RECORD: Reducing Register Traffic for Checkpointing in Embedded Processors

Tuo Li
School of Computer Science
and Engineering
University of New South Wales
Sydney, Australia
Email: tuol@cse.unsw.edu.au

Jude Angelo Ambrose
Canon Information Systems
Research Australia
Sydney, Australia
Email: angelo.ambrose@cisra.canon.com.au

Sri Parameswaran
School of Computer Science
and Engineering
University of New South Wales
Sydney, Australia
Email: sridevan@cse.unsw.edu.au

***Abstract*—Checkpoint/recovery, as a classic method, has been widely used for overcoming transient faults in computing systems. The basic function of checkpoint/recovery is to save the system states periodically and to restore the system states by using the saved states if a fault occurs. With the hardware-implemented checkpointing mechanism executing at runtime, a processor will have substantially increased register-file reads. For embedded processors, which typically have restricted design constraints on area, power, and performance, such increases might compromise the quality of the application greatly. In this paper, we present a checkpointing method, RECORD, aimed at reducing the resultant register traffic at runtime, by leveraging register data dependencies. The proposed checkpointing method can reduce redundant executions of register-file checkpointing. The experiments show that RECORD achieves improved register traffic reduction (20 %) along with reduced dynamic power consumption (approximately 20 %) in comparison to the state of the art with minimal area overhead. The leakage power increases marginally (about 2 %), but is more than compensated by the decrease in dynamic power.**

I. INTRODUCTION

Checkpoint/recovery [1] (also known as checkpoint/rollback) has been widely adopted to protect processor-based systems from transient faults [2]. Apart from reliability, checkpoint/recovery is also utilized in high-performance processors for speculative out-of-order execution [3], as a way to restore the system states from branch misspeculation. Note that this paper examines the reliability aspect of checkpoint recovery, rather than branch misspeculation. The functionalities of checkpoint/recovery are: one, saving the processor states periodically; and two, writing back saved system states after any fault is detected to restore the system. In a processor based system, checkpoint/recovery necessitates the storing of values at a particular point in time. Two types of values need to be stored: one, the values in register file (RF); and the other, the values in memory.

To accommodate the functionalities necessary for checkpoint/recovery, the baseline system must be extended, incurring sizable overheads, in terms of area, power, and performance. Such overheads come from: (1) the circuits implementing checkpoint storage and the control/data processing related to checkpointing and rollback, and, (2) the number of checkpointing executions at runtime. For example, if the system has more checkpointing executions, the checkpointing-related circuits will be switched on more often, resulting in more dynamic power consumption. Embedded processors usually have stringent constraints on area (i.e., logic gates), power, and performance [4].

The register-file (RF), which typically consumes 15% to 20% total power [5] in a processor, has been extensively studied for reducing power cost of processors [6]. Deploying and activating logging checkpoint/recovery in processors can increase *RF traffic* substantially, due to extra register reads from RF. The increased traffic will translate to considerable RF power overhead [6], compromising the quality of the application.

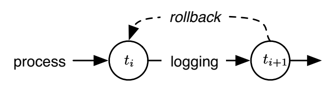


Fig. 1. Logging checkpoint/recovery

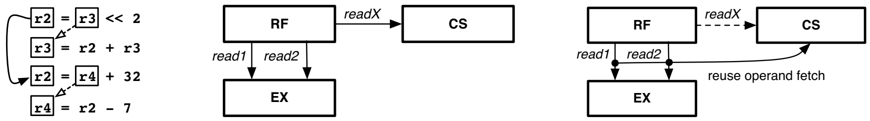
In this paper, we propose leveraging register data dependency to minimize the register traffic required by RF checkpoint/recovery. The proposed logging checkpoint/recovery scheme, named RECORD, considers various register data dependencies, which can potentially identify and eliminate the redundant executions of RF checkpointing at runtime. To the best of our knowledge, our approach is the first to realize a hardware-based logging checkpointing mechanism, which strategically utilizes the original processor executions to diminish the unnecessary checkpointing operations at runtime, for embedded processors. In order to evaluate the proposed scheme for embedded processors, we implemented RECORD in application-specific instruction-set processors (ASIPs), a representative type of embedded processor.[1]

II. RELATED WORK

Hardware-implemented checkpoint/recovery has been studied extensively and widely adopted in real-world applications. The RF checkpointing techniques in [7] and [8] are two typical full-separation techniques, which copy the entire RF to checkpoint storage with the same size as the original RF. A number of checkpointing executions are performed at the end of the checkpoint period. The number of checkpoint executions is usually the same as the number of registers in the RF. The checkpointing executions are not performed simultaneously with the normal instructions, thus checkpointing executions must occupy dedicated machine cycles (more performance overhead). The RF checkpointing technique in IBM S/390 G5 processor [9] is combined with lock-stepping (i.e., two pipelines executing with same data simultaneously), which checks the result at every instruction.

HP's RF logging-based checkpointing technique (as shown in Fig. 1), patented in [10], is a conventional logging technique. This technique allows the processor to perform extra reads from RF before writing to the registers in RF. The fetched value is then stored in the checkpoint storage. This technique

[1]Examples of commercial ASIPs include CADENCE/TENSILICA's XTENSA and SYNOPSYS's ASIP.



(a) Example code (b) Conventionl checkpointing (c) Reduced checkpointing

Scheme	*readX*	*writeX*
Conventional	4	4
RELI	3	3
RECORD	1	3

(d) RF traffic

Fig. 2. Motivation

is the basis of our study and is improved in this work (RECORD). RELI [11] is a more recent study on logging-based checkpoint/recovery, whose targets include both RF and memory. RELI's RF checkpointing is based on HP's [10] and reduced by using a one-bit history table, which indicates whether the register has been previously checkpointed in the checkpoint period or not.

FUJITSU'S SPARC64 processor [3] implements renaming checkpoint/recovery for handling branch misspeculation in out-of-order pipeline execution. By using a register alias table (RAT), the new register values are written to the renamed registers (a separate RF). These renamed registers will be discarded and the RAT will be restored if a fault is detected.

Besides, a large number of logging-based checkpoint/recovery studies [8], [12]–[14] have been proposed targeting cache and memory data. CARER [12] and SWICH [8] utilize dedicated cache lines to function as checkpoint storage. Hence, the cache replacement policy is modified. REVIVE [13] modifies the memory directory controller to allow some memory lines to function as the checkpoint storage for memory logging. SAFTYNET [14] handles RF, cache, and memory logging with additional checkpoint storages for each, in multiprocessor systems.

In comparison, RECORD differs from the previous RF checkpointing techniques in that RECORD requires much less checkpointing executions, hence less RF traffic. The closest existing technique, to RECORD, is RELI. The major advantage of RECORD over RELI is that RECORD considers various register data dependencies, and hence RECORD can leverage more opportunities for reducing checkpointing executions.

III. MOTIVATION

Fig. 2a presents a small piece of code from the SHA application which is part of the MiBench benchmark suite [15]. There are four instructions in the example code. In the conventional checkpointing method, the hardware for checkpointing could be described as shown in Fig. 2b. If the conventional checkpointing method were to be used, then for the first line of the code, the value of r2 will be stored in checkpoint storage before it is overwritten by r3<<2. This requires r2 to be read (*readX*) from the register file and then written (*writeX*) to the checkpointing storage (CS). r3 is to be read, shifted and stored in r2. We would need a total of one read from the register file over and above the normal reads, and one write to CS. In the next instruction, r3 will be stored, before being over written by r2+r3. Similarly, in the third instruction r2 will be stored again before being over written by r4+32. Finally, r4 will be stored before it is over written by r2-7. Thus, as shown in Fig. 2d, for this code segment we require 4 additional reads from the register file and 4 writes to the CS.

By the use of history table, the state-of-the-art method RELI reduced the number of additional reads from RF and the writes to the CS. In RELI, once r2 was stored in the first instruction, it was not stored again in the third instruction as long as both instructions happened to be within the same checkpointing period. For the same piece of code, the number of additional reads from the RF was 3 and the number of additional writes to CS was 3.

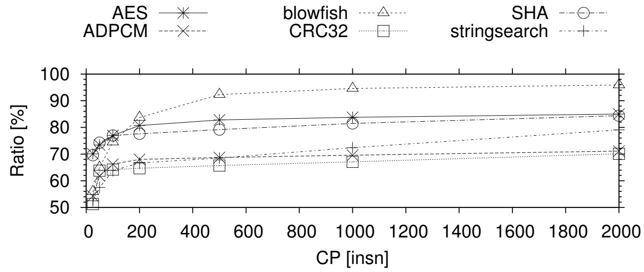


Fig. 3. Average register write-after-read ratio with 25, 50, 100, 200, 500, 1000, and 2000 checkpoint period length

In the RECORD method (shown in Fig. 2c) described in this paper, we were inspired by the fact that registers that are read from the RF, have to be often checkpointed later. For example, in Fig. 3, when the checkpointing interval is 1000, for all the applications examined, between 60 and 90 percent of registers read had to be checkpointed. In the example code given in Fig. 2a, in the first instruction, r2 is read from the register file and then written to CS just as in previous methods. However, r3 having been read from the RF to be used by the execution unit (e.g., ALU), is also stored in CS. Then, when the second instruction is executed, r3 is not stored again. This method at times might unnecessarily do checkpointing, however, as shown in Fig. 3, these are a small number, and that the additional reads from the RF are reduced significantly. By the use of the enhanced history table, for the RECORD method, we show that we need only one additional read from the RF with three additional writes for this example code segment.

IV. REDUCING REGISTER CHECKPOINTING

System Model: Let $RF = \{r_0, r_1, r_2, \ldots, r_N\}$ be the set of registers in register-file. Let $E = \{e_0, e_1, \ldots, e_M\}$ be the executed instruction sequence at runtime. Consider the processor has a built-in checkpointing mechanism. Depending on the checkpoint period length, in terms of instructions,[2] E is divided into subsequences of executed instructions $\{E_0, E_1, \ldots, E_K\}$ at runtime. Let $IN(e_i)$ and $OUT(e_i)$ be the source (input) registers and destination (output) register for the instruction e_i in the executed instruction sequence. At e_i, conventional checkpointing consists of two operations: (1) do additional read to RF to get the old value of $OUT(e_i)$, and, (2) write the old value of $OUT(e_i)$ to checkpoint storage.

Method 1: If $e_j \lhd e_i$[3] and $OUT(e_j) = OUT(e_i)$, there is a *write-after-write (WAW)* dependency (also known as output dependency) between e_j and e_i, noted as $e_j \delta^o e_i$. If $e_j \delta^o e_i$

[2] Such way of determining checkpoint period is widely used in practice [16].
[3] $\lhd$ means "precedes".

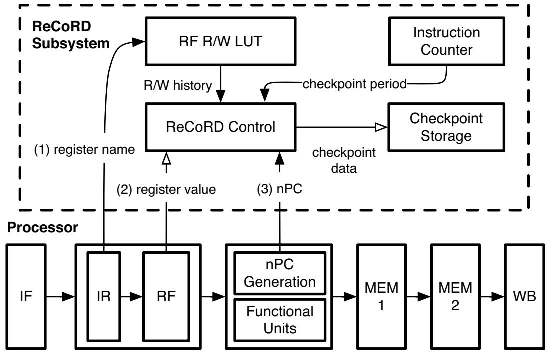


Fig. 4. Pipeline organization of the proposed architecture

exists and $e_i, e_j \in E_l$, the checkpointing execution at e_i is redundant and can be reduced. Method 1 removes one additional RF read and CS write. RELI functions similarly to this method.

Method 2: If $e_j \lhd e_i$ and $OUT(e_i) \in IN(e_j)$, there is a *write-after-read (WAR)* dependency (also known as anti-dependency) between e_j and e_i, noted as $e_j \delta^a e_i$. When $e_j \delta^a e_i$ exists and $e_i, e_j \in E_l$, checkpointing at e_i is redundant and can be reduced by reusing operand $IN(e_j)$ at e_j. Method 2 does not need additional RF reads to get the values of $IN(e_j)$, because $IN(e_j)$ is/are the operand(s), which is/are read from the RF by the native operations of the instruction. Note that Method 2 does need one additional write to CS.

V. RECORD ARCHITECTURE

A. Pipeline Organization

Fig. 4 presents a block diagram of the proposed RECORD architecture. The baseline processor at the bottom of the figure is augmented with RECORD subsystem at the top of the figure. At runtime, the program is executed across checkpoint periods. Every instruction is executed by the baseline pipeline for original operation and the RECORD subsystem for checkpointing. RECORD execution is in parallel with the baseline pipeline execution. The baseline pipeline's execution is intact when checkpointing is executed and halted when rollback is executed.

The baseline processor is an in-house implementation of PISA instruction set architecture [17], which is very similar to MIPS.[4] In PISA, the instruction width is 64 bits (only 32 bits are effectively used) and data width is 32 bits. The pipeline has six stages (two memory stages MEM1 and MEM2). The pipeline is an in-order single issue without delay slot. The details of the implementation will be discussed in Section VI.

The RECORD subsystem consists of four major components: the control unit, RF read/write history look-up table (R/W LUT), instruction counter, and checkpoint storage. The control unit is responsible for checking register R/W history and determining whether checkpointing is necessary for the current instruction. RF R/W LUT keeps the register read/write history of the executed instructions during one checkpoint period. Before entering a new checkpoint period, RF R/W LUT is cleared. This LUT has a 1-to-1 mapping with RF, with each register mapping to two registers. Hence, the size of this LUT is $2 \times N$ bits, where N is the number of registers in RF. The details of this LUT and the algorithm of how it is filled are given in the following sections. The instruction counter counts the number of executed instructions in a checkpointing period. This information is used by the control unit to determine the end of one checkpoint period. Checkpoint storage accommodates the backup values (checkpoint data). The checkpoint storage is a hardware stack. When a register is checkpointed, one register value and name are concatenated and pushed into CS as one checkpoint data. During rollback, the checkpoint data is popped out.

[4] http://imgtec.com/

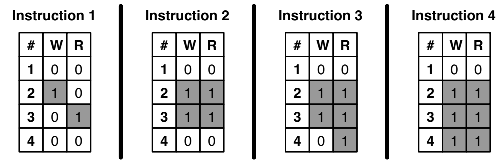


Fig. 5. Status of RF R/W LUT when executing the example code in Fig. 2a

The interface between the RECORD subsystem and the baseline system can be broken down into a few parts:

i. After instruction code is fetched and decoded, the baseline pipeline fetches register name of the destination register, i.e., $OUT(e_i)$, from instruction register (IR). The fetched $OUT(e_i)$, register name(s), is passed to RECORD. This register name is input to LUT and the associated value is the R/W history of this register during previous instructions. The R/W history is then given to control unit for analyzing dependencies and generating further control signals.
ii. The RECORD subsystem fetches (by RF read or operand reuse) the register value from baseline pipeline for checkpointing. If the checkpointing execution is redundant, this fetch will not happen.
iii. At the last instruction of a checkpoint period, if the system does not have errors (passes error checking), the next PC value (nPC) is sent to the RECORD subsystem as the backup PC. The backup PC is used in future rollback to restore the program back to the beginning of the current checkpoint period.

In order to support RECORD, some of the components in the baseline architecture are modified. The number of RF ports is determined by the maximum reads and writes of the RF at one pipeline stage. Logging checkpoint/recovery needs extra reads at the instructions that change RF values. Hence, the number of extra read ports is equal to the maximum RF writes in one instruction. With PISA, which has four read-ports and two write-ports originally, two extra read ports are required, determined by the worst-case instruction R-type `DLW` (double-load-word).

B. Runtime Control Algorithm

RECORD runtime mechanism is a hardwired implementation combining Method 1 and 2, which are elaborated in Section IV. This runtime mechanism is based on the use of RF R/W LUT. This LUT can be defined as an array $\vec{L} = \{\vec{s_0}, \vec{s_1}, \ldots, \vec{s_N}\}$, where each element $\vec{s} = \{R, W\}$ maps to one register in RF. The lower bit R (read history bit) indicates whether the register has been read in previous instructions of the current checkpoint period, while the higher bit W (write history bit) indicates whether the register has been written in previous instructions. In normal execution, R/W bits are checked to determine if the checkpointing execution is necessary. In rollback, only the registers with $W = 1$ will be restored using the values in CS. Fig. 5 demonstrates the change of LUT status during executing the example code in Fig. 2a, where **#** represents the index of LUT, *R* denotes the

Input: rs: source register name, rd: destination register name
Output: control signals

1: **if** $rs = rd$ **then** ▷ Check read/write same register
2: $e_i \delta^a e_i = 1$
3: **end if** ▷ Begin operand checkpointing
4: $\vec{s}_{rs} \leftarrow \vec{L}[rs]$ ▷ Fetch rs history
5: **if** $\vec{s}_{rs}[1] = 1$ **then**
6: Disable operand checkpointing on rs
7: **else if** $\vec{s}_{rs}[0] = 1$ **then**
8: Disable operand checkpointing on rs
9: **if** $e_i \delta^a e_i = 1$ **then**
10: $\vec{s}_{rs}[1] \leftarrow 1$ ▷ Change rs write history
11: **end if**
12: **else**
13: Enable operach checkpointing on rs
14: $\vec{s}_{rs}[0] \leftarrow 1$ ▷ Change rs read history
15: **if** $e_i \delta^a e_i = 1$ **then**
16: $\vec{s}_{rs}[1] \leftarrow 1$ ▷ Change rs write history
17: **end if**
18: **end if**
19: $\vec{L}[rs] \leftarrow \vec{s}_{rs}$ ▷ Update rs history in LUT
20: **if** $e_i \delta^a e_i = 1$ **then**
21: Disable checkpointing on rd
22: **else**
23: $\vec{s}_{rd} \leftarrow \vec{L}[rd]$ ▷ Fetch rd history
24: **if** $\vec{s}_{rd}[1] = 1$ **then**
25: Disable checkpointing on rd
26: **else if** $\vec{s}_{rd}[0] = 1$ **then**
27: Disable checkpointing on rd
28: $\vec{s}_{rd}[1] \leftarrow 1$ ▷ Change rd write history
29: **else**
30: Enable checkpointing on rd
31: $\vec{s}_{rd}[1] \leftarrow 1$ ▷ Change rd write history
32: **end if**
33: $\vec{L}[rs] \leftarrow \vec{s}_{rs}$ ▷ Update rd history in LUT
34: **end if**
35: **return**

Fig. 6. Algorithm for runtime control (simplified for the case of single source/destination register)

read history bit, and *W* denotes the write history bit. In this example, when executing the first instruction, as r2 is written and r3 is read, the W bit of r2 and the R bit of r3 are asserted. Similarly, at the later instructions, the W bits of r3 and the r4, as well as R bit of r4, are asserted.

Fig. 6 briefly describes the algorithm controlling ReCoRD runtime mechanism. This algorithm generates the control signals for checkpointing and updates the RF R/W LUT, in three major steps:

i. The first step (Line 1 to 3) is checking a very special case of Method 2, where the output register is also read ($OUT(e_i) \in IN(e_i)$) at the same instruction. If register r is also read, Method 2 can be applied to reduce this checkpointing.
ii. The second step (Line 4 to 19) determines whether operand register, i.e., $IN(e_i)$, checkpointing (Method 2) will be executed. During this step, RF R/W LUT is visited to look up the R and W bits of the input registers of the current instruction. For $r_k \in IN(e_i)$, unless both R and W bits are "0", r_k will not be checkpointed. If $W = 0$ and r_k is also written in the same instruction (result from the step 1), W will be asserted at this step.
iii. The third step (Line 19 to 34) determines whether the destination register, i.e., $OUT(e_i)$, checkpointing (Method 1) will be executed. During this step, RF R/W LUT is also visited to look up the R and W bits of the destination register. The destination register checkpointing will not be executed, unless both W and R bits are "0". If $R = 1$ and $W = 0$, the W bit must be asserted, to allow the corresponding register to be restored during rollback.

VI. Implementation

ReCoRD is implemented within an ASIP design flow as shown in Fig. 7. The baseline instructions set (using PISA instruction set [17]) is created using an architectural description language (ADL), which is then automatically converted

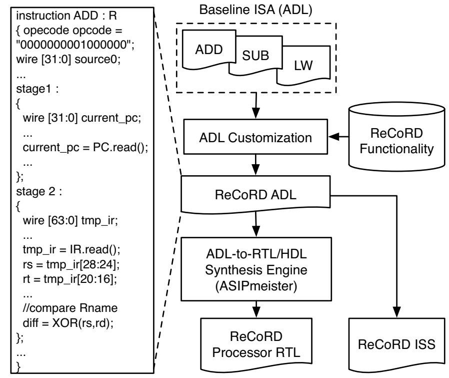


Fig. 7. Flowchart of ReCoRD processor implementation

to register-transfer level (RTL) description, and converted to a gate netlist using a logic synthesis engine. ADL is widely used for ASIP design to model a processor at a higher level than RTL. In this paper, we used PEAS [18] integrated in the processor design tool called ASIPmeister.

The baseline ISA's ADL is then customized to include ReCoRD functionality. This customization essentially modifies the ADL description of each instruction in the instruction set. The modifications include adding the pipeline resources for ReCoRD subsystem, adding the relevant data transfer, and adding data processing (bitwise logical operation, etc).

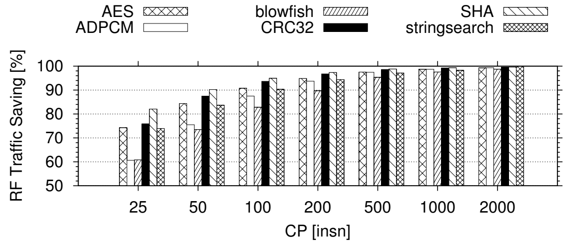
The resource-binding process is handled manually, during ADL programming. The CS and the RF R/W LUT are register-file type storages. The size of checkpoint storage can be determined in two ways. Aggressively, the depth can be determined by the maximum number of checkpointed registers by profiling the program. At worst case, the depth is equal to the number of registers in the RF. In this case, checkpoint data does not need register name and the CS does not need to be implemented as a stack. The instruction counter is a 1-increment-step counter. The width of the instruction counter is determined by the checkpoint period length. The control unit is automatically generated by ADL-to-RTL synthesizer, i.e., ASIPmeister, based on the description of ReCoRD in ADL. More details of this conversion process can be found in [18]. The resources and operations of ReCoRD are mainly scheduled in the instruction decode (ID) and execution (EX) pipeline stages. Based on the customized ADL, an instruction-set simulator (ISS) is implemented to simulate the system rapidly.

VII. Experiments and Results

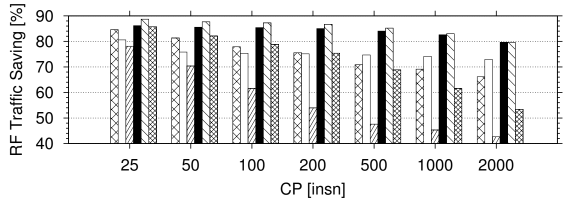
A. Experimental Setup

Two experiments are conducted to explore the efficacy of ReCoRD. The first experiment aims at measuring checkpointing events, RF traffic, and the resulting dynamic power for checkpointing with a set of benchmark applications. To enable this, we implemented an in-house Python instruction-set simulator (ISS), in which the instruction set architecture (ISA) is identical to the RTL version. Dynamic power resulted by RF traffic is estimated using an existing cache/memory estimator Cacti 6.5 [19], based on the events recorded during ISS simulation.

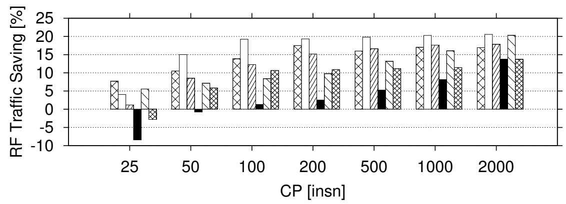
The second experiment aims at measuring and comparing the performance (pipeline critical path timing), area, power from gate-level netlist of the ReCoRD processor. Thus, we passed the processor RTL implemented in Section VI through



(a) RECORD vs. HP



(b) RECORD vs. FC



(c) RECORD vs. RELI

Fig. 8. RF traffic saving with checkpointing

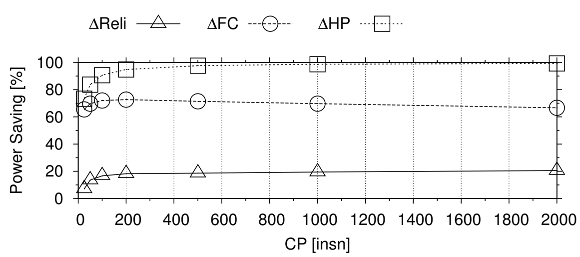


Fig. 9. Dynamic power reduction of checkpointing (ISS+CACTI)

logic synthesis provided by SYNOPSYS DESIGN COMPILER (DC) and obtained the gate-level netlist using the TSMC $65\,\mathrm{nm}$ technology node. Accurate measurement of dynamic power is stymied for two reasons . First, since the focus of this paper is front-end design, place-and-routing (P&R) is out of the scope of this paper and was not performed. Without P&R, the dynamic power is not accurate. Second, generating an accurate signal switching log (VCD/SAIF file) through gate-level HDL simulation is prohibitively slow. Therefore, we only report the static (leakage) power generated by SYNOPSYS DC and leave the dynamic power estimation to ISS+CACTI flow.

There are five types of processors implemented and tested: the baseline processor, HP's conventional checkpoint/recovery processor [10], full-copy (FC) checkpoint/recovery processor [7], the RELI processor [11], and the RECORD processor. The experiments consider six typical benchmark applications for embedded systems, five (`ADPCM`, `blowfish`, `CRC32`, `SHA`, `stringsearch`) from MIBENCH and one (`AES`) from CRYPTO.[5] The values for checkpoint period (CP) length

[5]http://www.cryptopp.com/benchmarks.html

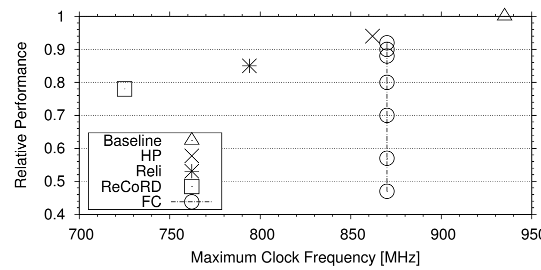


Fig. 10. Performance result

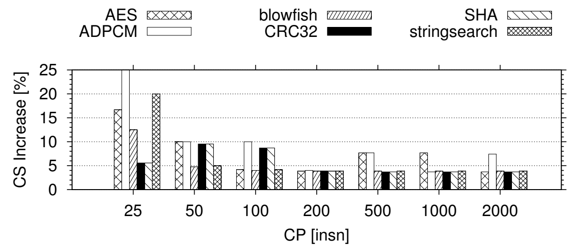


Fig. 11. CS increase of RECORD over RELI

are 25, 50, 100, 200, 1000 and 2000 instructions, which are also used by the study in [16].

B. RF Traffic Results

Fig. 8 shows the RF traffic saving results. We compared RECORD's RF traffic against HP (in Fig. 8a), FC (in Fig. 8b), and RELI (in Fig. 8c). RECORD showed improved RF traffic savings in comparison to HP checkpointing. The saving increases as CP increases, from $60\,\%$ in `AES` with CP $= 25$, to $99.6\,\%$ in `SHA` with CP $= 2000$. This huge saving is because HP naively executes checkpointing, while RECORD can take advantage of larger CP to reduce checkpointing execution. In comparison to FC checkpointing, RECORD also showed significant reduction of RF traffic. Different to the RF traffic saving from HP, the saving from FC decreases as CP increases. The largest saving from FC is $88.7\,\%$ in `SHA` with CP $= 25$, and the smallest saving is $42.7\,\%$ in `blowfish` with CP $= 2000$. RECORD has less RF traffic than RELI in most of the cases. The RF traffic saving generally increases as CP increases. The greatest saving is $21\,\%$ in `ADPCM` with CP $= 2000$. The special cases, where RECORD showed more RF traffic than RELI, are `CRC32` and `stringsearch` with CP $= 25$, and `CRC32` with CP $= 50$. This RF traffic increase is because `CRC32` has less register write-after-read occurrences. The average RF traffic reduction of RECORD in comparison to RELI is $1.2\,\%$ with CP $= 25$, $7.7\,\%$ with CP $= 50$, $11.0\,\%$ with CP $= 100$, $12.5\,\%$ with CP $= 200$, $13.7\,\%$ with CP $= 500$, $15.1\,\%$ with CP $= 1000$, and $17.0\,\%$ with CP $= 2000$.

Fig. 9 depicts the resultant average dynamic power saving caused by the RF traffic saving. The power numbers are generated by CACTI 6.5, where the worst-case CS size is used. For each CP, the number is averaged over six benchmark programs. These numbers generally reflect the results of traffic saving. While achieving significant power reduction when compared to HP and FC, RECORD reduces around $20\,\%$ dynamic power from that of RELI.

C. Performance Results

Fig. 10 presents the performance impact of RECORD in the processor due to checkpointing. We compared the performance

TABLE I. AREA AND LEAKAGE POWER ($f = 100\,\text{MHz}$, CS $= 32$)

Processor	A [μm^2]	A′ [μm^2]	ΔA′ [%]	P [μW]	P′ [μW]	ΔP′ [%]
Baseline	119 743	N/A	N/A	569	N/A	N/A
FC	137 345	123 498	3.1	649	590	3.7
HP	156 358	143 222	19.6	731	674	18.4
RELI	161 247	144 423	20.6	753	679	19.4
RECORD	167 656	146 872	22.6	778	690	21.3

(y-axis) and maximum clock frequency (x-axis) of the four checkpointing processors to the baseline processor (without checkpointing). Because the original multiplier and divider from ASIPmeister's library are single cycle and dominate the timing, it is not possible to observe the impact of checkpointing on pipeline's maximum clock frequency. Therefore, to measure performance overhead, we removed (using synthesis directives) the divider and multiplier from the processor during logic synthesis. The critical path timing (in ns) from synthesis and program runtime (in cycles) from ISS are used together to calculate the wall-clock time performance.

Due to the increased complexity in the pipeline organization, all four checkpointing processors had worse critical path timing than the baseline processor, which results in lower clock frequency. Among the four checkpointing processors, FC has the highest maximum frequency, 870 MHz, while RECORD has the lowest maximum frequency, 725 MHz, a bit lower than RELI's 793 MHz. For most of the processors, as the checkpointing is executed simultaneously with each instruction execution, the only factor, which determines the wall-clock time performance, is clock frequency. Hence, this frequency difference can be observed in relative performance numbers, where HP>RELI>RECORD. FC's performance is also determined by program cycle counts, because FC checkpoints all the registers together after fault check, which increases program cycles, depending on the checkpoint period length. Thus, FC has six performance numbers (six points) representing six CP length, 25, 50, 100, 200, 500, 1000 and 2000, from bottom to top. FC with CP $= 2000$ has the best performance among all FC processors with differing checkpoint period.

D. Hardware Results

For better clarity, we discuss hardware cost of checkpointing in two parts: CS and other logic circuits, "non-CS" (including history table and control). Fig. 11 presents the comparison of minimum CS size between RECORD and RELI. The other two processors are not discussed here because: (1) HP's CS size is much larger and grows fast over 32 entries from CP $= 50$, and, (2) FC's CS size is constantly 32 while CS size of both RELI and RECORD is never greater than 32. In comparison to RELI, RECORD generally requires larger CS, because RECORD has both source and destination registers checkpointed. However, as suggested in Fig. 3, increasing checkpoint period leads to more effective operand checkpointing (more write-after-read situations), which results in lower CS increase in larger CPs (around 5 % for 200 to 2000).

Table I shows the area and power results from logic synthesis, for understanding the cost of "non-CS" logic circuits. For simplicity, we synthesized all the processor targeting 100 MHz frequency, which is the commonly viable frequency with multiplier and divider in the pipeline. All the checkpointing processors have the same CS size (32). Column 1 gives the processor name. Column 2 and 3 are total area and the area without CS. Column 4 is the area overhead of "non-CS". Similarly, Column 5, 6 and 7 are the numbers for leakage power. When considering hardware cost from "non-CS", among all the checkpointing processors, FC is least costly in terms of both area and power, while the other three processors have around 20% overhead. RECORD has the largest overhead, 22.6 % in area and 21.3 % in leakage power. In comparison to RELI, RECORD only has 1.7 % more logic gates and 1.6 % more leakage power consumption.

VIII. CONCLUSION

In this paper, we have presented a runtime logging-based checkpointing technique, RECORD, which effectively reduces the register traffic for checkpointing, targeting embedded processors. RECORD leverages register data dependency at runtime to recognize and avoid redundant checkpointing executions. In our experiment, we tested RECORD in comparison to the conventional and state-of-the-art checkpointing techniques, through both ISS and netlist based evaluation flows. The experimental results have shown that RECORD is capable of minimizing the register traffic greatly, with minimal hardware and performance overheads.

REFERENCES

[1] E. N. M. Elnozahy *et al.*, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, Sep. 2002.

[2] A. Avizienis *et al.*, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, Jan 2004.

[3] H. Ando *et al.*, "A 1.3 ghz fifth generation sparc64 microprocessor," in *Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International*, Feb 2003, pp. 246–491 vol.1.

[4] J. Henkel and S. Parameswaran, *Designing Embedded Processors: A Low Power Perspective*, 1st ed. Springer Publishing Company, Incorporated, 2007.

[5] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 465–476.

[6] M. Pedram, *Power Aware Design Methodologies*, J. M. Rabaey, Ed. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[7] N. Bowen and D. Pradham, "Processor- and memory-based checkpoint and rollback recovery," *Computer*, vol. 26, no. 2, pp. 22–31, Feb 1993.

[8] R. Teodorescu, J. Nakano, and J. Torrellas, "Swich: A prototype for efficient cache-level checkpointing and rollback," *Micro, IEEE*, vol. 26, no. 5, pp. 28–40, Sept 2006.

[9] T. J. Slegel *et al.*, "Ibm's s/390 g5 microprocessor design," *IEEE Micro*, vol. 19, no. 2, pp. 12–23, Mar. 1999.

[10] E. Delano, "Checkpointing of register file," U.S. Patent 6 941 489, Sep. 6, 2005.

[11] T. Li, R. Ragel, and S. Parameswaran, "Reli: Hardware/software checkpoint and recovery scheme for embedded processors," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, March 2012, pp. 875–880.

[12] D. Hunt and P. Marinos, "A general purpose cache-aided rollback error recovery (CARER) technique," in *proceedings of the 17th international symposium on fault-tolerant coputing systems*, 1987, pp. 170–175.

[13] M. Prvulovic, Z. Zhang, and J. Torrellas, "Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, 2002, pp. 111–122.

[14] D. Sorin *et al.*, "Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, 2002, pp. 123–134.

[15] M. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, Dec 2001, pp. 3–14.

[16] N. Wang and S. Patel, "Restore: symptom based soft error detection in microprocessors," in *DSN'05*, June 2005, pp. 30–39.

[17] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, Jun. 1997.

[18] M. Itoh *et al.*, "Peas-iii: an asip design environment," in *Computer Design, 2000. Proceedings. 2000 International Conference on*, 2000, pp. 430–436.

[19] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," HP Laboratories, Tech. Rep. HPL-2009-85, 2009.