Copyright © 2019 American Scientific Publishers
All rights reserved
Printed in the United States of America

Journal of
Low Power Electronics
Vol. 15, 388–409, 2019

A Power-Driven Stochastic-Deterministic Hierarchical High-Level Synthesis Framework for Module Selection, Scheduling and Binding

Xiuyan Zhang, Ouwen Shi, Jian Xu, and Shantanu Dutt*
Dept. of ECE, University of Illinois at Chicago, Chicago, IL, 60607, United States

(Received: 21 February 2018; Accepted: 7 August 2018)

We present a power-driven hierarchical framework for module/functional-unit selection, scheduling, and binding in high level synthesis. A significant aspect of algorithm design for large and complex problems is arriving at tradeoffs between quality of solution and timing complexity. Towards this end, we integrate an improved version of the very runtime-efficient list scheduling algorithm called modified list scheduling (MLS) with a power-driven simulated annealing (SA) algorithm for module selection. Our hierarchical framework efficiently explores the problem solution space by an extensive exploration of the power-driven module-selection solution space via SA, and for each module selection solution, uses MLS to obtain a scheduling and (integrated) binding (S&B) solution in which the binding is either a regular one (minimizing number of FUs and thus FU leakage power) or power-driven with mux/demux power considerations. This framework avoids the very runtime intensive exploration of both module selection and S&B within a conventional SA algorithm, but retains the basic prowess of SA by exploring only the important aspect of power-driven module-selection in a stochastic manner. The proposed hierarchical framework provides an average of 9.5% FU leakage power improvement over state of the art (approximate) algorithms that optimize only FU leakage power, and has a smaller runtime by factors of 2.5–3x. Further, compared to a sophisticated flat simulated annealing framework and an optimal 0/1-ILP formulation for total (dynamic and leakage) FU and architecture power optimization under latency constraints, PSA-MLS provides an improvement of 5.3–5.8% with a runtime advantage of 2x, and has an average optimality gap of only 4.7–4.8% with a significant runtime advantage of a factor of more than 1900, respectively.

IP: 198.54.106.172 On: Tue, 21 Jul 2020 01:01:50
Copyright: American Scientific Publishers
Delivered by Ingenta

Keywords: Power Optimization, High-Level Synthesis, Operation Scheduling, Module Selection, Operation Binding, Simulated Annealing, Dynamic Power, Leakage Power.

1. INTRODUCTION

Power optimization is one of the most important issues at all levels of the hardware design hierarchy. The reduction of semiconductor size and the increase of clock frequency allow more components to be integrated into a unit area, and this causes power consumption per chip to be increased. The two significant categories of power consumption are dynamic (DP) and leakage power (LP). According to the study in Ref. [1], leakage power takes up to 42% of total power. Thus, both leakage and dynamic power significantly affect total power consumption. In this paper, the target is the optimization of dynamic plus leakage power, i.e., total power (TP) optimization. In high-level synthesis (HLS), power optimization is one of the main objectives. Some recent surveys[2,3] regarding low power designs in HLS studied and categorized low power design techniques and algorithms proposed in the last two decades. Most existing power-driven design techniques in HLS for module selection can be categorized into exact/optimal-techniques, constructive-heuristic techniques, and stochastic-heuristic techniques. Furthermore, some post-HLS techniques like clock gating,[29] power gating,[30] and thermal-aware floorplanning[31] are also discussed in Ref. [2]. Since in this paper we focus on the problem of module selection, scheduling, and binding (MSB), once we generate a complete MSB solution, the aforementioned post-HLS techniques, which are largely independent of the MSB solution generation algorithm, can also be applied to our MSB solution for further optimization.

With the significant increase chip design sizes, exact algorithms, like 0/1 integer linear programming (0/1-ILP),[4] are not able to obtain optimal solutions within a reason-

*Author to whom correspondence should be addressed.
Email: dutt@uic.edu

1546-1998/2019/15/388/022 doi:10.1166/jolpe.2019.1584

able runtime. Thus, to efficiently and effectively explore a problem solution space is the prime importance for the optimization problems. As a result, a large number of well-studied heuristics have been developed that can efficiently handle the optimization problems for large-size designs.

Constructive-heuristic approaches, like ASAP, ALAP, list scheduling (LS),[5] and force-directed scheduling (FDS)[6] have high efficiency in obtaining reasonable scheduling and binding (S&B) results. For the objective of power optimization in module selection in HLS, there are different constructive-heuristic approaches that can be broadly categorized into using multiple supply voltage (multi-V_{dd}) or multiple threshold voltage (multi-V_{th}) assignments for functional units (FUs). The multi-V_{dd} approaches[7–13] are used to reduce to mainly dynamic power and the multi-V_{th} approaches,[14–17] are used to reduce leakage power.

For dynamic power (DP) optimization, the technique discussed in Ref. [7] is a network flow based dual-V_{dd} binding algorithm that aims to reduce switching activity on interconnects between FUs. Chang et al.[8] proposed a dynamic programming based algorithm. It provides optimal scheduling solutions for trees in pseudo-polynomial times, but sub-optimal for general directed acyclic graphs. A force-directed scheduling (FDS) algorithm based two-stage power optimization approach with multi-V_{dd} assignments was presented in Ref. [10]. In the first stage, it generates a S&B solution by a modified FDS algorithm to minimize the dynamic power. In the second stage, it investigates all operations with high-V_{dd} and tries to re-assign certain operations to low-V_{dd} to reduce power without increasing the number of FUs determined in the previous stage.

For leakage power (LP) optimization,[14] proposed a maximum weighted independent set (MWIS) heuristic with dual-V_{th}'s. MWIS replaces a set of FUs in an initial S&B solution from low-V_{th} to high-V_{th} to achieve LP reduction. However, the quality of solutions (QoS) of final results is correlated to the QoS of the initial synthesis solution, which limits the solution space. Wang et al.[15] presented a min-cut flow based technique with dual-V_{th} assignments. It initially assigns all operations with high-V_{th}, and the result that the corresponding latency of the design could be larger than a given latency constraint. In this case, the technique re-assigns certain operations to low-V_{th} to fix timing violations such that the resulting power increase is minimized.

There are some design approaches that involve multi-V_{dd} and multi-V_{th} techniques to reduce both DP and LP, which is equivalent to the goal of total power (TP) optimization in this paper. In Ref. [13], a system of difference constraints (SDC) based simultaneous scheduling and module selection technique is presented for power minimization, but there is no explicit specification about which type of power is being optimized. They have two different types of power minimization objectives: (i) One based on only operation power (which can thus be interpreted as FU DP for executing the operations). Also, they can obtain an optimal solution in polynomial time using linear programming (the SDC's) for this formulation due to the constraint matrix being totally unimodular, implying that linear programming yields integral solutions of schedule time variables. (ii) A second objective based only on FU power independent of operations executing on them (which can thus be interpreted as FU LP). However, for the FU LP formulation, there is no binding or estimation of the number of FUs of each speed/delay-power type needed within the SDC mathematical programming framework. Binding seems to be done using a separate technique from Ref. [10], possibly as a pre-processing step to generate non-overlapping execution range constraints for operations bound to the same FU that is needed in their SDC framework. This lack of simultaneous binding in the SDC framework implies a potential sub-optimality. Their system is integrated with Magma tools, for both, final power estimation, and for performing lower-level gate sizing (apparently to further improve FU LP). It is thus also unclear how much of the power optimization they obtain is due to their SDC-based technique and how much due to the Magma tool's gate sizing, and also whether they perform the same gate-sizing optimizations for other techniques like optimal ILP that they compare to (their results are 6% from optimal). Finally, they use an FU library for which explicit speed-power values are not given (only continuous speed-power plots are specified), and internal DFG benchmarks that are not publicly available. Due to these issues, it is not possible to compare our technique in an apples-to-apples manner to the SDC-based technique of Ref. [13]. Mohanty et al.[18] presented an ILP-based power optimization technique with multi-V_{dd}'s for TP minimization. The technique proposed in Ref. [9] is a two-stage power minimization approach. In the first stage, it minimizes DP with a min-cost-flow-based technique that assigns most operations to the low-V_{dd}. Then, it uses a simple power gating scheme to reduce LP at the post-HLS stage. Both Refs. [9] and [10] optimize power in multi-steps where the QoS of the second step is limited by the QoS of the initial step and thus, the QoS of the final solution is tied to the initial step. Hence, for constructive-heuristics, the QoS is limited by the lack of solution space exploration.

For the purpose of improving the combined module selection and S&B QoS, stochastic-heuristics, like genetic algorithm,[19] simulated annealing (SA),[20,24] and ant colony,[21] apply controlled random exploration of a large solution space which, however, leads to a dramatic increase in runtime.

To obtain good QoS coupled with efficient runtime, this paper proposes a hierarchical framework with the objective of total power (dynamic and leakage power) optimization under given latency constraints. This framework integrates a significantly improved version of the list scheduling

IP: 198.51.106.173 On: Tue, 21 Jul 2020 01:01:30
Copyright: American Scientific Publishers
Delivered by Ingenta

(called modified list scheduling or MLS) algorithm with a modified power-driven simulated annealing algorithm (called PSA) for the most important power-driven exploration, that of module selection. The PSA generated module selection is followed by a S&B solution from MLS that satisfies the module selection configuration. The main concepts of the proposed hierarchical frameworks are:

1. Stochastic- and constructive-heuristics have complementary characteristics. Stochastic-heuristics, as the explorer in this hierarchical framework to obtain partial (module selection) solutions, have a greater potential in exploring the solution space of designs than constructive-heuristics have. Then, constructive-heuristics efficiently generate and provide reasonably good complete solutions to evaluate the stochastic-heuristic's partial solutions to improve the convergence runtime while still achieving a good QoS.
2. This framework has a high flexibility of integrating different techniques at the two levels of this optimization hierarchy. On the other hand, this framework also has the capability to be applied to various applications and objectives.

The rest of the paper is organized as follows: Section 2 defines our problem, gives various power models and formulations, and discusses the motivation for performing module selection for power optimization. The proposed hierarchical framework is presented in Section 3. Section 4 discusses a competitive flat simulated-annealing based power-driven module selection, scheduling and binding algorithm to compare to our hierarchical framework. Finally, Sections 5 and 6 provide the experimental results and conclusions, respectively.

IP: 198.54.106.178 On: Tue, 21 Jul 2020 01:01:50
Copyright: American Scientific Publishers
Delivered by Ingenta

2. MOTIVATION AND BASIC FORMULATIONS

In this paper, our goal is total (dynamic + leakage) power optimization for the power consumed primarily by functional units and secondarily (as an option) by muxes and demuxes connected to them. We formulate our power minimization problem as the follows.

Input: (1) A data flow graph (DFG), $G(V, E)$, where V is the operation set of all operations and E is the arc set that represents the precedence dependency among all operations. An arc from operation u to v represents the fact that u is the *immediate predecessor* of v (u's output is one of v's input operands) and v is the *immediate successor* of u. We define $parent(u)$ as the set of immediate predecessors of u and $child(u)$ as the set of immediate successors of u. Moreover, we define $Pr(u)$ as the set of all predecessors of u, and $Su(u)$ as the set of all successors of u. (2) A multiple design-based or multiple-V_{dd}/V_{th} based functional-unit (FU) library *lib* which contains possibly multiple module instances or FUs with different speed-power characterizations for each functional type. (3) An upper bound latency constraint l_c.

Output: A module selection, scheduling, and binding (MSB) solution that minimizes either total power (TP) consumed internally by FUs, or internally by FUs and muxes plus demuxes connected to them, and satisfies the upper-bound latency constraint l_c and all data dependencies of $G(V, E)$. Thus, in this paper, by the term "quality of solution (QoS)," we will mean quality in terms of minimizing the aforementioned power metric for the case of interest (FU only or FU + mux + demux power) for constraint-satisfying solutions.

2.1. Power Model Preliminaries

Both dynamic and leakage power models are well known. As in Ref. [23], the dynamic power consumption of a logic gate is a function of switching probability (α), load capacitance (C_L), supply voltage (V_{dd}), and clock frequency (f), and is given by:

$$dp = \frac{1}{2} \cdot \alpha \cdot C_L \cdot V_{dd}^2 \cdot f \tag{1}$$

The leakage power consumption of a logic gate is a function of leakage current (I_{leak}), and supply voltage (V_{dd}), and is given by:

$$lp = I_{leak} V_{dd} \tag{2}$$

Based on the early Berkeley short-channel insulated-gate field effect transistor model,[27] the leakage current of an nMOS transistor when $V_{gs} = 0$ is:

$$I_{leak} = \mu_0 C_{ox} \left(\frac{W}{L}\right) V_t e^{1.8} e^{(V_{gs} - V_{th})/nV_t} (1 - e^{-V_{ds}/V_t}) \tag{3}$$

where μ_0 is the effective carrier mobility, C_{ox} is the effective gate capacitance, *W/L* is the ratio of channel width to length of the transistor, V_t is the thermal voltage, and n is the sub-threshold slope coefficient. According to Ref. [13], the power-delay tradeoff can be captured as a convex curve. Theoretically, by adjusting parameters such as V_{dd}, V_{th}, and gate sizing (*W/L*) or using different design implementations (like the previous example of RCA and CLA), for each function type, it is possible to have a hardware implementation of a functional unit (FU) for every point on the tradeoff curve. Thus, we can unify the multiple design-based and multi-V_{dd}/V_{th}-based libraries by considering the multiple design-based FU library only. Therefore, we only consider the function type, delay (speed type) and power characteristics of an FU for our power model which will be discussed shortly.

Beyond scheduling and binding (S&B), module or functional-unit selection is an additional dimension of the design space in HLS. Dynamic power (DP) is consumed only during the period of operation execution, i.e., we consider the dynamic power consumed internally in FUs during the computation only. Thus, DP is related to which speed-power-type FU each individual operation is bound to, i.e., to unit dynamic power per clock cycle (cc), denoted

by *dp*, and delay of the FU that an operation is bound to. Leakage power, LP, is related to the number of FUs of each functionality-speed type used in a design, and specifically to the unit leakage power per cc, denoted as *lp*, of each FU used, the number of FUs with the same *lp* that is used, and the entire latency of the design. Note that, various other power optimization processes that are not directly related to MSB considerations, like power and clock gating, are post-HLS transformations of the design and need a number of considerations beyond the MSB framework. For example, power-gating requires formation of power islands of FUs with largely intersecting idle times that is difficult to address fully during regular HLS, and no work has done this to the best of our knowledge. We thus do not consider this issue in this paper, and assume that all FUs remain powered on for the entire duration of the design latency, including during their idle periods. Using different speed-power characterized FU design-based libraries can afford larger and more flexible optimization spaces than multi-$V_{\mathrm{dd}}/V_{\mathrm{th}}$ based FUs by offering a large range of speed-power parameters to assign to various operations. With the selection from a diverse set of FU designs, we can obtain a minimal power synthesis solution for a DFG. For example, an n-bit ripple carry adder (RCA) and an n-bit Kogge-Stone based carry lookahead adder (KS-CLA) have different speed and power parameters. Theoretically, to execute one operation with two n-bit inputs, an RCA has $\Theta(n)$ unit delay whereas a KS-CLA has $\Theta(\log n)$ unit delay. To simplify the analysis for the power parameters for different designs, we assume for the purpose of this discussion that both DP and LP are proportional to the total number of inputs across all gates in the FU (this is related to the number of transistors used in the FU—each transistor expends dynamic power when conducting, and the number of leakage paths within a gate or logic cell with a small input size is generally proportional to the number of transistors in it), which is $\Theta(n)$ for an RCA and $\Theta(n \log n)$ for a KS-CLA (we note that in the FU library we use for our experiments, the power and delay for different FUs are actually obtained more specifically via simulation of actual designs using Synopsys's Design Vision[35]). Therefore, an n-bit KS-CLA has $\Theta(n(\log n))$ units of dynamic power and $\Theta(n \log n)$ units of leakage power. Similarly, an n-bit RCA has $\Theta(n)$ units of dynamic power and $\Theta(n)$ units of leakage power. Furthermore, as analyzed in Ref. [22], a HLS design with fewer KS-CLAs can be used to replace k RCAs by a factor roughly proportional to their delay ratio of $\Theta((\log n)/n)$, if the replacement does not cause any violation of the latency and precedence constraints (scheduling an operation only after all its parent operations finish execution). In terms of total power, whether a set of RCAs or CLAs is better for a group of operations depends on the ratios of their total power consumed per cc, precedence constraints, i.e., the DFG structure, available slacks for these operations, and the fragmentation of available cc's that they impose on these FUs.

IP: 198.54.106.172 On: Tue, 21 Jul 2020 01:01:50
Copyright: American Scientific Publishers
Delivered by Ingenta

In general, considering a mix of CLAs and RCAs (and more generally, slow/low-power and fast/high-power FUs for each function type) should provide more power-efficient solutions under given latency constraints. Figure 1 illustrates an example of one CLA replacing three RCAs, where the CLA (fast FU) has a delay of 2 clock cycles (cc's) and an RCA (slow FU) has a delay of 4 cc's. Figure 1(a) shows a scheduling and binding solution using only RCAs for four operations u, v, w, x. The precedence constraint $w \rightarrow x$ and other precedence and the given latency constraint, which translate to the *asap* (the earliest schedule time) and *alap* (the latest schedule time) constraints shown for each operation executing on RCAs, necessitates the scheduling shown, resulting in the minimum allocation of 3 RCAs for this purpose. However, these operations can be scheduled and bound to only one CLA as shown in Figure 1(b), as the asap and alap constraints become more flexible (asap may decrease and alap increase)—for simplicity, only the alap for w on a CLA is shown. Hence, even if leakage power of one CLA is 2.5 times that of one RCA, the solution in Figure 1(b) still saves $0.5\times$ of the leakage power consumption of an RCA.

2.2. Component-Level Power Consumption and Architecture Generation

Here we discuss leakage and dynamic power consumption formulations for the major components of an HLS architecture: FUs, muxes, demuxes and registers.

2.2.1. FU Power Consumption

To evaluate the module selection, scheduling, and binding (MSB) solution for a design for the objective of total power optimization, obtaining both dynamic and leakage power from the corresponding MSB solution is required.

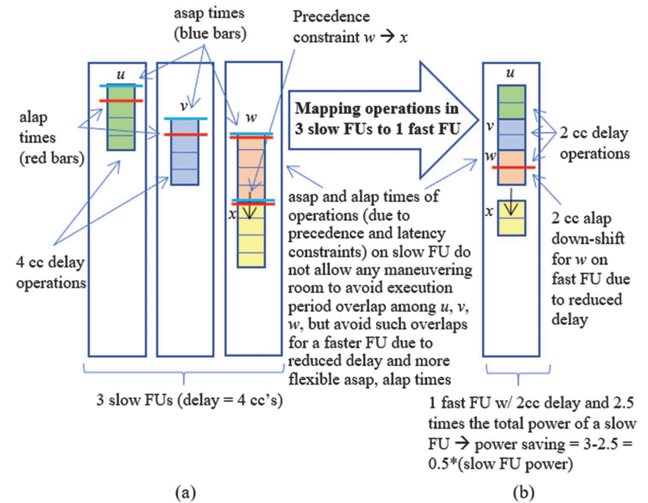

Fig. 1. An example of power reduction by utilizing a faster FU. The colors represent different operations, and they are bound to FUs that are represented by the larger white rectangles.

In the following, for notational and estimation convenience, by *power* of any type (leakage, dynamic, and total), we will mean *energy per clock cycle*, as opposed to energy per second (the two are related to each other by a factor, the clock frequency, and in any case, our power definition, is energy per some unit time, and thus, consistent with the standard meaning of power). We define a *configuration* $c(i)$ of an operation i as $c(i) = (t_i, s_i)$, where t_i is the function type of this operation and s_i represents the speed type of the FU that operation u_i is to be bound to. Each MSB solution includes a configuration for each operation u_i. For example, for a dual-speed *lib*, s_i is defined as:

$$s_i = \begin{cases} 0 & \text{if operation } u_i \text{ is bound to slow FU} \\ 1 & \text{if operation } u_i \text{ is bound to fast FU} \end{cases} \quad (4)$$

Moreover, we define a module selection solution for all operations for a DFG, as a *speed vector* $\text{SV} = [s_1, s_2, \ldots, s_i, \ldots, s_n]$, where s_i is the speed type of FU that operation u_i is to be bound to, and n is the number of operations of the given DFG. In practice, a multiple design-based library can have a heterogeneous number of speed types for different functional types. Further, for a functional type t that only has a single speed-type, all operations with function type t have only one option for their speed type s, and the corresponding entries in SV are constants (set to 0 and not subject to change during MS solution space exploration). We initialize SV with all operations assigned to the fastest possible speed-type FUs in the given FU library. Based on a specific speed vector SV that represents a module selection solution, we can compute the dynamic power consumption of FUs, denoted as DP_{FU} as:

IP: 198.54.106.172 On: Tue, 21 Jul 2020 01:01:50
Copyright: American Scientific Publishers
Delivered by Ingenta

$$\text{DP}_{\text{FU}} = f_D(c(1), \ldots, c(n)) = \sum_{i=1}^{n} \frac{dp(c(i)) \cdot d(c(i))}{l} \quad (5)$$

where $dp(c(i))$ and $d(c(i))$ are the dynamic power (dynamic energy per cc) and delay of operation u_i when bound to an FU of speed type $s_i \in c(i)$, respectively. $l \leq l_c$ is the achieved latency of a design.

Furthermore, based on an SV, we generate a S&B solution by using a constructive algorithm called modified list scheduling or MLS (this is a much better optimized version of classical list scheduling). From the corresponding set of required FUs in the S&B solution, we can compute the total leakage power consumption of all FUs, denoted as LP_{FU}, of an MSB solution as:

$$\text{LP}_{\text{FU}} = f_L(N_1, N_2, \ldots, N_m) = \sum_{i=1}^{m} lp(lib_i) \cdot N_i \quad (6)$$

where m is the number of FUs of different functionality-speed types (slow adder, fast adder, slow multiplier, faster multiplier, etc.) that are denoted by lib_i's, i.e., FUs of functionality-speed type $i = (s_i, t_i)$, $lp(lib_i)$ is the leakage power (leakage energy per cc) of a lib_i FU, and N_i is the number of lib_i FUs used. As we mentioned earlier, we assume that all FUs remain powered on during the entire design latency and thus, the leakage power of FUs is consumed over the entire design latency including the idle periods of FUs.

Thus, the total power consumed by FUs, i.e., the total power of an MSB solution, denoted as TP_{FU}, is:

$$\text{TP}_{\text{FU}} = \text{DP}_{\text{FU}} + \text{LP}_{\text{FU}} \quad (7)$$

We obtained the values of $dp(c(i))$, $d(c(i))$ and $lp(lib_i)$ for each functional-speed type FU by describing the corresponding designs using Verilog, followed by synthesis and power, delay report generation using Synopsys Design Vision;[35] these are reported in Table I in Section 5. Presumably Design Vision uses the transistor and gate level leakage and dynamic power models along the lines of those described in Eqs. (1)–(3) in Section 2.1 to obtain the aforementioned FU power and delay parameters.

2.2.2. Final Architecture Generation

Once we have a complete MSB solution, we can obtain the corresponding architecture to determine the power consumption of muxes, demuxes, and registers (dynamic power consumed on interconnects cannot be considered without layout/floorplanning the design and thus not considered here). From a complete MSB solution, we generate a corresponding architecture by performing register binding. In this paper, since we focus on the MSB problem and TP minimization of FUs, we use a tie-breaking variation of the basic left-edge algorithm[25] (either integrated with scheduling or post-regular-MSB in which only the basic left-edge algorithm is integrated with scheduling) that keeps the FUs used in the design to the exact number determined by the scheduling solution, but within this constraint, maximizes sharing of interconnects between FUs in order to reduce mux and demux sizes needed, and thus their power. Register allocation is also performed but post the above processes, and also uses the left-edge algorithm for minimizing the number of registers needed given the MSB solution.

Table I. Four-speed FU library via [23].

Function unit designs (16-bit)		d[*cc*]	*dp* [uW]	*lp* [uW]
Adder	Kogge-Stone	1	405.6	11.2
	Brent-Kung	2	149.7	8.2
	CarrySelect (RCA)	3	69.8	5.8
	RCA	6	23.0	3.8
Multiplier	CarrySave-Tree-RCA	3	972.9	80.3
	CarrySave-Tree-CSA	4	548.6	60.4
	Dadda-Kogge-Stone	5	432.5	59.5
	Wallace-CSA (RCA)	7	293.8	56.6
Divider	Radix-8 (Kogge-Stone)	8	560.6	123.4
	Radix-4 (Kogge-Stone)	12	210.3	69.5
	Radix-8 (Brent-Kung)	16	204.7	90.1
	Radix-4 (Brent-Kung)	24	86.9	57.4

Thus, we have two flows for obtaining a HLS architecture:
(1) Regular FU-power-driven MSB (binding integrated with scheduling but only to keep FUs used at the exact number determined by scheduling) → Post MSB tie-breaking left-edge-based FU re-binding to minimize mux + demux power but which also keeps FUs used at the exact number determined by scheduling → Left-edge based register allocation to minimize the number of registers used, assuming a register-architecture in which separate register banks are used for different FUs, i.e., registers in a bank store outputs of multiple FUs that are inputs to the concerned FU, and are thus shared by multiple input operands of the this FU. This phase also provides us the size of input muxes to registers as opposed to input muxes to FUs that we explicitly optimize in the aforementioned second stage of this flow.
(2) FU + mux-demux power-driven MSB (binding with mux-demux size/power considerations integrated with scheduling that also keeps FUs used at the exact number determined by scheduling) → Left-edge based register allocation to minimize the number of registers used as in flow 1.

Note that in flow 1, we explicitly only minimize TP_{FU} but report total architecture (FU + FUs' mux-demux + registers and their input muxes) power. In flow 2, we explicitly minimize FU + FUs' mux-demux power and also report total architecture power as in flow 1. Flow 2 provides lower architecture power solutions than flow 1, and our motivation to implement both flows are: (a) Flow 1 is a more standard flow in the research literature,[32] and enables us to directly compare its results to previous techniques employing the same flow (Section 5.2 and Section 5.3). (b) The QoS difference in architecture power (Section 5.4) between the two flows show the benefit yielded by the non-standard and more sophisticated integrated binding of flow 2 compared to the more common integrated binding of flow 1, albeit at an appreciably higher runtime (note again that the number of FUs of each lib_i type are the same in the final solutions of both flows). (c) The two flows provide a runtime-to-QoS tradeoff that allows a chip designer the flexibility of using the option the best fits his/her requirements.

Then, with the architecture, we can determine the complete interconnect information which includes information about which interconnects are connected to which muxes, demuxes and registers. Based on this, we compute power consumption of muxes, demuxes, and registers as follows.

2.2.3. Mux and Demux Power Consumption

Dynamic power is consumed in an FU or register mux when input data is generated due to the execution of an operation whose output is destined for the corresponding FU, since the data has to pass through a register mux and an FU mux for the FU. Thus, for a design with k muxes allocated in total with $M(i)$ the number of inputs of the i'th mux, the total dynamic power DP_{mux} consumed by muxes is:

$$DP_{mux} = \sum_{i=1}^{k} \frac{T(i) \cdot \lceil \log M(i) \rceil \cdot dp_{21mux} \cdot d_{21mux}}{l} \tag{8}$$

where $T(i)$ is the number of times that new input data for an FU passes through the i'th mux (which is either an input mux for one its input registers or is a mux directly connected to one of its inputs), $\lceil \log M(i) \rceil$ is the maximum number of 2-to-1 muxes in any path of this mux that is assumed to have a binary tree-structured design with 2-to-1 muxes as its nodes—data passes through exactly one such path of the mux. dp_{21mux} and d_{21mux} are the dynamic power and delay of a 2-to-1 mux, respectively. Similarly, we can determine the total dynamic power DP_{demux} consumed by demuxes—a demux consumes dynamic power whenever its source FU executes a new operation and thus generates a new output data. We define $DP_{mux/demux} = DP_{mux} + DP_{demux}$.

Moreover, the leakage power of a mux (demux) is related to the number of inputs (outputs) of this mux (demux), and the unit leakage power of a 2-to-1 mux (1-to-2 demux), denoted as lp_{21mux} ($lp_{12demux}$), An *m-to-1* mux (*1-to-m* demux) has *m–1* 2-to-1 muxes (1-to-2 demuxes). Similar to the assumption for LP_{FU}, all muxes and demuxes remain powered on over the entire design latency whether they are idling or not. Thus, for a design with k muxes and $M(i)$ the number of inputs of the i'th mux, the total leakage power LP_{mux} consumed by all muxes is:

$$LP_{mux} = \sum_{i=1}^{k} (M(i) - 1) \cdot lp_{21mux} \tag{9}$$

Similarly, we can determine the total leakage power LP_{demux} consumed by all demuxes. We define $LP_{mux/demux} = LP_{mux} + LP_{demux}$.

Thus, the total power $TP_{mux/demux}$ consumed by muxes and demuxes is:

$$TP_{mux/demux} = DP_{mux/demux} + LP_{mux/demux} \tag{10}$$

2.2.4. Register Power Consumption

The dynamic power of a register is related to the total number of input data stored in it, the unit dynamic power of a register, denoted as dp_{reg}, and the delay of a register, denoted as d_{reg}. For a design with q registers and $R(i)$ the total number of input data stored in the i'th register, the total register dynamic power DP_{reg} is:

$$DP_{reg} = \sum_{i=1}^{q} \frac{R(i) \cdot dp_{reg} \cdot d_{reg}}{l} \tag{11}$$

Moreover, the leakage power of registers in a design is related to the total number of registers are allocated and the

IP: 198.54.100.172 On: Tue, 21 Jul 2020 01:01:50
Copyright: American Scientific Publishers
Delivered by Ingenta

unit leakage power of the register, denoted as lp_{reg}. Again, we assume that all registers remain powered on over the entire design latency whether or not they are idling. For an architecture with q registers the total leakage power LP_{reg} is:

$$\text{LP}_{\text{reg}} = q \cdot lp_{\text{reg}} \tag{12}$$

Thus, the total power TP_{reg} consumed by all registers is:

$$\text{TP}_{\text{reg}} = \text{DP}_{\text{reg}} + \text{LP}_{\text{reg}} \tag{13}$$

Therefore, the total power of an architecture, denoted as TP_{arch}, is:

$$\text{TP}_{\text{arch}} = \text{TP}_{\text{FU}} + \text{TP}_{\text{mux/demux}} + \text{TP}_{\text{reg}} \tag{14}$$

We synthesized 16-bit 2-to-1 mux, 1-to-2 demux, and register using Synopsys Design Vision[35] and obtained the following delay and power parameters. They all have delays in picoseconds (much smaller than a cc), $dp_{\text{21mux/12demux}} = 0.312$ uW, $lp_{\text{21mux/12demux}} = 1.347$ uW, $dp_{\text{reg}} = 1.19$ uW, and $lp_{\text{reg}} = 3.6$ uW.

3. POWER-DRIVEN HIERARCHICAL FRAMEWORK

Here we discuss our power-driven hierarchical framework for module selection, scheduling and binding called PSA-MLS (Power-driven Simulated Annealing with Modified List Scheduling algorithm).

In this paper, one of the core techniques is called *modified list scheduling* (*MLS*). We develop MLS as a significantly augmented and sophisticated version of the very runtime-efficient list scheduling (LS) algorithm. The basic LS algorithm (which also performs binding) is given in Figure 2 with a slight modification for scheduling and binding an operation on an FU of a pre-selected speed type. MLS is given as part of our hierarchical framework's flowchart in Figure 3. Another main method is called

Algorithm *LIST_Sch*($G(V, E)$, l_c, SV) // $G(V, E)$ is the DFG,
V is the set of operation, E is the set of arcs in the DFG, l_c is the latency constraint.
SV is the speed vector.
1. $\mathbf{S}_{\mathbf{init}} = 1$ // Vector $\mathbf{S}_{\mathbf{init}}$ has 1 FU for each FU type-(functionality t, speed s)
2. j = 1 // the current clock cycle
3. Compute the ALAP time al^{lc} w.r.t. l_c
4. al_0^{lc} if < 0
5. return (not feasible)
6. Repeat
7. for each functional type t
8. for each speed type s
9. $\text{U}_{j,t,s}$ = available opers. of (t, s) in V.
10. Compute the slacks $\{sl_i = al_i^{lc} - j,\ \forall\, u_i \in \text{U}_{j,t,s}\}$
11. Schedule opers. with zero slack.
12. Bind opers. on an available FU. (if needed, allocated additional FUs for this) – this equivalent to left-edge binding; update $\mathbf{S}_{\mathbf{init}}$ (if new FUs added).
13. Schedule and bind additional opers. in $\text{U}_{j,t,s}$ with minimal slack under $\mathbf{S}_{\mathbf{init}}$ constraints.
14. j = j + 1.
15. Until all opers. in V are scheduled.
16. Return ($\mathbf{S}_{\mathbf{init}}$, S&B solution).

End Algorithm

Fig. 2. Simultaneous scheduling and binding using list scheduling and left-edge binding with a given speed vector SV.

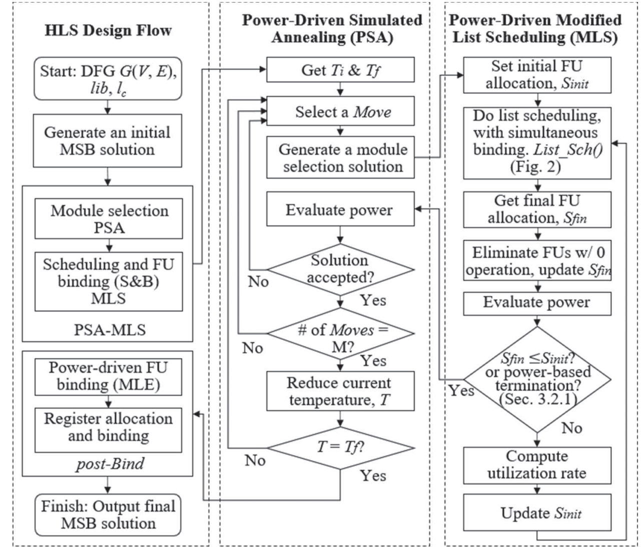

Fig. 3. The flow chart of our power-driven hierarchical HLS framework using power-driven simulated annealing for module selection and modified list scheduling for scheduling and binding.

power-driven simulated annealing (*PSA*) which combines with MLS to form the proposed hierarchical framework PSA-MLS.

Figure 3 shows the HLS design flow with our power-driven hierarchical framework PSA-MLS. The design processing starts from generating an initial module selection, scheduling, and FU binding (MSB) solution, where this initial MSB solution is the start point of the solution space that PSA-MLS will explore. To generate the initial MSB solution, any techniques and methods can be applied since theoretically, the initial MSB solution does not affect the QoS of the final complete MSB solution due to the fact that the randomness of PSA can explore the complete solution space of module selection (will be discussed shortly). In this paper, the initial MSB solution is generated as follows: (1) Module selection (initialization of SV): all operations are assigned to the fastest speed-type FUs in the given multiple design-based FU library *lib*. (2) Scheduling and Binding (S&B): The S&B result is generated by the proposed modified list scheduling. We tested four different initial SVs with all operations are assigned to: (a) the fastest speed; (b) the slowest speed; (c) mid-level speed; (d) random speed. According to our results, there is no significant power and runtime result difference among different initial SVs.

Copyright: American Scientific Publishers
Delivered by Ingenta

After generating the initial MSB solution, PSA-MLS starts to explore the solution space and generate the final complete MSB solution. From the final MSB solution, we do register allocation and binding as the post-MSB processing to obtain the complete architecture for obtaining the entire architecture power. Note that, the goal in this paper is the total power (dynamic + leakage) optimization where the total power is consumed by FUs only. Once the scheduling and module selection solution has been

generated, the QoS of the scheduling and module selection solution is already determined in that they specify an achievable lower-bound on the number of FUs needed for each function and speed-type combination. This lower bound can be exactly obtained by an optimal binding (assigning operations to particular FUs to pertain to the scheduling and module selection solution) algorithm such as the left-edge algorithm. Since we don't consider the power of muxes, demuxes and registers in this version of MLS (a later version in the *in-Bind* approach discussed in Sec. 5.4 does), no more optimization step is required e.g., some binding techniques like maximum weighted clique covering[26] can be applied for minimizing the power of interconnects (or muxes). However, post PSA-MLS (and thus outside of MLS), we used a modified left-edge (MLE) algorithm discussed in Section 3.3 for FU re-binding to minimize FU plus mux/demux power, and simple left-edge for register binding in a *post-Bind* (*post*-MSB *bind*ing) step shown in Figure 3. Next, we discuss the details of PSA-MLS.

The basic issue in module selection from multi-speed-power characterized FUs is what speed type an operation should be assigned so that the QoS of the final solution after scheduling and binding will be good. Since the speed vector (SV) captures the issue of speed type or module selection, with a given SV, the MSB problem is transformed to a variation of the standard S&B problem wherein the S&B solution needs to respect the operation assignments to speed-types specified in the SV.

IP: 198.54.106.172 On: Tue, 27 Jul 2020 01:01:50
Copyright: American Scientific Publishers
Delivered by Ingenta

To capture the issue of generating SVs and the corresponding S&B solutions, we employ a two-pronged approach in which there is a stochastic exploration of module selection space interspersed by an efficient constructive algorithm to perform S&B for the module selection solution. In this paper, we use a power-driven simulated annealing (PSA) algorithm to generate SVs and a modified list scheduling (MLS) algorithm to obtain the corresponding S&B solutions.

This framework called PSA-MLS avoids the very runtime-intensive problem of conventional SA for simultaneous module selection and S&B (i.e., for exploring a much larger solution space than that of module selection) to obtain good solutions, and also, maintaining the essential superiority of stochastic exploration of an important sub-space of the entire problem solution space. The flow chart of the two-pronged proposed hierarchical framework is given in Figure 3. The general procedures of the proposed PSA-MLS are the follows and all details will be discussed later:

1. PSA initializes the starting and final temperature.
2. PSA randomly generates a new module selection solution, SV′.
3. Invoke an efficient constructive S&B algorithm, like list scheduling (LS) in Figure 2, to extend SV′ with S&B for a new complete MSB solution p' that satisfies the speed-type assignments in SV′ with a corresponding total power consumption TP′. Note that, the simultaneous scheduling and binding (by left-edge algorithm) is described by line 11 to line 13 in Figure 2.
4. PSA decides the acceptance or rejection of p' based on the aforementioned power consumption TP′ by an acceptance probability, which is denoted as $prob(p')$. $prob(p') = \min\{1, e^{-(\mathrm{TP}'-\mathrm{TP})/T}\}$, where TP is the total power of the current complete solution p, and T is the current temperature.
5. Repeat step 2 to 4 until the system reaches the equilibrium state is reached (solution cost has not changed much over a certain number of moves—we set this limit of the number of moves to be equal to n, the number of operations in a given DFG), or the number of feasible moves (accepted + rejected) equals a threshold value of M.
6. Exit if current temperature T reaches to the freezing temperature T_f. Otherwise reduce the temperature according to a cooling rate and repeat steps 2 to 5.

3.1. Power-Driven Simulated Annealing

The power-driven simulated annealing algorithm (PSA) attempts to obtain the lowest power speed vector for the problem of speed-assignment for each operation (module selection) for power minimization. Starting from the initial MSB solution as mentioned before, PSA iteratively generates new random SVs by FU-centric moves (to be discussed shortly). For each new SV, denoted by SV′, S&B is performed by MLS to obtain a new complete MSB solution p' for the corresponding SV′. Then, we compute the total power consumption TP′ for p' by Eqs. (5) and (6). After that, PSA either accepts or rejects this SV′ by the acceptance probability of p', $prob(p') = \min\{1, e^{-(\mathrm{TP}'-\mathrm{TP})/T}\}$.

3.1.1. Initial and Freezing Temperature

One of our innovations in PSA is that it adaptively computes the initial and freezing temperature T_i and T_f based on the given DFG. The idea is that an average cost-increasing move at temperature $T = T_i$ should be accepted with a probability of 0.99 and with a probability of 0.01 at $T = T_f$. Accordingly, we choose moves from our move set (to be described shortly) according to their pre-assigned probabilities in a non-SA framework. The pre-assigned probability of selecting a move from the move set is uniformly distributed. Then, we compute the average power change ΔC_{avg} due to these moves over n random moves from some current solution as $\Delta C_{\text{avg}} = 1/n \sum_{i=1}^{i=n} C_i$ where C_i is the power change of the i'th move. Based on ΔC_{avg}, and the acceptance probability of $e^{(\Delta \mathrm{Cavg}/\mathrm{T})}$, we choose T_i and T_f such that the acceptance probabilities at these temperatures are 0.99 and 0.01, respectively. We use a cooling rate $r_c = 0.95$, which seems to be sufficient for a reasonably comprehensive search of the solution space.

3.1.2. Move Set for Speed Vector Generation

PSA has four types of moves to generate SV′, which is used to decide the speed type of an FU that an operation will be bound to. In general, for a given multiple design-based FU library *lib*, we define Q_t as the number of speed types for each function type t. As discussed before, a library can have heterogeneous speed types in which Q_t may be different across function types t. The speed type s_u for operation u with function type t is defined as $s_u = j$ if it is bound to (or to be bound to) the $(j+1)$'th fastest FU of the functional type that u has, where $0 \leq j \leq Q_t - 1$. Note that, for function types t with $Q_t = 1$, since all operations with function type t has only one possible speed type FU that they can be bound to, function type t is not involved in the SV generation by PSA Thus all such operations with function type t have a fixed speed type.

We now discuss the four different move types in PSA. The probability of selecting a move type is uniformly distributed to be 1/4.

- *Move* 1: Randomly choose one FU of speed type $s < Q_t - 1$ and increment s of these operations as well as that of the corresponding FU by 1. The probability of selecting a speed type s is $1/Q$ where Q is the sum of the Q_t's across all function types with $Q_t > 1$. Having selected a speed type s to increment, the probability of selecting an FU of speed type s is $1/m$ where m is the number of FUs with speed type s.
- *Move* 2: Randomly choose one FU with $s > 0$ and decrement s of the operations bound to that FU by 1. Both probabilities of selecting a speed type and selecting an FU of that speed type are the same as in Move 1.
- *Move* 3: Randomly choose two FUs with different speed types s_1 and s_2. Randomly select the same number, k, of operations in both FUs, and swap the speed types of these operations (i.e., an operation with speed-type s_1 is changed to speed-type s_2 and vice versa). The probability of selecting an FU with speed type s_1 is $1/m$ where m is the total number of FUs have been allocated across all functional-speed types with $Q_t > 1$. The probability of selecting an FU with speed type s_2 is $1/m'$ where m' is the number of the FUs with different speed-types than s_1 and with $Q_t > 1$. The probability of selecting a value for k is $1/k$, where k is the minimum value between the two numbers of operations bound to each FU.
- *Move* 4: Randomly choose a number k in $[1, n]$ with a probability of $1/n$, where n is the total number of operations in a DFG. Then, randomly choose a contiguous sequence of k elements in the current SV (randomly choosing the starting index for this sequence in $[1, n-k+1]$) by the probability of $1/(n-k+1)$), and change their s values randomly. For each selected element with function type t, the probability of selecting a new s value is $1/(Q_t - 1)$.

For simplicity of exposition, we illustrate the moves for the dual-speed case, i.e., with $Q_t = 2$ for all function types t with $Q_t > 1$. For a dual-speed *lib*, 0 and 1 represent slow and fast speed types, respectively. Figure 4. shows examples of the first three moves. Assuming the current speed vector SV = [0100101...1], operation 1, 3, 4 are in one slow FU with s value is 0, whereas operation 2, 5 are in one fast FU with s value is 1. Figure 4(a) shows that by *Move* 1, the new SV, SV′ = [**1**1**11**101...1], while Figure 4(b) shows that by *Move* 2, SV′ = [0**0**00**0**01...1], and Figure 4(c) shows that by *Move* 3, SV′ = [**10**1**10**01...1]. Finally, with $k = 5$, starting from operation 3 by *Move* 4, SV′ = [01**11010**...1].

Obviously, *Move* 1 to 3 are FU-centric speed-type changes and thus, SV′ would generally lead to a solution with a small power change, i.e., the new complete MSB solution p' is truly obtained incrementally from the solution p with the current speed vector SV. *Move* 4 is not FU-centric as other types of moves, and it may lead to large "jumps" in the solution space. The apparent randomness of the distribution of a subsequence of operations among different FUs is an interesting point in *Move* 4. This randomness will lead *Move* 4 to generate significant jumps for either better or worse from the current solution. Based on *Move* 4, it is possible to obtain any SV from the current module selection solution, and thus, using all the types of moves, the module selection solution space is connected, a desirable property for a simulated annealing algorithm.

IP: 198.54.106.172 On: Tue, 21 Jul 2020 01:01:50
Copyright: American Scientific Publishers
Delivered by Ingenta

3.2. Modified List Scheduling Algorithm

We develop and use a modified version of the very runtime-efficient list scheduling (LS) algorithm, called *modified list scheduling* (*MLS*), to obtain the S&B solution for a given SV generated by PSA. The basic list scheduling (LS) algorithm, which minimizes the total number of FUs used, has been well studied in HLS. The version of LS that respects a given SV is shown in Figure 2; henceforth, by LS, we will mean this version of LS. LS assigns a priority to all available operations (operations whose parent operations, if any, have finished execution) of a given DFG. The slack sl_u of an operation u is defined as:

$$sl_u = t_u^{al} - t \tag{15}$$

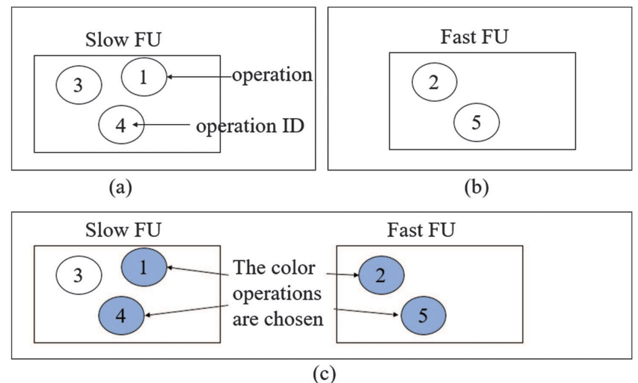

Fig. 4. Examples of: (a) *Move* 1; (b) *Move* 2; (c) *Move* 3.

where t_u^{al} is the alap time (as-last-as-possible or the latest time to schedule to meet the latency and precedence constraints) of operation u and t is the current clock cycle (cc).

LS proceeds chronologically by cc, and in each cc, for each functionality-speed type (t, s) in *lib*, where lib_i is the i'th combination of (t, s) in *lib*, LS schedules operations u whose $c(i) = lib_i(u)$ in an order that is inversely proportional to its slack. In case $sl_u = 0$, this operation must be scheduled immediately to meet the latency constraint on an available FU (that are not executing any operation in the current cc) of lib_i, or a newly-allocated FU of lib_iif there is none available. After all 0-slack operations are scheduled, if there are any available FUs of lib_i, the remaining unscheduled operations are scheduled in the current cc and bound to them in order of increasing slack (minimal slack first).

On the other hand, the FU binding by left-edge algorithm can be done simultaneously within the processing of LS. Once LS makes a scheduling decision, binding the just-scheduled operation to an available FU or newly allocated FU, as the case may be, of the right functionality speed type is equivalent to binding done post-scheduling by the left-edge binding algorithm (line 12 in Fig. 3).

However, the basic LS has well-known deficiencies compared to other scheduling algorithms. For example, unlike the force-directed scheduling algorithm,[6] LS does not have a global or semi-global view of S&B possibilities for resource/FU minimization (which is strongly correlated to leakage power minimization). In a greedy manner, LS performs the scheduling only based on the priorities of the operations that are available for scheduling in the current cc, and in many instances, schedules an operation only in a cc in which its slack $= 0$ even though it was available earlier. This creates a scheduling "jam" in many cc's where too many operations than available FUs need to be scheduled, leading to new FU allocations at that point; these new FUs will have low usage as they will be idle in all preceding cc's. Hence, LS can allocate a very high number of FUs, where many of these FUs have low usage, i.e., have only a small number of operations bound to them.

It is possible to reschedule such jammed-up 0-slack operations in an earlier cc by allocating new FUs earlier leading to higher utilization for them, and thus, as a corollary, have fewer FUs allocated in the final solution. We define the utilization rate as $ur_w(lib_i(w))$ of an FU w with the functionality-speed type $lib_i(w)$ as:

$$ur_w(lib_i(w)) = \frac{O_w \cdot d(lib_i(w))}{l} \tag{16}$$

where O_w is the number of operations that FU w has bound to it, $d(lib_i(w))$ is the delay of FU of functionality-speed type lib_i that w is bound to, and $l \leq l_c$ is the achieved latency of a design. Thus, the overall utilization rate $UR(lib_i)$ for the FUs of type lib_i is:

$$UR(lib_i) = \sum_{j=1}^{m} ur_j(lib_i) \tag{17}$$

where m is the total number of FUs allocated with lib_i. The overall utilization rate for type lib_i is a lower bound for the number of total FUs that will be used over the scheduling process if all needed FUs are allocated in cc 1 and thus better utilized (note that LS allocates only 1 FU of each speed-function type at the beginning). We define $S_{\text{init}}(lib_i)$ as the initial allocation for functionality-speed type lib_i at the beginning of each new iteration of a modified LS algorithm (MLS) which is shown in Figure 3. We define $S_{\text{fin}}(lib_i)$ as the final allocation result for functional-speed type lib_i in the currently complete scheduling iteration of MLS. Since we do not know the exact number of FUs that will be allocated in the final solution, at the beginning of each new iteration of MLS, we determine $S_{\text{init}}(lib_i)$ as:

$$S_{\text{init}}(lib_i) = \lceil UR(lib_i)/\min(1, (1+\beta)UR_{\text{avg}}(lib_i), UR_{\text{avg}}^{\text{Sinit}}(lib_i))\rceil \tag{18}$$

where $UR_{\text{avg}}(lib_i) = UR(lib_i)/m$ and $UR_{\text{avg}}^{\text{Sinit}}(lib_i) =$ the average utilization of FUs allocated in $S_{\text{init}}(lib_i)$ in the currently completed scheduling iteration of MLS. Note again that $UR(lib_i)$ is a lower bound on the number of FUs of lib_i that will be needed in any MSB solution, and $(1+\beta)UR_{\text{avg}}(lib_i)$ is the expected increase in the average utilization rate due to a better initial allocation of FUs of lib_i; we use $\beta = 0.2$ in our experiments. Further, it is reasonable to expect that the utilization of all FUs of type lib_i used in the final solution will not exceed the current utilization of all FUs of lib_i that were allocated in S_{init} in the current iteration, since these FUs have a relatively high utilization given that: (a) they were available for use starting from cc 1, and (b) in each cc, the ratio of the number of available operations to available FUs of lib_i should more or less be the maximum relative to future iterations of MLS in which S_{init} allocations either increase or remain unchanged for lib_i FUs—the larger this ratio, more is opportunity for better scheduling decisions among more available operations on a smaller set of FUs (due to a larger diversity of available operation slacks) thus increasing FU utilization. Hence, it is appropriate to use $\min(1, (1+\beta)UR_{\text{avg}}(lib_i), UR_{\text{avg}}^{\text{Sinit}}(lib_i))$ as an estimate of the average utilization expected in FUs of type lib_i in the next MLS iteration. This term is thus used in the denominator of the $S_{\text{init}}(lib_i)$ formulation for the next MLS iteration to provide a more realistic initial allocation of FUs than the lower bound of $UR(lib_i)$ needed FUs. This gives us a reasonably good estimation of FUs needed in the final solution and facilitates faster convergence (requiring fewer scheduling iterations) to a good solution.

Compared to LS, MLS (Fig. 3) has the above extra processing steps of initial FU re-allocation and final FU usage evaluation to improve the QoS of basic S&B solution, and extra scheduling iterations that terminate when the initial and final FU allocations converge. The overall runtime of MLS versus LS increases by a small average factor of 2.4× that is almost constant across DFG sizes, with MLS providing much better QoS—our experiments show that MLS has an average improvement of the total number of FUs used of 51.6% compared to LS.

3.2.1. *Power-Driven Modified List Scheduling*

To achieve the goal of power optimization in PSA-MLS, we minimize power with equations in Section 2 in a power-driven MLS (in which we can use different power metrics, e.g., leakage power, total power, FU power, architecture power.). As discussed before, MLS generates a S&B solution based on the module selection solution provided by PSA, where the functionality-speed type, for every operation has been fixed. Regular MLS attempts to minimize the numbers of FUs used for each functionality-speed type and this is related to total FU leakage power. However, in the mechanism of MLS, discussed earlier, we have two ways to reach a low power design: (1) by the original MLS termination criteria for minimizing number of FUs and thereby FU leakage power and (2) a more explicit way to minimize the total leakage power (Eq. (6)) across all functionality-speed types in which an increase in the number of some functionality-speed types is traded-off with a decrease in others so that total FU leakage power (or any other power metric mentioned above that can be estimated within MLS) is reduced.

IP: 195.54.106.172 On: Tue, 21 Jul 2020 01:01:50
Copyright: American Scientific Publishers
Delivered by Ingenta

The original MLS terminates when $S_{\text{fin}} \leq S_{\text{init}}$ in the current iteration. This condition is involved in the minimization of the number of FUs used for each functionality-speed type, since the $S_{\text{fin}} \leq S_{\text{init}}$ is a vectorized comparison which holds only when each element of the S_{fin} and S_{init} vectors satisfy the '$\leq$' relation. Thus a smaller S_{fin} always represents a lower power result. However, this is not the only way to obtain a lower power result. For example, a larger S_{fin} does not necessarily mean a larger power result because of the existing trade-off between power consumption of different functionality-speed types (as, for example, discussed and illustrated in Section 2). In fact, because of such a tradeoff a solution not satisfying $S_{\text{fin}} \leq S_{\text{init}}$ may have lower power than one that does, say, at a later iteration. Thus, to optimize power more directly, we define the following two OR conditions as the new termination criteria for MLS: (1) The original MLS condition, $S_{\text{fin}} \leq S_{\text{init}}$. (2) There have been k-consecutive MLS iterations such that the power result P of a current MLS iteration has a small factor difference of τ (we use $\tau = 0.1$ to limit runtime increase) compared to the best power result P_{best} in the current MLS processing (note that this is not necessarily the global best power result so far over the entire PSA-MLS processing). Besides allowing for a tradeoff in the power consumption across different functionality-speed type FUs (to reach an overall lower power solution), another idea of the second condition is that if we see small improvements over several consecutive iterations, it most probably means that the solution has at least reached close to a local minima and we will not see any more significant improvement. Moreover, to limit the extra runtime of additional iterations, k should be a small value, and in our experiments reported in this paper, we use $k = 2$.

3.3. Power-Driven FU Binding with Consideration of Mux and Demux Power

As mentioned before, the left-edge-based FU binding can be done simultaneously within the processing of LS and thus also MLS. LE-based FU binding obtains the exact number of FUs of each functionality-speed type that MLS-based scheduling yields, and thus it does not affect the QoS related to FU power. Once we have a complete MSB solution, we can generate the corresponding HLS architecture including needed interconnects, muxes, demuxes and registers to obtain the power consumption of muxes, demuxes, and registers. However, the LE-based binding will bind an operation to any available FU in the cc it is scheduled in. As a result, the increase of interconnects among FUs (equivalently, the increase of total mux and demux sizes) are not considered. Thus, the basic LE algorithm for FU binding is not appropriate for power optimization.

Therefore, we apply a simple modification on the basic LE algorithm (called MLE) for achieving the goal of minimizing the power of muxes and demuxes as a part of the goal of achieving a low power HLS architecture without changing the FU allocation results provided by an MSB solution.

The core idea of MLE is to select in the scheduling cc t of each operation u, the best operation-FU pair (u, F) from all available operation-FU pairs in cc t to bind such that F has the maximum number of existing fanin connections to F from all FUs that the *parent*(u) have been bound to. Thus, the increase of input mux size for F and output demux sizes of FUs to which *parent*(u) are bound to bind u to an available FU in cc t is minimized. After binding the selected (u, F), the interconnects and mux/demux sizes are of the affected FUs are updated, and the process repeated to bind the next best operation-FU pair in cc t, and so forth until all such pairs are bound. The binding them proceeds to operations scheduled in cc $t+1$. Thus, MLE achieves the goal of HLS architecture power minimization by minimizing the sizes of muxes and demuxes used. The power consumption of muxes and demuxes are computed by Eqs. (8) and (9). MLE can be used as a post-MSB step for performing FU re-binding (called *post-Bind*) as shown in Figure 3, or it can replace the LE-based FU binding step within MLS (called *in-Bind*), as discussed further in Section 5.

4. A COMPETING FLAT SIMULATED-ANNEALING BASED POWER-DRIVEN MSB ALGORITHM

For the purpose of comparing a good and flat simulated annealing technique for power-driven MSB to PSA-MLS, we developed an extended version of SALSA,[20] a well-known resource-minimizing scheduling and binding technique using simulated annealing, with the additional dimension of module selection called MS-SALSA (SALSA augmented with module selection moves and the goal of power optimization). MS-SALSA solves the module selection, scheduling, and binding (MSB) problem with the same objective as PSA-MLS but does it in a conventional flat (1-level) manner in a purely simulated annealing framework instead of a hierarchical framework employed by PSA-MLS.

We incorporated the move set of scheduling moves called *sch-set* (to be elaborated shortly) from SALSA and the move set for module selection in PSA called *spd-set* to expand conventional SALSA so that MS as well as S&B can be performed in a flat simulated annealing framework. We note that in the context of MS-SALSA, this move set, unlike in PSA, where it operates on a speed vector, operates on a complete MSB solution to essentially obtain another complete MSB solution. From the current MSB solution p, which has a complete scheduling, module selection and FU binding assignment for every operation, we can generate a new MSB solution p' with a move from either *sch-set* or *spd-set*. A *sch-set* move only changes the scheduling solution in p for some operation(s), while the MS solution remains the same. A *spd-set* move changes the MS solution for some operation(s), while the scheduling solution in p is generally not changed (there are different scenarios that will be discussed shortly). Once we have the new scheduling solution from a *sch-set* move or a new MS solution from a *spd-set* move, as mentioned before, the QoS of p' in terms of FU power is already determined in that they specify an achievable lower-bound on the number of FUs needed for each function and speed-type combination. We then use the left-edge FU binding algorithm to achieve this lower bound. Furthermore, for the changed parts of the new scheduling and MS (S&MS) solution resulting from one of the aforementioned moves, the optimal binding of operations to FUs can be done in linear time by using the left-edge algorithm incrementally only on functionality-speed types one or more of whose FUs have had their bound operations or their scheduled times changed. Unlike PSA, where a *spd-set* move only generates a new speed vector but not a schedule (and PSA thus performs S&B at a second-level hierarchy to further optimize and obtain a complete MSB solution p'), such a move in MS-SALSA only requires binding to be completed without any further optimization considerations (the optimization consideration is only within the simulated annealing based decision of accepting or rejecting the new MSB solution). Thus, both types of MS-SALSA moves generate complete new MSB solutions in a conventional 1-level simulated annealing framework.

IP: 198.54.106.172 On: Tue, 21 Jul 2020 04:01:50
Copyright: American Scientific Publishers
Delivered by Ingenta

Besides the two moves from SALSA in *sch-set*, we added one more scheduling move from another simulated-annealing based S&B technique.[24] Thus, the three moves in *sch-set* are: (1) randomly select an operation and increment its schedule time by 1 cc.[20] (2) randomly select an operation and decrement its schedule time by 1 cc.[20] (3) randomly select an operation and move it to a random cc inside its mobility range;[24] this randomness leads to large jumps in the solution space leading to a better exploration of the solution space within a given number of moves.

It is important to note that a move from either *spd-set* or *sch-set* may cause precedence violations in which case it is an *infeasible* move. A precedence violation occurs when an operation u has an execution range $ER(u)$ that overlaps the execution range of any predecessor (in predecessor set $Pr(u)$) or successor (in successor set $Su(u)$) operation of u. For an operation u, scheduled in cc j, with functionality-speed type $lib_i(u)$, $ER(u) = [j,\ j + d(lib_i(u)) - 1]$, where $d(lib_i(u))$ is the delay of a lib_i FU that u is bound to. A*spd-set* move that changes a set of operations from a faster speed to a slower speed may cause precedence violations, since some operation(s) in $Pr(u)$ or $Su(u)$ of such a changed operation u can result in the overlapped ERs with a larger-delay u. Similarly, a *sch-set* move that changes the schedule time of an operation u may also cause precedence violations where the new scheduling assignment of u can cause overlapped ER ranges among u and $Pr(u) \cup Su(u)$.

To fix precedence violations caused by *sch-set* moves, a greedy technique was presented by SALSA,[20] while other simulated annealing based techniques (e.g., Ref. [24]) discarded the infeasible move which leads to a significant runtime increase to find a feasible move. To fix the precedence violation, starting from the operation whose schedule time is changed by the move, SALSA recursively checks all predecessor and successor operations in a breadth-first manner, and shifts all the operations that are involved in precedence violations to the nearest available cc (nearest cc from its current schedule time that does not overlap the ERs of any predecessor or successor), allocating new FUs if needed. SALSA will stop fixing violations when either there is no violation (either precedence or latency constraint violation) exists or any operation violates the latency constraint to fix the precedence violation, i.e., for an operation u with a functionality-speed type $lib_i(u)$, the nearest available cc j is outside the mobility range $[t_u^{as}, t_u^{al}]$ for operation u, where t_u^{as} is the asap time of u and t_u^{al} is the alap time of u. t_u^{as} and t_u^{al} of an operation u that is either unscheduled or whose schedule is being changed are defined as:

$$t_u^{as} = \max\{t_w^{as} + d(lib_i(w)) \mid w \in parent(u)\} \tag{19}$$

$$t_u^{al} = \min\{t_w^{al} | w \in child(u)\} - d(lib_i(u)) \tag{20}$$

where $lib_i(w)$ is the functionality-speed type of operation w and $d(lib_i(w))$ is the delay of FU of type lib_i that w is bound to. Note also that if an operation w is currently scheduled, then t_w^{as} and t_w^{al} are its scheduled time. If an available cc within the mobility range of u is not available, then this obviously leads to an infeasible solution (latency constraint will be violated), and the corresponding move is discarded.

To fix an infeasible *spd-set* move, we develop and test two different violation fix techniques (to be discussed shortly) that are applicable in different scenarios, called *Viol-fix*1 and *Viol-fix*2. Similar to the failure scenario of the violation-fixing process of SALSA, an infeasible *spd-set* move cannot be rectified when latency constraint violations occur during the process of rectifying the initial violations.

If precedence violation(s) are introduced after making a move from either *spd-set* or *sch-set* and this move cannot be rectified by either of our aforementioned techniques, then this infeasible move is discarded, and another move of the same type (in *spd-set* or *sch-set*) is chosen.

Once we have a feasible or rectified move from either *spd-set* or *sch-set*, and the new S&MS solution is generated, then we can obtain the new complete MSB solution and the corresponding power by performing binding as discussed above. We then proceed along the same steps (4 to 6) as in PSA. It is important to note that, for the new S&MS solution, a native feasible *spd-set* move which causes 0 violation only changes the MS solution in p while the scheduling solution remaining the same. However, a rectified *spd-set* move not only generates a new MS solution but also leads to a new scheduling solution. On the other hand, in both, a native feasible or rectified *sch-set* move, only the scheduling solution is changed and the MS solution remains unchanged.

IP: 198.54.108.172 On: Tue, 21 Jul 2020 01:01:56
Copyright: American Scientific Publishers
Delivered by Ingenta

We next discuss our two violation fixing techniques in MS-SALSA, *Viol-fix*1 and *Viol-fix*2.

Viol-fix 1: *Viol-fix*1 rectifies infeasible *sch-set* moves by using the same technique as in SALSA with the added constraint that we can increase the number of FUs of each functionality-speed type by only one. Furthermore, to rectify infeasible *spd-set* moves, *Viol-fix*1 uses an extended version of the SALSA technique as follows. It is important to note that, a *spd-set* move generally changes more than one operation simultaneously, and thus can cause many precedence violations by having overlapped ERs with their predecessor and successor operations. Therefore, all such operations are required to be processed for fixing all violations. For a given DFG $G(V, E)$, we define the operation set V' as the set of operations whose speed type has been changed by a *spd-set* move and all their predecessor and successor operations. It is necessary to start checking for and fixing precedence violations from the operation(s) with the earliest asap time in V', and then, for each operation that is involved in precedence violations, *Viol-fix*1 uses the same technique as SALSA (with the aforementioned added constraint) to fix violations.

However, *Viol-fix*1 is neither very effective nor runtime-efficient for fixing precedence violations resulting from a *spd-set* move as only local corrections are made for each detected violation without consideration of other violations, and also since the process is iterated for each violation detected in an isolated way, resulting in potentially multiple schedule changes for several operations. To improve these shortcomings of *Viol-fix*1, we developed a second technique called *Viol-fix*2.

*Viol-fix*2: *Viol-fix*2 rectifies infeasible *sch-set* moves by using the same technique as in *Viol-fix*1. To rectify infeasible *spd-set* moves, *Viol-fix*2 performs list scheduling (LS) on the sub-DFG graph $G'(V', E')$ induced in the DFG $G(V, E)$ by V' (E' is the arc set that represents precedence dependency among all operations in V') with the caveat that precedence relations in arcs in E–E' that are incident on operations in V' are respected, and the information of FUs that are busy in each cc executing operations in V–V' (in addition to operations in V') is accounted for in scheduling operations in V' and allocating new FUs if needed. Since this invocation of LS does not schedule any operations in V–V', their schedules remain unchanged from those in the current MSB solution. Furthermore, as in *Viol-fix*1, here too we allow at most one extra FU of each functionality-speed type to be allocated by LS compared to the current MSB solution p; if LS needs to add more than one FU of each functionality-speed type, then the violation fix process fails.

Let R be a vector of numbers of each functionality-speed type FUs used in the current MSB solution p. We define $R_c = R + \boldsymbol{a}$ as the resource constraint for both *Viol-fix*1 and *Viol-fix*2 for the new MSB solution p' where $\boldsymbol{a}$ is a vector of the upper bound increases in the number of FUs allowed for each functionality-speed type. We tested different values for the upper bounds in $\boldsymbol{a}$ in the range of Refs. [1, 4] (all upper bounds in each tested $\boldsymbol{a}$ are uniform in our experiments). We also tested infinite values in $\boldsymbol{a}$ which is the same as conventional SALSA in not having any limit on the allocation of extra FUs for fixing violations. In our experiments, $\boldsymbol{a} =$ all-1 vector provides the best tradeoff between QoS and runtime: it provides an average total power improvement of 17% with only an average factor of 1.1× slower runtime compared to other $\boldsymbol{a}$ vectors we tested. Thus, in the rest of our experiments, we use $\boldsymbol{a} =$ all-1 vector to limit the allocation of extra FUs for p' compared to the current MSB solution p.

Finally, all the initial parameters, solution acceptance and rejection criteria in MS-SALSA are the same as PSA-MLS. Further, as in PSA-MLS, the upper bound on the number of feasible moves (both accepted and rejected moves) for a particular temperature is $M = h \cdot n$, where n is the total number of operations in the input DFG, and h is a constant to control the size of M and thus the runtime.

5. EXPERIMENTAL RESULTS

The proposed hierarchical framework (PSA-MLS) and two competing state-of-the-art power-driven module selection algorithms MWIS[14] and Min-Cut[15] were implemented in C++. All the runs were performed on an Intel Core i7-4710HQ Processor at 2.50 GHz with 16 GB RAM. We tested 13 media-bench DFGs from Ref. [33].

We constructed 2- and 4-speed 16-bit FU libraries, denoted by 2-*lib* and 4-*lib* respectively, based on various arithmetic FU designs in Ref. [12] for 32 nm CMOS technology. We described these designs using Verilog, and synthesized them and obtained their delay and power parameters via Synopsys Design Vision.[35] We use 0.33 ns as the clock period which is the delay of the fastest adder. Table I shows the design parameters of 2/4-*lib*s for the main operation functional types: addition, multiplication, and division; 2-*lib* has only the slowest and fastest speed types of 4-*lib*. For non-arithmetic operations (e.g., left shift, mux, demux), we only create one functionality-speed type with a delay of 1 cc, since for these simple operations there is not much design diversity that changes their delay and power significantly. We also synthesized 16-bit 2-to-1 mux, 1-to-2 demux, and register in Synopsys Design Vision and obtained the following delay and power values: they all have delays in picoseconds (much smaller than a cc), $dp_{21\text{mux}/12\text{demux}} = 0.312$ uW, $lp_{21\text{mux}/12\text{demux}} = 1.347$ uW, $dp_{\text{reg}} = 1.19$ uW, and $lp_{\text{reg}} = 3.6$ uW. The trends of the relative result differences among the techniques that we shortly present should hold for any given library as we have also seen for other analyzed or derived-from-literature libraries.

IP: 198.54.106.172 On: Tue, 21 Jul 2020 01:01:54
Copyright: American Scientific Publishers
Delivered by Ingenta

As mentioned before, the main goal of this paper is minimizing total (dynamic + leakage) power consumption that is consumed by functional units, and that comparison is presented here. Furthermore, within the context of minimizing only FU power, we also obtain the total architecture power for each MSB solution generated using different techniques and compare this as well. As we mentioned before, to obtain the total architecture power, we first generate the corresponding design of an MSB solution by performing *post*-MSB *bind*ing, *post-Bind*, (Fig. 3). Based on a given MSB solution, we use the proposed modified left-edge FU binding (MLE) to perform post-MSB FU re-binding (a default binding is performed during the earlier stage of module-selection and scheduling—MS&S) for reducing the sizes and hence power of muxes and demuxes. Then, we use the basic left-edge algorithm to perform register binding. Finally, we obtain the corresponding total architecture power TP_{arch} for the given MSB solution according to Eq. (13).

5.1. FU Total (Dynamic + Leakage) Power Comparison Among PSA and MS-SASLA with Viol-fix1 and Viol-fix2

We first discuss the FU power and runtime results among different simulated-annealing based techniques using the 4-speed library 4-*lib*. We also obtain the FU total power (TP_{FU}) results of an optimal 0/1-ILP formulation[18] that is executed using CPLEX[34] as the baseline for TP_{FU} result comparison among three simulated-annealing based techniques: MS-SALSA with *Viol-fix*1 and *Viol-fix*2, and PSA-MLS. We use $l_c = 1.2 \times l_{\text{as-slow}}$, where $l_{\text{as-slow}}$ is the ASAP latency (critical path latency) with all operations assigned to their slowest speeds; the coefficient 1.2 provides a moderately large solution space for the MSB problem for the purpose of exercising the different algorithms to a reasonable degree. Figure 5 and Table II show percentage increase in overall TP_{FU} of the simulated-annealing techniques compared to 0/1-ILP for different h values across all the DFGs from Ref. [33] (these are specified in Table III) except the largest DFG (*inv-matrix*), since for 4-*lib*, 0/1-ILP failed to generate TP_{FU} solution for *inv-matrix* due to insufficient memory. Since MS-SALSA has better TP_{FU} results and a small runtime disadvantage of 3% over all h values with *Viol-fix*2 than with*Viol-fix*1, we use the results of MS-SALSA with *Viol-fix*2 to represent the results of MS-SALSA for all subsequent result comparisons. Further, when we henceforth refer to MS-SALSA we will mean MS-SALSA with *Viol-fix*2.

We next compare MS-SALSA and PSA-MLS under three criteria: Equal h values, equal/similar runtime, and equal/similar QoS (TP_{FU} in this case). We use PSA-MLS result with $h = 4$ ($M = 4n$) as the baseline and determine execution parameters for MS-SALSA that provide equal or nearest possible values of the aforementioned metrics for the desired comparisons:

(1) Equal h value: The corresponding equal h value for MS-SALSA to compare to the aforementioned baseline PSA-MLS is $h = 4$. PSA-MLS has a TP_{FU} advantage of 8.3% and about a 2× runtime advantage compared to MS-SALSA. Furthermore, Figure 5 also shows that for different h values, PSA-MLS has significantly better power results than MS-SALSA.

(2) Equal/Similar runtime: This comparison corresponds to MS-SALSA with $h = 2$ for which its runtime is 165.22 seconds versus PSA-MLS's runtime of

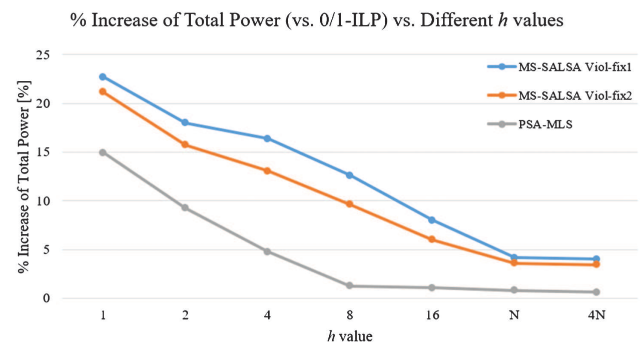

Fig. 5. Percentage increase of total power compared to 0/1-ILP of PSA, and MS-SALSA with *Viol-fix*1 and *Viol-fix*2 for different h values.

Table II. Percentage increase of FU total power compared to 0/1-ILP and runtime results of MS-SALSA with viol-fix 1 and viol-fix2, and PSA-MLS for different h values for 4-lib.

	% Increase of total power versus 0/1-ILP			Runtime [s]			Runtime ratio (vs. PSA)	
h value	MS-SALSA viol-fix 1	MS-SALSA viol-fix 2	PSA-MLS	MS-SALSA viol-fix 1	MS-SALSA viol-fix 2	PSA-MLS	MS-SALSA viol-fix 1	MS-SALSA viol-fix 2
1	22.71	21.18	14.95	73.75	68.6	40.41	1.83	1.7
2	18.01	15.74	9.27	181.51	165.22	88.92	2.04	1.86
4	16.36	13.08	4.77	300.09	291.61	151.4	1.98	1.93
8	12.62	9.65	1.26	656.04	590.05	287.55	2.28	2.05
16	8.03	6.02	1.07	1102.8	950.8	458.37	2.41	2.07
N	4.17	3.61	0.81	6950.4	7075.4	3813.03	1.82	1.86
4N	4.02	3.46	0.63	31211.1	30138.7	12473.6	2.5	2.42
Avg.	12.96	10.39	4.68	5782.24	5611.48	2473.33	2.34	2.27

151.4 seconds for $h = 4$. In this configuration, PSA-MLS has 11% TP_{FU} improvement over MS-SALSA.
(3) Equal/Similar QoS: For a similar QoS result compared to PSA-MLS with $h = 4$ (4.77% from optimal), we consider MS-SALSA's results with $h = 16$ (6.02% from optimal). PSA-MLS has 6× runtime advantage compared to MS-SALSA.

Therefore, PSA-MLS has comprehensive advantages over MS-SALSA: (1) Computational efficiency: For similar QoS's as well as for similar solution space as represented by equal h values, PSA-MLS is several factors faster than MS-SALSA (in the range of [4, 45] for equal/similar QoS's, and in the range of [1.7, 2.5] for equal h values). (2) QoS Efficacy: Much better QoS is obtained by PSA-MLS than MS-SALSA under similar runtime limits.

IP: 198.54.106.172 On: Tue, 21 Jul 2020 01:01:50
Copyright: American Scientific Publishers
Delivered by Ingenta

5.2. FU and Architecture Leakage Power Comparisons

Next, we compare PSA-MLS to two constructive state-of-the-art heuristic techniques (MWIS[14] and Min-Cut[15]) for leakage power (LP) minimization, which the latter two techniques perform For these comparisons, we choose $h =$ 4 for both MS-SALSA PSA-MLS which provides a good tradeoff between QoS and runtime. We first compare total FU leakage power (LP_{FU}) results among different techniques (0/1-ILP, MS-SALSA, MWIS, Min-Cut, and PSA-MLS) with 2-*lib*, since MWIS and Min-Cut techniques obtain leakage power and work for two speeds only (to extend these techniques to work with a 4-speed library, will need significant changes to the original algorithms to the extent that these extensions become quite different algorithms). For the comparison to MWIS, which is mainly a power-driven module selection technique, we implemented the algorithm, and as in Ref. [14], experimented with different initial S&B solutions that it needs. For the comparison to Min-Cut, there is a requirement that $l_c < l_{\text{as-slow}}$ but such that it is satisfiable by using faster FUs in the library. We use $l_c = 0.6 \times l_{\text{as-slow}}$, which satisfies the above requirements for library 2-*lib*. For both MS-SALSA and PSA-MLS, the results correspond to an h value of 4.

Table III shows the LP_{FU} and runtime results for 5 techniques for all the DFGs from Ref. [33] with 2-*lib* and $l_c = 0.6 \times l_{\text{as-slow}}$. Compared to MS-SALSA, PSA-MLS has an average LP_{FU} improvement of 7.1% with a runtime advantage of 2× (recall that the earlier comparison to MS-SALSA was for 4-*lib*, $l_c = 1.2 \times l_{\text{as-slow}}$, and total power minimization). Compared to MWIS and Min-Cut, PSA-MLS has 11.2% and 7.8% LP_{FU} improvement, while having runtime advantages of 2.5× and 3×, respectively. Compared to 0/1 ILP, PSA-MLS has an average optimality gap of 9.7% with a runtime advantage of 209×.

Next, we generate the architecture using *post-Bind* for the MSB solutions for each technique and compare the total leakage power of the HLS architecture among all techniques. In Table IV, the leakage power of registers (LP_{reg}), mux and demux ($LP_{mux/demux}$), and the total leakage power of the architecture ($LP_{arch} = LP_{FU}$ (from Table III)$+ LP_{reg} + LP_{mux/demux}$) are presented. The power results in Table IV show that: (a) compared to other approximate techniques, the LP_{arch} power improvement of PSA-MLS are similar to its LP_{FU} power improvements over them, and (b) PSA-MLS's gaps for LP_{arch} and LP_{FU} with respect to the 0/1 ILP method are similar. Note that the runtime of the left-edge-based *post-Bind* is extremely fast, and thus we do not show the *post-Bind* runtime in Table IV (for the largest size DFG, the runtime for *post-Bind* is less than one second).

5.3. FU and Architecture Total Power Comparisons

We also ran 0/1-ILP, MS-SALSA, and PSA-MLS for minimizing total FU total power TP_{FU}. Then, we generate the HLS architecture using *post-Bind* for each technique and obtain also the total power of the architecture (TP_{arch}) among all techniques. Table V shows the total power of FUs (TP_{FU}), registers (TP_{reg}) (Eq. (13)), muxes and demuxes ($TP_{mux/demux}$) (Eq. (10)), and the entire architecture (TP_{arch}) for the above techniques with 4-*lib* and a latency constraint of $l_c = 1.2 \times l_{\text{as-slow}}$ (note that MWIS and Min-Cut do not optimize DP_{FU} and are thus not included in these comparisons). For both MS-SALSA and PSA-MLS, the h value is 4. For 4-*lib*, 0/1-ILP cannot

IP: 198.54.106.172 On: Tue, 21 Jul 2020 01:01:50
Copyright: American Scientific Publishers
Delivered by Ingenta

ults for 0/1-ILP,[18] MS-SALSA, MWIS,[14] Min-Cut,[15] and PSA-MLS for 2-*lib*. $l_c = 0.6 \times l_{\text{as-slow}}$. $h = 4$ for MS-SALSA and PSA-MLS. A positive (negative) %
crease).

LP_{FU} results for 2-lib [mW]				% LP_{FU} imp. of PSA-MLAS versus others				Runtime [secs]				
MS-SALSA	MWIS	Min-cut	PSA-MLS	0/1-ILP (%)	MS-SALSA (%)	MWIS (%)	Min-cut (%)	0/1-ILP	MS-SALSA	MWIS	Min-cut	PSA-MLS
0.23	0.23	0.23	0.23	0.0	0.0	0.0	0.0	0.14	0.13	0.11	0.14	0.07
0.18	0 19	0.17	0.17	−13.2	5.0	9.9	−0.9	0.46	1.30	1.4	1.2	0.63
0.31	0.30	0.30	0.31	−4.7	1.0	−4.7	−4.7	0.49	1.17	1.7	1.5	0.5
0.09	0.10	0.09	0.09	−9.2	0.0	14.4	2.8	13.5	2.2	4.8	3.9	0.9
0.35	0.36	0.33	0.32	−7.9	7.4	11.2	2.3	16.1	2.1	8.5	8.4	0.96
0.38	0.44	0.37	0.36	0.0	3.8	17.6	3.0	20.0	1.9	9.4	7.7	0.8
0.29	0.34	0.31	0.29	−25.8	0.0	14.1	5.8	31.1	5.6	6.3	5.8	2.7
0.52	0.59	0.51	0.51	−8.9	1.9	14.1	0.1	37.4	6.0	10.8	8.6	2.9
0.48	0 55	0.46	0.42	−7.6	12.1	23.0	7.9	125.3	7.1	14.1	10.7	3.3
0.24	0.26	0.24	0.21	−24.0	10.1	19.1	10.9	146.8	9.5	13.7	12.3	4.1
0.36	0.38	0.35	0.27	−14.3	22.7	26.9	20.6	3327.4	16.5	22.3	20.1	7.3
0.72	0.79	0.76	0.66	−6.9	9.3	17.1	14.0	3646.8	26.1	29.4	19.8	10.4
1.33	119	1.40	1.23	−12.2	7.0	−3.5	11.9	6419.2	58.8	79.5	65.3	31.5
5.46	5.71	5.50	5.07	−9.7	7.1	11.2	7.8	13784.7	138.4	202.0	165.5	66.1

IP: 198.54.106.172 On: Tue, 21 Jul 2020 01:01:50
Copyright: American Scientific Publishers
Delivered by Ingenta

P$_{arch}$) results for 0/1-ILP, MS-SALSA, MWIS, min-cut, and PSA-MLS for 2-*lib* $l_c = 0.6 \times l_{as\text{-}slow}$ $h = 4$ for MS-SALSA and PSA-MLS.

LP results for 2-lib with post-bind														LP$_{arch}$ % imp. of PSA versus other			
LP$_{reg}$ [mW]				LP$_{mug/demux}$ [mW]					LP$_{arch}$ [mW]					LP$_{arch}$ % imp. of PSA versus other			
-SA	MWIS	Min-cut	PSA-MLS	0/1-ILP	MS-SALSA	MWIS	Min-Cut	PSA-MLS	0/1-ILP	MS-SALSA	MWIS	Min-cut	PSA-MLS	0/1-ILP (%)	MS-SALSA (%)	MWIS (%)	Min-cut (%)
08	0.008	0.008	0.008	0.001	0.001	0.001	0.001	0.001	0.23	0.23	0.23	0.23	0.23	0.0	0.0	0.0	0.0
13	0.015	0.013	0.015	0.005	0.004	0.004	0.005	0.004	0.17	020	021	0.19	0.19	−11.8	3.8	8.8	−1.4
23	0.024	0.023	0.024	0.005	0.005	0.003	0.004	0.003	0.32	0.34	0.32	0.32	0.34	−4.7	1.0	−4.2	−4.4
11	0.009	0.011	0.011	0.006	0.006	0.005	0.006	0.006	0.10	0.11	0.12	0.11	0.11	−7.6	0.0	10.5	2.4
30	0.037	0.035	0.033	0.009	0.009	0.008	0.008	0.008	0.34	0.39	0.41	0.37	0.36	−7.1	6.2	10.8	2.6
23	0.020	0.020	0.019	0.007	0.005	0.007	0.006	0.007	0.38	0.40	0.46	0.40	0.39	−0.4	4.1	16.8	2.7
69	0.075	0.071	0.069	0.016	0.019	0.010	0.010	0.019	0.31	0.38	0.43	0.39	0.38	−22.9	0.0	10.6	2.9
20	0.068	0.072	0.047	0.021	0.005	0.020	0.020	0.014	0.55	0.54	0.68	0.60	0.57	−3.7	−4.8	16.2	5.2
72	0.047	0.042	0.050	0.016	0.009	0.147	0.014	0.015	0.44	0.56	0.74	0.51	0.48	−9.1	13.3	34.4	5.4
71	0.054	0.057	0.048	0.014	0.012	0.014	0.016	0.014	0.23	0.32	0.33	0.31	0.27	−17.3	14.2	17.0	12.0
57	0.064	0.054	0.061	0.019	0.015	0.020	0.023	0.024	0.31	0.43	0.46	0.42	0.36	−15.5	15.9	21.9	15.0
91	0.088	0.093	0.107	0.036	0.033	0.034	0.034	0.031	0.73	0.85	0.91	0.89	0.79	−8.3	6.3	13.1	10.8
31	0.137	0.133	0.120	0.053	0.048	0.047	0.052	0.053	1.27	1.50	1.37	1.58	1.41	−10.5	6.6	−2.3	11.2
20	0.644	0.632	0.612	0.207	0.170	0.321	0.198	0.199	5.40	6.25	6.67	6.33	5.88	−9.0	5.8	11.9	7.1

IP: 198.54.106.172 On: Tue, 21 Jul 2020 01:01:50
Copyright: American Scientific Publishers
Delivered by Ingenta

) results of for 0/1-ILP, MS-SALSA and PSA-MLS for 4-*lib*. $l_c = 1.2 \times l_{\text{as-slow}}$. $h = 4$ for MS-SALSA and PSA-MLS.

TP (DP+LP) results for 4-lib										% TP imp. of PSA-MLS versus				Runtime [secs]		
W]	TP_{reg} [mW]			$TP_{mux/demux}$ [mW]			TP_{arch} [mW]			TP_{FU} [mW]		TP_{arch} [mW]				
PSA-MLS	0/1-ILP	MS-SALSA	PSA-MLS	0/1-ILP	MS-SALSA	PSA-MLS	0/1-ILP	MS-SALSA	PSA-MLS	0/1-ILP (%)	MS-SALSA (%)	0/1-ILP (%)	MS-SALSA (%)	0/1-ILP	MS-SALSA	PSA-MLS
0.883	0.008	0.008	0.009	0.001	0.001	0.001	0.892	0.892	0.892	0.0	0.0	0.0	0.0	3.3	0.5	0.2
0.710	0.015	0.013	0.013	0.005	0.003	0.005	0.759	0.903	0.757		16.5	0.3	16.1	6.2	2.0	1.4
1.300	0.025	0.025	0.026	0.005	0.005	0.005	1.298	1.371	1.431	−2.5	3.1	−2.5	3.0	14.4	2.5	1.3
0.285	0.008	0.007	0.008	0.005	0.005	0.004	0.284	0.366	0.497	−5.3	19.4	−4.8	18.7	21.5	3.0	1.7
1.494	0.036	0.035	0039	0.009	0.009	0.009	1.518	1.716	1.542	−1.4	10.6	−1.6	10.1	23.7	5.6	3.0
0.953	0.026	0.026	0.026	0.007	0.008	0.074	0.899	1.352	1.052	−10.1	27.7	−17.3	22.0	179.3	6.4	3.4
0.726	0.063	0.063	0.060	0.017	0.016	0.014	0.565	0.746	0.800	−49.7	−8.9	−41.7	−7.3	217.7	28.9	13.8
2.633	0.070	0.073	0.076	0.020	0.020	0.020	2.624	2.875	2.729	−3.9	5.3	−4.0	5.1	623.4	28.4	13.9
2.180	0.047	0.043	0.050	0.016	0.017	0.015	2.186	2.299	2.245	−2.7	2.6	−2.7	2.3	16031	29.1	16.4
0.988	0.066	0.064	0.050	0.018	0.018	0.015	1.038	1.106	1.053	−3.6	3.6	−1.5	4.9	*86.4k	39.7	21.4
1.424	0.065	0.064	0.058	0.024	0.023	0.024	1.459	1.687	1.506	−3.9	11.0	−3.2	10.8	*86.4k	59.9	29.3
3.354	0.094	0.089	0.103	0.034	0.036	0.034	3.372	3.619	3.491	−3.4	4.0	−3.5	3.6	*86.4k	112.3	56.9
6.540	**	0.136	0.148	**	0.055	0.053	**	6.881	6.741	**	2.2	**	2.0	**	354.9	175.1
16.96	0.52	0.51	0.52	0.16	0.16	0.22	16.89	18.93	17.70	−4.6	7.1	−4.8	6.5	276.3k	318.1	162.6
23.50	–	0.65	0.67	–	0.22	0.27	–	25.81	24.44	–	5.8	–	5.3	–	673.0	337.7

per-bound runtime of 24 hours, and thus the runtime results are 86.4 k seconds.

IP: 198.54.106.172 On: Tue, 21 Jul 2020 01:01:50
Copyright: American Scientific Publishers
Delivered by Ingenta

$_{\text{ch}}$) for MS-SALSA and PSA-MLS with *post-Bind* and *in-Bind* for 4-*lib*. $l_c = 1.2 \times l_{\text{as-slow}}$. $h = 4$ for MS-SALSA and PSA-MLS.

P) results for 4-lib with post-blind									TP (DP + LP) results for 4-lib with in-blind													
	$TP_{mux/demux}$ [mW]		TP_{arch} [mW]		% Imp. PSA versus MS-SALSA		TP_{FU} [mW]		TP_{reg} [mW]		$TP_{mux/memux}$ [mW]		TP_{arch} [mW]		% Imp. PSA versus MS-SALSA		TP_{sum} % Imp. in-bind versus post-bind		Runtime with in-bind [Secs]	
SA- ILS	MS-SALSA	PSA-MLS	MS-SALSA	PSA-MLS	TP_{FU} (%)	TP_{arch} (%)	MS-SALSA	PSA-MLS	MS-SALSA	PSA-MLS	MS-SALSA	PSA-MLS	MS-SALSA	PSA-MLS	TP_{FU} (%)	TP_{arch} (%)	MS-SALSA (%)	PSA-MLS (%)	MS-SALSA	PSA-MLS
009	0.001	0.001	0.89	0.89	0.0	0.0	0.88	0.88	0.008	0.008	0.001	0.001	0.89	0.89	0.0	0.0	0.0	0.0	1.1	0.74
013	0.003	0.005	0.90	0.76	16.5	16.1	0.94	0.76	0.014	0.013	0.004	0.003	0.95	0.78	18.4	18.3	−5.7	−3.0	7.2	5.13
026	0.005	0.005	1.37	1.33	3.1	3.0	1.38	1.36	0.028	0.024	0.004	0.005	1.42	1.39	2.0	2.2	−3.3	−4.1	7.7	4
008	0.005	0.004	0.37	0.30	19.4	18.7	0.31	0.32	0.013	0.008	0.006	0.005	0.33	0.33	−2.0	0.0	8.7	−12.3	10	6.8
039	0.009	0.009	1.71	1.54	10.6	10.1	1.62	1.57	0.038	0.038	0.008	0.009	1.67	1.62	3.2	3.1	2.7	−4.9	19.8	5.9
026	0.008	0.074	1.35	1.05	27.7	22.0	0.92	0.91	0.025	0.026	0.008	0.008	0.95	0.95	0.7	0.5	29.6	10.1	26.2	8.5
060	0.016	0.014	0.75	0.80	−8.9	−7.3	0.64	0.53	0.061	0.063	0.013	0.016	0.72	0.61	18.0	15.2	4.1	24.2	107.5	31.1
076	0.020	0.020	2.87	2.73	5.3	5.1	2.73	2.59	0.079	0.073	0.020	0.020	2.83	2.68	5.1	5.2	1.6	1.7	94.3	33.5
050	0.017	0.015	2.30	2.25	2.6	2.3	2.16	2.14	0.052	0.047	0.016	0.016	2.22	2.20	0.8	0.9	3.3	1.9	111.2	65.7
050	0.018	0.015	1.11	1.05	3.6	4.9	1.05	0.99	0.055	0.064	0.017	0.017	1.12	1.07	5.5	4.3	−0.9	−1.5	165.7	101.4
058	0.023	0.024	1.69	1.51	11.0	10.8	1.46	1.44	0.065	0.068	0.022	0.024	1.55	1.53	1.3	0.9	8.4	−1.7	253.5	115.5
103	0.036	0.034	3.62	3.49	4.0	3.6	3.65	3.51	0.103	0.093	0.034	0.035	3.79	3.64	4.0	4.1	−4.7	−4.1	442	211.2
148	0.055	0.053	6.88	6.74	2.2	2.0	6.44	6.24	0.148	0.137	0.52	0.058	6.64	6.44	3.1	3.1	3.5	4.5	1037.5	479.2
666	0.217	0.217	25.81	24.14	5.8	5.3	24.18	23.24	0.689	0.662	0.207	0.218	25.08	24.12	3.9	3.8	2.8	1.3	2283.7	1068.7

generate optimal results for three large DFGs in the given runtime bound of 24 hours, and thus we took the near optimal solution determined at this upper-bound runtime, which are labeled with a '$*$' suffix in Table V. Also, 0/1-ILP failed to generate solutions for the largest DFG (*inv-matrix*) due to insufficient memory, and this situation is labeled by "$**$". Furthermore, for comparing to 0/1-ILP, TP_{FU} and runtime results of all three techniques are provided in the row, called "Total-1," of Table V that does not include the largest DFG (*inv-matrix*). Moreover, for comparing between MS-SALSA and PSA-MLS, the results of MS-SALSA and PSA-MLS, which include all DFGs, are shown in the row called "Total-2" of Table V. For TP_{FU} which all techniques compared here directly optimize PSA-MLS compared to 0/1-ILP has an average optimality gap of 4.7% with a runtime advantage of 1900$\times$ (similar to the prior LP comparison section, the runtime results in this table only consider the runtime of MSB processing since the *post-Bind* processing is extremely fast). Compared to MS-SALSA, PSA-MLS obtains 5.8% smaller TP_{FU} with a runtime advantage of 2$\times$. For TP_{arch}, PSA-MLS has an average optimality gap of 4.8% compared to 0/1-ILP and a 5.3% TP_{arch} advantage compared to MS-SALSA. The similarity of the comparative results among the techniques for TP_{FU} and TP_{arch} results show that optimizing FU power is strongly related to the architecture power optimization.

5.4. Results from Simultaneous Minimization of FU and Mux, Demux Power

Recall that the optimization function in PSA-MLS with *post-Bind* in the previous subsection is FU power only Now we consider directly optimizing the entire architecture power within the two simulated annealing techniques, PSA-MLS and MS-SALSA with changes that involve performing both power-driven LE-based FU binding and LE-based register binding in the evaluation step of a simulated annealing move (this binding process is called *in-Bind* since it is done within the optimization wrapper, rather the after it). This way, the entire architecture power is determined for each move, and thus each simulated annealing technique optimizes this power function. To make PSA-MLS address architecture power minimization, the changes needed are: (1) The basic left-edge binding in MLS is replaced by the power-driven modified left-edge binding (MLE) that was described in Section 3.3. Thus, the power consumption of muxes and demuxes (or, equivalently, the total size of muxes and demuxes) can be considered simultaneously within the scheduling result of each MLS iteration. (2) Before the power evaluation step in PSA, we perform basic left-edge-based register binding for the scheduling and FU binding solution returned by MLS for the current move.

Similarly, to make MS-SALSA address architecture power minimization, MLE-based FU binding, and LE-based register binding are processed after each new MSB solution has been generated for the current move.

We compare the TP_{arch} power results for both PSA-MLS and MS-SALSA between *in-Bind* and *post-Bind* approaches (i.e., between directly optimizing TP_{arch} and TP_{FU} only, respectively) in Table VI. For both PSA-MLS and MS-SALSA, with the consideration of overall architecture power (*in-Bind*), TP_{arch} results are further improved by 2.8% and 6.2% over optimizing TP_{FU} only (*post-Bind*) but with runtime increases by factors of 3.39$\times$ and 3.16$\times$, respectively (note that PSA-MLS's power is still appreciably lower than MS-SALSA's using *in-Bind*). The runtime increases significantly as the binding steps have significant-enough complexity (linear in the number of operations in the DFG) that have to be incurred for every move unlike they being only incurred once in the *post-Bind* approach. The small TP_{arch} improvements using *in-Bind* show that optimizing TP_{FU} only (wherein we use *post-Bind*) has a strong correlation to optimizing TP_{arch}, and given the *post-Bind* based approach's significantly more efficient runtime, it is a much more attractive strategy within PSA-MLS for optimizing the total architecture power.

6. CONCLUSIONS

In this paper, we presented a hierarchical framework for the purpose of efficiently and effectively exploring a reasonably extensive functional-unit (FU) power solution space for the HLS problem of combined module selection, scheduling and binding (MSB). With a hierarchical framework using a stochastic-constructive algorithm combination, PSA-MLS efficiently explores module (speed) selection solutions using simulated annealing, while, for each such solution, an effective constructive algorithm, MLS, provides a good scheduling and binding solution to obtain a complete MSB solution. The proposed framework can obtain optimal or near-optimal power solutions with very efficient runtimes compared to an optimal 0/1-ILP formulation. For comparison, we also developed a good flat simulated annealing technique MS-SALSA for the power-driven MSB problem by combining and significantly extending prior simulated annealing techniques for the scheduling and binding problem. PSA-MLS also provides significantly better FU as well as complete HLS architecture (FU + mux/demux + register) power solutions at faster runtimes than competing state-of-the-art approximate algorithms like MWIS. Min-Cut, and MS-SALSA. PSA-MLS's appreciable power and runtime advantages over MS-SALSA also underscores the basic thesis of this paper about the advantage of a hierarchical stochastic-deterministic algorithm combination over a flat stochastic-only algorithm. We also showed that optimizing FU power within PSA-MLS is strongly correlated to optimizing the HLS architecture power, and explicitly optimizing the architecture power within PSA-MLS only provides small improvements in the architecture power compared to optimizing FU power only, but has a significant enough

IP: 198.54.106.172 On: Tue, 21 Jul 2020 01:01:51
Copyright: American Scientific Publishers
Delivered by Ingenta

runtime increase that it does not provide a good tradeoff between QoS and runtime.

Acknowledgment: This work has in part been made possible by NSF grant CCF-1248945. We also thank the anonymous reviewers for their comments, which have improved the paper.

References

1. ITRS Working Group, 2011 International Technology Roadmap for Semiconductors (ITRS): Design, Semiconductor Industry Association, Washington, DC, August **(2011)**.
2. Z. Zhang, D. Chen, S. Dai, and K. Campbell, High-level synthesis for low-power Design. *IPSJ Transactions on System LSI Design Methodology* 8, 12 **(2015)**.
3. S. M. Logesh, D. S. Ram, and M. C. Bhuvaneswari, Survey of high-level synthesis techniques for area, delay and power optimization. *International Journal of Computer Applications* 32, 3935 **(2011)**.
4. W. T. Shiue and C. Chakrabarti, ILP-based scheme for low-power scheduling and resource binding, *Proc. Int. Symp. Circuits and Systems* **(2000)**, Vol. 3, pp. 279–282.
5. A. M. Sllame and V. Drabek, An Efficient List-Based Scheduling Algorithm for High-Level Synthesis, *Proc. of IEEE Euromicro Symposium on Digital System Design: Architecture Methods and Tools*, September **(2002)**, pp.316–323.
6. P. G. Paulin and J. P. Knight, Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* 8, 661 **(1989)**.
7. D. Chen, J. Cong, Y. Fan, and J. Xu, Optimality study of resource binding with Multi-Vdds. *Proc. DAC* 580 **(2006)**.
8. J. M. Chang and M. Pedram, Energy minimization using multiple supply voltages. *IEEE Transactions on VLSI* 5, 436 **(1997)**.
9. D. Chen, J. Cong, and J. Xu, Optimal module and voltage assignment for low-power. *ASP-DAC* 850 **(2005)**.
10. A. K. Allam and J. Ramanujam, Modified force-directed scheduling for peak and average power optimization using multiple supply-voltages. *ICICDT'06* 2006, 1 **(2006)**.
11. Z. Gu, Y. Yang, J. Wang, R. P. Dick, and L. Shang, TAPHS: Thermal-aware unified physical level and high-level synthesis, *Proceedings of ASP-DAC*, IEEE **(2006)**.
12. H. Liu, W. Lee, and Y. Chang, A provably good approximation algorithm for power optimization using multiple supply voltages, *DAC*, IEEE **(2007)**.
13. W. Jiang, Z. Zhang, M. Potkonjak, and J. Cong, Scheduling with integer time budgeting for low-power optimization, *ASP-DAC*, IEEE **(2008)**.
14. X. Tang, H. Zhou, and P. Banerjee, Leakage power optimization with dual-V_{th} library in high-level synthesis, *Proc. DAC* **(2005)**, pp. 202–207.
15. N. Wang, S. Chen, and T. Yoshimura, Min-cut based leakage power aware scheduling in high-level synthesis, *ISQED* 164 **(2013)**.
16. K. S. Khouri and N. K. Jha, Leakage power analysis and reduction during behavioral synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 10.6, 876 **(2002)**.
17. Y. Chen, Y. Xie, Y. Wang, and A. Takach, Minimizing leakage power in aging-bounded high-level synthesis with design time multi-V_{th} assignment, *ASP-DAC*, IEEE **(2010)**.
18. S. P. Mohanty and N. Ranganathan, Simultaneous peak and average power minimization during datapath scheduling. *IEEE TCAS* 52, 1157 **(2005)**.
19. A. Sengupta, R. Sedaghat, and P. Sarkar, A multi structure genetic algorithm for integrated design space exploration of scheduling and allocation in high level synthesis for DSP kernels. *Swarm and Evolutionary Computation* 7, 35 **(2012)**.
20. J. A. Nestor and G. Krishnamoorthy, SALSA: A new approach to scheduling with timing constraints. *IEEE Transactions on Computer-Aided Design* 12, 1107 **(1993)**.
21. F. Ferrandi, C. Pilato, D. Sciuto, and A. Tumeo, Mapping and scheduling of parallel C applications with ant colony optimization onto heterogeneous reconfigurable MPSoCs, *ASP-DAC* **(2010)**, pp. 799–804.
22. S. Dutt and O. Shi, Co-exploration of unit-time leakage power and latency spaces for leakage energy minimization in high-level synthesis. *J. Low Power Electronics* 12, 295 **(2016)**.
23. P. Behrooz, *Computer Arithmetic: Algorithms and Hardware Designs*, New York, Oxford University Press **(2000)** Vol. 19, pp. 512583–512585.
24. J. C. Alves and J. S. Matos, A simulated annealing approach for high-level synthesis with reconfigurable functional units, *Proc. of the 38th Midwest IEEE Symposium* **(1995)**, pp. 314–317.
25. A. Hashimoto and J. Stevens, Wire routing by optimizing channel assignment within large apertures, *8th DAC Workshop* **(1971)**.
26. J. Midwinter, Improving interconnect for the behavioral synthesis of ASICs, M.Sc. Thesis, Carleton University, April **(1988)**.
27. B. J. Sheu, D. L. Scharfetter, P.-K. Ko, and M.-C. Jeng, BSIM: Berkeley short-channel IGFET model for MOS transistors, *IEEE J. Solid-State Circuits* 22, 558 **(1987)**.
28. D. Chen, J. Cong, and J. Xu, Optimal simultaneous module and multivoltage assignment for low power. *ACM Trans. on Design Automation of Electronic Systems* 11, 362 **(2006)**.
29. S.-H. Huang, W.-P. Tu, and B.-H. Li, High-level synthesis for minimum-area low-power clock gating. *Journal of Information Science and Engineering* 28, 971 **(2012)**.
30. D. Dal and N. Mansouri, Power optimization with power islands synthesis. *TCAD* 28, 1025 **(2009)**.
31. D.-C. Juan, Y.-L. Chuang, D. Marculescu, and Y.-W. Chang, Statistical thermal modeling and optimization considering leakage power variations, *Proc. DATE* **(2012)**, pp. 605–610.
32. D. Gajski, N. Dutt, A. Wu, and S. Lin, High-Level Synthesis, Introduction to Chip and System Design, Kluwer Academic Publishers, Norwell, MA **(1992)**.
33. ExPRESS benchmark, http://express.ece.ucsb.edu/benchmark.
34. CPLEX–IBM, https://www.ibm.com/software/commerce/optimization/cplex-optimizer/.
35. Design Vision (A GUI version of Design Compiler), https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-synthesis.html.

IP: 198.54.106.172 On: Tue, 21 Jul 2020 01:01:59
Copyright: American Scientific Publishers
Delivered by Ingenta

Xiuyan Zhang

Xiuyan Zhang *received his B.S. degree in electrical engineering from University of Illinois at Chicago in 2013. He is currently working toward the Ph.D. degree at the Department of Electrical and Computer Engineering, University of Illinois at Chicago. His research interests include CAD for VLSI and optimization algorithms.*

Ouwen Shi

Ouwen Shi *was a Ph.D. student and received M.S. degree in electrical and computer engineering from University of Illinois at Chicago in 2017. He is currently a software engineer at Cadence Design Systems working on timing- and power-driven optimization in physical design automation, with a focus on routing congestion and instance density monitoring, and layer assignment.*

Jian Xu

Jian Xu *received the Ph.D. degree (F'12-S'19) in Computer Engineering from University of Illinois at Chicago.*

Shantanu Dutt

Shantanu Dutt *is a professor at the Department of Electrical and Computer Engineering, University of Illinois at Chicago. He received his Ph.D. in computer science and engineering from the University of Michigan, Ann Arbor, his M.Tech. in computer engineering from Indian Institute of Technology, Kharagpur, and his B.Tech. in electronics and communication engineering from the M.S. University of Baroda, India. Professor Dutt was awarded a Research Initiation Award by the National Science Foundation. He has received a Best-Paper award at the Design Automation Conference (DAC), 1996, a Most-Influential-Paper award from the Fault-Tolerant Computing Symposium (FTCS) in 1995 (for an FTCS'88 paper), a Best-Paper nomination at DAC 2004, and was a featured speaker (1 of 2) at the Int'l Conference on CAD (ICCAD), 2006. His research is or has been funded by NSF, DARPA, AFOSR and companies like Xilinx and Intel. He has published about 80 papers in well-recognized archival journals and refereed conferences in all the above areas. His current technical interests include CAD for sub-micron VLSI circuits, optimization algorithms, fault-tolerant computing, and testing and trusted design for VLSI and FPGA systems. He has been part of various conference committees, and recently has been the EDA TPC Chair of the ICCD'19 conference, and the panel chair of the SLIP'19 workshop that was held in conjunction with the DAC'19 conference.*

IP: 198.54.106.172 On: Tue, 21 Jul 2020 01:01:50
Copyright: American Scientific Publishers
Delivered by Ingenta