An Efficient Compression Scheme for Checkpointing of FPGA-Based Digital Mockups

Ting-Shuo Chou, and Tony Givargis
Dept. Computer Science
University of California, Irvine
Irvine, CA 92697
{tingshuc, givargis}@uci.edu

Chen Huang, Bailey Miller, and Frank Vahid
Dept. Computer Science & Engineering
University of California, Riverside
Riverside, CA 92521
{chuang, bmiller, vahid}@cs.ucr.edu

Abstract - This paper outlines a transparent and nonintrusive checkpointing mechanism for use with FPGA-based digital mockups. A digital mockup is an executable model of a physical system and used for real-time test and validation of cyber-physical devices that interact with the physical system. These digital mockups are typically defined in terms of a large set of ordinary differential equations. We consider digital mockups impelemented on field-programmable gate arrays (FPGAs). A checkpoint is a snapshot of the internal state of the model at a specific point in time as captured by some controller that resides on the same FPGA. We require that the model continues uninterrupted execution during a checkpointing operation. Once a checkpoint is created, the corresponding state information is transferred from the FPGA to a host computer for visualization and other off-chip processing. We outline the architecture of a checkpointing controller that captures and transfers the state information at a desired clock cycle using an aggressive compression technique. Our compression technique achieves 90% reduction in data transferred from the FPGA to the host computer under periodic checkpointing scenarios. The checkpointing with compression yields 15-36% FPGA size overhead, versus 6-11% for checkpointing without compression.

I. Introduction

Cyber-physical systems (CPSs) are systems where computational elements closely integrate and interact with physical environments. Examples of CPSs include aerospace, automotive, and medical systems. The functioning of a cyber device within its physical environment creates a challenge during the test and validation phase. Specifically, to test and validate a cyber device, one may conduct the test within a real physical setting, hence losing key debug capabilities such as slower/faster than real-time execution, pause and resume, step-by-step debugging, and deterministic rewind and replay. Alternatively, a cyber device can be tested within a simulated environment, where all aspects of the test and validation are controllable. This paper focuses on the latter.

A case for simulating the physical environment for the purpose of test and validation of a cyber device is offered by Miller et al. using so called digital mockups [7]. These digital mockups are fast and accurate models of physical systems implemented to execute in real-time (or faster) on an FPGA. Similar digital mockups are shown to be feasible for the purpose of test and validation of cyber devices, given their high degree of configurability, observability, and controllability. Works such as that proposed by Pimentel [3] and Huang [8] which utilize FPGAs to simulate physical models containing large number of ordinary differential equations (ODEs) further illustrate the feasibility of using digital high-speed models during test and validation phase of CPS devices.

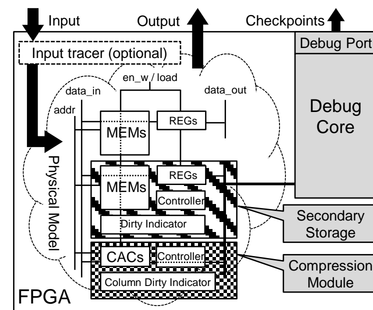

Fig. 1. Overview

To increase the usefulness of FPGA-based digital mockups, we introduce the design and architecture of a checkpointing controller, intended to reside on the FPGA and transparently capture and transfer checkpoints at a desired clock cycle using an aggressive compression technique. A checkpoint in this paper is a snapshot of the internal state of the model at a specific point in time, captured by the checkpointing controller and transferred to a host computer for visualization and other off-chip processing. A preferred checkpointing mechanism allows the model to execute in an uninterrupted manner during the checkpointing operation.

In essence, our problem is that of capturing a selected subset of the internal state of the FPGA, including the contents of memory blocks and flip-flops. Wheeler [9] and Graham [10] have solved the problem of retrieving and restoring data in memory blocks and flip-flops through design instrumentation. Nevertheless, continuously creating checkpoints of digital mockups poses new challenges compared to conventional approaches.

First, FPGA-based digital mockup are compute intensive, usually solving a large number (hundreds to thousands) of ODEs in real-time for sufficient model accuracy. The corresponding state of such complex models may include hundreds to thousands of variables, making capture and off-chip transfer of the state impractical. Instead, a common solution is the use of on-chip memory to hold a checkpoint prior to transfer, limiting the number of checkpoints. Second, a digital mockup must continue execution during checkpointing operations as the mockup is connected to a cyber device that is expecting continuous behavior. Therefore, the clock applied to a physical model cannot be suspended for any period of time; instead, the instantaneous

capture of a consistent checkpoint, namely, the creation of a complete copy of all model variables in a single cycle, is necessary. Third, a checkpoint must be transferred to the host computer in a timely manner. Reducing the minimum time interval between two consecutive checkpoints, thus increasing the overall checkpointing rate, is desirable [3, 8, 11]. An aggressive compression method, as proposed in this paper, plays an important role in increasing this rate.

In this paper, we address these challenges using a checkpointing controller as shown in Fig. 1. We use secondary storages for duplicating the state of the physical model. We read data at a certain clock cycle by setting the secondary storages to read-only mode while the physical model runs normally. If the instrumented storage is of memory type (i.e., having an address input), we also insert compact controllers for keeping data in secondary storages consistent with those in the physical model. The check-pointing controller incorporates a compression scheme that reduces the size of a checkpoint with minimal additional circuit (area) overhead. Compression substantially shortens the checkpoint transfer time. To support our scheme, we introduce specialized caches called Column Accessible Caches (CACs) that can be written as a row addressable memory and read as a column addressable memory. We demonstrate our technique in a framework that currently supports checkpointing on any Xilinx FPGA-based digital mockup. The framework consists of a debug soft core (namely MicroBlaze provided by Xilinx Inc.) and two design templates, which are secondary storage modules and compression modules written in VHDL with interfaces as shown in Fig. 1. The framework minimizes modifications to the original physical model. We use a human lung and a medical ventilator cyber device as an example to demonstrate our techniques [7].

The rest of this paper is organized as follows. Section 2 describes our checkpoint mechanism, the added logic circuits, and secondary storages. Section 3 describes our compression scheme and how CACs work. Section 4 provides the performance of our scheme in terms of compression rate. Section 5 concludes.

II. Checkpointing Architecture

The main component of the proposed checkpointing mechanism are secondary storage structures to mirror the model state variables. Here, the model always executes using its primary memories while the debug core strictly manipulates the duplicated data in secondary storage. Hereafter the original memory blocks of the model are called primary memory. The duplicated memory blocks are called secondary memory.

Fig. 2(a) shows an example of design instrumentation on a typical memory block with inputs and outputs such as *data_in*, *data_out*, *we* (write enable) and *addr*. The gray lines and boxes represent the original model circuit and the remaining circuitry represents the new signals added to accommodate the debug core. *Din_1*, *Addr_1*, *Dout_1* and *WE_1* are input to the original model circuit while the debug

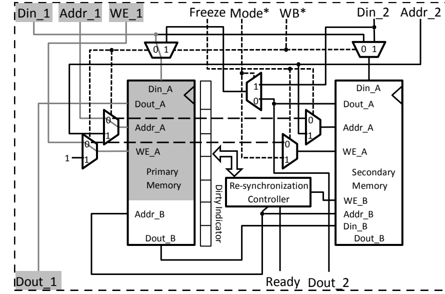

Fig. 2(a). Example of instrumentation

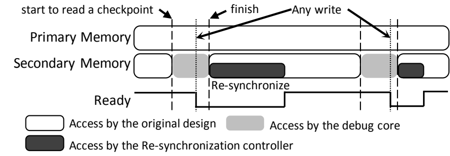

Fig. 2(b). Timing diagram

core uses the control signal *Freeze*, *WB* (write back) and *Mode* to execute four operations, as summarized in Table I. These four operations, originating from the debug core, use *Din_2*, *Addr_2* and *Dout_2* to retrieve or restore a checkpoint. The status signal *Ready* reflects whether the primary and the secondary memories are consistent. This signal is necessary because any write on primary memory during a checkpoint operation will make secondary memory inconsistent with the primary memory. During a checkpoint operation the debug core asserts *Freeze* to set all secondary storages to read-only mode. The timing diagram during a checkpoint operation is illustrated in Fig. 2(b).

Table I

Checkpointing controller signal description

Freeze	WB*[1]	Mode*	Description
0	0	0	Normal use. Data at Din_1 are duplicated
1	0	0	Read data in secondary memory
1	1	0	Write back using data in secondary memory
1	1	1	Write back using data at Din_2

A data inconsistency between the primary and secondary memories triggers the execution of the re-synchronization process at the end of the checkpoint operation. The re-synchronization controller, dirty indicator, and the extended read port of primary memory in Fig. 2(a) are inserted to resolve consistency between the primary and secondary memories. The dirty indicator is an array of flags (one per word) that is set during a checkpoint operation on a write to a corresponding primary memory word. Using the dirty indicator, the re-synchronization controller copies the dirty data from the primary memory to the secondary memory using extended read/write ports. Fig. 3 shows the pseudo code of the re-synchronization controller. The execution time of the re-synchronization controller, at most, is equal to the depth (number of words) of the primary memory. Moreover, the checkpoint operation is nonintrusive as the primary memory is never accessed by the debug core or the re-synchronization controller as shown in Fig. 2(b).

[1] *WB** may be set to (*Freeze* AND *WB*) and *Mode** may be set to (*Freeze* AND *WB* AND *Mode*) in order to eliminate unnecessary control signal combinations.

```
1 FOR addr = 0 to DEPTH_OF_MEMORY
2   IF dirty_indicator[addr] == true AND (ADDR@primary_memory != addr
3      OR WE@primary_memory == false) THEN
4      secondary_memory[addr] = primary_memory[addr]
5   ELSE
6      goto Line 2
7   END IF
8 END FOR
```

Fig. 3. Re-synchronization controller algorithm

The re-synchronization activity is also nonintrusive, as it utilizes the dual port (2W/2R) memory supported by most FPGAs.

III. Compression Scheme

Many algorithms have been applied to compress plain text, images, video, scientific graphs, and so on. According to the pigeonhole principle, no lossless compression algorithm can efficiently compress completely random data. In this paper, we take advantage of the fact that the state information of a digital mockup is comprised of data generated solving a large number of ODEs. These ODE circuits output continuous time values that drift slightly when sampled frequently. Moreover, when digital mockups are used to replace physical models, such a human lung or heart, the solution of these ODEs is restricted to be within a certain range (e.g., the volume and pressure at any branch of a human lung can only vary within some narrow range). Based on these observations, our compression scheme takes a data differencing approach to reduce data transmission to the host under periodic checkpointing. Specifically, we outline a computationally efficient architecture that supports delta (difference between two consecutive samples) encoding [15].

A. Column Accessible Cache

A Column Accessible Cache (CAC) is a memory structure containing K rows of N-bit words. As with traditional memories, the CAC words are accessed using log(K) row address bits. However, a CAC as proposed in this work allows addressing the data using log(N) column address bits as well.

For our application, using a word as the unit for data differencing will waste too many bits in common cases. Assuming only one nibble (4-bits) changes within a word during the time between two consecutive checkpoints, we have to encode the changed nibble with seven dummy (unchanged) nibbles plus the indexing bits since eight nibbles (i.e., a word) should be encoded as a whole. In this case, the ratio of indexing bits plus dummy (unchanged) bits to data (changed) bits is 8.75 (i.e., (7 + 28) / 4). An unrealistic 8.75 bits are needed to encode a single data bit. Thus, using a word as the unit of differencing is not efficient. Therefore, we choose a nibble to be the unit for data differencing. To ideally track the data changes as a function of time, eight flags are needed to track the changed nibbles in a 32-bits word. With these flags, we place the dirty words with fewer than four changed nibbles into a specialized cache and then access and encode them using column addressing, using CACs.

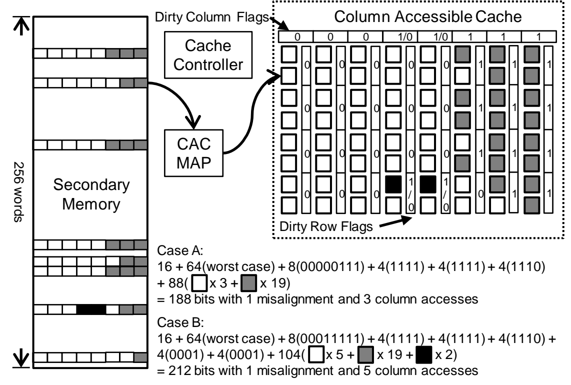

Fig. 4. CAC architecture

Fig. 4 shows the CAC architecture and a dedicated cache controller that manages the contents of the CAC. A practical CAC may be organized as an 8-by-8 two dimensional matrix of nibbles. Eight single bit dirty column flags are used for tracking changes in each column. Likewise, every eight nibbles, within a word, share a dirty column flag. There are also dirty row flags but, note that, a dirty row flag is not shared by a row of nibbles; a dirty row flag is shared by the two nibbles of a column byte, which is a byte in the order of column (see the CAC architecture in Fig. 4). Each column in the CAC has one column word, four column bytes, or eight nibbles. There are 32 dirty row flags in a CAC. The 32 dirty row flags are actually the bit map of dirty nibbles. The 8 dirty column flags together with the 32 dirty row flags are used to encode the dirty nibbles.

The CAC Map in Fig. 4 is used to record the address mappings between secondary memory and the CAC. For instance, a 7-bits address in a 128-words secondary memory is mapped to 3-bits address in a CAC. The size of a CAC Map is the logarithm of the memory depth of secondary memory multiplied by 8 (i.e., the number of rows in a CAC). A CAC Map also has 8 additional bits used to indicate which mapping is valid.

After a checkpoint, data is transmitted to the host using an encoding scheme as depicted in Fig. 5. The first data field is ValidEntries that indicates which entry is valid. The field is 8 bits long and mandatory. If there is no valid entry, the field's value is zero. The second field is the 8-bits long NewMapEntryTable that indicates which entry has a new mapping. If no entry needs update or no entry is valid, this field can be omitted. The third field is MapEntries, included only when NewMapEntryTable is set. The field contains new address mappings as indicated by NewMapEntryTable. N is the round up value of the logarithm of memory depth. The fourth field is 8-bits long DirtyColumnFlags, indicating the dirty columns. The fifth field is DirtyRowFlags. The dirty row flags are included only when their corresponding dirty column flag is set. The last field is Nibbles and it

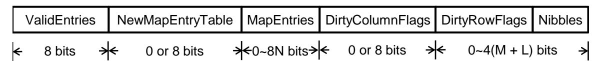

Fig. 5. Encoding format

contains the actual data. A nibble is included only when its corresponding dirty row flag is set. Note that the nibbles that share the same row dirty flags are included or excluded together. M and L are integers reflecting the total number of dirty column and row flags that are set.

Two cases in Fig. 4 are given to illustrate the benefits of CACs. In Case A, the dirty nibbles are the gray squares. The total encoded size based on the encoding formant in Fig. 5, in worst case, will be 188 bits, where we assume all entries in the CAC Map are valid and require update. The dirty nibbles are the black and gray squares in Case B. The total encoded size, in the worst case, will be 212 bits. If a CAC is not applied, the total size will be 320 bits, which are eight dirty words plus their 8-bit indexing data. In these two examples, a CAC yields at least 41% and 34% data size reduction.

A CAC serves as a cache of dirty nibbles of the secondary storage. A cache controller is required for choosing promising entries (i.e., a word with fewer than four dirty nibbles). The cache controller executes a simple but efficient periodic procedure. At the beginning of the procedure, the controller searches the CAC for entries with more than four dirty nibbles and removes them to make room for promising entries. If a dirty word has more than four dirty nibbles, leaving the dirty word in its original memory block and reading it through a row access, as well as encoding it, are more efficient. The cache controller also scans the secondary memory and adds promising entries into a CAC from top to bottom on a first come first serve basis. The periodic cache refreshing procedure always keeps a CAC in good condition, where all entries in a CAC are very likely to have dirty nibbles fewer than four. Furthermore, this periodic procedure carried out by the cache controller of each CAC offloads (or amortizes) the computation of encoding all dirty nibbles and words carried by the debug core. Specifically, the cache controller of each CAC reads the secondary memory which it attaches to and chooses promising entries to put in a CAC during the period of normal execution (i.e., the white rounded rectangles in Fig. 2(b)). Without these cache controllers, the debug core would choose promising entries of each secondary memory in a sequential manner during a checkpoint operation (i.e., the light gray rounded rectangles in Fig. 2(b)).

B. Distributed CACs and Global CAC

Intuitively, we achieve better compression rate if a CAC is filled with dirty words that have fewer than four dirty nibbles. However, a physical model may have distributed memory blocks and the number of dirty words with dirty nibbles fewer than four may not be sufficient to fully utilize each CAC. Thus, we also propose a global CAC (GCAC), which consists of several CACs and a single controller that scans all dirty flags of every memory block.

IV. Experimental Results

We use the Weibel lung model as a case study. The number of ODEs of a Weibel lung model increases exponentially as the number of generations (levels) of the lung model increases. Therefore, the amount of data presenting states of a Weibel lung model also increases exponentionally.

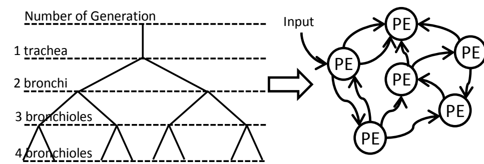

Fig. 6. Weibel lung model

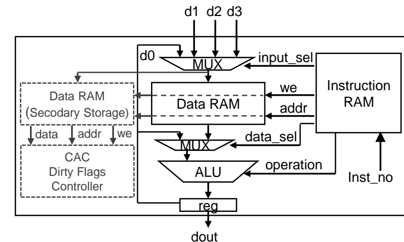

Fig. 7. PE architecture and instrumentation

A. Weibel Lung Model

The Weibel lung model was proposed by E. R. Weibel [12]. The left part of Fig. 6 depicts a 4-generation lung model, including the trachea, the bronchi, and the bronchioles lung elements. The accuracy of this model, as well as the number of ODEs, increases as the number of generations increases. 4-generation, 6-generation, 8-generation, and 10-generation lung models are represented by 56, 251, 1019 and 4091 ODEs, respectively. In Huang's work [8], these ODEs are mapped to several processing elements (PEs) that can solve ODEs through the Runge-Kutta method. The values of flow, pressure, and volume at the joining of two branches are evaluated at one millisecond time resolution. To meet this time resolution, more processing elements (PE) [8] are required as the number of generations increases.

Fig. 7 shows the architecture of a PE. A PE consists of instruction RAM, data RAM, and an ALU. The instruction RAM is read only and the values of flow, pressure, and volume are stored in the data RAM. *d1*, *d2*, and *d3* are incoming data lines from other PEs. *dout* is the output data line. To checkpoint the lung model and support compression, we manually insert the secondary storage, the CAC, the dirty flags, and the controller. The wires needed to connect to the inserted logic are *data*, *addr* and *we*, as described earlier.

B. Compression Rate Analysis

To explore the benefits of CACs, we built a customized simulator that cycle-accurately simulates PEs and their network. The simulated lung models include a 4-generation Weibel lung mapped to 4 PEs (we4_pe4). we*X*_pe*Y* indicates the Weibel lung model of *X*-generation mapped to *Y* PEs. We also simulate the input pressure ranging from -500 to 500 mmHg. This range is sufficient to cover possible pressure

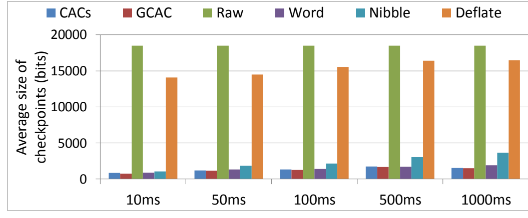

Fig. 8. Average size of checkpoints

values of a human lung. The input pressure oscillates at the frequency of around 15 times/minutes, which is the general breath rate of an adult. We summarize our simulation parameters and their possible values in Table II.

Table II
Simulation parameters and possible values

Input parameter	Possible value
lung model	we4_pe4, we6_pe8, we8_pe16, we10_pe196
input pressure	-500 ~ 500 mmHg
input oscillation rate	10 ~ 20 times / minute
input shape	square and sine
interval between checkpoints	10 ~ 1000 ms

The size of checkpoints is used to identify the performance of our compression method. We compare the size of checkpoints of our approaches (CACs and GCAC) to four other approaches, which are Raw, Deflate, Dirty Word Tracking (DWT) and Dirty Nibble Tracking (DNT). Raw represents the size of checkpoints without any compression. Deflate stands for the famous data compression algorithm used in zlib [14] and gzip [13]. We serialize the data in memory blocks and feed the result into Deflate. DWT and DNT are the basic data differencing techniques using dirty flags to track dirty words or nibbles. As a unit of differencing, DWT-approach uses a word whereas DNT-approach uses a nibble. In order to have a fair comparison, we include complete indexing bits for each approach. The calculation of the size of each approach is listed as follows.

Raw= Σ (NumberPEWidth+PEDataRAMSize)

Deflate= Σ (NumberPEWidth+PEDataRAMComressedSize)

DWT= Σ (NumberPEWidth+MaxNumberDirtyWordWidth+ (WordEncodingWidth × NumberDirtyWord))

DNT= Σ (NumberPEWidth+MaxNumberDirtyNibbleWidth+ (NibbleEncodingWidth × NumberDirtyWord))

CACs= Σ (NumberPEWidth+CACEncoding+ MaxNumberDirtyWordWidth+ (WordEncodingWidth × NumberDirtyWordInRAM))

GCAC=GCACEncoding + Σ (NumberPEWidth+MaxNumberDirtyWordWidth+ (WordEncodingWidth × NumberDirtyWordInRAM))

Here, NumberPEWidth is the logarithm of the number of PEs. For a lung model mapped to 256 or fewer PEs (common case), an 8-bit byte is used. PEDataRAMSize presents the size of the data RAM of a certain PE. PEDataRAMCompressedSize presents the compressed data size of the data RAM of a certain PE based on the Deflate algorithm. MaxNumberDirtyWordWidth is the logarithm of maximal number of dirty words, which is the size of a data RAM block. For PEs having data RAM up to 256 words (32-bits), a byte is used. WordEncodingWidth is MaxNumberDirtyWordWidth plus 32 (i.e., the size of a word). The calculation for MaxNumberDirtyNibbleWidth is

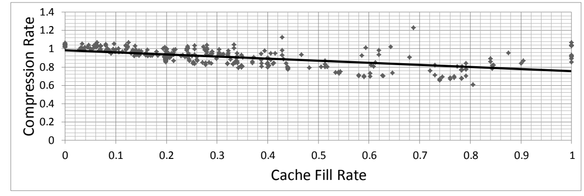

Fig. 10(a). Cache fill rate v.s compression rate (CACs)

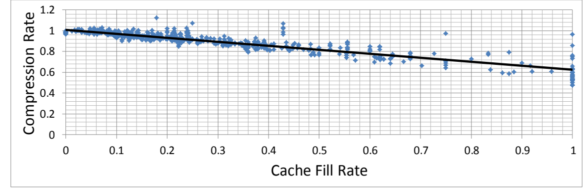

Fig. 10(b) Cache fill rate v.s compression rate (GCAC)

similar to MaxNumberDirtyWordWidth. The length of CACEncoding and GCACEncoding are calculated based on the encoding format in Fig. 5.

Fig. 8 shows the average size of a checkpoint for different intervals between checkpoints. Results are obtained from averaging the size of 800 checkpoints under these parameters: the lung model is we6_pe8 and input oscillation rate is 15 times per minute; the intervals between two successive checkpoints are 10, 50, 100, 500, and 1000ms and the interval between cache refreshing is set to tenth of the interval between checkpoints. The average size of checkpoints generated by each approach increases as the interval between checkpoints increases; intuitively, data is more likely to be modified over larger time intervals. We observe that differencing techniques, including CACs-, GCAC-, DWT- and DNT-approach, outperform the Raw- and Deflate-approach in the majority of cases. Hence, we conclude that for FPGA-based mockups of physical systems, data differencing techniques can reduce the size of a checkpoint by 90%.

We also compare the performance of the CACs-approach with that of the GCAC-approach. Here, the compression rate is defined as the size of checkpoints generated by CACs- and GCAC-approach divided by the size of checkpoints generated by the DWT-approach. We further define the cache fill rate to be the ratio of the number of entries inside CACs or GCAC to the maximal number of entries they can hold. A CAC fully filled with valid entries has cache fill rate equaling to one. We perform exhaustive search on input parameters and get thousands of checkpoints for analysis. Fig. 10(a) and 10(b) shows the relations between the compression rate and cache fill rate. The black line in Fig. 10, generated by linear line-fitting, indicates the trend of data. As the cache fill rate increases, the compression rate drops. The slopes of the black lines in Fig. 10(a) and 10(b) are -0.24 and -0.36, respectively. Generally, GCAC-approach yields lower compression rate than CACs-approach at the same cache fill rate. Statistically, we can conclude that GCAC-approach works better than CACs-approach. Another observation from Fig. 10(a) and 10(b) is that most points have a compression rate smaller than one, indicating that the size of checkpoints generated by the CACs/GCAC-approaches is smaller than that of the

DWT-approach in the majority of cases.

C. FPGA Area Overhead Analysis

To further validate our architecture and evaluate the area overhead in terms of FPGA resources, we implemented checkpointing and synthesized it using the Xilinx ISE 13.2 tool chain. Table III lists the FPGA resources taken by the original design (2nd and 4th columns) and resources taken by our chekcpointing module (3rd and 5th columns). RAM128X32S refers to the checkpointing module used to instrument a 128-words memory block. The other modules are also named similarly. The resource overhead includes secondary storages, dirty indicators, and all the controllers. We see that the module used to instrument bigger memory blocks has smaller overhead since the overhead of a controller is nearly constant while the overhead of the dirty indicators and the secondary memory are proportional to the size of the corresponding memory block that is instrumented. Table III also lists the FPGA resources consumed by a CAC and compression engines (i.e, CAC_32X1, CAC_64X1 and CAC_128X1). CAC_32X1 includes a CAC, the cache controller, the CAC Map and the dirty flags attached to 32-words secondary memory. CAC_64X1 and CAC_128X1 are named literally. The required resources do not increase excessively when compared to a CAC.

Table III
FPGA area overhead of checkpointing and compression engine

	LUTs	LUTs (modified)	Slice	Slices(modified)
BRAM36	0/1	200/2	0	72
RAM128X32S	100	434 (334%)	31	170
RAM64X32S	66	319 (383%)	21	112
RAM32X32S	16	177 (1006%)	4	57
RAM32X16S	8	119 (1387%)	2	54
CAC	320	-	87	-
CAC_32X1	457	-	138	-
CAC_64X1	476	-	147	-
CAC_128X1	484	-	144	-

Table IV
FPGA area of Weibel lung models

	LUTs	Slices	BRAM
we4_pe4	7317	2832	52
we4_pe4_D	7779(6%)	3132	52 (0%)
we4_pe4_DC	8408(15%)	3363	52 (0%)
we6_pe8	8769	3720	56
we6_pe8_D	9875(13%)	3841	56 (0%)
we6_pe8_DC	11235(28%)	4505	56 (0%)
we8_pe16	12963	4747	64
we8_pe16_D	14431(11%)	5459	64 (0%)
we8_pe16_DC	17621(36%)	6656	64 (0%)

Table V summarizes the resource overhead on different generation Weibel lung models. we*X*_pe*Y* stands for the *X*-generation lung model mapped to *Y* PEs. The suffix _D refers to the version with instrumentation of checkpointing mechanism. The suffix _DC refers to the version with checkpointing mechanism and compression support. The resource overhead is mostly LUTs and slices with minimal BRAM requirements.

V. Conclusions

In this paper, we presented a transparent checkpointing mechanism and an application-specific compression scheme, targeting FPGA digital mockups of physical systems. We proposed a Column Accessible Cache (CAC) to support the compression scheme. We evaluated the size reduction of checkpoints through distributed CACs and global CAC (GCAC). We observed that a compression approach based on GCAC works better than CACs. Statistically, both schemes provide an additional compression rate of 5% to 20% using our proposed data differencing approach. Data differencing combined with GCAC achieves 90% reduction in the size of periodic checkpoints. We evaluated the resource overhead of our checkpointing architecture using a digital mockup of a human lung mapped to a Xilinx Virtex5 FPGA. We observed a reasonable 6% to 11% increase in the FPGA area utilization. When compression support was included, we observed a 15% to 36% resource overhead in terms of FPGA area utilization. This additional overhead may be justified in applications where transparent, high frequency checkpointing is required.

Acknowledgements

This work was supported by the National Science Foundation (1016789, 1136146).

References

[1] Michigan Instruments. Training and Test Lung (TTL) and PenuView http://www.michiganinstruments.com/resp-ttl.htm, 2009.
[2] Drosdol, Johannes, F. Panik, "The Daimler-Benz Driving Simulator: A Tool for Vehicle Development," Society of Automotives Engineers, 1985.
[3] J. C. G. Pimental, Y. G. Tirat-Gefen, "Hardware Acceleration for Real Time Simulation of Physiological Systems," Engineering in Medicine and Biology Society, 2006.
[4] Ashish Gholkar, Amitay Isaacs, Hemendra Arya, "Hardware-In-Loop Simulator for Mini Aerial Vehicle," Sixth Real-Time Linux Workshop, NTU, Singapore, Nov. 3-5, 2004.
[5] B.M. Hanson, M.C. Levesley, K. Watterson, P.G. Walker, "Hardware-in-the-loop-simulation of the cardiovascular system, with assist device testing application," Medical Engineering & Physics, Volume 29, Issue 3, pp. 367-374, April 2007.
[6] Z. Jiang, M. Pajic, A. Connolly, S. Dixit, R. Mangharam, "Real-Time Heart Model for Implantable Cardiac Device Validation and Verification," 22nd Euromicro Conference on Real-Time Systems, pp. 239-248, 2010.
[7] B. Miller, F. Vahid, T. Givargis, "Digital Mockups for the Testing of a Medical Ventilator," in IHI, 2012.
[8] C. Huang, F. Vahid, T.Givargis, "A custom FPGA processor for physical model differential equation solving," Embedded Systems Letters, 2011.
[9] T. Wheeler, P. Graham, B. Nelson, B. Hutchings, "Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification," in FCCM, 2001.
[10] P. Graham, B. Nelson, B. Hutchings, "Instrumenting bitstreams for debugging FPGA circuits," in FCCM, 2001.
[11] M. Yoshimi, Y. Osana, T. Fukushima, H. Amano, "Stochastic simulation for biochemical reactions of FPGA," in FPL, 2004.
[12] E. R. Weibel, Morphometry of the Human Lung. Berlin, Germany: Springer-Verlag, 1963.
[13] L. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," IEEE Transactions on Information Theory, May 1977
[14] zlib: Compression library http://zlib.net/.
[15] S.T. Klein, T.C. Serebro, D. Shapira, "Modeling delta encoding of compressed files," in DCC, 2006.