Stacked FSMD: A Power Efficient Micro-Architecture for High Level Synthesis

Khushwinder Jasrotia, Jianwen Zhu
Department of Electrical and Computer Engineering
University of Toronto, Ontario M5S 3G4, Canada
{ksj,jzhu}@eecg.toronto.edu

Abstract— **In this paper, we argue that the classic micro-architecture model, namely finite state machine with datapath (FSMD), cannot handle procedure abstraction needed by complex applications. This presents one of the major obstacles for the adoption of high-level synthesis technology in practice. We propose a simple extension of FSMD, called stacked FSMD, which mimics the procedure linkage concepts in software. We demonstrate that the new micro-architecture can not only fully support procedure calls, but also be made power efficient by a technique called region-based partitioning, which can be applied directly at the behavioral level with the assistance of simple metric evaluated at the behavioral level. With a rigorous experimental procedure, we show that the controller power saving achieved can range from 12% to 68% with modest overhead in area.**

I. Introduction

With transistor densities of over one hundred million gates and clock frequencies in the gigahertz range, digital systems today are truly complex. Systems of such complexity are impossible to design at the transistor and gate level, and increasingly difficult to design at the register-transfer level (RTL). It is therefore natural to raise design automation to higher levels of abstraction.

High-level synthesis (HLS) is an automated refining process from an abstract description of a digital design at the algorithm level to detailed implementation at the RTL level [1], [2]. Despite the intensive research efforts invested in the last decade, the notion of HLS unfortunately remains in the hands of academia and a few EDA companies, rather than the design community. One of the reasons for such a reluctance is that the size or complexity of the application that current academic or commercial tools can accept is too small to justify the departure from the mature RTL design flow.

More specifically, classic HLS tools accept only behavioral VHDL or Verilog, which lack the expressive power of traditional programming languages, such as C/C++ and Java, to help combat complexity. One of key constructs lacking is *procedure abstraction*, where functionalities are encapsulated in procedures, also known as subroutines or functions. Even though VHDL/Verilog have modest support for procedures, chances are rare for system designers to write complex applications using these languages. Furthermore, the classic HLS tools assume a micro-architecture of FSMD, as formalized in [1], which contains a controller and a datapath. The monolithic nature of FSMD controller implies that only one procedure can be supported. When synthesizing a complex application with multiple procedures, HLS tools have to resort to *inlining*, or the expansion of callee at every call site, resulting poor designs in terms of area and power consumption.

In this paper, we make several new contributions. First, we propose an extension of the FSMD micro-architecture, called *stacked FSMD*, which contains a set of controllers, each corresponding to a procedure; a hardware stack to support procedure linkage; as well as a *shared* datapath. Second, we show that by applying a technique called *region-based partitioning* at the behavioral level to redefine the procedure boundary, the stacked FSMD micro-architecture can be made power-efficient. Third, we propose a simple metric, called the *behavioral power index*, which can be evaluated at the behavioral level, to help assist the partitioning decision. We experimentally demonstrate that this metric has *high fidelity*, or correlates extremely well with the real power figure measured at the gate level.

The rest of the paper is organized as follows. Section II discusses related work. Section III provides a detailed description of the proposed micro-architecture. Section IV discusses the region based partitioning method as well as the behavioral power index. We give experimental result in Section V to demonstrate the power efficiency of the proposed method as well as the fidelity of the proposed metric. We conclude the paper in Section VI with suggestions for future work.

II. Related Work

Several previous work have addressed the support of procedure abstraction. In [3], [4], procedures are synthesized into independent hardware modules, with the calling mechanism implemented by introducing a wait state in the control unit of the calling module. A more comprehensive treatment is presented in [5], where procedures with fixed delay and variable delay are further distinguished. In [6], a common bus is used to transfer address and parameter information between them. While useful for a wide range of practical problems, these early efforts cannot support nested or recursive calls. Furthermore, while always mutual exclusive in time, different procedures always use separated datapaths, resulting unnecessary waste of hardware resources.

A comprehensive survey of power reduction techniques is out of the scope of this paper. The most relevant to ours are [7] and [8], where a monolithic controller or FSMD is partitioned into smaller ones, thereby achieving similar power reduction effect as ours. The key difference we make in our method is that we directly exploit the high-level loop information at the behavioral level, whereas [7] and [8] have to recover the loop information by expensive analysis.

III. The Stacked FSMD Microarchitecture

The SFSMD model extends the classic FSMD microarchitecture model to include support for procedure abstraction. Specifically, this model supports sequential procedures (in the sense of a sequential programming languages like *C*).

In the SFSMD model, procedures are implemented as separate controllers (FSMs) sharing a common datapath unit. The key feature is a special *stack controller* that controls the interactions between the controllers and also allows the datapath unit to be shared. The structure of the SFSMD model is shown in Figure 1. All components share the same clock. Note that the figure omits the external input and output signals of the controllers and datapaths for clarification purposes only.

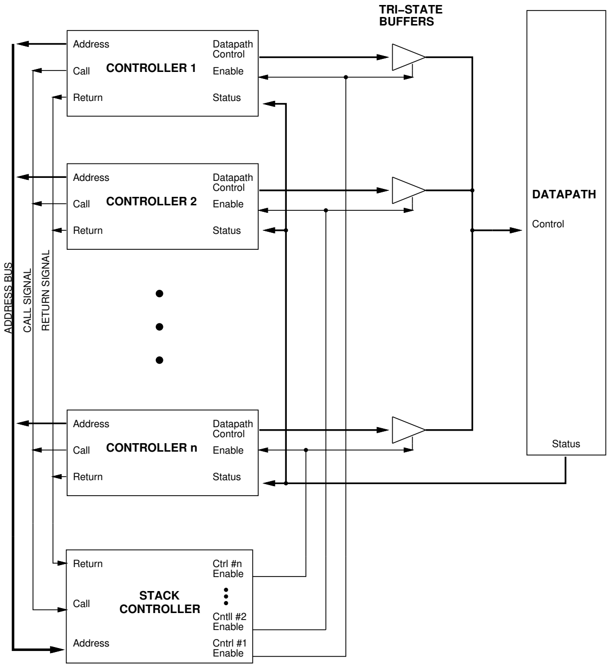

Fig. 1. Stacked FSMD Model

The stack controller is used to handle procedure calls and returns in an analogous fashion to how stack mechanisms are used for procedure linkage in microprocessors. The value stored on the top of the stack represents the address of the currently active FSM. This allows the stack controller to activate that particular FSM and halt the rest. Procedure calls are performed by pushing the address of the called FSM onto the stack, and returns are made by popping the stack so that control can be passed back to the caller FSM.

The stack controller controls the activation of the FSMs through the use of enable signals. It decodes the address value at the top of its stack to generate a dedicated enable signal for each FSM. Each enable signal is connected to the *enable* inputs of the state-registers of its corresponding FSM. When the enable signal is asserted, the FSM is able to operate normally, but when the signal is negated, the FSM is halted at the current state since its state-registers are unable to update. Only one enable signal is active at any time due to the sequential nature of procedures.

The enable signals are also used to control access to the datapath unit. Each FSM's datapath control signals are tri-state buffered to the inputs of the datapath unit. The enable signals are used to activate the tri-state buffer for the corresponding FSM so that the datapath components can be accessed. Again, due to the sequential nature of the procedures, only one set of datapath control signal is always active and driving the datapath components. The sharing of the datapath can also be implemented by a multiplexor, in which case encoded values of the enable signals are needed to drive the select inputs of the multiplexor. All controllers have access to the *status* signals of the datapath. Unshared datapath components can be directly controlled by their corresponding FSMs - these connections are not indicated in Figure 1.

The FSMs transmit data to the Stack Controller via a shared unidirectional bus consisting of an *address bus*, a *call* line, and a *return* line. If the design consists of N FSMs, then *address bus* consists of $\lceil log_2 N \rceil$ lines, and is used to transfer the address of the called FSM to the stack controller. *Call* is a single line used to indicate a valid address on the *address bus* for a procedure call. *Return* is a single line used to indicate a procedure return. For procedure returns, only the *return* signal is used, the address lines are not driven. Only one FSM controls the bus at a time, with the others providing high-impedance values. For the *call* and *return* signals, external pull-up or downs can be used to provide valid logic levels for the inputs of the stack controller when these signals are not being driven. The stack controller generates N dedicated *enable* signals, one for each FSM and its corresponding datapath control tri-state buffer.

Support for recursion would require a modification: A calling FSM would need to additionally stack the identity of the control-step succeeding a procedure call and the the identity of the control-step in the called procedure. The stack-controller can be used to stack this information or a memory device can be instantiated in the datapath to implement the stack. Obviously, support for recursion makes the design more complicated, but it is nevertheless possible in the SFSMD model.

Since procedures may require the passing and returning of parameters, a set of input and output registers can be defined for each procedure. Prior to each procedure call, an additional control step may be required to copy the actual parameters to the input register, and another control step to receive any outputs stored by the procedure. In order to support recursion, the parameters will need to be stacked in memory.

IV. Region Based Partitioning

A. Partitioning Methodology

Due to the prevalence of loops in a program, most of the execution time is spent computing a small number of operations. By extracting such loops and implementing them as separate controllers in the SFSMD model, significant power savings can be achieved. This is because the controller for each loop is smaller than the single controller implementing the entire system, and since only one controller is running at any given time, the remaining ones can be deactivated, thus saving power [8].

We propose a partitioning scheme that operates at the behavioral level prior to synthesis. The partitioner redefines the procedure boundaries for the original specification by first *exlining* loops. Exlining can be defined as the inverse of inlining in which a sequence of statements are replaced by procedure calls [9]. Each exlined loop is then implemented as a separate controller in the SFSMD model.

If the loops account for a major portion of the overall execution time, significant power savings can be achieved since overall switching activity is reduced to the localized activities of each smaller individual controller for the loops. The ability to localize the controller activity is inherent in the SFSMD model where each inactive controller can be disabled by stopping its clock.

The partitioning scheme can be generalized by defining it as a transformation of the original design via a series of exlining *and* inlining operations. Inlining helps improve parallel-level optimizations by incorporating acyclic instructions from existing procedures into the main body. Inlining also exposes loops within the procedures. The result is a design where the main body is composed entirely of either, a series of acyclic instructions, or procedure calls to exlined loops. This is illustrated in Figure 2.

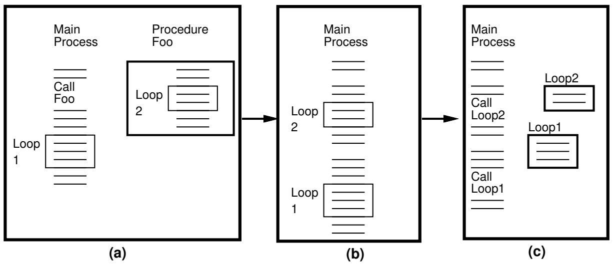

Fig. 2. Inlining and Exlining Transformations. (a) Original Specification, (b) After inlining *foo*, (c) After loop exlining.

The extraction of loops introduces extra power overhead due to inter-procedural communication between the controllers and due to loss of control-step optimizations across procedure boundaries. Communication overhead occurs due to extra states [1] that need to be inserted to implement procedure calls and returns, switching activities involved in the stack-controller, and the activities of tri-state buffers used for sharing the datapath. This means that loop exlining cannot be done indiscriminantly since high-power overhead loops can result in a design with increased power requirements. What is required is a method for selecting a subset of loops for exlining, such that the reduction of of power far outweighs the power increase due to communication [8].

B. Behavioral Power-Index

The loops from a behavioral specification can be extracted for SFSMD implementation in any number of ways. If there are n loops at the root-level, then there are 2^n possible ways of partitioning the code - nested loops further increase this value.

The partitions can be represented by a tree structure as indicated in Figure 3. Each node in the tree, except for the root, corresponds to a loop region. A child of a node corresponds to a nested loop within the node. The parent of a node corresponds either to an outer loop or the main process in which the node resides. The root of the tree always corresponds to the main process. Each edge can be weighted with the accumulated number of iterations of the parent node. This represents the total number of calls made to its child.

[1] Due to the shared datapath, extra states are not required for parameter passing since all controllers have access to the same variables.

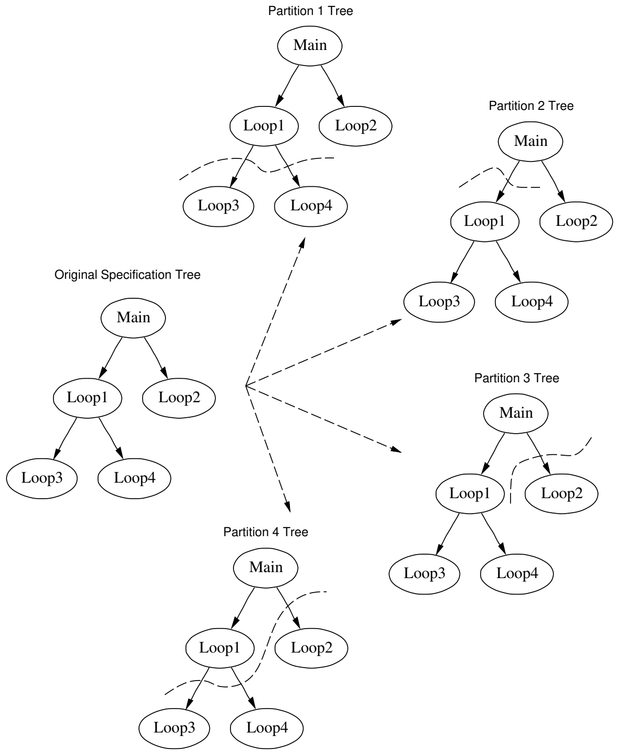

Fig. 3. Tree Representation of Partitioning

A partition can be simply considered as a "cut" across the nodes of the tree. The cut can be made in any direction - sideways, vertically, diagonally, etc, so there are exponential number of partitions possible. The *regions* of a partition are formed by splitting the tree at points where the cut intersects the edges of the nodes. At each split point, a call instruction has to be added in the parent loop for each child. The total number of calls made to each child is the same as the weight of the edge that was cut.

Assuming a set of candidate partitions exist, the problem then is of selecting one that maximizes power reduction. Formally, a partition is defined as a collection of k regions R.

$P(S) = \{R_1, R_2, \ldots, R_k\}$ such that $R_i \cap R_j = \varnothing$ for $i \neq j$, and $\bigcup_{i=1}^{k} R_i = S$.
Where R_i consists of the set of basic-blocks corresponding to an exlined loop or main process.

For each partition P_j, a power index is defined as:

$$\mathscr{P} = \sum_{i=1}^{k} |States(R_i)| \cdot Cycles(R_i) + K \cdot Calls(R_i) \quad (1)$$

$$\text{where } R_i \in P_j$$

$|States(R_i)|$ is the number of control-steps used in the FSM of the controller for region i. It is used to represent the relative "power complexity" of the region. Generating control-steps requires that scheduling be performed on the basic-blocks of the region. Scheduling is typically performed at a later stage during high-level synthesis, and is strongly intertwined with with the allocation phase. However, fast scheduling can be performed at the behavioral level which does not require allocation. For example, "as soon as possible" (ASAP) scheduling can be used to get a quick estimate of the number of control-steps. It does not take resource constraints into consideration (assumes unlimited resources) and follows a simple rule that an assignment to a variable can execute only after the values of its operands have been computed. $Cycles(R_i)$ is the accumulated number of control-cycles spent in region i and can be determined by simulating the original specification. $Calls(R_i)$ represents the number of calls made to region i, and K is a constant for controlling the relative weight of the contribution. $Calls(R_i)$ can be computed by simulating the original specification and measuring the frequency of transitions to the region.

Since the regions are active only one at a time, the index expresses the total power of a partition as a sum of the "energy" contributions of the individual regions. The first term represents the energy expended by the controller of the region, and the second term adds the energy for communication. The partition with the lowest power index should be chosen for SFSMD implementation.

V. Experimental Result

A. Experiment Procedure

The region based partitioning was applied to various *C* benchmark kernels in order to evaluate its impact on power consumption. The benchmarks used were based on the Livermore kernels [10]. These kernels were chosen because their loop-intensive nature made them appropriate for loop extraction, and also because they could be synthesized in reasonable amounts of time.

As discussed earlier, the number of partitions of a program is exponential in the number of loops present. Since we were manually extracting and processing the partitions, the search space had to be decreased due to time constraints. This was done by limiting the number of partitions of a kernel to the maximum loop depth in the specification. For example, in a design with three root-level loops, only two partitions would be considered: The first partition would consist of only one *region* - the original specification without exlining. The second partition would be composed of four *regions* consisting of three exlined loops along with the main procedure which calls the loops.

The partitioning strategy can also be visualized by using the tree-representation that was described in the previous chapter. The partitions are formed by making horizontal cuts across the tree. Due to the tree-structure, horizontal cuts allow the widest range of region-granularity to be exercised in the shortest number of steps. Therefore, for each kernel, power figures are available for partitions with regions ranging from the highest granularity (unpartitioned) all the way to very low (the deepest level nested loop have been exlined).

This strategy is effective in finding low power partitions for designs in which most of the execution time is spent in the inner most loops. This is because the horizontal cuts ensure that the deepest nested level loops will be ultimately extracted and implemented as separate controllers. Since these controllers represent regions of the finest granularity, their power consumption will be very low.

The power of a partitioned design was calculated by summing the energy contributions of the individual regions and dividing it by the total time of the simulation. The equation used to calculate the power for each partition is shown:

$$Power = \frac{1}{Cycles_{Tot}} \sum_{i=1}^{k} Cycles_{Ri} \cdot Pwr_{Ri} + Pwr_{Stack} \cdot Calls_{Ri} \tag{2}$$

where Ri is a region of the partition

$Cycles_{Tot}$: This is the total number of cycles used to complete the simulation of the unpartitioned design. Cycle overhead for regions calls and returns were added for the different partitions.

$Cycles_{Ri}$: This is the accumulated number of cycles spent executing region i. This was obtained from simulations of the unpartitioned design. Cycle overhead for region calls and returns were added for the different partitions.

Pwr_{Ri}: This is the intrinsic power of the controller associated with region i. This value was obtained by using Synopsys *Power Analyzer* to report power for the synthesized controller of the region. *Power Analyzer* calculates power based on the switching activity of the nets in the design.

Pwr_{Stack}: This is the intrinsic power of the Stack controller. It was obtained by synthesizing a Stack controller and measuring its power with *Power Analyzer*.

$Calls_{Ri}$: This is the number of calls made to region i. This was obtained by profiling the kernels.

The value in the summation represents the total energy contributed by each region i. The first term represents the energy expended exclusively by the region's controller, and the second term adds the energy overhead of calling the region. Adding up the energy contributions of all the regions and dividing it by the total number of cycles gives the effective power of the partition. Notice that for the unpartitioned design, this equation is simply equal to the power of the unpartitioned controller.

B. Power Reduction Results

The power results for partition levels 1, 2, and 3 are shown in Tables I, II, and III, respectively. The first column lists the benchmark kernels used. The increase in execution time for each partition over the unpartitioned implementation is shown in the second column. The next two columns list the power and energy values of the partitions, and the last two columns show the power and energy decrease over the unpartitioned design. Partition 1 consists of designs in which root level loops have been extracted for separate implementation. Of the twenty-three benchmarks[2], only thirteen had nested loops, and were hence, able to be partitioned into level 2. Of these, five kernels had another level of loop nesting, and were able to be partitioned into level 3. In this partition, the deepest nested loops were extracted and implemented as separate controllers.

[2] LL17_int was not synthesized since the entire kernel consists of just one loop and so partitioning cannot be applied.

Benchmark	Partition 1				
	Time Increase (%)	Power $(mWatt)$	Energy $(nJoule)$	Power Decrease (%)	Energy Decrease (%)
LL1_int	4.6	1.1	10.3	13.2	9.2
LL2_int	0.7	2.7	154.7	12.0	11.4
LL3_int	7.0	1.2	7.1	19.2	13.6
LL4_int	1.7	2.0	47.7	14.6	13.1
LL5_int	3.4	1.2	13.9	16.5	13.6
LL6_int	1.1	1.9	69.9	12.3	11.4
LL7_int	2.1	2.0	38.4	33.6	32.2
LL8_int	1.8	6.9	511.6	20.0	18.6
LL9_int	5.0	2.8	47.0	19.6	15.6
LL10_int	1.6	3.3	85.0	31.4	30.3
LL11_int	6.3	1.0	6.4	31.9	27.6
LL12_int	6.5	1.0	6.4	33.4	29.1
LL13_int	1.1	2.7	104.8	32.9	32.1
LL14_int	0.8	1.5	85.0	42.7	42.2
LL15_int	1.3	2.3	73.0	48.7	48.1
LL16_int	6.0	2.5	26.2	56.7	54.1
LL18_int	0.3	6.5	1028.0	13.0	12.8
LL19_int	1.5	1.9	49.4	14.2	12.9
LL20_int	0.3	3.1	469.9	16.0	15.8
LL21_int	0.9	2.0	99.1	24.5	23.8
LL22_int	1.7	1.1	25.9	45.3	44.4
LL23_int	1.7	3.4	82.7	20.6	19.3
LL24_int	5.0	1.5	12.5	29.9	26.4
Average	2.7	2.4	132.8	26.2	24.2

TABLE I

CONTROLLER POWER RESULTS FOR PARTITION LEVEL 1

Benchmark	Partition 2				
	Time Increase (%)	Power $(mWatt)$	Energy $(nJoule)$	Power Decrease (%)	Energy Decrease (%)
LL2_int	1.4	2.4	138.4	21.8	20.7
LL4_int	4.8	1.9	457.5	20.5	16.7
LL6_int	1.1	1.6	57.8	27.5	26.7
LL8_int	2.6	6.3	471.4	26.9	25.0
LL14_int	3.8	1.7	102.6	32.8	30.3
LL15_int	2.6	2.1	66.4	53.3	52.7
LL18_int	1.8	2.4	387.8	67.7	67.1
LL19_int	5.4	1.6	44.6	25.4	21.4
LL20_int	0.5	2.8	426.6	24.0	23.6
LL21_int	1.3	1.9	94.9	28.0	27.1
LL22_int	2.5	1.0	24.1	49.5	48.2
LL23_int	2.1	3.3	79.4	24.2	22.6
LL24_int	7.5	1.3	10.9	40.4	36.0
Average	2.9	2.3	150.0	34.0	32.1

TABLE II

CONTROLLER POWER RESULTS FOR PARTITION LEVEL 2

Benchmark	Partition 3				
	Time Increase (%)	Power $(mWatt)$	Energy $(nJoule)$	Power Decrease (%)	Energy Decrease (%)
LL2_int	3.9	1.5	87.9	51.5	49.7
LL6_int	5.0	1.4	51.9	37.3	34.2
LL15_int	2.5	2.1	67.5	53.1	51.9
LL21_int	1.7	1.7	84.4	36.2	35.1
LL23_int	2.5	3.0	72.9	30.6	28.9
Average	3.1	1.9	72.9	41.8	40.0

TABLE III

CONTROLLER POWER RESULTS FOR PARTITION LEVEL 3

The reduction in power consumption across all partitions ranged between 12.0% and 67.7%, with a corresponding reduction in energy ranging between 11.4% and 67.1%. The average power reduction for partition levels 1, 2, and 3, were 26.2%, 34.0%, and 41.8%, respectively, while the energy reduction for the partitions were 24.2%, 32.1%, and 40.0%. Due to the low cycle-time overhead for *call* and *return* operations, the energy reduction for each benchmark was very close to its power reduction (averaging within 2%). As per the results, power reduction improves with increased partitioning levels. This makes sense because the majority of execution time for the kernels was spent in the inner-most loops. Since these loops represent the finest-grained regions, they have the smallest controllers and consequently consume the least amount of energy. Furthermore, the power contribution of the stack controller was found to be very small, so the communication overhead for the higher partition were insignificant.

It is observed that partition level 3 out-performs partition level 2 for majority of the benchmarks. One exception is benchmark *LL15_int*, in which partition level 2 results are slightly better than level 3. This is a situation in which the power figure reported for a larger circuit (partition 2) is lower than the power reported for a smaller one (partition 3). This anomaly is generated because the power values happen to be outside the noise-margins of the power measurement technique.

C. Behavioral Power Index Fidelity

The fidelity of the power-index is also evaluated. We define the fidelity as a measure of how well the calculated power index value of a partition correlates with its actual power. This is done by comparing the power plots of the partitions against the power-index plots as indicated in Figure 4. Fidelity is checked by confirming that for each benchmark, the ordering of the relative increase or decrease of the partitions in the index matches the ordering of the partitions in the power plots.

In the majority of the cases, the results of the power-index matched the partitioned power results. An exception was *LL15_int*, for which the power index choose the wrong partition due to the anomaly mentioned earlier for this particular benchmark. Higher values for K were also tested for the power-index equation with no significant reduction in fidelity. This was due to the low power consumption of the stack-controller and the relatively few number of calls that were made.

D. Area Results

The area overhead plots for the different partitions are shown in Figure 5, Figure 6 and Figure 7. The average area increase for partitions 1, 2, and 3 over the unpartitioned design are 5.96%, 6.70%, and 3.96%, respectively.

Interestingly enough, partitions for benchmarks *LL8_int, LL14_int, LL15_int, LL16_int, LL20_int,* and *LL22_int*, result in area decrease over the unpartitioned design. One possible explanation for this phenomenon is that since the synthesis tool is dealing with smaller designs, it can to a better job in terms of resource sharing due to the smaller search space it has to deal with.

VI. CONCLUSION

We draw three main conclusions. First, the SFSMD model provides a good basis for procedure abstraction. Second, by extracting loops with high execution count, the region-based partitioning technique can help to reduce controller power. Third,

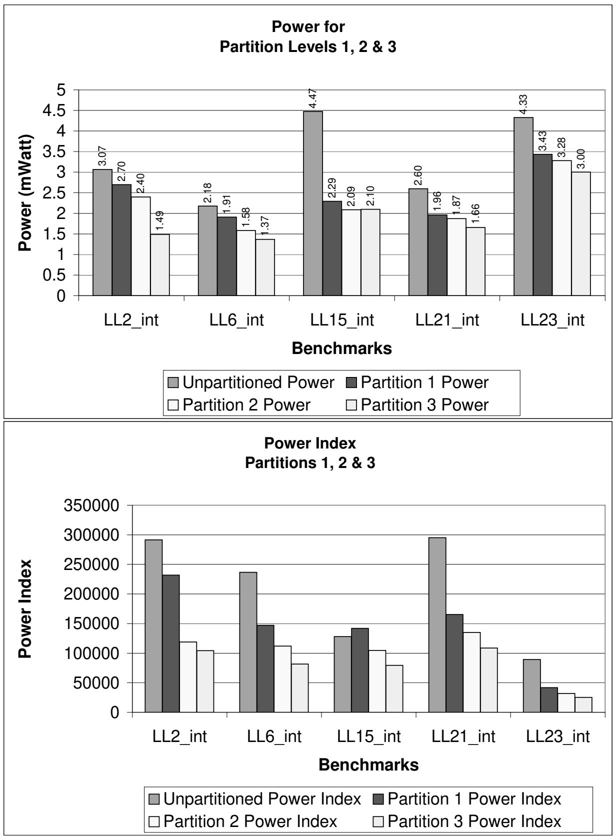

Fig. 4. Power measurement vs Behavioral Power Index for Partition Levels 1,2 and 3

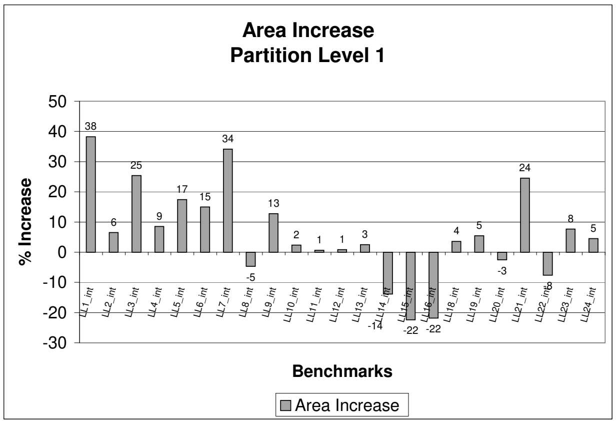

Fig. 5. Area Overhead for Partition Level 1

due to the strong correlation between the power-index values and the actual measured power, the proposed behavioral power-index can be used to effectively guide the partitioning decisions of a high-level partitioning tool.

There is a lot of scope for future work in this area. An important step would be the integration of the SFSMD model into a synthesis engine. This would automate the transformation of procedural descriptions into an SFSMD model, thereby enabling more comprehensive studies of this model to be performed. Tools could be developed to directly estimate the design area, speed, and power of this implementation style, assisting the designers in choosing the best design option. Similarly, a tool can be

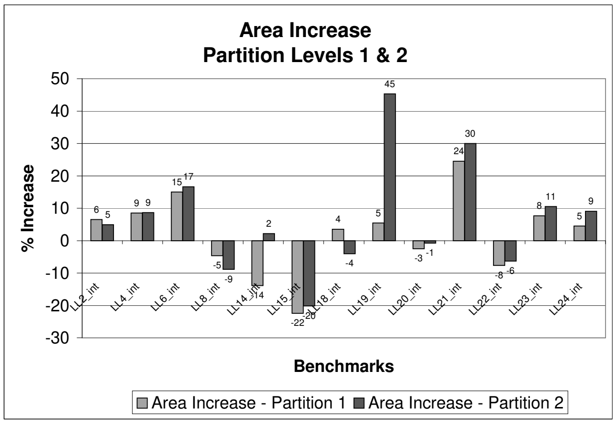

Fig. 6. Area Overhead for Partition Levels 1 and 2

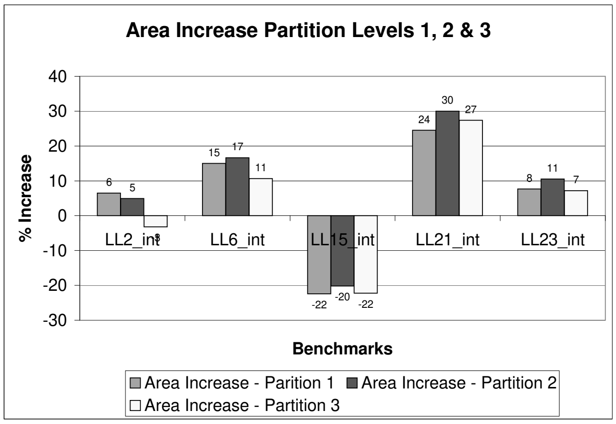

Fig. 7. Area Overhead for Partition Levels 1, 2 and 3

developed to perform loop region based partitioning. Such a tool would partition the code into a power optimal configuration before forwarding it to the synthesis engine for SFSMD implementation.

REFERENCES

[1] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

[2] G. De Micheli, *Synthesis And Optimization of Digital Circuits*, McGraw Hill, 1994.

[3] R. Camposano, L. Saunders, and R. Tabet, "VHDL as input for high level synthesis," *IEEE Design and Test of computers*, pp. 43–49, March 1991.

[4] R. Camposano and J. van Eijndhoven, "Partitioning a Design in Structural Synthesis," *Proceedings of the International Conference on Computer Design*, 1987.

[5] L. Ramachandran, S. Narayan, F. Vahid, and D. Gajski, "Synthesis of Functions and Procedures in Behavioral VHDL," *Proceedings of the European Design Automation Conference*, 1993.

[6] F. Vahid, "I/O and Performance Tradeoffs with the FuctionBus during Multi-FPGA Partitioning," *International Symposium on FPGAs*, pp. 27–34, February 1997.

[7] L. Benini, P.Vuillod, G. De Micheli, and C. Coelho, "Synthesis of low power selectively-clocked systems from high-level specifications," *International Symposium on System Synthesis*, pp. 57–63, November 1996.

[8] F. Vahid, E. Hwang, and Y. Hsu, "FSMD Functional Partitioning for Low Power," *Design Automation and Test In Europe*, pp. 22–28, March 1999.

[9] F. Vahid, "Procedure Exlining: A New System-Level Specification Transformation," *European Design Automation Conference*, pp. 508–513, September 1995.

[10] LiverMore Benchmark WebPage, `http://parallel.ru/ftp/benchmarks/livermore/livermorec.c`.