_data_state = 4'b0110;

        end
        4'b0010: begin //make sure SRAM is working
            if(DMA_OP_ICAP & dne_r) begin
                if(BUSY & END & ~BUSY0) stream_data_state = 4'b0100;
            end
            else stream_data_state = 4'b0110;

        end
        4'b0100: begin
            if(drdy_wire) stream_data_state = 4'b0101;

        end
        4'b0101: begin
            if(size_reg_dma == 0) begin //done
                dne_r_dma = 1'b1;
                dma_busy = 1'b0;
                dmaf_we = 1'b0;
                if(~DMA_OP_ICAP) begin
                    stream_data_state = 4'b0000;
                end
            end
            else if(~DMA_OP_ICAP) begin //pause

                if(~drdy_wire) begin

```

```

        dmaf_we = 1'b0;
        stream_data_state = 4'b0110;
    end
    else if(~DMAF_FULL) begin
        dmaf_we = 1'b1;
        dmaf_din = data_out_wire;
        size_reg_dma = size_reg_dma - 1;
    end
end
else begin
    if(drdy_wire & ~DMAF_FULL) begin
        dmaf_we = 1'b1;
        dmaf_din = data_out_wire;
        size_reg_dma = size_reg_dma - 1;
    end
end
end
4'b0110: begin
    dmaf_we = 1'b0;
    dma_busy = 1'b0;
    if(DMA_OP_ICAP & dne_r) begin
        dne_r_dma = 1'b0;
        stream_data_state = 4'b0001;
    end
end
endcase
end
/*****/

```

In ICAP_CONTROLLER:

1. Modifications to the ICAP input/output ports to support DMA operations

```

/***** SL *****/
input [31:0] DMAF_DOUT, //data from DMA fifo
input DMAF_EMPTY, //DMA fifo empty
input [3:0] DMAF_DATA_COUNT, //DMA fifo data count
output DMAF_RE, //DMA fifo read enable
output DMA_OP_SRAM, //start DMA signal to SRAM
output [31:0] ADDR_OUT_SRAM, //address out to SRAM
output [31:0] SIZE_OUT_SRAM //size out to SRAM
/*****/

```

2. New registers/wires for DMA data and control signals

```

/**SL**/
/***** SL *****/
reg [31:0] dma_size_reg; //DMA size register
reg [31:0] dma_addr_reg; //DMA starting address
parameter DMASIZEREG = 16'h0008;
parameter DMAADDRREG = 16'h000c;
reg [2:0] state;
reg dma_start;
wire dma_done;
wire [31:0] data_out_icap;
wire icap_cen;
wire icap_wen;

```

```

//DMA counter
reg [31:0] dma_counter;
parameter DMACOUNTER = 16'h0010;
assign REG_M_0 = (ADDR_IN[BASEr-1:0] == DMACOUNTER)?dma_counter:32'bz;
reg [31:0] dma_eff_counter;
wire [31:0] dma_eff_count_wire;
parameter DMAEFFCOUNTER = 16'h0014;
assign REG_M_0 = (ADDR_IN[BASEr-1:0] == DMAEFFCOUNTER)?dma_eff_counter:32'bz;
/*****

```

3. Addition of the DMA controller

```

/**SL**/
/*****DMA controller *****/
dma_controller dmacntrl(
    //inputs
    .CLK(CLK),
    .START(dma_start), //command to start DMA operation from

    .ADDR_IN(dma_addr_reg), //DMA starting address
    .SIZE(dma_size_reg), //DMA transfer size
    .DMAF_DOUT(DMAF_DOUT), //data in from SRAM
    .DMAF_EMPTY(DMAF_EMPTY),
    .DMAF_DATA_COUNT(DMAF_DATA_COUNT),
    //outputs
    .DMAF_RE(DMAF_RE),
    .DONE(dma_done), //DMA done
    .DMA_OP(DMA_OP_SRAM), //indicate DMA start op to SRAM
    .ADDR_OUT(ADDR_OUT_SRAM), //address out to SRAM
    .SIZE_OUT(SIZE_OUT_SRAM), //size out to SRAM
    .DATA_OUT(data_out_icap), //data out to ICAP
    .ICAP_CE_N(icap_cen), //clock enable signal to ICAP
    .ICAP_WE_N(icap_wen), //write enable signal to ICAP
    .DMA_EFF_COUNT(dma_eff_count_wire), //effective bandwidth cycle counter
    .ICAP_BUSY(BUSY_ICAP)
);

```

ICAPcntrl

4. Modification of the ICAP controller state machine to start DMA operations

```

DMASIZEREG : begin
    dma_size_reg = data_in_m_reg; //update dma transfer size
    dner = 1'b1;
end
DMAADDRREG : begin
    if(dma_done) begin
        dma_addr_reg = data_in_m_reg;
        dma_start = 1'b1;
        dner = 1'b1;
    end
    else dma_start = 1'b0;
end

```

5. Addition of an independent state machine to control DMA operations

```

always@(posedge CLK) begin //state machine for DMA

    CE_n = icap_cen;
    WRITE_n = icap_wen;
    ICAP_datain = data_out_icap;

```

```

        case (state)
        3'b000: begin
            if(dma_start) begin
                state = 3'b001;
                dma_counter = 32'h00000000; //reset DMA_counter
                dma_eff_counter = 32'h00000000; //reset DMA effective counter
            end
        end
        3'b001: begin //make sure DMA starts
            dma_counter = dma_counter + 1;
            if(~dma_done) state = 3'b010;
        end
        3'b010: begin //wait for DMA to finish
            dma_counter = dma_counter + 1;
            if(dma_done) begin
                dma_eff_counter = dma_eff_count_wire;
                state = 3'b000;
            end
        end
    endcase
end

```

In ***DMA_CONTROLLER***:

The master fully stream DMA engine: it is able to decompress the compressed bitstreams and performs DMA operations.

```

module dma_controller(
    input CLK,
    input START, //start DMA operation
    input [31:0] ADDR_IN, //DMA start address
    input [31:0] SIZE, //DMA size

    //DMA fifo (DMAF)
    input [31:0] DMAF_DOUT, //Data out from DMA_FIFO
    input DMAF_EMPTY, //DMA_FIFO empty
    input [3:0] DMAF_DATA_COUNT,
    output DMAF_RE, //DMA_FIFO ready enable

    output DMA_OP, //DMA operation signal to SRAM (arbitration)
    output [31:0] ADDR_OUT, //address to SRAM
    output [31:0] SIZE_OUT, //size to SRAM
    output DONE, //DMA operation done

    output [31:0] DATA_OUT, //data to ICAP
    output ICAP_CE_N, //clock enable signal to ICAP
    output ICAP_WE_N, //write enable signal to ICAP
    output [31:0] DMA_EFF_COUNT, //DMA effective counter
    input ICAP_BUSY

);

/*****
    reg [2:0] state;
    reg dmaf_re;
    reg [31:0] size, size_original, addr;
    reg dma_op, done, icap_cen, icap_wen;
    reg [1:0] bytes;
    wire [31:0] data_out_wire;
    reg [31:0] data_out_reg;

```

```

reg [31:0] addr_debug;                                //debug

assign DMAF_RE = dmaf_re;
assign DONE = done;
assign DMA_OP = dma_op;
assign ADDR_OUT = addr;
assign SIZE_OUT = size_original;
assign ICAP_CE_N = icap_cen;
assign ICAP_WE_N = icap_wen;

//intelligent ICAPctrl
reg [15:0] size_decode;
assign DATA_OUT = data_out_reg;
reg [31:0] dma_eff_count;
assign DMA_EFF_COUNT = dma_eff_count;

/*****
/* Endian Flip incoming data */
endianflip32 ed(
    .IN(DMAF_DOUT),
    .EN(1'b1),
    .OUT(data_out_wire) //(DATA_OUT)
);
*****/

/*****
initial begin
    dmaf_re <= 1'b0;
    size <= 32'h00000000;
    size_original <= 32'h00000000;
    addr <= 32'h00000000;
    state <= 3'b000;
    dma_op <= 1'b0;
    done <= 1'b1;
    icap_cen <= 1'b1;
    icap_wen <= 1'b1;

    //intelligent ICAPctrl
    size_decode <= 16'h0000;
    data_out_reg <= 32'h00000000;
    dma_eff_count <= 32'h00000000;
end
*****/

always @(posedge CLK) begin
    case (state)
        3'b000: begin                                //latch dma info, wait for start

            dma_op <= 1'b0;
            icap_cen <= 1'b1;
            icap_wen <= 1'b1;
            dmaf_re <= 1'b0;
            if(START) begin
                dma_eff_count <= 0;
                done <= 1'b0;
                size <= SIZE;
                size_original <= SIZE;
                addr <= ADDR_IN;
                state <= 3'b001;

                addr_debug <= ADDR_IN; //debug
            end
        end
    end
end

```



```

end
3'b001: begin                                //start dma operation
    dma_eff_count <= dma_eff_count + 1;
    icap_cen <= 1'b1;
    icap_wen <= 1'b1;
    dma_op <= 1'b1;    //start
    if(~DMAF_EMPTY) begin                    //read from fifo
        dmaf_re <= 1'b1;
        state <= 3'b010;
    end
end
3'b010: begin                                //wait for fifo to become ready
    dma_eff_count <= dma_eff_count + 1;
    state <= 3'b011;
end
3'b011: begin
    if(size == 0) begin                    //done
        state <= 3'b000;
        dma_op <= 1'b0;
        done <= 1'b1;
        icap_cen <= 1'b1;
        icap_wen <= 1'b1;
        dmaf_re <= 1'b0;
    end
    else if(data_out_wire[31:16] == 16'hecde) begin        //special instruction caught

        size_decode <= data_out_wire[15:0];
        dmaf_re <= 1'b0;
        icap_cen <= 1'b1;
        icap_wen <= 1'b1;

        if(DMAF_EMPTY) begin
            size <= size - 1;
            state <= 3'b101;
        end
        else begin
            dma_op <= 1'b0;
            size <= size - 2;
            state <= 3'b100;
        end
    end
end
else begin
    dma_eff_count <= dma_eff_count + 1;
    icap_cen <= 1'b0;
    icap_wen <= 1'b0;
    data_out_reg <= data_out_wire;
    size <= size - 1;
    addr_debug <= addr_debug + 4;                //debug

    if(DMAF_EMPTY) begin
        dmaf_re <= 1'b0;
        if(size > 1) begin                    //need to resync
            state <= 3'b001;
        end
    end
end
end
3'b100: begin                                //start intelligent ICAPctrl
    dmaf_re <= 1'b0;
    if(size_decode == 0) begin
        icap_cen <= 1'b1;

```

```

        icap_wen <= 1'b1;

        if(DMAF_DATA_COUNT >= size) dma_op <= 1'b0;
        else dma_op <= 1'b1;

        if((~DMAF_EMPTY) || (size==0)) begin
            dmaf_re <= 1'b1;
            state <= 3'b010;
        end
    end
else begin
    icap_cen <= 1'b0;
    icap_wen <= 1'b0;
    data_out_reg <= data_out_wire;
    size_decode <= size_decode - 1;
end
end
3'b101: begin
    dma_eff_count <= dma_eff_count + 1;
    icap_cen <= 1'b1;
    icap_wen <= 1'b1;
    dma_op <= 1'b1;
    if(~DMAF_EMPTY) begin
        dmaf_re <= 1'b1;
        state <= 3'b110;
    end
end
3'b110: begin
    dma_eff_count <= dma_eff_count + 1;
    dmaf_re <= 1'b0;
    dma_op <= 1'b0;
    size <= size - 1;
    state <= 3'b100;
end
end
endcase
end
endmodule

```

APPENDIX 2: Step-by-Step Demo

Note: please email to shaoshal@uci.edu if you have questions about this process

Test 1: test fully streaming DMA engine without compression

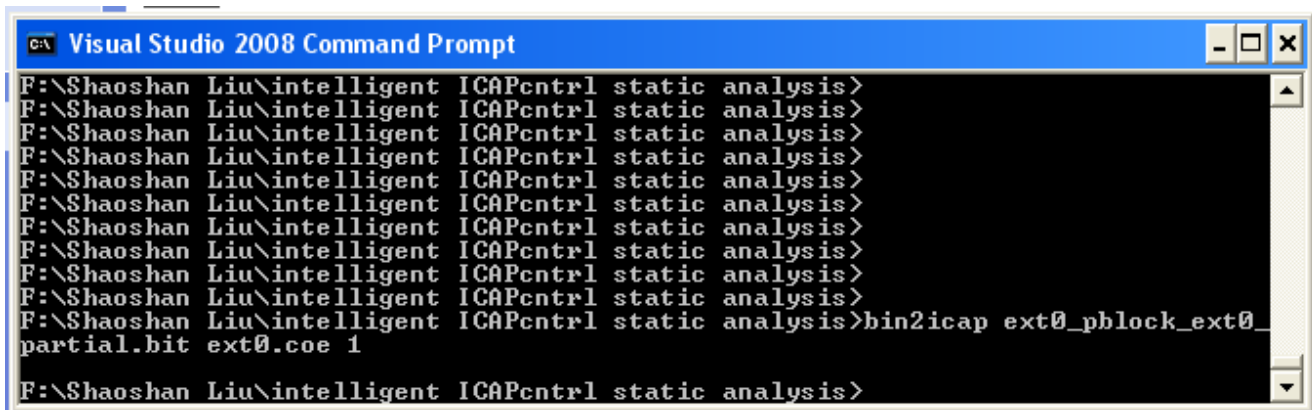
Steps:

1. Grab the bitstream files from
“\Shaoshan_Liu\emIPSV1.1_PR_ICAP_DMA_streamingFIFO_PR\project_PR\project_PR2.runs\floorplan_1\merge”

In this case, we grab the file “ext0_pblock_ext0_partial.bit” and copy it over to the folder
“\Shaoshan Liu\intelligent ICAPcntrl static analysis”

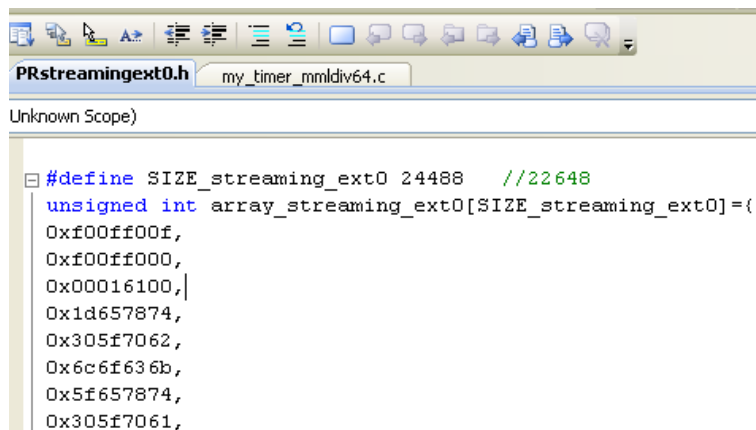
2. Generate the uncompressed bitstream file

To generate the uncompressed bitstream file in ASCII format, use the following command:
“bin2icap ext0_pblock_ext0_partial.bit ext0.coe 1”



```
C:\ Visual Studio 2008 Command Prompt
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>bin2icap ext0_pblock_ext0_
partial.bit ext0.coe 1
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
```

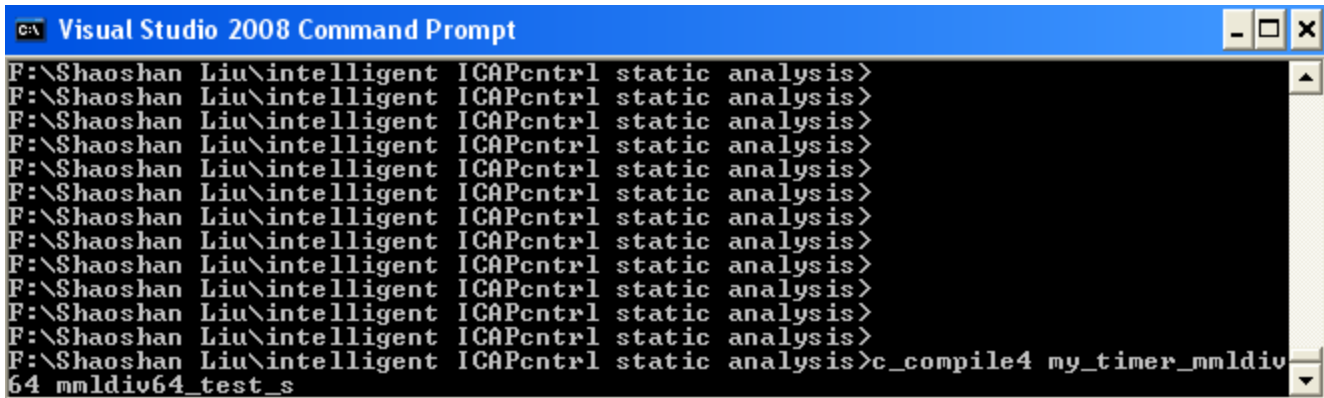
3. Store the uncompressed bitstream file into the file “\Shaoshan Liu\emIPS Tests\PRstreamingext0.h” and set the file size to 24488.



```
PRstreamingext0.h my_timer_mmldiv64.c
Unknown Scope)

#define SIZE_streaming_ext0 24488 //22648
unsigned int array_streaming_ext0[SIZE_streaming_ext0]={
0xf00ff00f,
0xf00ff000,
0x00016100,
0x1d657874,
0x305f7062,
0x6c6f636b,
0x5f657874,
0x305f7061,
```

4. Compile the file “\Shaoshan Liu\emIPS Tests\my_timer_mmldiv64.c”

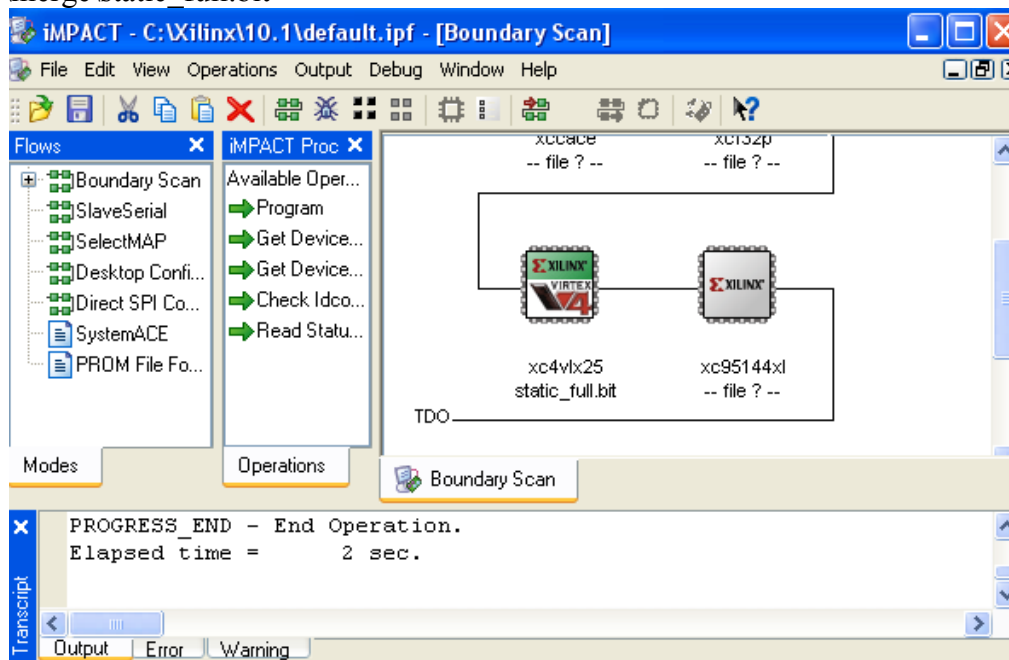


```
Visual Studio 2008 Command Prompt
F:\Shaoshan Liu\intelligent ICAPctrl static analysis>
F:\Shaoshan Liu\intelligent ICAPctrl static analysis>
F:\Shaoshan Liu\intelligent ICAPctrl static analysis>
F:\Shaoshan Liu\intelligent ICAPctrl static analysis>
F:\Shaoshan Liu\intelligent ICAPctrl static analysis>
F:\Shaoshan Liu\intelligent ICAPctrl static analysis>
F:\Shaoshan Liu\intelligent ICAPctrl static analysis>
F:\Shaoshan Liu\intelligent ICAPctrl static analysis>
F:\Shaoshan Liu\intelligent ICAPctrl static analysis>
F:\Shaoshan Liu\intelligent ICAPctrl static analysis>
F:\Shaoshan Liu\intelligent ICAPctrl static analysis>
F:\Shaoshan Liu\intelligent ICAPctrl static analysis>c_compile4 my_timer_mmldiv
64 mmldiv64_test_s
```

5. Use IMPACT to configure the chip such that only the baseline TISA is loaded

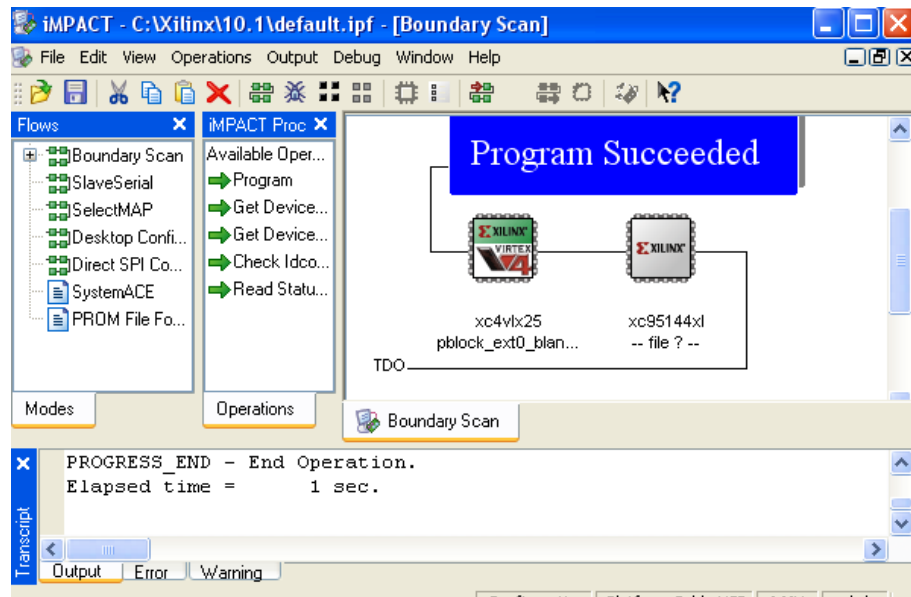
First do a full chip configuration by downloading the following file to the chip:

“\Shaoshan_Liu\emIPSV1.1_PR_ICAP_DMA_streamingFIFO_PR\project_PR\project_PR2.runs\floorplan_1\merge\static_full.bit”



Then take out the extension by downloading the following file to the chip:

“\Shaoshan_Liu\emIPSV1.1_PR_ICAP_DMA_streamingFIFO_PR\project_PR\project_PR2.runs\floorplan_1\merge\pblock_ext0_blank.bit”



By now we should have only the baseline TISA design running on the V4 chip, and in the next few steps we show how to do run-time partial reconfiguration and test it with the mmldiv64 test.

6. Download the test file to the chip by typing the following command: "download com1: my_timer_mmldiv64.bin && serplexd -n -r -s com1:"

```

C:\ Visual Studio 2008 Command Prompt
F:\Shaoshan Liu\emIPS Tests>
F:\Shaoshan Liu\emIPS Tests>
F:\Shaoshan Liu\emIPS Tests>
F:\Shaoshan Liu\emIPS Tests>
F:\Shaoshan Liu\emIPS Tests>
F:\Shaoshan Liu\emIPS Tests>
F:\Shaoshan Liu\emIPS Tests>
F:\Shaoshan Liu\emIPS Tests>
F:\Shaoshan Liu\emIPS Tests>
F:\Shaoshan Liu\emIPS Tests>
F:\Shaoshan Liu\emIPS Tests>
F:\Shaoshan Liu\emIPS Tests>
F:\Shaoshan Liu\emIPS Tests>
F:\Shaoshan Liu\emIPS Tests>download com1: my_timer_mmldiv64.bin && serplexd -n
-r -s com1:

```

7. Get results from the command prompt

The results show that the ICAP-DMA counter is 5fb9 (24505 in decimal), whereas the ICAP-DMA effective counter is 5fb7 (24503 in decimal). For these two counters, the clock runs at 100 MHz, and ICAP-DMA counter counts the number of cycles the ICAP takes to finish configuration, whereas the ICAP-DMA effective counter counts the number of cycles the fully streaming DMA engines take to complete data transfer. Since we are not using the compressed bitstream in this case, these two numbers are just off by 2 cycles. If we do the calculation, the file size is 24488 words (each word is 4 bytes) and it takes 24505 cycles to complete configuration, thus the ICAP throughput is 399.7 Mbytes/s.

After configuration, the rest of the program performs mmldiv64 operations as well as normal integer operations. The results also show the time taken for these operations. Note that different from the previous counter numbers, these performance numbers are generated using the on-chip timer, which runs at 10 MHz instead of 100 MHz.

```

.....Download complete, 114236 bytes s
ent
Will NOT attempt to use the NIC
Will exchange un-encoded data ['raw' model].
Will talk to a BigEndian client (mips,ppc,...)
Console Thread ...

MMLDIU64 test (with extension) starts

    End Configuration

    ICAP-DMA Counter (cycles): 00005fb9
    ICAP-DMA effective Counter (cycles): 00005fb7    mmldiv64 TIME =
0005f1a0

normal integer test (no extension) starts

    int TIME =        03183d36

```

8. Double check functionality with the standalone mmldiv64 test by typing the following command: "download com1: mmldiv64_test2.bin && serplexd -n -r -s com1: "

```

    int TIME =        37c435cc
    int TIME =        3add62a0
    int TIME =        3df68f82

    int TIME =        3df77f89
    mmldiv64 TIME =        00000000

    total TIME =        3df77f89 11
F:\Shaoshan Liu\eMIPS Tests>download com1: mmldiv64_test2.bin && serplexd -n -r
-s com1:
.....

```

9. Results of double-check

```

    BASE FINISH =        00000000.0a648faa
    BASE TIME =        0551e6ec

EXTENSION TEST RESULTS
    EXT START =        00000000.0a673a56
    EXT FINISH =        00000000.0cef0031
    EXT TIME =        0287c5db

PERFORMANCE over base (base/ext) =        00000002
PERFORMANCE over original (noext/ext) =        00000001
OVERHEAD over original (base/noext) =        00000001
TEST PASSED SUCCESSFULLY 1_

```

Test 2: test fully streaming DMA engine with compression (intelligent ICAP controller)

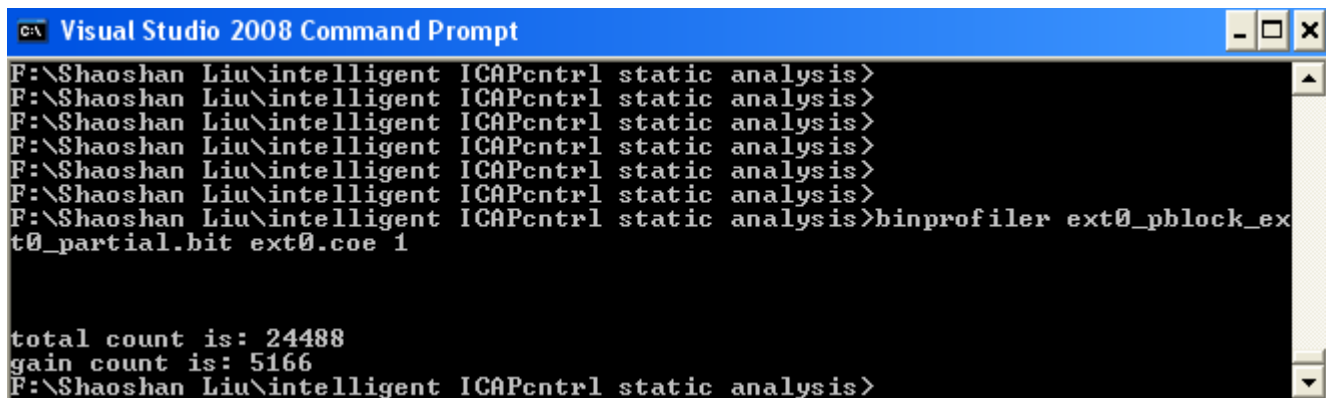
Steps:

1. Grab the bitstream files from
“\Shaoshan_Liu\emIPSV1.1_PR_ICAP_DMA_streamingFIFO_PR\project_PR\project_PR2.runs\floorplan_1\merge”

In this case, we grab the file “ext0_pblock_ext0_partial.bit” and copy it over to the folder
“\Shaoshan Liu\intelligent ICAPcntrl static analysis”

2. Generate the uncompressed bitstream file

To generate the compressed bitstream file in ASCII format, use the following command:
“binprofiler ext0_pblock_ext0_partial.bit ext0.coe 1”



```
C:\ Visual Studio 2008 Command Prompt
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>binprofiler ext0_pblock_ext0_partial.bit ext0.coe 1

total count is: 24488
gain count is: 5166
F:\Shaoshan Liu\intelligent ICAPcntrl static analysis>
```

3. Store the compressed bitstream file into the file “\Shaoshan Liu\emIPS Tests\PRstreamingext0.h” and set the file size to 22648, note that the original size is 24488, thus the compression ratio is 1.08.

```
my_timer_ICAPtest.c PRstreamingext0.h my_timer_mmldiv64.c
(Unknown Scope)

#define SIZE_streaming_ext0 22648 //24488
unsigned int array_streaming_ext0[SIZE_streaming_ext0]={
0x00000000,
0xf00ff00f,
0xf00ff000,
0x00016100,
0x1d657874,
0x305f7062,
0x6c6f636b,
0x5f657874,
0x305f7061,
0x72746961,
0x6c2e6e63,
0x64006200,
0x0c34766c,
```

4. Same as that in the previous section
5. Same as that in the previous section
6. Same as that in the previous section
7. Get results from the command prompt

The results show that the ICAP-DMA counter is 6281 (25217 in decimal), whereas the ICAP-DMA effective counter is 59a8 (22952 in decimal). Thus, the effective transfer throughput is 426.8 Mbytes/s, whereas the ICAP throughput is 388.4 Mbytes/s.

```
.....Download complete, 106876 bytes sent
Will NOT attempt to use the NIC
Will exchange un-encoded data ['raw' model].
Will talk to a BigEndian client (mips,ppc,...)
Console Thread ...

MMLDIU64 test <with extension> starts

End Configuration

ICAP-DMA Counter <cycles>: 00006281
ICAP-DMA effective Counter <cycles>: 000059a8
mmldiv64 TIME = 0005f1a0

normal integer test <no extension> starts

int TIME = 03183d36
```

8. Same as that in the previous section

9. Same as that in the previous section