Rescheduling of Power Gating instructions for reduction of In-rush current

Sumanta Pyne
Department of Computer Science and Engineering
National Institute of Technology, Rourkela
Sundergarh, Odisha - 769008, India
Email: pynes@nitrkl.ac.in

***Abstract*—The present work introduces two compilation techniques for reduction of in-rush current in processors with power gating (*PG*) facility. These are done by rescheduling the *PG* instructions responsible in turning on multiple components from sleep to active mode at overlapped time intervals. The first method eliminates overlapped wake-up of the components resulting lesser in-rush current at the cost of performance degradation due to increase in program size. The second method allows overlapped wake-up as long as the resultant in-rush current is tolerable by the system with lesser increase in delay and program size. Algorithms are designed to automate these methods. The efficacy of the proposed methods are evaluated on MiBench and MediaBench benchmark programs. The original program with *PG* and their translated versions are executed on gem5 which simulates ARM Cortex-M4F processor enhanced with *PG*. McPAT is used to find the values of power consumed and in-rush current. The first and second methods reduce in-rush current by an average of 54% and 35%, respectively with corresponding average performance loss of 21% and 9%.**

I. Introduction

The emergence of deep submicron process technology with the decrease in dimensions of the transistor has increased the transistor count and speed of operation at the cost of greater device leakage currents. Power gating (*PG*) is a technique used to reduce power consumption of VLSI chips, by shutting off the blocks that are not in use, thus reducing stand-by or leakage power. Shutting down of the blocks can be initiated either by software or by hardware. Many modern processors are equipped with *PG* instructions to switch-off and switch-on different blocks to sleep and active modes, respectively. This allows the compiler and operating system to reduce runtime leakage power.

When a power gated block is switched on from sleep mode to active mode it draws a huge amount of in-rush current due to simultaneous charging of its internal capacitors. In-rush current is several times higher than the actual current required by the block to function in active mode. The flow of in-rush current may cause permanent damage to the circuit and also lead to higher power consumption. It can reduce the battery life for battery-operated systems due to rise in load current.

There exists several hardware schemes to reduce in-rush current. Some of them are usage of multiple power switches, daisy chaining, soft start with voltage regulators, discrete and integrated load switches. Most of them are not practical in *PG* design industry since they require more space and higher design cost.

The present work proposes two software based approaches for reduction of in-rush current due to overlapped wake-up caused by *PG* instructions that simultaneously activates multiple components. The first method eliminates overlapped wake-up, while the second one allows overlapped wake-up with guaranteed tolerable in-rush current. These techniques can enable a compiler to generate target code that will reduce in-rush current to tolerable levels.

The existing works on management of in-rush current in *PG* systems are discussed in Sec. II. The proposed compilation techniques are explained in Sec. III. Section IV covers explaination of the experimental setup with analysis of the results. Finally, Sec. V concludes the present work with future directions.

II. Related Works

The existing research and development works on reduction of in-rush current in systems with *PG* are based on hardware techniques at circuit level. Some of the techniques are discussed in the following paragraph.

In [1] the authors proposed *PG* structures in which sleep transistors are turned on in a non-uniform stepwise manner to reduce the magnitude of peak current and voltage glitches in the power distribution network requiring minimum time to stabilize power and ground. In this case, the rush current gradually increases as the number of switches is turned on. However, the rush current can be large unless the daisy chain is very slow. On the other hand, such a long daisy chain can cause long propagation delay and the slowly rising voltage can introduce other problems such as hot electron effects [2]. Another approach is to separate the power switches into two passes: a weak transistor pass and a strong transistor pass [4]. At wake-up, the weak transistors are turned on first so as to slowly turn on the rush currents. When the design is discharged (charged) to a voltage close to zero (V_{dd}), the strong transistor pass is turned on ready for normal operation. In [2] the authors introduced a timing and voltage drop analysis tool named CoolTime. It can guide the designer in setting power switch structure and sequence for controlling wake-up (rush current and wake-up time). An in-rush current limiter circuit [3] invented by Ball can sense the increased load current and produces sense current having a load current - sense current ratio of 1000:1, hence reducing the in-rush current. Kiong et al. introduced the in-rush current optimization power up flow analysis with PFET removal algorithm [5] to improve the in-rush current. In [6] the authors introduces two intermediate scheme to reduce wake-up time. During wake-up procedure their *PG* scheme implements a small transistor to control the sleep transistor (*ST*), has two stages: relaxation stage and completely turn-on stage. During the relaxation stage, the drain to source voltage (V_{ds}) of the *ST* reduces significantly while limiting the current exponentially as the V_{ds} of *ST* changes. During the complete turn-on stage, the small control transistor is turned off, and the *ST* acts like a current source. This two-stage transition method reduces the peak voltage fluctuations in the virtual ground and virtual power, and it also reduces the circuit wake-up time. They also introduced two circuit schemes with intermediate states to further reduce the ground bounce based on their proposed power gating circuit scheme. Meanwhile, the intermediate state saves more charges by charge recycling while allowing the virtual ground or virtual power floating between V_{dd} and ground. In [7] the authors proposed model memory access power gating (*MAPG*), a low-overhead technique to enable power gating of an active core when it stalls during a long memory access. They described a programmable two-stage power gating switch design that can vary a core's wake-up delay while maintaining voltage noise limits and leakage power savings with controlled in-rush current. A novel framework for generating a proper power-up sequence of the switches to control the in-rush current of a power-gated domain has been introduced in [8]. It also minimizes the power-up time and reduces the dynamic IR drop of the active domains. A detailed study of cause and effects of in-rush current with remedies to reduce it has been discussed in [9]. The authors explain in-rush current reduction techniques like soft-start with the help of voltage regulators to increase rise time. They explained that power switching with a controlled rise time can be accomplished by using discrete circuitry. Effective usage of integrated load switches in place of the discrete solution was also highlighted. In [10] Kim et al. discussed the reduction of in-rush current by turning on each switch cell at different times. They showed that in-rush current can be reduced even more if signal transition time to switch each cell is adjusted.

The existing software methodologies to reduce leakage current on systems with *PG* are mainly based on compilation techniques [11], [12], [13], [15], [14], [16] which deals in scheduling of *PG* instructions. Task scheduling with *PG* is proposed in [17]. Reduction of in-rush current has not been addressed in these works.

The current status of research carried in this field shows that there is a scope to perform investigation on software techniques for in-rush current management in systems with *PG*. Existing hardware techniques add extra circuitry for in-rush current management increasing design cost, design space and average power consumption. The software techniques can address these drawbacks.

III. Present Work

An arrangement for instruction controlled *PG* is shown in Fig. 1. It has n *PG* components $C_0, C_1, \cdots, C_{n-1}$. *PG* is done with the help of the header p-MOS transistors having higher threshold voltage (V_T). The header switches are controlled by an n-bit power gating control register (PGCR) placed in the power gating controller (PGC). The bits $0, 1, \cdots, n-1$ are the *PG* bits of $C_0, C_1, \cdots, C_{n-1}$, respectively. If any of these bits $\alpha \in \{0, 1 \cdots, n-1\}$ is '0', then the component C_α is in active mode, otherwise C_α is in sleep mode. Let there be two *PG* instructions *switch_off* and *switch_on* each consuming three clock cycles - one cycle in each of instruction fetch (*IF*), instruction decode (*ID*) and execution (*EX*) stages of the instruction pipeline. To put C_α in sleep (or power gated) mode the instruction *switch_off*(C_α) is used to set the value of α^{th} bit of PGCR. C_α in sleep mode can be put to active mode with the help of the instruction *switch_on*(C_α) which resets the value of α^{th} bit of PGCR. Hence, a program can use this *PG* facility.

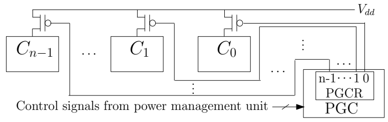
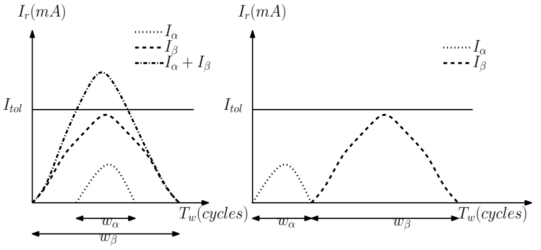


Fig. 1. An arrangement for instruction controlled *PG* system



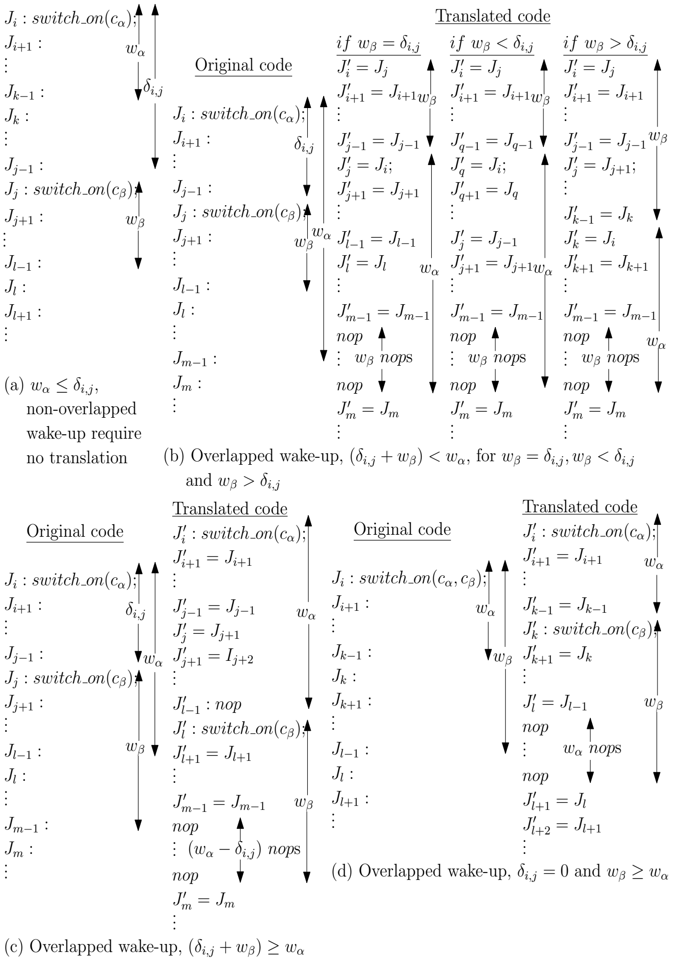
(a) Overlapped wake-up (b) Non-overlapped wake-up

Fig. 2. In-rush current for overlapped and non-overlapped wake-up

When C_α in sleep mode is switched on using *switch_on*(C_α) it draws in-rush current I_α for a period of w_α cycles where w_α is the wake-up time (T_w) of C_α. It is considered that $I_\alpha \leq I_{tol}$ for wake-up of any individual *PG* component C_α, where I_{tol} is the maximum tolerable in-rush current for a given system. The problem of intolerable in-rush current may arise during wake-up of multiple components during an overlapped time interval. Fig. 2(a) shows in-rush current (I_r) in milliampere (mA) for overlapped wake-up of two components C_α and C_β where $\beta \in \{0, 1, \cdots, n-1\}$ and $\beta \neq \alpha$. w_β and I_β are the wake-up time and in-rush current of C_β, respectively. The resultant in-rush current is $I_\alpha + I_\beta$. Simultaneous overlapped wake-up of several components can lead to higher flow of in-rush current resulting higher peak power dissipation and reduction of chip reliability. Hence, it is better to have non-overlapped wake-up as shown in Fig. 2(b).

A. PG with Non-overlapped Wake-up (PGNW)

A program with *switch_on* instructions may cause overlapped wake-up of *PG* components. Considering an assembly language program with M instructions where i^{th} instruction J_i is *switch_on*(C_α) and j^{th} instruction J_j is *switch_on*(C_β),



(a) $w_\alpha \leq \delta_{i,j}$, non-overlapped wake-up require no translation

(b) Overlapped wake-up, $(\delta_{i,j} + w_\beta) < w_\alpha$, for $w_\beta = \delta_{i,j}, w_\beta < \delta_{i,j}$ and $w_\beta > \delta_{i,j}$

(c) Overlapped wake-up, $(\delta_{i,j} + w_\beta) \geq w_\alpha$

(d) Overlapped wake-up, $\delta_{i,j} = 0$ and $w_\beta \geq w_\alpha$

Fig. 3. Elimination of overlapped wake-up using *PGNW*

where $i, j \in \{0, 1, \cdots, M-1\}$ and $i \leq j$. Let it take $\delta_{i,j}$ cycles from J_i to reach J_j. If $i = j$, then $\delta_{i,j} = 0$. For $i < j$, $\delta_{i,j} = \sum_{p=i}^{j-1} cycles(I_p, I_{p+1})$, where $cycles(I_p, I_{p+1})$ is the time gap (in cycles) between the entry of J_p and J_{p+1} in the *EX* stage. J_i and J_j will result non-overlapped wake-up of C_α and C_β if $i < j$ and $w_\alpha \leq \delta_{i,j}$. Hence, no translation of the program is required as shown in Fig. 3(a). Overlapped wake-up occurs if $w_\alpha > \delta_{i,j}$. Rescheduling of *switch_on* instructions for elimination of overlapped wake-up are shown in figures 3(b), 3(c) and 3(d) where $0 \leq i \leq q \leq j \leq k \leq l \leq m < M$. No-operation (*nop*) instructions are inserted in the translated code to ensure the usage of a component after completion of its wake-up. An *nop* is assumed to consume one byte of memory space. It takes three clock cycles - one cycle in each of *IF*, *ID* and *EX* stages. An *nop* being an immediate successor of a *switch_on* acts as a delay of one clock cycle after the execution of *switch_on*. This adds time and space overheads of $O(w_\alpha)$ if $(\delta_{i,j} + w_\beta) < w_\alpha$. The time and space overheads are $O(w_\beta)$ for $(\delta_{i,j} + w_\beta) \geq w_\alpha$ as well as in case of $\delta_{i,j} = 0$ and $w_\beta \geq w_\alpha$. The space overhead can be reduced to $O(1)$ by replacing *nop*s with a loop having empty body or a single *nop* depending on the values of w_α and w_β.

The proposed Algorithm 1 (*PGNW* Algorithm) produces the code for *PG* with non-ovelapped wake-up. It takes an assembly language level code fragment of the source program having $M'(M' \leq M)$ instructions with overlapped wake-up of $n'(n' \leq n)$ components. As output it produces an equivalent *PGNW* code fragment with rescheduled *switch_on* instructions leading to non-overlapped wake-up for n' components. The time taken by M' instructions of input code fragment is T clock cycles. The input and output code fragments always start with a *switch_on* instruction.

Algorithm 1 *PG* with non-overlapped wake-up (*PGNW*)

```
Input: A code fragment of $M'$ instructions with overlapped wake-up of $n'$
       components, where $M' \leq M$ and $n' \leq n$.
Output: A code fragment with rescheduled switch_on instructions leading to
       non-overlapped wake-up of $n$ components.
Initialization: (i) Store the switch_on instructions in S in non-decreasing order of
                earliest finishing time of the wake-up.
                (ii) Consider the input code fragment as PGNW code.
1:  $\alpha \leftarrow S[1].component$
2:  At cycle 1, replace the switch_on instruction with switch_on($C_\alpha$).
3:  $t \leftarrow w_\alpha + 1$
4:  for ($u \leftarrow 1; u \leq n'; u \leftarrow u + 1$) do
5:      $i \leftarrow S[u].instruction$
6:      $j \leftarrow S[u-1].instruction$
7:      $\alpha \leftarrow S[u].component$
8:      $\beta \leftarrow S[u-1].component$
9:      $\tau \leftarrow t - w_\beta + \delta_{i,j}$
10:     if $\delta_{i,j} + w_\beta < w_\alpha$ then
11:         At cycle $\tau$, remove switch_on($C_\beta$).
12:         Move all instructions starting from cycle $\tau + 1$ backward by one step.
13:         Move all instructions starting from cycle $t$ forward by one step.
14:         At cycle $t$, insert switch_on($C_\alpha$).
15:         Move all instructions starting from cycle $t + w_\alpha$ forward by $w_\beta$ steps.
16:         Insert $w_\beta$ nops from cycle $t + w_\alpha - w_\beta$ to cycle $t + w_\alpha - 1$.
17:     else if $\delta_{i,j} + w_\alpha \geq w_\beta$ then
18:         At cycle $\tau$, remove switch_on($C_\alpha$).
19:         Move all instructions starting from cycle $\tau + 1$ backward by one step.
20:         if $\delta_{i,j} > 0$ then
21:             At cycle $t + w_\beta - 1$, insert an nop.
22:         end if
23:         Move all instructions starting from cycle $t$ forward by one step.
24:         At cycle $t$, insert switch_on($C_\alpha$).
25:         Move instructions starting from cycle $t + w_\alpha$ forward by $w_\beta - \delta_{i,j}$ steps.
26:         Insert $w_\beta - \delta_{i,j}$ nops from cycle $t + w_\alpha - (w_\beta - \delta_{i,j})$ to cycle $t + w_\alpha - 1$.
27:     end if
28:     $t \leftarrow t + w_\alpha$
29: end for
```

PGNW Algorithm considers an array data structure *S* having n' elements representing *switch_on* instructions for n' components. $S[u]$ is the u^{th} *switch_on* instruction, where $u \in \{1, 2, \cdots, n'\}$. *S* stores the *switch_on* instructions belonging to the given code fragment in non-decreasing order of earliest finishing time of the wake-up of components involved with *switch_on* instructions. $S[u].instruction \in \{0, 1, \cdots, M'\}$ is the instruction number of *switch_on* instruction assigned to $S[u]$. The value representing the component turned on by $S[u]$ is denoted by $S[u].component \in \{0, 1, \cdots, n-1\}$. The algorithm begins by considering the given input code fragment *PGNW* code.

The input and output code fragments are considered to start at clock cycle $t = 1$. Here, t and τ denotes number of clock cycles starting from cycle 1. In step 2 the first instruction of the input code fragment which runs during cycle 1 is replaced by *switch_on*(C_α). Steps 11 and 18 are involved in removal of *switch_on* instructions which run during cycle τ. This allow steps 12 and 18 to move all instructions starting from cycle $\tau + 1$ b bytes backward to lower memory addresses of *PGNW* code, where b is the size of the instructions *switch_on*(C_α) and *switch_on*(C_β). Hence, in *PGNW* code these instructions will begin at cycle τ. Steps 13 and 23 moves all the instructions starting at cycle t forward to higher memory address by b bytes, delaying them by one cycle. This enable steps 14 and 24 to schedule *switch_on*(C_α) at cycle t. Steps 15 and 25 move all instructions staring at cycle t forward by w_β and $w_\beta - \delta_{i,j}$ bytes, respectively. Thus delaying them by same number of cycles. This allows steps 16 and 26 to insert w_β and $w_\beta - \delta_{i,j}$ bytes, respectively.

Algorithm 1 considers all cases for overlapped wake-up as shown in Fig. 3. At the beginning of each iteration of the **for** loop of steps 4-29 indexed by *u*, the subarray consisting of elements $S[1], S[2], \cdots, S[u-1]$ constitutes the currently scheduled *switch_on* instructions that produces non-overlapped wake-up, the element $S[u]$ constitute the *switch_on* instruction to be scheduled in the current iteration *u* for non-overlapped wake-up with previous $u-1$ *switch_on*s, and the remaining elements of subarray $S[u+1], S[u+2], \cdots, S[n']$ constitutes the *switch_on* instructions to be scheduled in next $n' - u$ iterations for non-overlapped wake-up. This property forms the loop invariant which is preserved during initialization (in steps 1-3), maintenance (in steps 4-29, for $u \in \{2, 3, \cdots, n'\}$) and termination (when $u = n' + 1$) phases to ensure the correctness of the algorithm.

In each iteration the movement of $O(M')$ instructions in $O(M' \times w_\beta)$ time results insertion of w_β *nop*s in the *PGNW* code. w_β is $O(w_{max})$, where $w_{max} = max(w_0, w_1, \cdots, w_{n-1})$. Hence, it takes $O(n' \times M' \times w_{max})$ time to generate a *PGNW* code with $O(n' \times w_{max})$ *nop*s.

The proposed *PGNW* method is strict in elimination of overlapped wake-up. It is suitable for systems where reliability has higher priority than delay. It may not be suitable for safety-critical and real-time systems where apart from reliability lower delay is crucial. Sec. III-B introduces a method to deal with these issues.

B. PG with Tolerable In-rush current (PGTI)

PGNW guarantees tolerable in-rush current at the cost of increase in delay and program size. These overheads can be reduced by allowing overlapped wakepus within the limitation maximum tolerable in-rush current. For each *PG* component C_α an in-rush current table (IT_α) is maintained. The tuple $t \in \{1, 2, \cdots, w_\alpha\}$ of IT_α denoted by $IT_\alpha[t]$ stores the value of I_α during t^{th} cycle of wake-up of C_α. I_α is minimum during cycles 1 and w_α. I_α is maximum or at peak during cycle $\frac{w_\alpha}{2}$.

Algorithm 2 *PG* with tolerable in-rush current (*PGTI*)

```
Input: (i) A code fragment of $M'$ instructions with overlapped wake-up of $n'$
           components, where $M' \leq M$ and $n' \leq n$.
       (ii) In-rush current tables (ITs) for each $n'$ components.
       (iii) Maximum tolerable in-rush current ($I_{tol}$).
Output: A PGTI code with rescheduled switch_on instructions leading to
        overlapped wake-up of $n$ components with tolerable in-rush current.
Initialization: (i) $I_{tot}[t] \leftarrow 0\ \forall t | t \in \{1, 2, \cdots, T\}$.
                (ii) Store the switch_on instructions in S using following rules:
                     (a) in order of their occurrences in the code fragment.
                     (b) if a switch_on is having more than one component then store
                         switch_on for each component in non-decreasing order of
                         wake-up time.
                (iii) Consider the input code fragment as PGTI code.
1:  $\alpha \leftarrow S[1].component$
2:  for ($t \leftarrow S[1].start; t \leq S[1].end; t \leftarrow t+1$) do
3:      $I_{tot}[t] \leftarrow IT_\alpha[t - S[1].start + 1]$
4:  end for
5:  for ($u \leftarrow 2; u \leq n'; u \leftarrow u+1$) do
6:      $\alpha \leftarrow S[u].component$
7:      $\beta \leftarrow S[u-1].component$
8:      $\Delta t \leftarrow 0$
9:      $t \leftarrow S[u].start$
10:     while ($t \leq S[u].end$) do
11:         if ($I_{tot}[t + \Delta t] + IT_\alpha[t - S[u].start + 1]) > I_{tol}$ then
12:             if $S[u].start = S[u-1].start$ then
13:                 Move all successors of switch_on($C_\beta$) forward by one step.
14:                 Insert switch_on($C_\alpha$) next to switch_on($C_\beta$).
15:             else
16:                 Move switch_on($C_\alpha$) and all its successors forward by one step.
17:                 Insert an nop as immediate predecessor of switch_on($C_\alpha$).
18:             end if
19:             $\Delta t \leftarrow \Delta t + 1$
20:             goto Step 9.
21:         end if
22:     end while
23:     for ($t \leftarrow S[u].start; t \leq S[u].end; t \leftarrow t+1$) do
24:         $I_{tot}[t + \Delta t] \leftarrow I_{tot}[t + \Delta t] + IT_\alpha[t - S[u].start + 1]$
25:     end for
26: end for
```

The proposed Algorithm 2 (*PGTI* Algorithm) produces the code for *PG* with tolerable in-rush current. It takes an assembly language level code fragment of the source program having $M'(M' \leq M)$ instructions with overlapped wake-up of $n'(n' \leq n)$ components, in-rush current tables (*IT*s) for each n' components and maximum tolerable in-rush current (I_{tol}) as inputs. As output it generates an equivalent *PGTI* code fragment with rescheduled *switch_on* instructions that guarantees atmost I_{tol} amount of in-rush current due to overlapped wake-up of n' components. The input and output code fragments always start with a *switch_on* instruction. *PGTI* Algorithm considers two array data structures I_{tot} and *S*. I_{tot} is an array of *T* elements, where *T* is the total number of clock cycles required by M' instructions belonging to the code fragment given as input. $I_{tot}[t] \in \mathbb{R}_{\geq 0}$ is the total in-rush current due to overlapped wake-up during cycle $t \in \{1, 2, \cdots, T\}$. Initially, all the elements of I_{tot} are assigned with zero. The array *S* has n' elements representing *switch_on* instructions for n' components. $S[u]$ is the u^{th} *switch_on* instruction, where $u \in \{1, 2, \cdots, n'\}$. *S* stores the *switch_on* instructions in order of their occurrences in the given code fragment. In case of a *switch_on* instruction is having more than one component, the *switch_on* instruction for each component are stored in non-decreasing order of wake-up time. $S[u].start$ and $S[u].end$ are the respective staring and finishing times (or cycles) of wake-up due to $S[u]$, where $S[u].start, S[u].end \in \{1, 2, \cdots, T\}$ and $S[u].start \leq S[u].end$. The value representing the component turned on by $S[u]$ is denoted by $S[u].component \in \{0, 1, \cdots, n-1\}$. The algorithm begins by considering the given input code fragment as the *PGTI* code.

Steps 1-4 assigns the in-rush current values during the wake-up of $S[1]$ to the array I_{tot}. In each iteration of the **for** loop covering steps 5-26 reschedules $S[u]$ for tolerable in-rush current. The inner **while** loop comprising of steps 10-22 check the possiblity of intolerable in-rush current caused by wake-up of $S[u].component$ overlapped with the wake-up of $S[1].component, S[2].component, \cdots S[u-1].component$ considered in previous $u-1$ iterations. For a particular cycle

$t \in \{S[u].start, S[u].start+1, \cdots, S[u].end\}$, if the total in-rush current is found to be greater than I_{tol}, then $S[u]$ is delayed by one cycle using an *nop* and the delay counter Δt is increased by 1. Another trial for checking of tolerable in-rush current for $S[u]$ is done. This goes on until $S[u]$ is rescheduled to guarantee tolerable in-rush current. On finding a tolerable in-rush current the steps 23-24 adds in-rush current produced by rescheduled $S[u]$ to the total in-rush in I_{tot}.

The steps 13 moves all successive instructions of *switch_on*(C_β) forward to higher addresses by b bytes in memory of *PGTI* code. Step 14 inserts a b byte instruction *switch_on*(C_α) as immediate succesor of *switch_on*(C_β). Similarly, step 16 moves *switch_on*(C_β) and all its successive instructions forward to higher addresses by 1 byte in memory of *PGTI* code. This creates room for an *nop* inserted as immediate predecessor of *switch_on*(C_α) in step 17.

At the beginning of each iteration of the **for** loop of steps 5-26 indexed by *u*, the subarray consisting of elements $S[1], S[2], \cdots, S[u-1]$ constitutes the currently scheduled *switch_on* instructions that produces tolerable in-rush current due to overlapped wake-up, the element $S[u]$ constitute the *switch_on* instruction to be scheduled in the current iteration *u* for tolerable in-rush current due to overlapping with previous $u-1$ wake-up, and the remaining elements of subarray $S[u+1], S[u+2], \cdots, S[n']$ constitutes the *switch_on* instructions to be scheduled in next $n'-u$ iterations for tolerable in-rush current. This property forms the loop invariant which is preserved during initialization (in steps 1-4), maintenance (in steps 5-26, for $u \in \{2, 3, \cdots, n'\}$) and termination (when $u = n'+1$) phases to ensure the correctness of the algorithm.

In each iteration the inner **while** loop of steps 10-22 take $O(M')$ time due to movement of $O(M')$ instructions to insert an *nop* in *PGTI* code. In each iteration of the outer **for** loop, the inner **while** takes $O(T \times M')$ time inserting $O(T)$ *nop*s and the inner **for** loop of steps 23-25 take $O(T)$ time. Hence, it takes $O(n' \times T \times M')$ time to generate a *PGTI* code with $O(n' \times T)$ *nop*s.

IV. Experiment and Results

To establish the efficacy of the proposed approach, simulations are carried out on **gem5** [18] architecture simulator. McPAT [20] is used for obtaining power values. **gem5** is configured with the instruction set and functional units (*FU*s) of the ARM Cortex-M4F processor [19]. The processor has seven *FU*s. Integer ALU (*ialu*) is not power gated because it is used in majority of the instructions. The bits 0, 1, 2, 3, 4 and 5 of PGCR are the *PG* bits of Floating Point Divider (*fpdiv*), Floating Point Multiplier (*fpmul*), Floating Point Adder (*fpadd*), Integer Divider (*idiv*), Integer Multiplier (*imul*), and Barrel Shifter (*bshf*), respectively as shown in Fig. 4. The size of the instruction cache is considered to be 32 KB.

McPAT is configured with the power model of ARM Cortex-M4F based on 32nm process technology, where the leakage power dissipation is almost 70% of the total power consumption. Here, the processor clock frequency $f_{clk} = 1.0$ GHz, and power supply voltage $V_{dd} = 0.9$ V. V_T of processor's n-MOS and p-MOS transistors are $V_{tn} = 0.18$ V and $V_{tp} = -0.18$ V, respectively. V_T of p-MOS transistors which act as header switches are -0.45 V. $I_{tol} = 200$ mA. Table I show the values of load capacitance ($c_l^{(\alpha)}$), maximum operating current ($I_{op}^{(\alpha)}$), w_α and peak I_α (I_α^{pk}) for each C_α belonging to ARM Cortex-M4F with *PG*.

TABLE I
Values of $c_l^{(\alpha)}, I_{op}^{(\alpha)}, w_\alpha$ and I_α^{pk}

C_α	*fpdiv*	*fpmul*	*fpadd*	*idiv*	*imul*	*bshf*
$c_l^{(\alpha)}$(in nF)	6.58	5.9	3.89	2.24	1.81	0.8
$I_{op}^{(\alpha)}$(in mA)	17.24	15.47	12.84	9.63	8.12	4.72
w_α(in $cycles$)	32	30	24	18	16	10
I_α^{pk}(in mA)	185	177	146	112	102	72

The high level *PG* instructions *switch_off* and *switch_on* are designed to support *PG* in high level languages. An assembly language level instruction ***pg pgcr_bit_vector*** has been added to the instruction set, where *pgcr_bit_vector* is a 32-bit vector representing the *PG* bits of PGCR. Its size is five bytes. It sets/resets corresponding *PG* bits of PGCR to switch OFF/ON the *FU*s with *PG* in three clock cycles - one cycle in each of *IF*, *ID* and *EX* stages. The GCC compiler for ARM Cortex-M4F [21] is extended to replace high level *switch_on* and *switch_off* instructions with equivalent ***pg pgcr_bit_vector*** instruction. The features leading to generation of basic *PG* (using [12], [14]), *PGNW* and *PGTI* codes are also added to the GCC compiler for ARM Cortex-M4F.

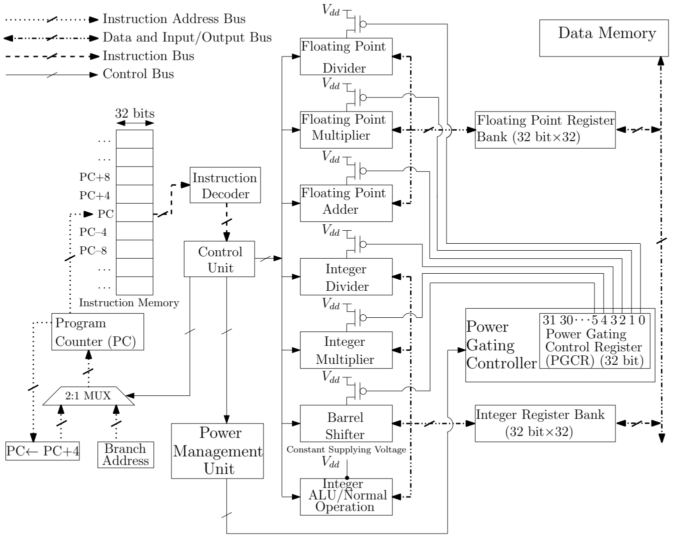
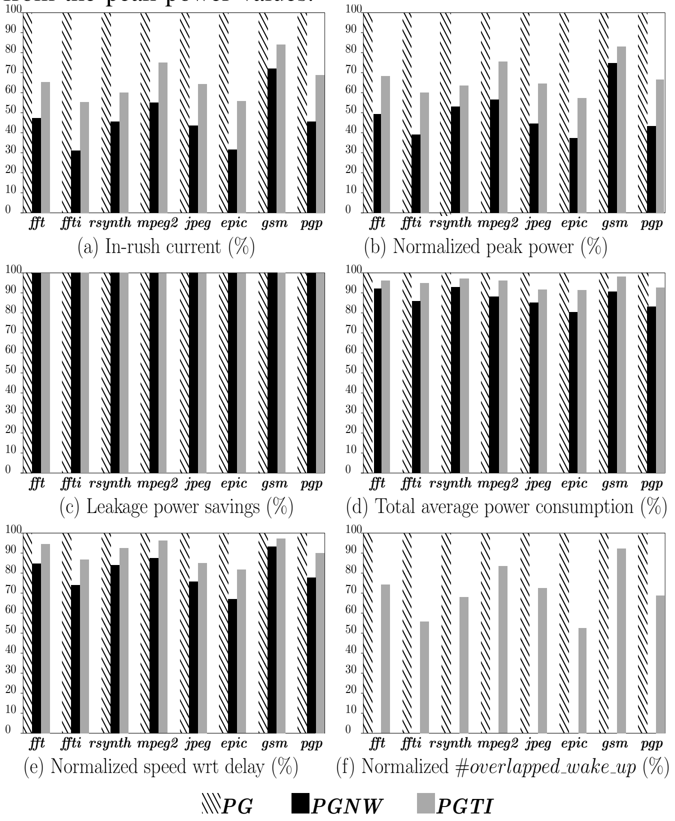


Fig. 4. A machine architecture model with *PG* control

TABLE II
Benchmark description

Program	*fft*	*ffti*	*rsynth*	*mpeg2*	*jpeg*	*epic*	*gsm*	*pgp*
Bench	MiBench	MiBench	MiBench	Media	Media	Media	Media	Media
Category	Telecomm	Telecomm	Office	Video	Image	Image	Speech	Crypto
#cfow	15	8	15	18	14	7	23	9

The proposed techniques are tested on MiBench [22] and MediaBench [23] benchmark programs as shown in Table II, where #cfow is the number of code fragments with overlapped wake-up. The benchmark programs are compiled using updated GCC compiler. The generated target code is executed on gem5 behaving as ARM Cortex-M4F processor. The performance values are generated by gem5. These performance values along with process technology and power related parameters of ARM Cortex-M4F act as input to McPAT in a prescribed XML file format. McPAT produces the power trace with the help of information in the XML file. The values of peak, average, dynamic and leakage power are produced by McPAT. The values of overlapped in-rush current are obtained from the peak power values.



(a) In-rush current (%)

(b) Normalized peak power (%)

(c) Leakage power savings (%)

(d) Total average power consumption (%)

(e) Normalized speed wrt delay (%)

(f) Normalized $\#overlapped\_wake\_up$ (%)

Fig. 5. Comparison of experimental results

The experimental results are shown in Fig. 5. *PGNW* and *PGTI* are compared with respect to the normalized values of power and performance related to basic *PG* program. Leakage power savings achieved by *PGNW* and *PGTI* are similar to that of *PG*. Peak power dissipation $\propto$ in-rush current $\propto$ number of overlapped wake-up ($\#overlapped\_wake\_up$). For *PGNW* $\#overlapped\_wake\_up = 0$ and for *PGTI* it is lesser than that of *PG*. Hence, peak power and in-rush current for *PGTI* are lesser than *PG* but higher than *PGNW*. Reduction of in-rush current and peak power dissipation experienced by (i) *PGNW* lies within 28-68% and 25-65%, respectively, and (ii) *PGTI* lies within 16-47% and 18-45%, respectively. This leads to reduction in total average power consumption for *PGNW* and *PGTI* which lies within 7-21% and 2-9%, respectively. The addition of *nop*s in *PGNW* and *PGTI* codes increase execution time. The loss in performance experienced by *PGNW* and *PGTI* lie within 5-35% and 2-18%, respectively.

V. Conclusion

The present work introduces two compilation techniques for reduction of in-rush current in *PG* sytems. The proposed method *PGNW* reduces in-rush current by eliminating overlapped wake-up at the cost increased delay and program size. To address these issues *PGTI* has been introduced. *PGTI* allows ovelapped wake-up within the limitations of tolerable in-rush current. These methods are evaluated on standard benchmark programs. *PGNW* achieves higher reduction in in-rush current. *PGTI* is better in terms delay and increase in code size. The future work will investigate to reduce time and space overheads of *PGNW* and *PGTI* codes. Thus making them fit for real-time and safety-critical embedded systems.

References

[1] S. Kim, S. V. Kosonocky and D. R. Knebel, "Understanding and minimizing ground bounce during mode transition of power gating structures," *Proc. of Int. Symp. on Low Power Electronics and Design (ISLPED '03)*, 27-27 Aug. 2003.

[2] K. Choi and J. Frenkil, "An Analysis Methodology for Dynamic Power Gating," Sequence Design Inc. https://pdfs.semanticscholar.org/09d3/db89fffc3dd250c6584b54a1925bafc1b27d.pdf

[3] A. Ball, "Integrated in-rush current limiter circuit and method," *US patent US20040090726 A1*, May 13, 2004.

[4] P. Royannez, H. Mair, F. Dahan and U. Ko, "90nm Low Leakage SoC Design Techniques for Wireless Applications", *Proc. of IEEE Int. Solid-State Circuits Conf.*, Feb. 2005, pp. 138 - 139.

[5] T. S. Kiong and U. C. Kong, "Power Gate Optimization Method for In-Rush Current and Power Up Time," *Intel Corporation*, https://dac.com/sites/default/files/App_Content/files/48/48_07U_1.pdf

[6] K. He, R. Luo and Y. Wang, "A power gating scheme for ground bounce reduction during mode transition," *Proc. of 25th Int. Conf. on Computer Design, ICCD 2007*, 7-10 Oct. 2007, Lake Tahoe, USA, pp. 388-394.

[7] K. Jeong, A. B. Kahng, S. Kang, T. S. Rosing and R. D. Strong, "MAPG: Memory access power gating," *Proc. of Design, Automation, Test & Exhibition in Europe Conf., DATE 2012*, Dresden, Germany, Mar. 12-16, 2012. pp. 1054-1059.

[8] S. H. Chen, Y. L. Lin and M. C. T. Chao, "Power-Up Sequence Control for MTCMOS Designs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 21, No. 3, pp. 413-423 , Mar. 2013.

[9] A. Kaknevicius and A. Hoover, "Managing Inrush Current," *Application Report SLVA670A, Texas Instruments*, May 2015, www.ti.com/lit/an/slva670a/slva670a.pdf

[10] S. Kim, S. Paik, S. Kang and Y. Shin, "Wake-up scheduling and its buffered tree synthesis for power gating circuits," *Integration, the VLSI journal, Elsevier*, Vol. 53, pp. 157-170, 2016.

[11] Y. P. You, C. Lee and J. K. Lee, "Compiler Analysis and Supports for Leakage Power Reduction on Microprocessors," *Proc. of 15th Int. Conf. on Languages and Compilers for Parallel Computing (LCPC) 2002, Springer LNCS*, Vol. 2481, pp. 45-60, 2005.

[12] Y. P. You, C. Lee and J. K. Lee, "Compilers for Leakage Power Reduction," *ACM Transactions on Design automation of Electronic Systems (TODAES)*, Vol. 11, No. 1, pp. 147-164, 2006.

[13] S. Roy, S. Katkoori and N. Ranganathan, "A compiler-based leakage reduction technique by power-gating functional units in embedded microprocessors," *Proc. of 20th Int. Conf. on VLSI Design and 6th Int. Conf. on Embedded Systems*, pp. 215-220, 2007.

[14] S. Roy, S. Katkoori and N. Ranganathan, "A Framework for Power-Gating Functional Units in Embedded Microprocessors," *IEEE TVLSI*, Vol. 17, No. 11, pp. 1640-1649, 2009.

[15] Y. P. You, C. Lee and J. K. Lee, "Compilation for Compact Power-Gating Controls," *ACM TODAES*, Vol. 12, No. 4, Article No. 51, 2007.

[16] D. Park, J. Lee, N. S. Kim and T. Kim. "Optimal algorithm for profile-based power gating: a compiler technique for reducing leakage on execution units in microprocessors," *Proc. of Int. Conf. on Computer-Aided Design (ICCAD 2010)*, pp. 361-364, 2010.

[17] Y. Wang, J. Xu, Y. Xu, W. Liu and H. Yang, "Power Gating Aware Task Scheduling in MPSoC," *IEEE TVLSI*, Vol. 19, No. 10, pp. 1801-1812, October 2011.

[18] http://gem5.org/Main_Page

[19] http://infocenter.arm.com

[20] http://www.hpl.hp.com/research/mcpat/

[21] https://developer.arm.com/open-source/gnu-toolchain/gnu-rm

[22] http://vhosts.eecs.umich.edu/mibench//

[23] http://mathstat.slu.edu/˜fritts/mediabench/