Hierarchical Dynamic Power-Gating in FPGAs

Rehan Ahmed[1,2](✉), Steven J.E. Wilton[1,2], Peter Hallschmid[1,2], and Richard Klukas[1,2]

[1] The University of British Columbia, Vancouver, Canada
[2] Blackcomb Design Automation, Vancouver, Canada
{rehan.ahmed,richard.klukas}@ubc.ca, stevew@ece.ubc.ca, peter.hallschmid@blackcomb-da.com

Abstract. Dynamic power-gating has been shown to reduce FPGA static leakage power significantly. In this paper, we propose a high-level synthesis (HLS) compiler-assisted framework that automatically detects the hierarchical power-gating opportunities, and turns off accelerators when they are not required. Unlike previous work which considers turning off entire accelerators when they are not required, our technique is more fine-grained, in that it allows turning off a portion of an accelerator when other parts of an accelerator are running. Results on CHStone benchmarks show that hierarchical power-gating can save up to 31 % of static energy when the parent and descendant accelerators are power-gated independently. An additional savings of up to 25 % can be achieved if the parent accelerator is power-gated while the sub-accelerator runs.

1 Introduction

As Field-Programmable Gate Arrays (FPGAs) are migrated to advanced processing nodes, power has become a first-class concern for many applications. Compared to an Application-Specific Integrated Circuit (ASIC), an FPGA implementation typically dissipates 14x more power [7]. One of the most effective techniques to reduce the power dissipation in an FPGA is to turn off (power-gate) part of the design when it is idle. Recent work has presented techniques for dynamic power-gating (DPG) [3], in which parts of the FPGA can be turned off at run time, under control of an on-state power controller.

Dynamic power-gating, however, presents a formidable challenge to a designer. In ASIC design, in which power-gating is common, power-gating opportunities are identified manually and typically described in a standard format, such as Unified Power Format (UPF). For FPGAs, many designers may not be willing or able to invest the effort into this manual identification, reducing the effectiveness of overall power-gating. Although techniques have been proposed for identifying some power-gating opportunities from a netlist or dataflow graph, these techniques typically focus on power-gating opportunities which are as short as several cycles [2,6,8], limiting the effectiveness of power-gating. Automatically

K. Sano et al. (Eds.): ARC 2015, LNCS 9040, pp. 27–38, 2015.
DOI: 10.1007/978-3-319-16214-0_3

identifying significant power-gating opportunities from a netlist or RTL design remains a difficult challenge.

Identifying power-gating opportunities can be much easier, however, if the design is created using a higher-level design tool. In particular, high-level synthesis (HLS) methodologies are increasingly being used to raise the abstraction level of a design and improve designer productivity. In such a methodology, a designer specifies a design using a language such as C, and the tool generates a circuit. Importantly, the overall architecture of the resulting digital system is determined by the high-level synthesis compiler. The compiler not only knows about the structure of the circuit, but also its temporal behaviour which can be used to identify power-gating opportunities automatically.

In [1], a methodology is presented in which the schedule from a high-level synthesis tool is used to determine the idle periods of individual hardware accelerators in a synthesized system. The predicted length of these idle periods is used to determine whether the power saved by power-gating the accelerator is more than the overhead of turning the accelerator off and then on again at the end of the idle period. Based on this knowledge, individual accelerators can then be power-gated when it is deemed profitable. In this previous work, however, the granularity of power-gating is fixed at the accelerator level. An entire accelerator is turned off or on as a unit. For very large accelerators, it may be profitable to power gate at a finer granularity. If an accelerator has many phases, each of which is implemented by a separate logic circuit (which we call a sub-accelerator), then it may be desirable to turn off individual sub-accelerators when parent or other sub-accelerators are running.

In this paper, we present a HLS compiler-assisted framework that automatically detects the hierarchical power-gating opportunities from an application expressed in C language. We present a study in which we vary the granularity of power-gating and quantify the benefits of making power-gating decisions for each sub-accelerator separately, rather than simply at the accelerator level. We show that for some applications, this finer-granularity results in more effective power-gating, providing more power savings than the previous technique.

When determining whether an idle period is long enough to make power-gating worthwhile, [1] assumes a static schedule constructed using a single set of input vectors. If these input vectors change, the idle times experienced during the run of the application may deviate from the predicted idle times, potentially leading to sub-optimal power-gating decisions. For the accelerators considered in [1], the idle periods are long enough that this is not likely to be a concern. Finer-grained power-gating, however, implies shorter idle periods, meaning the optimal power-gating decisions may be more sensitive to changes in the inputs. In this paper, we investigate whether this is an issue as the granularity of power-gating decreases. Thus, this paper has three contributions:

1. We enhance the design framework in [1] to support hierarchical power-gating. Section 3 presents our compiler-assisted framework for automatically generating hierarchical power-gating decisions.

2. We study the impact of reducing the power-gating granularity to consider power-gating sub-accelerators as well as accelerators. Section 4 quantifies the impact of hierarchical power-gating.
3. We investigate whether the compiler-assisted power-gating approach used in [1] is sufficient as the granularity of power-gating decreases. Section 5 presents our findings across a large number of input patterns.

2 Context

Although our work will apply to any FPGA which provides dynamic power-gating control, and any high-level synthesis tool, we describe it in the context of the Dynamic Power-Gated FPGA architecture from [3] and the LegUp high-level synthesis tool from [4]. In this section, we present a brief background about each.

2.1 Dynamic Power-Gated Architecture

The DPG architecture from [3] is a typical island-style FPGA in which the logic and routing fabric have been augmented with header switches and associated control logic that allow regions of the chip to be selectively powered-down, under the control of signals from elsewhere on the chip (typically from a power-state controller). A Power-Gated Region (PGR) is the basic unit of power-gating which is turned on or off as a unit. Each of these regions consists of a small number of CLBs, as shown in Fig. 1. The flip-flops within each logic element are not turned off as this allows for a more rapid power-up sequence since state is retained. Table. 1 shows the simulated architecture parameters used in our experiments.

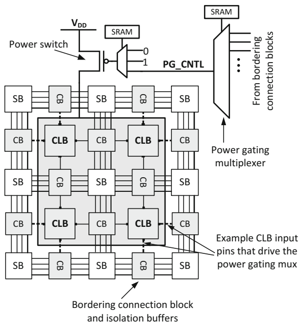
V_DD
Power switch
SRAM
0
1
PG_CNTL
SRAM
From bordering connection blocks
Power gating multiplexer
SB
CB
CLB
Example CLB input pins that drive the power gating mux
Bordering connection block and isolation buffers

Fig. 1. Example power gating region supporting DPG [3]

Table 1. Simulated Architecture Parameters used in Experiments of Sect. 4

		PGR	SB
$P_{1\rightarrow 0}$	Power to Turn-Off	1.22E-04	1.45E-05
$P_{0\rightarrow 1}$	Power to Turn-On	5.98E-04	5.53E-05
$T_{1\rightarrow 0}$	Time to Turn-Off	6.59E-09	4.12E-10
$T_{0\rightarrow 1}$	Time to Turn-On	7.14E-09	4.46E-10
$P_{ON-leak}$	On Leakage Power	6.70E-05	5.36E-06
$P_{OFF-leak}$	Off Leakage Power	2.03E-06	2.93E-07

2.2 LegUp High-Level Synthesis Framework

Our work is based on *LegUp*, a high-level open source synthesis framework [4]. LegUp automatically generates an SoC consisting of a MIPS processor and one or more accelerators from an application expressed in *C*. The application is first profiled on a hardware profiler to identify the functions that would benefit from hardware implementation. Based on this information, the tool then compiles each identified function into a *hardware accelerator*. The portions of the algorithm that are not selected for acceleration are mapped to the MIPS processor and will run as software. These accelerators and a MIPS processor are then combined using an Avalon fabric, creating an accelerated version of the original C code.

3 Hierarchical Power-Gating

As described above, LegUp automatically identifies functions in the input C code that are suitable for acceleration. The work in [1] considers each of these accelerators, along with a static schedule, and determines whether the benefit of turning off the accelerator is more than the overhead of doing so. In our work, we consider not only each function in the C code, but their sub-functions as well. In the hardware generated by LegUp, each sub-function is implemented as a separate hardware unit (which we refer to as a *sub-accelerator*) and our approach considers turning off each of these hardware units separately.

Our framework is shown in Fig. 2. We identify all the profitable idle phases across all the hierarchical accelerators and generate a static power-gating schedule. This is then passed on to HLS compiler which then generates Verilog descriptions of the datapath and controller circuit. The various phases of the framework are discussed below.

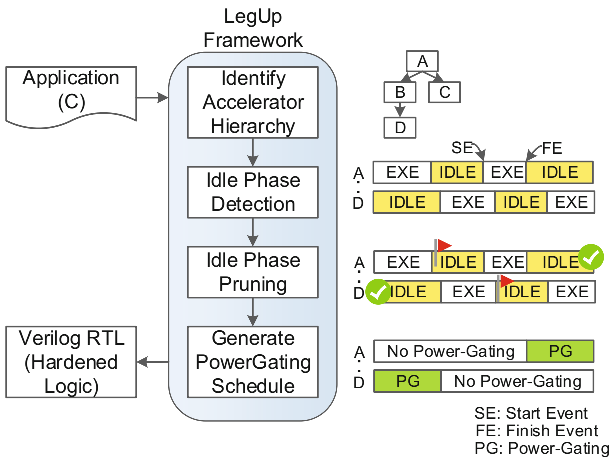

Fig. 2. HLS Compiler Assisted Hierarchical Power-Gating Framework

3.1 Identifying Accelerator Hierarchy

The (HLS) compiler operates on a C program and converts annotated (profitable) C-functions into hardware accelerators. In doing so, if the function has descendant functions, they are also converted into sub-accelerators. We keep track of the function calls in the program and build a hierarchical call tree for all the functions (annotated for acceleration) in the program. This enables us to identify all the hierarchical accelerators in the program, which later get synthesized into hardware modules. In this paper, we refer to the top accelerator in the hierarchy as a *parent accelerator* and the accelerators called by the top accelerator as *child accelerators*.

3.2 Idle Period Detection

Each accelerator (parent or child) typically has active and idle periods. The idle periods are potential power-gating opportunities. The goal of this phase is to identify all idle periods based on a static schedule, as determined by the scheduling algorithm in the HLS compiler, and to identify the start and end events associated with each idle period. Note that the actual duration of the idle period will typically vary from run to run as input patterns change. In Section 5, we evaluate the impact of changing inputs on the number of idle periods and their duration.

3.3 Pruning Idle Periods

The previous phase produces a list of all potential power-gating opportunities. However, many of these may not be profitable. Powering down (and later powering up) a power-gated region incurs both energy and delay overhead. Thus, for each power-gating opportunity, we must determine whether a power-gating event should be generated.

In general, an accelerator should be turned off when idle if the power saved by power-gating the accelerator is more than the overhead of turning the accelerator off and then on again at the end of the idle period. To make this determination, we first find the accelerator's energy break-even time – the minimum idle time at which the leakage-savings compensate the energy penalty for mode transition. If the predicted idle time duration, as determined in Section 3.2, is more than break-even time, we generate a power-gating event. Below, we briefly describe how we find an accelerator's break-even time.

Determining Accelerator's Energy Break-Even Time: Every time an accelerator transitions between the *sleep* and *execution* modes, there is an energy penalty. An accelerator should only be power gated if it will be idle long enough to compensate for this penalty. The energy break-even time is the minimum idle time for which an accelerator should be power gated, and can be calculated as:

$$T_{break-even} = \frac{(P_{1\rightarrow 0} \times T_{1\rightarrow 0}) + (P_{0\rightarrow 1} \times T_{0\rightarrow 1})}{P_{ON-leak} - P_{OFF-leak}} \tag{1}$$

where $P_{1\rightarrow 0}$, $T_{1\rightarrow 0}$, is the power and time required to enter power-saving mode. Similarly, $P_{0\rightarrow 1}$ and $T_{0\rightarrow 1}$ is the power and time required to exit the power-saving mode. $P_{ON-leak}$ and $P_{OFF-leak}$ is the leakage power in the turned-on and turned-off state respectively.

In context of our target DPG architecture, as discussed in Sec.2.1, the power-gated region (PGR) is the basic unit of granularity. Thus, an accelerator occupies an integral number of PGRs and switch-blocks (SBs). Therefore, the switching energy and power-saved can be expressed as,

$$\begin{aligned} E_{switch} = {} & NumPGR \times (P_{switchPGR} \times T_{switchPGR}) \\ & + NumSBs \times (P_{switchSB} \times T_{switchSB}) \end{aligned} \tag{2}$$

$$\begin{aligned} P_{saved} = {} & NumPGR \times (P_{PGR-on-leak} - P_{PGR-off-leak}) \\ & + NumSBs \times (P_{SB-on-leak} - P_{SB-off-leak}) \end{aligned} \tag{3}$$

where in (2) and (3), $NumPGR$ and $NumSBs$ is the number of PGRs and SBs, respectively, occupied by the accelerator. We find this by performing an initial mapping of the accelerator to the fabric, however, estimation techniques could also be used.

3.4 Power-gating Schedule Generation

Once all the profitable power-gating opportunities across all the hierarchical accelerators have been identified, a schedule with these decisions is generated. The power-gating schedule records the conditions (start and end events) that trigger a particular idle period in an accelerator. Each time these conditions are met at run-time, power-gating is triggered which would put the accelerator in sleep mode.

4 Experimental Results

4.1 Experimental Setup

We use the CHStone benchmarks suite developed for C-based high-level synthesis (HLS); these benchmarks represent diverse real-world application domains. The C-based source of each benchmark was provided as input to the Legup HLS framework, augmented with our tool flow, as shown in Fig.2. The RTL output from the framework, which contains datapath, controller and hierarchical power-gating schedule, is then mapped to the dynamic power-gating FPGA architecture from [3] to get the number of PGRs and SBs occupied by the accelerators. Since we have not fabricated an FPGA with our power-gating architecture, we do not generate a bitstream. However, the HSPICE simulated architecture parameter values, given in Table. 1, allows us to quantify the impact of hierarchical power-gating.

4.2 Hierarchical Power-Gating Evaluation

In order to quantify the impact of hierarchical power-gating, we apply the following power-gating policies to the parent and child accelerators in CHStone benchmarks suite [5] and compare their impact:

Accelerator-Level Power-Gating; Policy-P1: In this policy, the entire accelerator, including all the sub-accelerators of this accelerator, is considered as one power-gating unit. The sub-accelerators have no separate power-gating control. Whenever the parent accelerator is active at the end of it's idle period, all it's sub-accelerators in the hierarchy become active as well. This approach is similar to [1] and serves as the reference.

Parent Runs while Child Runs; Policy-P2: This approach represents intra-accelerator power-gating in which the parent and descendant accelerators are treated as independent units for power-gating i.e. each hierarchical accelerators can be switched independently. In this policy, the parent accelerator remains active and does not sleep after initiating the sub-accelerator. The sub-accelerator is power-gated, however, if only the parent is required to be active.

Parent is Power-Gated while Child Runs; Policy-P3: This approach also represents intra-accelerator power-gating. In this approach, we power-gate the parent accelerator after it is has started a sub-accelerator. Once the sub-accelerator has finished processing, the parent accelerator is woken up.

The proposed power-gating policies are evaluated based on the total leakage energy consumed by an accelerator in power-gating mode, denoted as E_PG, in which the accelerator is turned-off during it's idle period if it is deemed profitable. The leakage energy in all execution and idle phases is added together to estimate E_PG. By design, the DPG architecture in [3] suffers a 10% performance degradation due to the presence of power-gating circuitry; to account for this, we increase the idle and execution time periods by this degradation factor when calculating E_PG. Of the circuits in the CHStone suite, we present results for two benchmarks below.

JPEG: The hierarchical call tree of JPEG benchmark is shown in Fig. 3. There are three parent-level and two child-level accelerators. We evaluate the above mentioned power-gating policies for *Decode_Block (DB)* accelerator as it has two descendants. The occupancy stats of various JPEG accelerators are reported in Table. 2. The columns labeled DTF show the number of power-gated regions (PGRs) and switch-blocks (SBs) for each accelerator that can be *dynamically*

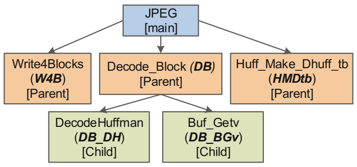

Fig. 3. JPEG Hierarchical Call Tree

Table 2. JPEG Occupancy Stats

Accelerator	Total PGRs	PGRs (DTF)	SBs (DTF)	EBT	EBC	W.Cyc
W4B	162	139	1576	4.43E-08	3	4
DB	222	155	659	6.01E-08	4	4
HMDtb	48	28	260	4.78E-08	4	1
DB_DH	31	21	96	5.91E-08	4	1
DB_BGv	25	19	54	6.50E-08	5	1

turned off (DTF) (typically not all PGRs and switch blocks can be turned off even when an accelerator is idle, since routing switches within a switch block or other parts of a PGR must remain active to implement parts of the circuit that have not been turned off). The fifth column shows the accelerator's energy-break even time (EBT) which is converted into energy-break even cycles (EBC) assuming a 66Mhz clock. The last column is the number of cycles to transfer from sleep to normal execution mode, denoted as *wake-up cycles* (W.Cyc), and are used to quantify the impact on execution length.

Fig. 4a shows the execution and idle time percentages across the three power-gating policies. The *Decode_Block (DB)* appears as a single accelerator in policy $P1$, as the parent and it's descendants are treated as one combined hardware module, while it's descendants are visible in the other two policies. As can be observed from Fig. 4a, the descendant accelerators executes briefly and remain idle most of the time during which they can be power-gated. The parent accelerator is power-gated in $P3$ while it's descendants are running which presents an additional power-savings opportunity. Fig. 4b shows the *E_PG* consumption comparison across power-gating policies. The *E_PG* for the parent and child accelerators are added together in $P2$ and $P3$. As can be observed in Fig. 4b, *E_PG* in $P2$ is 31.88% less than *E_PG* in $P1$ which indicates that in JPEG benchmark it would be worth considering the parent and child accelerators as stand-alone units for power-gating. Further, in this benchmark, the child accelerators have significant execution times, meaning that by turning off the parent when the children are executing (policy $P3$), further power reductions are possible. As shown in Fig. 4b, *E_PG* in $P3$ is 49.30% less than *E_PG* in $P1$.

SHA: The hierarchical call tree of SHA benchmark is shown in Fig. 5 and it's accelerators sizes are given in Table. 3. Note that the *Sha_Tansform* accelerator appears as both parent and child, and is instantiated twice when hardware is generated. We distinguish the child-level *Sha_Tansform (SU_ST)* accelerator by adding its parent name before it.

As can be observed in Fig. 6a, the child accelerator, *Sha_Tansform (SU_ST)*, is busy executing most of the time; during this time, it's parent, *Sha_Update (SU)*, can be turned off (Policy $P3$). The value of *E_PG* for each policy is shown in Fig. 6b. As can be seen, *E_PG* for policy $P3$ is 46.11% less than for policy $P1$, as the parent remains turned off 89% of the time. This comparison suggests that it is best to treat the parent and child as independent accelerators which, if

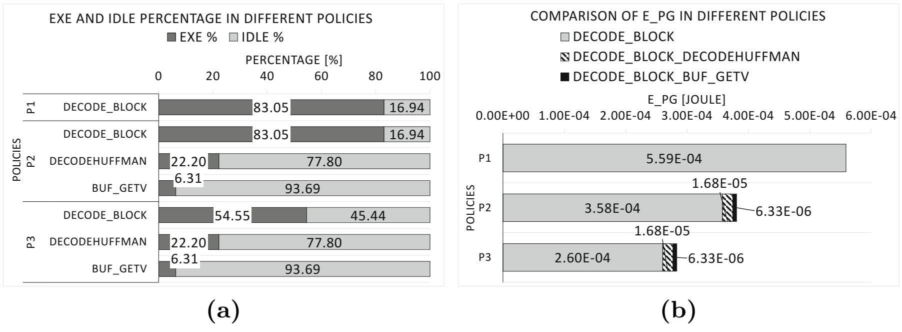

Fig. 4. JPEG Benchmark (a) Percentage of Execution and Idle Times (b) Comparison of Total Leakage Energy Consumed - in various Power-Gating Policies

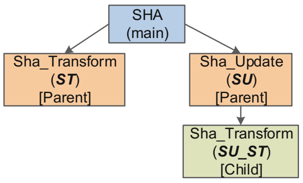

Fig. 5. SHA Hierarchical Call Tree

Table 3. SHA Occupancy Stats

Accelerator	Total PGRs	PGRs (DTF)	SBs (DTF)	EBT	EBC	W.Cyc
SU	52	38	350	4.79E-08	4	1
ST	79	65	476	5.17E-08	4	2
SU_ST	75	68	570	4.95E-08	4	2

are power-gated during their respective idle times can reduce, the leakage energy by 46%. Therefore, in case of SHA benchmark, policy $P3$ remains best overall.

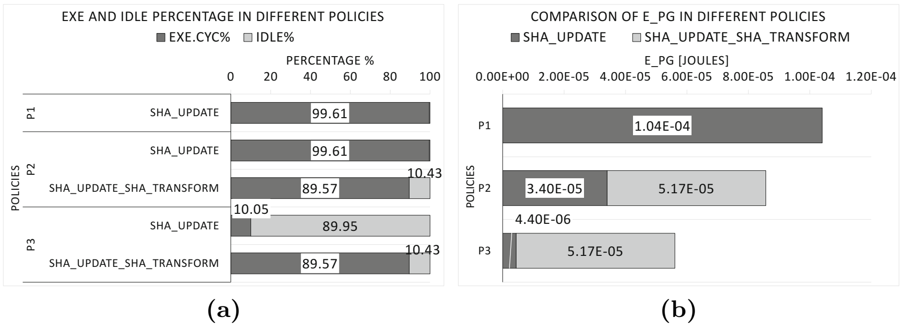

Fig. 6. SHA Benchmark (a) Percentage of Execution and Idle Times (b) Comparison of Total Leakage Energy Consumed - in various Power-Gating Policies

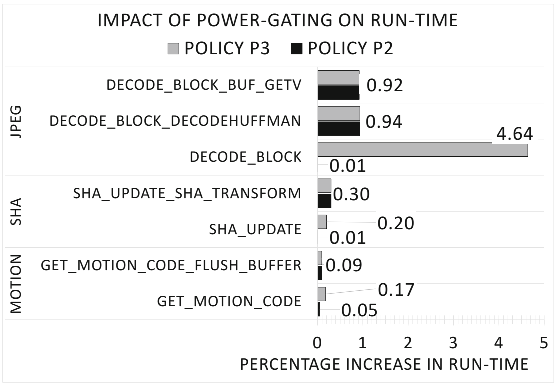

Fig. 7. Impact of Power-Gating on Run-Time

4.3 Impact of Power-Gating on Execution Run-time

Each power-gating event incurs extra cycles to transition between power-gating modes. Fig. 7 shows the percentage increase in run-time for various accelerators due to power-gating overhead in policy P2 and P3. In policy P3 the power-gating of the parent while the child runs introduces more idle periods for the parent which increase the overall overhead. As a result, the execution length of the parent accelerator increases more in P3 than in P2. As shown in Fig. 7, the run-time of the *decode-block* in policy P3, which is a parent accelerator in JPEG, is increased by 4.6% due to large number of calls to the two child accelerators during which the parent is power-gated.

5 Impact of Input Patterns on Static Power-Gating Decisions

As discussed in Sec.3.4, a single set of input vectors is used when constructing the static power-gating schedule. When the application is run, it may experience different input patterns than those assumed during scheduling. This may cause changes in the idle periods of accelerators and sub-accelerators. This may mean that power-gating decisions made during scheduling are sub-optimal. If the idle period of an accelerator is much smaller than predicted, it may be that more energy is wasted turning off and on the region than is saved while power-gated. If the idle period of an accelerator is longer than predicted, it may be that the scheduler decides that power-gating is not worthwhile, when in actuality, it would have been. To investigate this concern, we vary the input vectors ten thousand times and observe the impact on the number and duration of idle periods in an accelerator. In particular, we are interested in whether the change in input can reduce the idle period duration below energy break-even cycles leading to a sub-optimal decision.

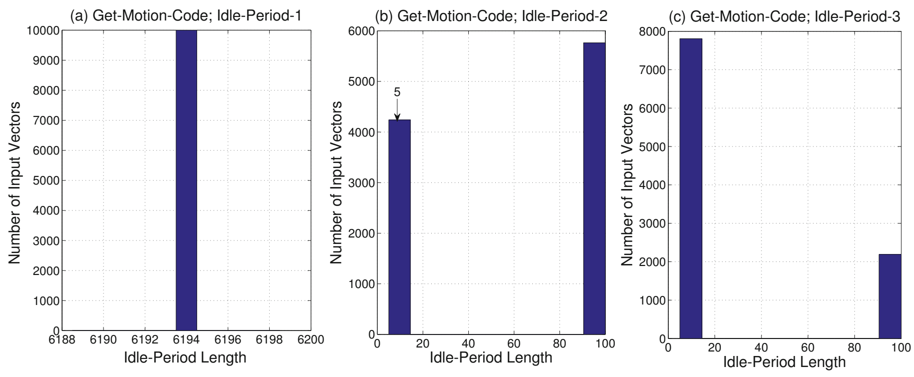

Fig. 8. Histograms showing the Variation in the Idle-Period Duration of Get-Motion-Code Accelerator (a) Idle-Period-1 (b) Idle-Period-2 (c) Idle-Period-3

We performed the experiment across nine accelerators in three of the CHStone benchmarks circuits. For each accelerator, we varied the input pattern by randomly selecting values for each input independently (the allowable range of values for each input was determined by examining the program). Ten thousand input sets are generated, and for each, the total number of and the duration of idle periods were recorded.

We found that in eight (out of nine) accelerators, there was no variation in the number of idle periods and the duration of each idle period remained the same. As a result, for these accelerators, the static power-gating scheduler makes the optimal decisions.

The ninth accelerator, *Get_Motion_Code (GMC)* in the Motion benchmark, exhibited variation in the duration of idle periods. This particular accelerator has three idle periods and the static power-gating schedule determines that power-gating is appropriate for all of them. Changing the input vector $10K$ times reveals the variation as illustrated in the histograms of Fig. 8. The X-axis of each histogram represents the number of idle cycles in an idle period and the Y-axis is the number of input vectors that led to that number of idle cycles. The first idle period does not show any variation, however, the second and third idle periods do show variation. In both cases, the periods were either 5 cycles long or 100 cycles long, depending on one of the input signals. For period 2, about 40% of input samples led to an idle period of 5 cycles. In this accelerator, the energy break-even point is exactly 5 cycles. As a result, the static scheduler always assumes that the accelerator should sleep for this idle period. If the period happens to be 5 cycles, the overall cost and benefit of power-gating are equal, but if the idle period is 100 cycles, significant power savings are obtained by power-gating. In either case, however, the decision to power gate during this idle period is optimal. This situation is the same for the third idle period, in which 20% of input vectors lead to long idle periods.

It is possible that this could occur if the minimum idle phase duration goes below energy break-even point, however, in none of our experiments did this occur.

6 Conclusion and Future Work

This paper demonstrates the potential of hierarchical power-gating to save static leakage power. We show that fine-grained control over the sub-accelerators allow them to sleep when only the parent context is required. Similarly, power-gating the parent accelerator when the child is running, creates more power saving opportunities but increases the program length. Reducing the granularity, however, reduces the duration of the idle periods. We show that the compiler-assisted power-gating decisions remain optimal as long as the idle period experienced at run-time is greater than the break-even point. The static power-gating predictions can be greatly improved if the application workload is known at compile-time. Adapting the power-gating decisions with varying workload at run-time, however, would require a dynamic power-gating predictor which is an interesting area for future investigation.

References

1. Ahmed, R., Bsoul, A.A.M., Wilton, S.J.E., Hallschmid, P., Klukas, R.: High-level synthesis-based design methodology for dynamic power-gated fpgas. In: 24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, September 2–4, 2014, pp. 1–4 (2014). http://dx.doi.org/10.1109/FPL.2014.6927433
2. Bharadwaj, R., Konar, R., Balsara, P., Bhatia, D.: Exploiting temporal idleness to reduce leakage power in programmable architectures. In: Proceedings of the 2005 Asia and South Pacific Design Automation Conference, ASP-DAC 2005, vol. 1, pp. 651–656, January 2005
3. Bsoul, A., Wilton, S.J.E.: An FPGA architecture supporting dynamically controlled power gating. In: Proc. of the 2010 Intl. Con. on Field-Programmable Technology (FPT), pp. 1–8 (2010)
4. Canis, A., Choi, J., Fort, B., Lian, R., Huang, Q., Calagar, N., Gort, M., Qin, J.J., Aldham, M., Czajkowski, T., Brown, S., Anderson, J.: From software to accelerators with legup high-level synthesis. In: Proc. of the 2013 Intl. Con. on Compilers, Architecture and Synthesis for Embedded Systems (CASES), pp. 1–9, September 2013
5. Hara, Y., Tomiyama, H., Honda, S., Takada, H., Ishii, K.: Chstone: a benchmark program suite for practical c-based high-level synthesis. In: ISCAS, pp. 1192–1195. IEEE (2008). http://dblp.uni-trier.de/db/conf/iscas/iscas2008.html#HaraTHTI08
6. Ishihara, S., Hariyama, M., Kameyama, M.: A low-power FPGA based on autonomous fine-grain power gating. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **19**(8), 1394–1406 (2011)
7. Kuon, I., Rose, J.: Measuring the gap between FPGAs and asics. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems **26**(2), 203–215 (2007)
8. Usami, K., Ohkubo, N.: A design approach for fine-grained run-time power gating using locally extracted sleep signals. In: International Conference on Computer Design, ICCD 2006, pp. 155–161, October 2006