Functional Partitioning for Reduced Power

Enoch Hwang, Frank Vahid, Yu-Chin Hsu

Technical Report CS-98-03
May 13, 1998

Department of Computer Science
University of California
Riverside, CA 92521
{ehwang,vahid,hsu}@cs.ucr.edu

Abstract

Power consumption in VLSI systems has become a critical metric for design evaluation. Although power reduction techniques can be applied at every level of design abstraction, most automated power reduction techniques apply to the lower levels of design abstraction, such as the register-transfer or gate level. We therefore investigated the power reduction attainable by the evolving automated behavioral-level technique of functional partitioning, in which a behavioral process is automatically divided into several smaller, mutually-exclusive, interacting processes. We demonstrate through several experiments that functional partitioning, which has already been shown to yield improvements in solving problems of I/O constraint satisfaction, synthesis complexity, and hardware/software partitioning, can also yield substantial reduction in power consumption (on average 41% total power reduction) with some overhead in size and performance.

Functional Partitioning for Reduced Power

Enoch Hwang Frank Vahid Yu-Chin Hsu
Department of Computer Science
University of California, Riverside, CA 92521
ehwang@cs.ucr.edu vahid@cs.ucr.edu hsu@cs.ucr.edu

Abstract

Power consumption in VLSI systems has become a critical metric for design evaluation. Although power reduction techniques can be applied at every level of design abstraction, most automated power reduction techniques apply to the lower levels of design abstraction, such as the register-transfer or gate level. We therefore investigated the power reduction attainable by the evolving automated behavioral-level technique of functional partitioning, in which a behavioral process is automatically divided into several smaller, mutually-exclusive, interacting processes. We demonstrate through several experiments that functional partitioning, which has already been shown to yield improvements in solving problems of I/O constraint satisfaction, synthesis complexity, and hardware/software partitioning, can also yield substantial reduction in power consumption (on average 41% total power reduction) with some overhead in size and performance.

1. Introduction

Power reduction of VLSI systems is becoming an increasingly-important goal of system design. The shrinking sizes of integrated circuits calls for reduced power consumption, in order to extend battery life or prevent excessive heat generation. While power reduction techniques can be applied at nearly every design abstraction level, most of the recent power estimation and reduction tools focus on the lower levels of abstraction, such as the register-transfer (RT) or gate levels. There is therefore still much opportunity to reduce power by developing tools that focus on higher levels, such as the behavioral level, where major reductions may be gained.

A number of behavior-level power reduction techniques have been proposed recently, most of which seek to reduce switching activity through either behavior transformations or through partitioning. Behavior transformations reduce the number of cycles in a computation by subsequent behavioral synthesis, as in [1] and [2]. Partitioning includes disabling mutually-exclusive or inactive portions of the circuit when not needed in the execution during a certain time, as in [3], [4], and [5]. In [4], a finite-state machine (FSM) is decomposed into mutually exclusive FSM's, where inactive FSM blocks have their clock signal disabled. In [5], the clock signal to inactive flip-flops is disabled. Other power reduction techniques at the behavioral level include resource allocation as proposed in [6] to increase the temporal correlation, thereby reducing switching activity. In [7], redundant adders are added to reduce the output activity. [8] and [9] introduce coding techniques for reducing the switching activities on the I/O pins and address busses. [10] and [11] provide optimum supply voltage and/or mixed voltages to the modules. Software may be compiled so as to minimize the power dissipation when it is executed on a given hardware platform, as in [12].

In this paper, we demonstrate power reductions at the behavioral level through functional partitioning. This work was inspired by the FSM partitioning approach in [4], and is applicable to an entire behavioral process which includes both control and data, rather than just a FSM which includes only control. Techniques for automated functional partitioning have already been developed to solve problems of satisfying I/O constraints, reducing synthesis runtime, and hardware/software partitioning [13], with order-of-magnitude reductions in I/O and synthesis runtimes reported compared with the traditional approach of structural partitioning. Our functional partitioning technique can be used as an initial global optimization and reduction technique, after which the earlier localized techniques can be applied for further power reductions.

The rest of this paper is organized as follows. In section 2, we give a background of the power consumption problem and how partitioning can reduce switching activity. Our functional partitioning technique is discussed in section 3. Section 4 shows our experimental results, and we conclude in section 5.

2. Background

Power consumption of a CMOS circuit is composed mainly from dynamic power due to capacitive charging and discharging when a signal toggles as computed from the equation

$$P_d = \frac{1}{2} C V^2 f\ N \qquad (1)$$

where C is the loading capacitance of a gate output, V is the supply voltage, f is the clock frequency, and N is the switching frequency of the gate, i.e. the number of gate output transitions per clock cycle.

For a given behavioral level circuit design, all the variables except for *N* are usually fixed by a design decision. Thus, the power reduction of a circuit at this level is obtained mainly by reducing the switching frequency *N* from the outputs of the gates.

When a circuit is synthesized from a behavioral description to a gate level netlist, only one control unit and one datapath is generated. What this means is that when there is a signal change at the primary input, the entire datapath and control unit may be affected. This results in much unnecessary switching activities. For example, if a processor contains two procedures, *Procedure1* and *Procedure2*, with common inputs as in Figure 1 (a), and we want to evaluate function *Procedure1* followed by *Procedure2*. While performing *Procedure1*, the input signals will also propagate to the gates in *Procedure2* because they are all inter-connected. A hypothetical switching activities for the gates of the control units and datapaths for both *Procedure1* and *Procedure2* are shown in Figure 1 (b). Thus, power is being used by *Procedure2* even though the result from it is not needed. In fact, the amount of power used by *Procedure2* is the same regardless of whether the result is needed or not. Similarly when *Procedure2* is being performed, *Procedure1* is consuming power but the result from it is not needed. If we take a hypothetical situation where both functions require 1μW of power and 1μsec to execute, then for both functions to execute sequentially, a total of 2μW of power is required for 2μsec resulting in a total of 4μJ of energy

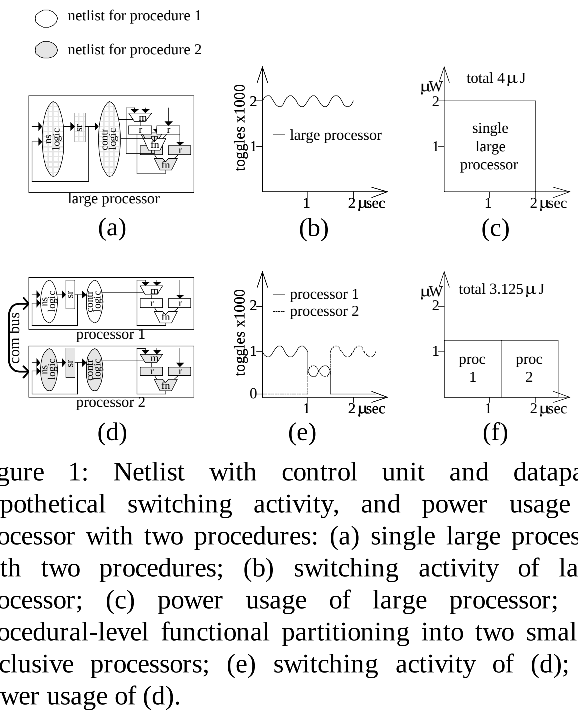

Figure 1: Netlist with control unit and datapath, hypothetical switching activity, and power usage of processor with two procedures: (a) single large processor with two procedures; (b) switching activity of large processor; (c) power usage of large processor; (d) procedural-level functional partitioning into two smaller, exclusive processors; (e) switching activity of (d); (f) power usage of (d).

being consumed by the single large processor as illustrated in Figure 1 (c).

One solution is to partition the circuit during synthesis so that the switching activities can be localized within a subset of gates. While numerous earlier techniques described above seek to partition the controller or the datapath of a single custom hardware processor, functional partitioning instead seeks to partition the processor into multiple simpler mutually-exclusive communicating processors, each with its own controller and datapath. These processors are exclusive because they together implement a single sequential process (akin to the FSM decomposition approach mentioned earlier). Because each processor is smaller than one large processor, and switching occurs in only one processor at any given time, switching activity due to computations carried out within a processor will be reduced. However, partitioning introduces new switching activity for interprocessor communication. For example, we would put the control units and datapaths for *Procedure1* and *Procedure2* into two separate processors so that the input to one will not affect the input to the other as in Figure 1 (d). With several smaller mutually-exclusive processors, the total amount of switching activities at any one time is reduced as shown in Figure 1 (e), and the power is reduced as shown in Figure 1 (f).

Thus, we focus on the problem of functionally partitioning such that power is minimized. We assume that we are partitioning among homogeneous parts within a single piece of silicon, and that the number of parts, usually two or three, is given (future work might explore finding the best number of parts for a given example). The problem that must be solved is one of partitioning such that the reduction in switching activity for computations far outweighs the switching activity increase for communication, while also minimizing the increase in total circuit size and execution time.

3. Functional Partitioning

Functional partitioning may be best introduced by contrasting it with the well-known technique of structural partitioning. In structural partitioning (also known as

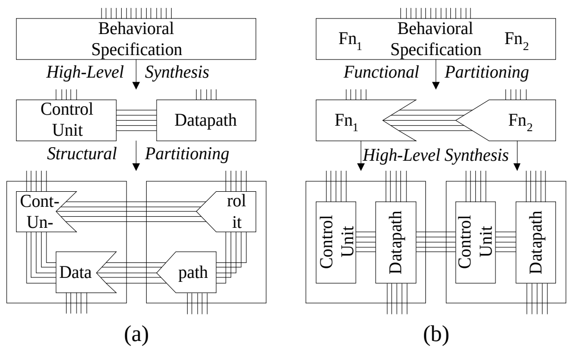

Figure 2: Partitioning: (a) structurally; (b) functionally.

circuit or netlist partitioning), see Figure 2 (a), we first synthesize a behavioral process, usually written in behavioral VHDL or Verilog, down to an RT or gate level netlist. Second, we partition the netlist among parts, which may represent physical packages on which the entire netlist would not fit, or blocks on a single piece of silicon. Structural partitioning is known to be I/O limited [15], meaning that the I/O capacity is usually exceeded during partitioning before the gate capacity (in other words, it's very hard to partition a netlist without introducing hundreds or thousands of wires between parts). From a power reduction perspective, structural partitioning does not reduce switching activity. The reason is that synthesis converts one process into one custom hardware processor, consisting of a controller and datapath. Even though partitioning creates more than one physical partition, there is still logically only one datapath and one control unit. When a primary input signal changes, it may affect many of the gates in the datapath regardless of which partition they are in. Moreover, the gates in the control unit must also be active in order to provide the correct control signals to the datapath.

In functional partitioning, the behavioral process is first partitioned into several smaller processes, as shown in Figure 2 (b). Each process is then synthesized to its own custom processor, each with its own controller and datapath. Such partitioning has been shown to reduce I/O as well as synthesis runtimes by nearly an order of magnitude [13], as well as being suited for hardware/software partitioning. (Note that, while many researchers have demonstrated advantages of functionally-partitioning multiple processes, we focus on and have shown the advantages of partitioning one large process). From a power reduction perspective, functional partitioning can significantly reduce switching activity. The reason is that each processor is smaller than one large processor implementing the entire process, and only one processor is executing a computation at any given time. Thus, at a given time, switching activity is limited to one small processor; the other processors will be idle and power is saved as shown in Figure 1 (d) – (f).

```
input x;
count := 2;
while ((count * count) <= x) loop
  mod(x,count,mod_result);
  divide(x,count,divide_result);
  is_prime(count,prime_result1);
  is_prime(divide_result,prime_result2);
  if (mod_result = 0) and (prime_result1 = 1) and
    (prime_result2 = 1) then
    answer1 <= divide_result;
    answer2 <= count;
    exit;
  else
    count := count + 1;
  end if;
end loop;
```

Figure 3: Sample unpartitioned pseudo-code.

```
part 1
  …
  while ((count * count) <= x) loop
    call and wait for result from part 2;
    call and wait for result from part 3;
    if …
  end loop;

part 2
  wait for part 1 to call;
  get parameters x and count;
  mod(x,count,mod_result);
  divide(x,count,divide_result);
  return mod_result and divide_result;

part 3
  wait for part 1 to call;
  get parameters count and divide_result;
  is_prime(count,prime_result1);
  is_prime(divide_result,prime_result2);
  return prime_result1 and prime_result2;
```

Figure 4: Sample partitioned pseudo-code.

3.1 Partitioning model

In our approach, we first parse a VHDL process into an access-graph representation (similar to a call graph). The access graph is a directed acyclic graph in which each node represents a procedure, and each edge represents a call from one procedure to another. If the user-written procedures did not provide satisfactory granularity, automated procedure inlining and exlining can be performed before creating the access graph [16]. The graph is then annotated with edge access frequencies, edge bit sizes, node execution times, and node sizes, obtained through profiling and synthesis during a pre-estimation phase. Then, partitioning hueristics are applied to the graph, and annotations are combined in estimates of size, performance, and I/O during an online-estimation phase. We have found a modified version of the Kernighan/Lin heuristic to work best. The heuristics are guided by a weighted-sum cost function seeking to obtain balanced part sizes, minimized I/O, and minimized execution time (by minimizing communication). Details of the functional partitioning approach are extensive and have been published elsewhere; we refer the reader to [17]. Note that there is no explicit power metric in the cost function. Because we are dealing with homogeneous parts, we have found that minimizing communication between parts, while maintaining balanced part sizes, is a good approach to minimizing power. This indirect approach should be replaced by a direct power metric when a suitable power estimation becomes available.

For example, given the pseudo-code in Figure 3, we can partition it into three parts as in Figure 4. The process calls three separate procedures, *mod*, *divide*, and *is_prime*. We can put *mod* and *divide* in part two, *is_prime* in part three and the remaining code in part one. Part one controls the entire program flow, performing the primary I/O, and calling the procedures in the other parts when required.

3.2 FunctionBus

The model we used for connecting the parts together is a modified version of the *FunctionBus* model described

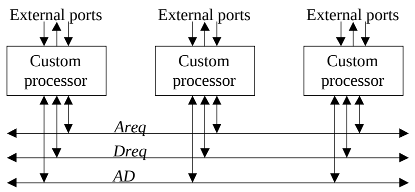

Figure 5: FunctionBus architecture.

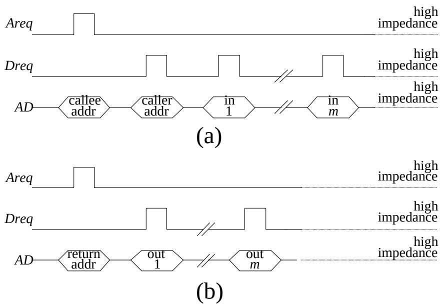

Figure 6: Timing diagrams: (a) function call; (b) function return.

in [18]. The FunctionBus architecture is shown in Figure 5. The parts are connected together by a bus consisting of the address request (*Areq*), data request (*Dreq*), and address/data (*AD*) lines. *AD* consists of *N* lines, used to carry address and data. *Areq* is a single line used to indicate a valid address on *AD*. *Dreq* is a single line used to indicate valid data on *AD*. Only one processor will control the bus at a time, with the others providing high-impedance values. This bus is used for all handshaking and data transfers between the parts. A system executes as a series of function calls. If a function *A* calls another function *B*, *A* sends *B*'s address over the bus, followed by input parameters to *B*. *B* captures these parameters and then executes. Upon completion, *B* sends *A*'s address over the bus, followed by any output parameters. *A* captures these parameters and resumes execution.

The protocol used in the FunctionBus is shown in Figure 6. A *function call* sends the address, possibly a return address, and input parameters as shown in Figure 6 (a). A *function return* sends the return address and output parameters as shown in Figure 6 (b). Both communications begin by placing the address of the receiver function on *AD* and pulsing *Areq*. The function call protocol then places the return address on *AD* and pulses *Dreq* (if necessary). Both then send a sequence of data chunks by placing a chunk on *AD* and pulsing *Dreq*.

With this model, a system executes as a series of function calls. Such an execution model has the feature that only one part is active at a time. As long as there is no activity on the FunctionBus, the inputs to all the parts will be at a constant value. If the inputs to a part do not change, then there will be no switching activities within that part. Thus, all switching activities will be localized to just the one active part.

4. Experiment

We want to compare the power consumption of the partitioned system with that of the unpartitioned system. First, we describe a system at the behavioral level using VHDL. For the unpartitioned system, we simply synthesize the behavioral description to the gate level and calculate the power consumed. For the partitioned system, we first partition the behavioral description into two or more functional parts. We then add the FunctionBus with the neccessary handshaking protocol to each part. The parts are individually synthesized to the gate level. The same synthesis optimizations applied to the unpartitioned system are also applied to the partitioned system. At the gate level, we join these parts back together by adding common wires for the FunctionBus. Power consumption is then calculated.

For comparing the power consumption, we used both the average power per clock period and the total power. In our power calculation, a zero-delay model is assumed and the transition probabilities are used. As derived from equation (1), the average power is computed as

$$P_{average} = \frac{1}{2T_c} Vdd^2 \sum_{\forall i} C_i D_i \quad \text{watts} \qquad (2)$$

where T_c is the clock period, C_i is the total capacitance at node *i*, and D_i is the transition probability which is computed as

$$D_i = \frac{T_c N_i}{\text{(total execution time)}} \qquad (3)$$

where N_i is the number of toggles at node *i*. The total power is

$$\begin{aligned} P_{total} &= P_{average} \times \text{(total execution time)} \\ &= \frac{1}{2} Vdd^2 \sum_{\forall i} C_i N_i \quad \text{joules} \end{aligned} \qquad (4)$$

In our experiments, we described five examples at the behavioral level using VHDL. MEBS [19], a behavioral synthesizer, was used to synthesize the partitioned and unpartitioned systems from the behavioral level to the gate level. Purespeed [20], an event-driven simulator, was used to collect the switching frequency data for the power calculation. The loading capacitance for each node was obtained from the synthesized gate level netlist and a low-power technology library. For the power calculation, the voltage was assumed to be at 5V and the clock frequency at 20MHz.

Table 1 shows the sizes of these examples in terms of the number of gates. *Fac*, shown in Figure 3, is a factorization program. *Ch* is to evaluate the Chinese Remainder Theorem. *Diffeq* is an example from the HLSynth MCNC benchmark. *Volsyn* is a volumne-measuring medical instrument controller. *NLoops* is a fabricated example with nested while loops. For the *Fac* example, two different ways of partitioning the system was done. The first way is shown in Figure 4. For the *Ch* example, three different ways of partitioning the system was done. The second and third columns show the gate count for the unpartitioned and partitioned system. In the partitioned column, the gate count for the individual modules are further broken down. The fourth column shows the number of times each part is called via the FunctionBus. For example, for the *Fac1* example, part one is called one time and parts two and three are called *n* times to denote that it is dependent on the input value. The last column shows whether there is a loop in that part and if so how many times it loops around. Again, *n* denotes that it is dependent on the input value.

The results are summarized in Table 2. The table shows the gate count, the execution time, the average and total power used as a ratio of the partitioned to unpartitioned examples. Columns six and seven are the absolute power for the partitioned system. Since the switching frequency is dependent on the inputs, the results shown are averages from several runs. The average power column shows the average power used in one clock period and the total power is the total power used for the entire execution.

In all cases, both the average and total power is reduced. The reduction in average power ranges from 27% to as much as 78% as in the case for *Fac2*. The reduction in total power ranges from 5% to 66%. Even with this drastic power reduction, the tradeoff is not too terrible. The gate count is increased by 32% on average. If we consider only the best partitioning for the *Fac* and *Ch* examples, then the gate count is only increased by 21% on average and 49% in the worse case. Execution time is increased by 53% in the worse case and only 22% on average. The reason for the size and execution time increase is because of the extra communication overhead. The execution time overhead should be less than reported because we used a fixed clock in the calculation, but the critical path for the modules are actually less.

It is interesting to note that in the *Fac2* example, the gate count for the partitioned system is increased by only 4%. This increase is very insignificant. In fact, a decreased was observed in [18]. A possible reason is that NSYN can optimize a small design much better than a large design.

For both the *Fac* and *Ch* examples, we see that power is further reduced just by adding an extra part. For the *Fac* example, as much as 10% total power was further reduced. By adding an extra part, we have increased the total size because of the communication overhead. However, the individual size of each part is smaller. This causes the switching activity to be even more localized and confined within fewer gates. Thus, a reduction in the power is seen.

From this result, we can see that it is better to have more smaller parts rather than bigger but less parts, as long as the size and performance overhead is kept within the limit. The rationale is that smaller parts will have less switching activities when the part is active. The rest of the dormant parts do not contribute any dynamic power due to capacitive charging and discharging because there are no switching activities. Of course, more parts mean more communication on the FunctionBus and longer execution time, but this overhead is small, as can be seen from the results.

The results do take into consideration of the fact that the FunctionBus capacitance for communications between parts are larger than internal capacitance. In our power calculation, we have used a bus capacitance that is four times the internal capacitance. Early investigation of synthesis for low power using gated clock to memory units resulted in small power reductions when compared to our functional partitioning technique. However, further investigation on the comparisons will be performed.

Figure 6 and Figure 7 show a plot of the switching activities for the unpartitioned and partitioned *Fac1* example as shown in Figure 3 and Figure 4 respectively. Here we actually see the decreased in switching activities as a result of the partitioning. In the unpartitioned case, Figure 6, the number of toggles can go as high as 2000 in

Table 1: Example size in number of gates.

Ex	gate count		FB	loops
	Unpart	Part= P1+P2+P3+...	calls	
Fac1	15251	17172=8918+3051+5203	1/n/n	1/2/2
Fac2	15251	15802=9260+1349+1695+3498	1/n/n/n	1/1/1/1
Ch 1	19766	32460=14965+3679+13816	1/1/1	4/3/n
Ch 2	19766	34111=11116+5499+3679+13817	1/1/1/1	1/3/3/n
Ch 3	19766	29551=11694+2345+1695+13817	1/3/3/1	1/1/1/n
Diffeq	11487	12245=1340+10905	1/12	1/n
Volsyn	11193	13163=10798+2365	1/n	n/n
NLoops	2622	3307=1811+1496	1/n	n/n

Table 2: Comparison of power consumption.

Ex	ratio of partitioned / unpartitioned				absolute partitioned	
	gate count (ratio)	execution time (ratio)	average power (ratio)	total power (ratio)	average power (µW)	total power (µJ)
Fac1	1.13	1.23	0.36	0.44	22.65	2242.95
Fac2	1.04	1.53	0.22	0.34	13.84	1694.67
Ch 1	1.64	1.02	0.55	0.56	16.03	3604.99
Ch 2	1.73	1.05	0.52	0.55	15.16	3352.01
Ch 3	1.49	1.16	0.45	0.52	13.12	3035.78
Diffeq	1.07	1.30	0.73	0.95	40.76	7771.52
Volsyn	1.18	1.24	0.71	0.88	10.38	2587.22
NLoops	1.26	1.20	0.41	0.49	3.70	2157.51
Average	1.32	1.22	0.49	0.59	16.96	3305.83

a clock cycle. The total power consumption is 5097 µJoules. Whereas, in the partitioned case, Figure 7, the maximum number of toggles in a clock cycle is only 650. The total power consumption for this case is only 2,243 µJoules, a 56% reduction.

In the partitioned graph, Figure 7, parts two and three are called twice each. Part one is the main controlling module which is active only during the communication with the other two parts. We see that while part two is active, parts one and three does not have any switching activities whatsoever. Similarly, when part three is active, parts one and two are completely inactive. The only time when all three parts have switching activites is when the communication is occuring on the FunctionBus.

This confirms the idea that our FunctionBus model does completely isolate the switching activities within the parts.

5. Conclusions

We have considered the problem of reducing power consumption at the behavioral level. Power consumption is reduced when the amount of switching activities for the circuit is reduced. We have presented a functional partitioning model where the behavioral level description of a system is partitioned into several functional parts. These functional parts are synthesized individually and executed sequentially. Since these parts are mutually exclusive, each containing its own control and datapath, switching activities can be isolated to just one part at a time. Only one part needs to be active at a time and the inputs to that part does not affect the inputs to the other parts, the rest of the parts are dormant and do not contribute any dynamic power due to capacitive charging and discharging. The FunctionBus was used to isolate the switching activities within the parts. Thus, the total amount of switching activities for the entire circuit is reduced and power is saved.

We have shown that the total power reduction for the partitioned system using our FunctionBus model is on average 41% less than the unpartitioned counterpart. Furthermore, the tradeoff in the system size is increased by 32% on average and the execution time is increased by 22% on average. Functional partitioning has already been shown to solve problems of satisfying I/O constraints, reducing synthesis runtime, and hardware/software partitioning. These results demonstrate the additional power reduction benefits obtainable by introducing automated functional partitioning into a synthesis environment.

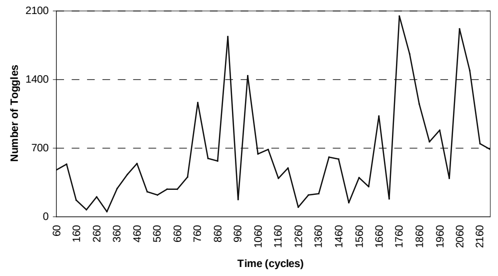

Figure 7: Plot of switching activities for the *Fac1* unpartitioned example. Total power is 5097 µJoules.

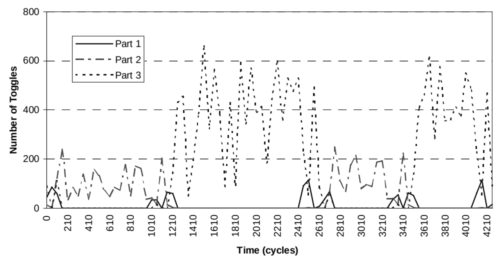

Figure 8: Plot of switching activities for the *Fac1* partitioned example. Total power is 2243 µJoules.

References

[1] A. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, & R. Brodersen, “Optimizing Power Using Transformations,” *IEEE Trans. on CAD of IC and Systems*, pp. 12-31, January 1995.

[2] A. Raghunathan, & N. Jha, “ILP formulation for Low Power Based on Minimizing Switched Capacitance during Data Path Allocation,” *Proc. of the Int. Sym. on Circuits & Systems*, 1995.

[3] Vaishnav & Pedram, “Delay Optimal Partitioning Targeting Low Power VLSI Circuits,” *Intl. Conf. on CAD*, pp. 638-643, 1995.

[4] L. Benini, P. Vuillod, G. De Micheli & C. Coelho, “Synthesis of Low-Power Selectively-Clocked Systems from High-Level Specification,” *International Symposium on System Synthesis*, pp. 57-63, Nov. 1996.

[5] T. Lang, E. Musoll, & J. Cortadella. “Reducing Energy Consumption of Flip-Flops,” *Universitat Politècnica de Catalunya DAC Technical Report 4*, 1996.

[6] J. Chan and M. Pedram, “Register allocation and binding for low power,” *Proceedings of the Design Automation Conference*, pp. 29-35, June 1995.

[7] T. Lang, E. Musoll, & J. Cortadella. “Redundant Adder for Reduced Output Transitions,” *Universitat Politècnica de Catalunya DAC Technical Report 22*, 1996.

[8] C. Su, C. Tsui, & A. Despain, “Low power architecture design and compilation techniques for high-performance processors,” *COMPCON 1994 Digest of Technical Papers*, pp. 489-498, 1994.

[9] M. Stan & W. Burleson, “Limited-weight codes for Low Power I/O,” *Proc. of the 1994 Int. Workshop on Low-Power Design*, pp. 209-214, 1994,

[10] A. Chandrakasan, S. Sheng & R. Brodersen, “Low-power techniques for portable real-time DSP applications,” *Proceedings of VLSI Design*, 1992.

[11] R. Martin & J. Knight, “Power-profiler: optimizing ASICs power consumption at the behavioral level,” *Proceedings of the Design Automation Conference*, pp. 42-47, 1995.

[12] V. Tiwari, S. Malik & A. Wolfe, “Compilation Techniques for Low Energy: an overview,” *Proc. of IEEE Symposium on Low Power Electronics*, pp. 38-39, October 1994.

[13] F. Vahid, T. Le, & Y.C. Hsu, “A Comparison of Functional and Structural Partitioning,” *International Symposium on System Synthesis*, pp. 121-126, November 1996.

[14] A. Chandrakasan, T. Sheng, & R. Brodersen, “Low Power CMOS Digital Design,” *Journal of Solid State Circuits*, Vol. 27, No. 4, pp. 473-484, April 1992.

[15] F. Johannes, “Partitioning of VLSI Circuits and Systems,” *Proceedings of the Design Automation Conference*, pp. 83-87, 1996.

[16] F. Vahid, “Procedure Exlining: A Transformation for Improved System and Behavioral Synthesis,” *International Symposium on System Synthesis*, pp. 84-89, September 1995.

[17] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and design of embedded systems*, Prentice Hall, 1994.

[18] F. Vahid, “I/O and Performance Tradeoffs with the FunctionBus during Multi-FPGA Partitioning,” *Int. Sym. on FPGA*, pp. 27-34, February, 1997.

[19] Y. Hsu, T. Liu, F. Tsai, S. Lin, & C. Yu, “Digital design from concept to prototype in hours,” in *Asia-Pacific Conference on Circuits and Systems*, December 1994.

[20] PureSpeed Simulator, An event-driven simulator from FrontLine Design Automation, Inc.