ACS: an Addressless Configuration Support for Efficient Partial Reconfigurations

Jenny Yi-Chun Kuo Anderson Kuei-An Ku Jingling Xue Oliver Diessel Usama Malik*
School of Computer Science and Engineering, University of New South Wales, NSW 2052, Australia
{*kuoy, kua, jingling, odiessel*}*@cse.unsw.edu.au*
* Endace Technology Limited, Hamilton 3204, New Zealand
usama.malik@endace.com

Abstract

This paper presents a complete design of a reconfigurable architecture support system, called ACS (an Addressless Configuration Support), which provides efficient access to non-contiguous reconfigurable locations in reconfigurable systems. ACS reduces the amount of partial reconfiguration information required by removing a large amount of addressing information and padding as found in Virtex-4 bitstreams.

ACS improves significantly on the distTree architecture previously proposed by us. ACS introduces the selector block which connects the leaf nodes to a consecutive block of reconfiguration locations called a frame set. The system allows any number of leaf nodes customised to the size of the device, thereby providing much more flexibility. The hardware costs have also been reduced significantly over the distTree design. Together with the new marker loading mechanism, ACS is readily applicable to SRAM-based FPGAs. This new ACS system is benchmarked using eight real-world applications against a Virtex-4 device and the results show 6.83%-15.07% speedups when the reconfiguration granularity is set to a Virtex-4 frame.

1. Introduction

Reconfigurable systems are systems which consist of arrays of reconfigurable hardware such as Field Programmable Gate Arrays (FPGAs). The two main components of an FPGA are the logic blocks and the switch boxes that provide routing between these logic blocks. These switch boxes as well as the contents of the logic blocks are reprogrammable after fabrication to perform different tasks at different times.

Applications nowadays often have more intricate functionalities and require more hardware resources than are available. As a result, designs often cannot fit onto a single FPGA device [8]. The solutions are often to use more logic by either networking multiple FPGAs or performing some form of run-time reconfiguration [7]. He *et al.* [9] proposed a solution by finding optimal strategies in networking multiple FPGA devices using crossbars. However, having multiple FPGAs not only increases the costs but also power consumption. The on-chip/off-chip latencies and synchronization between the devices are also of concern. In this paper, we take the second approach and present a new hardware support system, called *ACS* (an Addressless Configuration Support), which aims to facilitate fast run-time partial reconfigurations with little extra hardware costs.

1.1. Background

A run-time reconfiguration can either be a full or partial reconfiguration. Each configuration is referred to as a context and the context is stored in the form of a bitstream. A full reconfiguration requires a complete configuration of the whole system while a partial reconfiguration requires only the differences between the old and the new configurations, resulting in a smaller bitstream length and a smaller reconfiguration latency. The bitstreams are swapped in and out of an FPGA whenever new functionalities are required. Although run-time reconfigurable systems offer great flexibilities and the ability to accommodate larger designs than the physical capacity of the device, the time it takes to perform a reconfiguration can take up a large proportion of the total execution time resulting in inefficient throughput. This delay is called the reconfiguration latency and is proportional to the length of the bitstream, which depends on the area to be reconfigured and the sparsity of the reconfiguring blocks. This leads to one of the major shortcomings in current FPGA technologies [13] [14] [15].

1.2. Motivation

In reconfigurable systems, an unused reconfiguration context is usually stored in a memory external to the working hardware and is only swapped in when required. As a

result, considerable amount of information may need to be transferred between the internal and external memories and thus incur a significant run-time reconfiguration latency. Also, the current reconfigurable architectures impose considerable overheads by loading more reconfiguration bits than required. For example, a frame is the smallest reconfigurable unit in a Virtex device [2] and a bitstream of the size of a complete frame must be loaded onto the device even when only a small part of a frame is changing. Malik and Diessel [12] experimented with smaller reconfiguration units with vector addressing and found that the smaller the granularities, the shorter the bitstream despite of the increased addressing overhead.

Lange and Middendorf [11] proposed a two-level reconfigurable architecture that aims to reduce the addressing overhead incurred in systems with finer granularities by adding extra hardware. The architecture links the locations that are required to be reconfigured by short-circuiting the non-reconfiguring locations. However, this architecture may be difficult to implement in practice for two reasons. First, because the reconfiguring locations may be far apart, signals have to be strong enough to maintain their integrity. The frequency of the system is also limited by the longest distance between two reconfiguring locations. Second, there is little parallelism in the architecture and the efficiency depends heavily on the locality of the reconfiguring locations. As a result, the reconfiguration latency may still be large in extreme cases.

In the previous paper [10], we proposed a preliminary idea of using a hardware architecture support called *distTree* for partial reconfigurations. While allowing reconfiguration of non-contiguous frames in an FPGA with minimum addressing information from outside, *distTree* is too costly to be practical. First, *distTree* has to be a perfectly balanced binary tree. If there are $2^m + 1$ reconfigurable locations, the tree will have 2^{m+1} leaves while a large amount of hardware is not used. Second, the granularity of tree leaf nodes is assumed to be one bit with no solutions to coarser granularities being explored. At this finest granularity, *distTree* can be costly. For example, the smallest Virtex-4 device consists of 4.7Mb of configuration bits, which would result in a *distTree* with 4.7 million leaf nodes. Third, *distTree* does not address how the markers, which is used to indicate the reconfiguring sites, should be loaded into the system and has always assumed their availability when required. Addressing these limitations may impose extra hardware and timing overheads that were not accounted in the *distTree* design. Finally, *distTree* was evaluated using hypothetical bitstreams with random reconfiguration sites which may not have reflected real-world scenarios.

In this paper, we present an improved system called *ACS*, which addresses all the limitations above by including a marker loading mechanism, a selector block and a refined *distTree* now called the *BinTree*. *ACS* is not only efficient but also practical as demonstrated using eight real-world benchmarks from different application domains. The results are compared with a Virtex-4 device.

In summary, this paper extends on the previous *distTree* and makes the following contributions:

- **The Selector Block:** The selectors are introduced to reduce the size of the *BinTree* in the *ACS* system and offer more flexibility. A significantly smaller *BinTree* can now be used instead of a tree with the same number of leaf nodes as the number of reconfiguration locations in a device. The granularity is now set to one Virtex-4 frame instead of one bit in the original *distTree* design.
- **The Marker Loading Mechanism:** The *ACS* design includes a loading mechanism for the markers. An array of shift registers used as marker buffers is implemented to accept the input from outside. These extra hardware costs and the communication latency imposed by the marker buffers are adequately modelled.
- **The *BinTree*:** The design of the *BinTree* is optimized to save the hardware costs so the leaf nodes can now be of any size for flexible customisation. If there are $2^m + 1$ reconfigurable locations, the *BinTree* will only need to be of size $2^m + 1$ instead of 2^{m+1}.
- **Validation against Virtex-4:** Eight real-world benchmarks from different application domains are used to evaluate the effectiveness of the *ACS* system. The bitstreams used for the system include the reconfiguration data as well as the markers to make the timing simulation realistic. The *ACS* design is evaluated by comparing with a Virtex-4 device using bitstreams generated by Xilinx ISE. The results show that *ACS* has an average of 6.83%-15.07% speedup over a Virtex-4 device.

1.3. Paper Organization

Section 2 describes the hardware components of the *ACS* system including selectors, I/O connections and memory requirements. It also describes the system work flow and the design of each component. Section 3 introduces the methodologies used in designing the system and the experimental setup. Section 4 evaluates and analyses its effectiveness. Section 5 concludes the paper.

2. The *ACS* Architecture

The *ACS* system, as shown in Figure 1, is defined in terms of four key architectural parameters f, k, s, and N. f denotes the number of reconfigurable units in a device, for example, a frame in Virtex-4 devices. k denotes the reconfiguration port size and the width of the link between marker buffers and marker memory. s represents the width of the selector block output and N denotes the number of leaf nodes in the *BinTree* as well as the size of marker buffers. In this paper, s and N are the two variables in

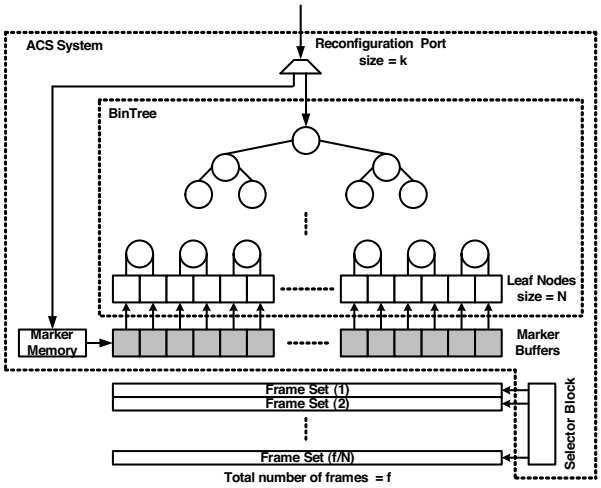

Figure 1. Basic diagram of an ACS system

designing a suitable $\boldsymbol{ACS}$ system while the other two parameters are set to constants for fair comparisons with the target Virtex-4 device (i.e. $f = 3600$ and $k = 8$).

The $\boldsymbol{ACS}$ architecture comprises of three main components: the $\boldsymbol{BinTree}$, which is in the form of a balanced binary tree, an array of shift registers used as marker buffers and a selector block. The leaf nodes of the $\boldsymbol{BinTree}$ are represented with white square boxes and the root and internal nodes are marked with circles. The leaf nodes serve as the dedicated memory for the $\boldsymbol{ACS}$ system and I/O's to the reconfigurable units. Each leaf node connects to f/N reconfigurable units where each of these units is called a frame set. The marker buffers are made up of shift registers of size k. They are marked with grey boxes in Figure 1, and are used to receive reconfiguration information from the marker memory and transfer this information to the leaf nodes during partial reconfigurations. The selector block is used to activate the current reconfiguring frame set when N is designed to be smaller than f. The width of the selector block output is represented with s where $s = f/N$.

In this paper, we compare $\boldsymbol{ACS}$ with a Virtex-4 device. Frames in Virtex-4 are arranged in a 2D matrix across the device. A frame has 41 32-bit words and the height of a frame spans over the height of 16 CLBs, 4 DSPs or 4 BRAMs [6].

2.1. The *BinTree*

The $\boldsymbol{BinTree}$ is constructed in a way that it can take any number of leaf nodes but always satisfies the conditions of a balanced binary tree. Examples of non-perfectly balanced $\boldsymbol{BinTree}$'s are shown in Figure 2. There are three different node design modules for three different kinds of tree nodes, the root node, the internal nodes, and the leaf nodes.

The root node acts as the control center for the $\boldsymbol{ACS}$ system and the $\boldsymbol{BinTree}$. It contains a finite state machine

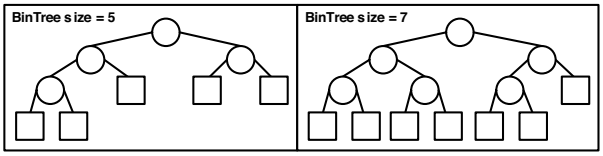

Figure 2. Examples of $BinTree$ with 5 and 7 leaf nodes in ACS systems

(FSM) and two counter registers. The two counter registers at the root, the *leftCounter* and *rightCounter*, are used to store the *flows* of its left and right child, respectively. The *flow* refers to the total number of data that would go through a node. These two counters each stores values up to half of the number of leaf nodes, i.e., $Root.RegisterSize = lg(N/2)$. Because each leaf node in the $\boldsymbol{ACS}$ system is now connected to a frame set, the counter registers should only decrement after each frame set worth of data is sent. A *counter_decrement* signal is sent with the last data for this purpose.

Each internal node consists of an FSM, a counter register, which is used to store the *flow* F and the *threshold* value T, which is the total number of data that should be delivered to the left child of a node, and an adder. Each internal node needs to sum up the F value from both of its children and to update the threshold value T using its left child's flow F value. It also needs to forward data from the parent to the correct child according to the T value. If T is larger than 0, the data should be sent to the left child, otherwise to the right child. The threshold counter T decrements only when $T > 0$ and *counter_decrement* signal is received. The size of the adder depends on the level of the node, i.e., $AdderInput = level$ and $AdderOutput = level + 1$.

A leaf node in $\boldsymbol{BinTree}$ contains an FSM and a 1-bit register to store the marker M. In our experiments, a leaf node is connected to a reconfigurable unit equivalent to a frame in a Virtex-4 device. The leaf nodes interface with their parents as well as the marker buffers. A new set of markers is required when all the leaf nodes either have their current M set to 0 to start with or have received *counter_decrement* signals from their parents. When a new set of markers is required, a *get_next_marker* signal is issued and a new set of markers is shifted into the marker buffers in blocks of k bits from the marker memory as shown in Figure 1.

2.2. The Marker Loading Mechanism

The markers consist of a series of 0's and 1's. They are used to identify the leaf nodes which should be reconfigured and will be discussed in Section 2.4.1. The length of a set of markers equals the total number of leaf nodes that an $\boldsymbol{ACS}$ system has. The markers are stored in a small dedicated on-chip memory and are always loaded into the marker buffers first before moving into the leaf nodes. The marker buffers

communicate with the leaf nodes by the *get_next_marker* signal. When this signal is received by the buffer, a new set of markers is requested and the current content in the marker buffers will be transferred into the leaf nodes.

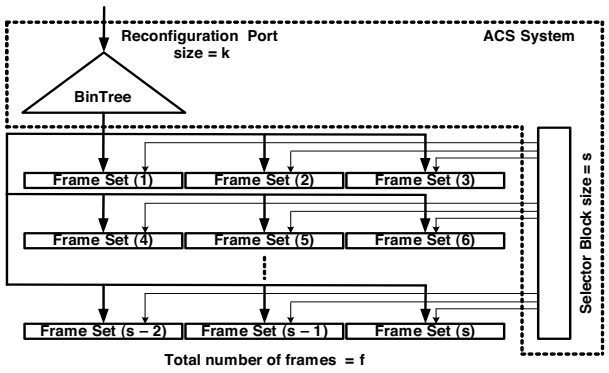

Figure 3. 2-D frame set structure using a small $BinTree$ combined with a selector block

2.3. The Selector Block

In case when N, the number of leaf nodes in ACS, is smaller than the total number of frames available in a device, a selector block is employed to activate the current reconfiguring frame set. It is similar to the traditional write enable signal of a memory. The selector block connects its input to the *get_next_marker* signal and outputs to a $s = \lceil f/N \rceil$-bit-wide output where each bit connects to a set of reconfigurable frames. The selector block is designed with a simple shifter. As soon as the *get_next_marker* signal is 1, the markers for the next frame set overwrite the contents in the marker buffers and the selector block enables the next frame set. Figure 3 shows how the frame sets are arranged when incorporating the whole ACS system into an existing 2-D SRAM-based FPGA.

2.4. System Operation

The ACS system improves the performance of partial reconfigurations by parallelizing the counter setup, data delivery and marker loading stages. The bitstreams for the system consist of reconfiguration data and markers. The reconfiguration data enters the system through the reconfiguration port during reconfigurations while the markers are pre-loaded into a small dedicated on-chip marker memory as shown in Figure 1.

2.4.1. Initialization

To intialize the system, the markers are loaded into the marker memory in k-bit blocks via the reconfiguration port first. The marker buffers then accept markers for the first two frame sets from the marker memory and put the first set into the leaf nodes as their counters while keeping the second set in the marker buffers. All the leaf nodes then issue a *flow_ready* signal to their parents to signify that the markers are ready. The markers mark the locations where reconfiguration data should be placed and are treated not only as the flows F but also the thresholds T of the leaf nodes. If the marker of a leaf node is 1, then the frame connected to that particular leaf node should expect new reconfiguration data, otherwise the frame remains unchanged. After initialization, counter setup, data delivery and any further marker loading stages will run in parallel (Figure 4). The reconfiguration port is now solely dedicated for reconfiguration data loading.

2.4.2. Counter Setup

The aim is to set up all the counters in the $BinTree$ so that incoming data can be directed to the correct reconfiguration locations. The process starts from the internal nodes at the lowest level and proceeds up the $BinTree$ one level at a time until the root is reached. Upon receiving *flow_ready* signals from both children, an internal node calculates its flow by adding up the flows from both children and sends a *flow_ready* signal to its parent. Therefore, $F(Node) = F(LeftChild) + F(RightChild)$. When the root node receives *flow_ready* from both children, it takes the flow from its left child to be the *leftCounter* and the flow from its right child to be the *rightCounter*. The root node is now ready to accept data via the reconfiguration port. At the same time, the root node sends a *get_thresh* signal which is propagated down the $BinTree$ to all internal nodes one level at a time until the lowest level internal nodes are reached. Every internal node, upon receiving the *get_thresh* signal, sets its threshold value to the flow value of its left child, i.e. $T(Node) = F(LeftChild)$. The internal node then forwards the *get_thresh* signal to its children. Note that data can begin travelling down the system at the same time as the *get_thresh* signal so that the first data arrives at its destination leaf node exactly one cycle after the completion of the counter setup.

2.4.3. Data Delivery

The data delivery stages are run in parallel with the counter setup and marker loading stages (Figure 4). In this stage, the root node accepts data from the reconfiguration port and forwards the data down the $BinTree$ as described in Section 2.1. The *leftCounter* and *rightCounter* decrement when the last data word of a frame is received. The data that leaves the root node is then redirected down the $BinTree$ by the internal nodes. The data delivery stage finishes when both the *leftCounter* and the *rightCounter* at the root node become 0, which means that no more data is required for the current configuration.

2.5. Marker Loading and Selector Block

After the markers for the first frame set are used by the internal nodes, the content in the marker buffers (i.e., markers for the second frame set) is shifted into the leaf nodes. Each leaf node then makes a request for a new marker value and update its counter either when its current counter value is 0, or when it receives a valid data from the parent. A leaf node updates its counter value by replacing the current counter value by the marker stored in the marker buffers. All the requests for a new set of markers are collected with an AND gate and the output of the AND gate is an 1-bit wire used as the *get_next_marker* signal to load the markers for the next frame set. The *get_next_marker* signal is also connected as the input to the selector block to activate the next frame set as the recipient of any incoming data.

2.6. The Parallelism Overview

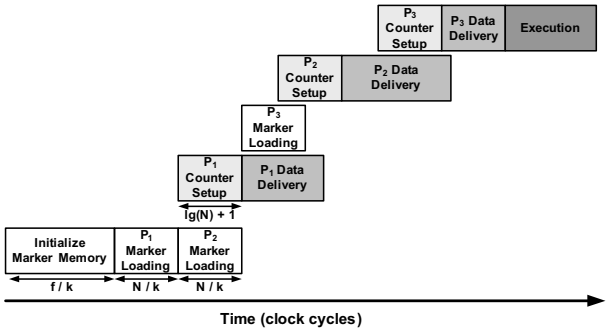

Figure 4. Parallelism of the ACS system

The parallelism of the three stages discussed in Section 2.4 is shown in Figure 4. The latency at the start-up of the system is called the initialization latency and includes the marker loading from external source to the marker memory and a counter setup. This latency is inevitable in the $\boldsymbol{ACS}$ system and is exactly $f/k + N/k + lg(N) + 1$ cycles, where f is the number frames and k is the size of the reconfiguration port and the size of the link between marker buffers and marker memory. Every subsequent marker loading stage takes N/k cycles and every subsequent counter setup stage takes $lg(N) + 1$ cycles to complete. For a complete partial reconfiguration, the $\boldsymbol{ACS}$ system needs to go through the iteration of marker loading $\rightarrow$ data delivery $\rightarrow$ counter setup f/N times. The example in Figure 4 shows a complete partial reconfiguration that requires only three iterations, where P_j represents the j^{th} iteration and $j = 1, 2, or\ 3$. Data delivery stages shown in the figure commence from when the first data word enters the root node, and complete when the last data word in the bitstream reaches its destined leaf node.

3. Methodology

In our experiments, we have chosen eight benchmarks from OPENCORES.ORG [1] representing a variety of application domains. $\boldsymbol{ACS}$ is compared with the Virtex-4 XC4VLX15 device using partial bitstreams generated from the Xilinx ISE and the BitGen tool. The hardware costs, timing requirements and power consumptions of the $\boldsymbol{ACS}$ system are simulated using the Synopsis® Design Compiler® .

3.1. Virtex-4 Partial Reconfigurations

Virtex-4 performs partial reconfigurations in three main stages [4]: setup, bitstream loading and startup sequence. In this paper, we focus on the configuration data loading phase of the bitstream loading stage. The Virtex-4 reconfiguration memories are tiled around the device in units of a frame which is the smallest unit that must be reconfigured. Partial reconfigurations can be loosely referred to as the processes of loading the bitstream into the frames in a device. The Virtex-4 bitstream consists of a header, reconfiguration commands, reconfiguration data, pad zeros and a tail. The reconfiguration commands and reconfiguration data are the parts of a bitstream that are of interest here.

BitGen is a tool provided by Xilinx as part of the ISE design suite that can be used to manipulate the bitstream files [3]. The BitGen command: *BitGen [target_conf.ncd] -r [initial_conf.bit]* generates a file called *target_conf.bit* which is the partial reconfiguration bitstream that changes the configuration from *initial_conf* to *target_conf*. For Virtex-4 devices, the partial reconfiguration bitstreams consist of a number of Type 1 and/or Type 2 packets [5]. These packets contain a header with information on the starting address of configuration frame and the number of words to be loaded from the starting address as in a DMA burst model. The number of frames to be reconfigured can be calculated by finding all Type 1 and Type 2 packet headers in the *.bit* files. This is checked against the bitstreams generated for $\boldsymbol{ACS}$ as described in Section 3.2.

3.2. ACS Partial Reconfigurations

$\boldsymbol{ACS}$ also does partial reconfigurations by loading bitstreams into the frames connected to the leaf nodes. The information of the Virtex-4 bitstream composition was obtained from [5]. The following steps are taken to generate the $\boldsymbol{ACS}$ bitstreams for partial reconfigurations:

1. Synthesize the configurations written in any hardware descriptive languages in Xilinx ISE for the target device. In our experiments, we have used XC4VLX15 as our target device. This device is chosen as it is the smallest device that fits all benchmarks. We have kept all the default settings in the ISE unless otherwise

specified. We keep the resulting *.ncd* and *.bit* files from all configurations.

2. Take the resulting *.bit* files from Step 1 and set BitGen options to -g Binary:Yes to generate *.bin* files. Look for the Type 1 packet header 0x30004000. This denotes the start of the configuration data. The next word that follows contains the number of 32-bit words in this packet. Since these *.bin* files are the initial configuration bitstream files and the XC4VLX15 device has 3600 configuration frames where each frame is 41 32-bit words long, the number of words in the packet is always 0x90240.

3. Use a Java application developed for ACS, taking the two *.bin* files as inputs and generate the differences between two configurations into a *.mkr* and a *.dta* files. The *.mkr* is the marker file that is used to mark the frames to be reconfigured. The length of the *.mkr* file is exactly 3600 bits long for our chosen device. The *.dta* file contains only the reconfiguration data to be loaded into the device. The *.mkr* and *.dta* files are the input files to the ACS system.

3.2.1. Virtex-4 Reconfiguration Architecture

Virtex-4 devices accept the bitstreams through either an 8-bit or a 32-bit SelectMap [5] bidirectional data bus clocked at 100MHz. In our experiments, we choose to compare with the SelectMap 8-bit reconfiguration method. The speedup of ACS over Virtex-4 will remain the same with larger buses as the reconfiguration latency is inversely proportional to the width of the reconfigurable port. The Virtex-4 bitstreams contain commands, addresses and reconfiguration data. The commands are loaded first to set up the device before accepting addresses or data. Then, addresses which include the starting frame address and the number of consecutive frames to be reconfigured are sent to a Frame Address Register (FAR) and the address decoder. Reconfiguration data is then shifted into a Frame Data Register (FDR) and then into the destination frame(s) depending on the addresses given [4]. Each reconfiguring frame either takes data straight from the FDR if it is the first frame in a packet, or otherwise from its preceding frame.

The ACS system replaces the FAR, FDR, address decoders and any wires that are used for these components.

3.3. Experimental Setup

The proposed ACS system was written in VHDL, synthesized using Synopsis® Design Compiler® with Tower $0.18\mu m$ Standard Cell library (tsl18fs120) and simulated using ModelSim™ SE 5.7e by Mentor Graphics. The tsl18fs120 library is optimized for the TOWER semiconductor which uses the 0.18 micron process. The results from the Synopsis Design Compiler are presented and analyzed in Section 4.

Benchmarks Name	Application Area	Slice	Slice (%)	IOB	IOB (%)
cf_fir	Signal Processing	272	4.43	178	74.17
colorconv	Image Processing	670	10.90	153	63.75
bluetooth	Communication	728	11.85	85	35.42
fir	Signal Processing	1148	18.68	103	42.92
fm	Communication	422	6.87	22	9.17
huffman	Compression	755	12.29	23	9.58
basicrsa	Encryption	527	8.58	132	55.00
t65	Processor Core	510	8.30	57	23.75

Table 1. Benchmarks

All the benchmarks used in the experiments are downloaded from OPENCORES.ORG [1] and their basic information is shown in Table 1. All benchmarks utilize 5% to 20% of the resources available which mimics the scenario where small functional units are swapped in and out of the reconfigurable area during run-time partial reconfigurations. The number of slices used, slice%, number of IOB used and IOB% are obtained from the Xilinx ISE with the target device set to XC4VLX15, which has 6144 slices and 240 IOB's in total.

4. Experimental Results

N	Sel Size	*BinTree* Area	Marker Buffer	Sel Area	Total Area	Sel %	Total Power	Max Clock
ACS								
8	450	6154	58	3230	9442	34.2	11.3	1.88
12	300	8342	117	2149	10608	20.25	11.86	1.69
16	225	10337	117	1608	12062	13.33	13.07	1.69
18	200	11519	172	1428	13119	10.88	13.81	1.67
20	180	12643	172	1284	14099	9.1	14.57	1.67
36	100	21180	294	707	22181	3.18	21.57	1.98
72	50	40343	540	347	41230	0.84	38.5	2.11
144	25	79121	1001	167	80289	0.2	73.06	2.34
distTree								
4096	0	1766756	34120	0	1805536	0	-	-

Table 2. Hardware costs, total power and maximum clock for ACS and $distTree$

4.1. Hardware Costs

Table 2 shows the hardware costs, longest path timing requirements and power consumptions for ACS of various sizes. N denotes the number of leaf nodes an ACS system has and "Sel Size" equals to the width of the selector block output so that $Sel\ Size = 3600/N$. The areas presented here include the logics and any wiring area associated with the component. They are presented in terms of number of tsl18 cells which are 2-input NAND gates. Sel % column shows the hardware costs of selector blocks with respect to the total ACS area. The selector block area can be quite significant when N is small and is inversely proportional to N. When N equals to the total number of frames in a device,

Time (ns)	cf_fir	colorconv	bluetooth	fir	fm	huffman	basicrsa	t65
cf_fir	-	2,161,830	2,628,380	2,543,340	2,841,650	2,763,780	2,200,760	1,601,300
colorconv	2,161,830	-	2,773,430	2,958,990	3,179,300	3,429,630	2,606,330	2,153,630
bluetooth	2,628,380	2,773,430	-	3,273,340	3,084,170	3,690,900	2,958,080	2,662,540
fir	2,543,340	2,958,990	3,273,340	-	3,315,850	3,183,340	3,070,240	2,728,900
fm	2,841,650	3,179,300	3,084,170	3,315,850	-	3,657,930	3,165,270	2,964,090
huffman	2,763,780	3,429,630	3,690,900	3,183,340	3,657,930	-	3,338,560	3,264,060
basicrsa	2,200,760	2,606,330	2,958,080	3,070,240	3,165,270	3,338,560	-	2,316,880
t65	1,601,300	2,153,630	2,662,540	2,728,900	2,964,090	3,264,060	2,316,880	-

Table 3. Estimated Times for Virtex-4 partial reconfigurations

Time (ns) / Speedup (%)	cf_fir	colorconv	bluetooth	fir	fm	huffman	basicrsa	t65
cf_fir	-	1,897,850 / 13.91	2,374,130 / 10.71	2,210,260 / 15.07	2,511,330 / 13.15	2,519,470 / 9.70	1,960,410 / 12.26	1,460,670 / 9.63
colorconv	1,897,850 / 13.91	-	2,548,160 / 8.84	2,636,990 / 12.21	2,889,650 / 10.02	3,129,600 / 9.59	2,377,860 / 9.61	1,963,140 / 9.70
bluetooth	2,374,130 / 10.71	2,548,160 / 8.84	-	2,995,750 / 9.27	2,812,510 / 9.66	3,404,610 / 8.41	2,740,130 / 7.95	2,483,940 / 7.19
fir	2,210,260 / 15.07	2,636,990 / 12.21	2,995,750 / 9.27	-	3,038,630 / 9.12	2,922,250 / 8.93	2,787,680 / 10.14	2,467,630 / 10.59
fm	2,511,330 / 13.15	2,889,650 / 10.02	2,812,510 / 9.66	3,038,630 / 9.12	-	3,373,330 / 8.44	2,884,770 / 9.72	2,694,620 / 10.00
huffman	2,519,470 / 9.70	3,129,600 / 9.59	3,404,610 / 8.41	2,922,250 / 8.93	3,373,330 / 8.44	-	3,067,300 / 8.84	3,055,350 / 6.83
basicrsa	1,960,410 / 12.26	2,377,860 / 9.61	2,740,130 / 7.95	2,787,680 / 10.14	2,884,770 / 9.72	3,067,300 / 8.84	-	2,133,510 / 8.59
t65	1,460,670 / 9.63	1,963,140 / 9.70	2,483,940 / 7.19	2,467,630 / 10.59	2,694,620 / 10.00	3,055,350 / 6.83	2,133,510 / 8.59	-

Table 4. Times and percentage speedups for partial reconfigurations using ACS

which is the case for the previously proposed $distTree$ proposed in [10], the selector block can be eliminated.

As mentioned in Section 3.2.1, ACS replaces the FAR, FDR, address decoders and wirings around these components in a Virtex-4 device. The actual reconfiguration architecture for Virtex-4 is unknown and a lot of optimizations have gone into their design. A rough estimation using Synopsys Design Compiler with the same library shows that the area of these components occupies about 18931 cells which is larger than any ACS systems with sizes < 20 and is twice as large as an ACS system of size 8. The "Total Power" column shows the rough estimation of the total dynamic power required for the whole ACS system in mW. The last column from Table 2 shows the longest path of ACS systems in ns when ACS is running on its own. The ACS system is not only able to meet the timing requirement, which is set to 10ns clock cycles as for Virtex-4 devices, but also gives at least 50% of slack.

The hardware cost of the previously proposed $distTree$ structure is shown in the last row of Table 2. For $distTree$ to reconfigure 3600 frames, it is required to have 4096 leaf nodes, which makes the total area almost 200 times more than an ACS system of size 8. The power consumption and the longest path for $distTree$ are unknown in this case as the Synopsys Design Compiler available at this time cannot simulate designs that require more than 4GB of memory.

4.2. Comparison with Virtex-4

Let us now examine the performance of ACS compared with that of the XC4VLX15 device. The reconfiguration time for ACS is taken from the result of ModelSim simulation using methodologies described in Section 3.2.

Table 3 shows the reconfiguration time that XC4VLX15 would take when running the same partial reconfigurations. The time is calculated by generating the partial bitstream using the method of Section 3.2 and converting the partial *.bit* files into partial *.bin* files, which takes away any header information that is not loaded during partial reconfigurations. By dividing the size of the *.bin* files by the reconfiguration port size, which is 8-bit in this case, we obtain the partial reconfiguration time required for XC4VLX15.

Table 4 shows the total reconfiguration times for an ACS of size 8 to reconfigure a XC4VLX15 device and the speedups in percentages. The reconfiguration time taken from configuration A to configuration B is the same as from configuration B to configuration A. Each leaf node in the ACS system connects to 450 41 x 32-bit frames. The size of ACS is set to 8 because we found that the reconfiguration latency intrinsic to the ACS architecture is less than 0.04% of the total reconfiguration time, so a small ACS system is sufficient to demonstrate the efficiency while introducing as little extra hardware as possible. Table 4 shows that ACS has achieved speedups ranging from 6.83% to 15.07% over a XC4VLX15 device.

# Frames Per Packet	cf_fir	colorconv	bluetooth	fir	fm	huffman	basicrsa	t65
cf_fir	-	22.52	18.35	18.70	23.56	16.95	18.76	21.80
colorconv	22.52	-	13.78	17.86	17.81	13.70	12.60	17.74
bluetooth	18.35	13.78	-	14.18	14.92	11.41	10.09	14.35
fir	18.70	17.86	14.18	-	15.31	12.69	13.09	13.99
fm	23.56	17.81	14.92	15.31	-	16.64	16.38	19.10
huffman	16.95	13.70	11.41	12.69	16.64	-	16.88	14.82
basicrsa	18.76	12.60	10.09	13.09	16.38	16.88	-	13.75
t65	21.80	17.74	14.35	13.99	19.10	14.82	13.75	-

Table 5. Number of frames per packet for different partial reconfiguration benchmarks

To understand where the speedups of ACS come from, we have written a Java application that parses the partial bitstreams produced by BitGen and calculates the number of frames per packet in the bitstream (shown in Table 5).

These packets can be either Type 1 or Type 2 and each packet contains frame data that belongs to a consecutive block of frames. The more packets there are in a partial bitstream, the more sparse the reconfiguration is and the better ACS is expected to perform. This is because ACS essentially removes the 23-byte header and tail and at least a frame worth of padded zeroes associated with each packet. For Virtex-4, this overhead can occupy up to 50% of the bitstream. The result shows that there are 75-182 packets for all the partial bitstreams generated. Take an example of the partial reconfiguration from "cf_fir" to "colorconv" There are 144 packets in the partial bitstream which means that there are $144 * 23 = 3312$ bytes of packet header and tail and at least $144 * 41 * 4 = 23616$ bytes of padding zeros. Therefore, the partial reconfiguration overhead for Virtex-4 occupies at least $\frac{3312+23616}{bitstream\_size} = 12.46\%$ of the bitstream. For ACS, the partial reconfiguration overhead is just the number of frames thus the overhead is only $\frac{3600/8}{1156*41*4+3600/8} = 0.24\%$ where 1156 is the number of frames to be reconfigured. The percentage speedup can be estimated by $\frac{bitstream\_size(ACS)}{bitstream\_size(Virtex4)} * 100\% = 13.95\%$, which is in line with the 13.91% shown in Table 4. The difference between 13.95% and 13.91% comes from the intrinsic reconfiguration delay of the ACS system. Because the size of the system N is chosen to be 8, the system itself needs to be reconfigured 450 times in order to complete one partial reconfiguration for the target device. This results in some delay that is included in Table 4 but cannot be seen in the theoretical analysis.

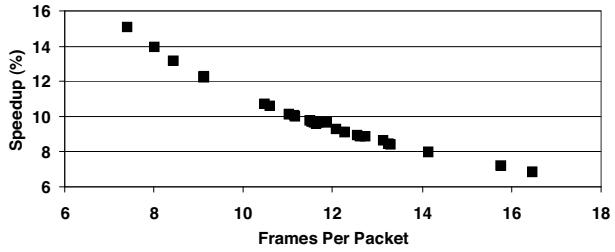

Figure 5. Speedups for ACS v.s. the number of frames per packet in Virtex-4 bitstreams

Sparsity here refers to the number of frames per packet in the bitstream. When there are less frames in a packet, the average overhead per frame in a bitstream will be larger and better speedup should be achieved by ACS. Figure 5 plots the speedups of ACS versus the sparsity and shows that the more sparse the partial bitstream is, the better performance ACS can achieve over Virtex-4.

5. Conclusions

In this paper, we presented a complete design for the ACS system that has practical benefits. We simulated and evaluated the system using eight real-world benchmarks in different application domains. The system can easily be extended onto larger devices with the introduction of the selector block so extra hardware overhead incurred will be small. The ACS system is now complete with marker loading mechanism and the results are compared with a XC4VLX15 device using partial bitstreams generated by Xilinx ISE. The results show that 6.83%-15.07% speedups can be achieved while the hardware cost is no greater than that of a XC4VLX15 device. The ACS system shows a promising potential for future reconfigurable architectures and may provide even better speedups on systems with reduced granularities.

References

[1] Opencores. http://www.opencores.org.

[2] Virtex series configuration architecture user guide, 2000.

[3] Xilinx, Development system reference guide, 2005.

[4] Virtex-4 user guides, 2005. www.xilinx.com/support/ documentation/user_guides/ug070.pdf.

[5] Xilinx, Correcting single-event upsets in Virtex-4 platform FPGA configuration memory, 2008.

[6] T. Becker, W. Luk, and P. Y. K. Cheung. Enhancing relocatability of partial bitstreams for run-time reconfiguration. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 35–44, 2007.

[7] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[8] E. El-Araby, I. Gonzalez, and T. El-Ghazawi. Performance bounds of partial run-time reconfiguration in high-performance reconfigurable computing. In *1st International Workshop on High-performance Reconfigurable Computing Technology and Applications*, pages 11–20, 2007.

[9] F. He, X. Song, M. Gu, G. Yang, W. Hung, and J. Sun. Probabilistic optimization for FPGA board level routing problems. *IEEE Transactions on Circuits and Systems*, 53(4):264–268, April 2006.

[10] J. Kuo, H. ElGindy, and A. Ku. A novel network architecture support for fast reconfiguration. In *International Conference on Field-Programmable Technology*, pages 353–356, 2007.

[11] S. Lange and M. Middendorf. Hyperreconfigurable architectures for fast run time reconfiguration. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 304–305, 2004.

[12] U. Malik and O. Diessel. On the placement and granularity of FPGA configurations. *IEEE International Conference on Field-Programmable Technology*, pages 161–168, 2004.

[13] W. H. Mangione-Smith. Atr from UCLA. *Personal Communications*, 1999.

[14] E. M. Panainte, K. Bertels, and S. Vassiliadis. FPGA-area allocation for partial run-time reconfiguration. In *Proceedings of ProRISC*, pages 415–420, November 2005.

[15] T. T. Ye, L. Benini, and G. D. Micheli. Packetized on-chip interconnect communication analysis for MPSOC. In *International Conference on Design, Automation and Test in Europe*, page 10344, 2003.