IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 13, 2019, accepted December 9, 2019, date of publication December 23, 2019, date of current version January 6, 2020.

Digital Object Identifier 10.1109/ACCESS.2019.2961568

Compiler-Directed Parallelism Scaling Framework for Performance Constrained Energy Optimization

YUNG-CHENG MA, (Member, IEEE)
Department of Computer Science and Information Engineering, Chang-Gung University, Taoyuan 333, Taiwan
e-mail: ycma@mail.cgu.edu.tw

This work was supported in part by the Ministry of Science and Technology of Taiwan under Grant 105-2221-E-182-037.

ABSTRACT Evolution of semiconductor manufacturing technology leads to the rising trend of leakage current and the end of Dennard scaling. At the dark silicon era, aggressive power gating scheme with quantitative management on power-gated hardware resources is required. This paper proposes a novel approach —parallelism scaling — to control static energy on power-gated parallel hardware. This work presents performance-constrained optimization method to power off the greatest possible amount of hardware. As a first trial, this paper examines the idea on VLIW-style architecture exploiting instruction-level parallelism. This paper establishes a theoretical foundation to realize parallelism scaling. The mathematical programming theory includes (1) topological model to control the granularity of program partitioning, (2) optimal partitioning on well-structured control flow graph in polynomial time, and (3) decision support for parallelism through item packing model guided by energy density. Evaluation conducted on EEMBC Denbench benchmark suite shows at least 15% to 53% saving on static energy compared to non-power-gated architecture. Compared to the state-of-art resource management scheme, our approach saves about 20% to 30% energy to meet the same performance demand. The evaluation reveals noteworthy opportunity to save static energy for future dark-silicon architecture design.

INDEX TERMS Power gating, instruction-level parallelism, energy efficiency, VLIW, static energy, leakage power, dark silicon.

I. INTRODUCTION

Architects have been facing dark silicon challenge in recent years. Moore's law, together with Dennard scaling [1], pushed the performance growth of computer systems for several decades: chip designers utilized fast feature-size shrinking to design processors with lower supply voltage and faster clock frequencies while maintaining the same power densities. However, the trend of Dennard scaling stopped at around 2005 due to the rapid growth of leakage current in deep-submicron manufacturing process [2]. Researchers forecast the coming of dark silicon era: with 7nm process node, over 50% of the chip area must be powered off [3]. As a consequence, architects turn to power gating technologies and the design of heterogeneous accelerators.

Power-gating [4], [5] is a circuit technology to control the dissipation of static energy resulted primarily from the growth of leakage current. With power gating, the chip area is divided into power domains and special transistors are deployed to serve as current switches to cut off the leakage current flowing through temporarily unused power domains. Several circuit technologies have been proposed for power gating [4], [5], such as MTCMOS (multi-threshold voltage CMOS) and adaptive body bias. However, the following overheads induce challenges to power gating design [4]: (1) the chip area overhead on power rail routing and isolation cells, and (2) the state-transition overhead on time and energy to activate and deactive power-gated hardware. Compiler algorithms were also proposed to control the current switches (see Section II). Researchers [4], [6] indicates that the energy to activate and deactive a hardware component once is equivalent to the static energy of keeping the hardware power-on for hundreds of cycles. The compiler has to trade-off between computational energy (the energy for normal computation) and the state-transition overhead.

The associate editor coordinating the review of this manuscript and approving it for publication was Michele Magno.

There are various styles of accelerators, such as VLIW (very long instruction word) processors, vector units, graphics processing units (GPUs), and tensor processing units (TPUs) [7]. A common feature of the accelerators is that they use a pool of parallel hardware to accelerate the applications. In many applications (such as autonomous driving [8]), the accelerators are deployed to meet certain (explicit or implicit) real-time performance demand. To meet performance demands from various applications, the system is usually over designed with provision of sufficient amount of hardware which may result in energy waste. To reduce the energy waste, we seek for the methods to depower part of the parallel hardware that exceeds performance demand.

Various accelerators in system-on-chips (SoCs) are in very-long-instruction-word (VLIW) architecture. A VLIW processor speeds up the program execution through exploiting instruction-level parallelism (ILP): multiple instructions (or operations) are packed into a single instruction bundle to be executed in a single cycle. The compiler is responsible for finding out parallel instructions and ensuring the correctness of the execution at off-line. At 2000s, various digital signal processors are VLIW architecture [9]–[13]. In recent years, various deep-learning accelerators are based on VLIW architectures enhanced with vector and SIMD instructions, such as Qualcomm Hexagan [14], CEVA XM6 [15], Tensilica visual DSP [16]. There are also computer vision applications accelerated by multiple VLIW cores [7], [17], such as Google's pixel visual core [7]. Key features of VLIW-like architectures are (1) simple hardware for execution control, and (2) exposing all hardware for software control. The features make VLIW architectures attractive for applications requiring good energy efficiency.

This paper studies the control over power-gated parallel hardware with real-time performance demand. A novel compiler algorithm, **parallelism scaling**, is proposed. As a first trial, we experiment the algorithm on a VLIW-style architecture. The compiler algorithm, consists of program partitioning and parallelism decision, is designed for performance-constrained energy optimization (PCEO): minimize the energy cost subject to a given deadline on program execution time. Consequently, the execution parallelism changes with time and the parallelism decision determines the power dissipation of the processor. As a result, a general mathematical programming theory is established and the framework has the potential to be applied to other styles of parallel hardware. Recall that high ratio of the chip area has to be powered off [3]. Through the mathematical programming model, we study how much energy can be saved through the best possible algorithm.

PCEO raises challenges to parallelism control. The previous work [18] proposes parallelism adaption algorithm to reduce energy consumption at peak-performance mode: each program region is simply assigned its speedup-saturation parallelism. For PCEO, a new algorithm is required to choose parallelism from a set of possible decisions. Nevertheless, the parallelism assignment interferes the program partitioning stage on the effect of controlling state-transition energy. To resolve the chicken-egg problem, we propose the multi-pass SRE-ED algorithm (see Section IV) that makes decision by examining solutions resulted from various program partitioning granularity. Addressing these issues, this paper makes the following contributions:

(1) A novel granularity control scheme for program partitioning, called *SRE-criteria*, to control the state-transition energy through a topological model,
(2) A new program partitioning algorithm, *PGR composition*, that finds an optimal solution for the N-way multi-cut problem in polynomial time when the control flow graph is derived from a well-structured high-level language program, and
(3) The GPED (greedy packing by energy density) algorithm to assign parallelism for PCEO.

This paper is organized as follows. Section II gives the background and related work. Section III gives an overview and the problem modeling of the proposed approach. Section IV describes our strategy to devise the algorithm for PCEO. The compiler algorithm on the theoretical aspects is established at Section V and Section VI deals with the practical realization issues. Section VII presents the evaluation results and the conclusion is given at Section VIII.

II. BACKGROUND AND RELATED WORK

Here we present fundamental concepts on power gating and surveys related work on compiler-directed power-gating control.

A. FUNDAMENTALS: POWER GATING FOR DARK SILICON

The evolution of semiconductor manufacturing process brings the dark silicon challenge. Power dissipation of a chip can be divided into two parts: (1) the dynamic power, or switching power, resulted from voltage swings on the gates and wires, and (2) the static power, or leakage power, comes from the leakage current flowing through transistors even when the circuit has no switching activities. According to the report of Industry Technology Roadmap for Semiconductors (ITRS) [19], static power becomes the dominant part of the total power dissipation in today's deep-submicron manufacturing process. The raise of leakage current causes the failure of Dennard scaling [1] and leads to the era of dark silicon [2]. It is reported that, with 7nm manufacturing process, over 50% of the chip area must be powered off [3]. The major design trends in the dark silicon era are (1) widely deployment of power-gating circuitry to control the leakage current, and (2) the use of heterogeneous accelerators in domain-specific architectures to meet performance demand with good energy efficiency [7], [20], [21].

Power gating is the technology to cut-off leakage current flowing through temporarily unused hardware components. The concept is illustrated in Figure 1. Circuits for normal operations are divided into power domains and special transistors (different from the transistors used for normal opera-

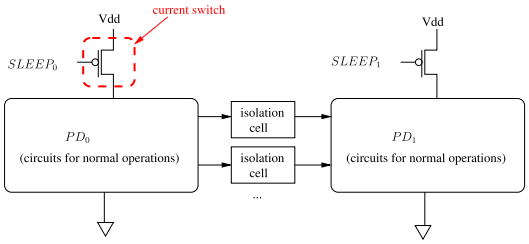

FIGURE 1. Concepts of power-gating circuit technology.

tions) are deployed to serve as current switches. Each power domain PD_i has its own set of current switches. When PD_i is idle, the sleep signal $SLEEP_i$ is asserted to make the current switches cut-off leakage current flowing through PD_i. Several circuit technologies have been proposed for power gating [4], [5], such as MTCMOS (multi-threshold voltage CMOS) and adaptive body bias. This paper focuses on the architecture design and software control over the power-gating circuitry.

Major architectural design concerns come from the following overheads of power gating:

1) Timing overhead. To prevent the circuit been burned out by the rush current, a gradual wake-up scheme is deployed to activate a sleeping power domain [4]–[6]. This introduces the overhead of state-transition time on a power-gated architecture. We adopt the design proposed by [6], which requires one additional clock cycle to execute a power-gating instruction.
2) Chip area overhead. Aside from the area occupied by current switches and power rails, isolation cells also contribute to the chip area overhead [4], [5]. To prevent the circuit being destroyed during activation, an isolation cell is inserted for each net that crosses different power domains. The chip area overhead raises the granularity issue on power domain partitioning.
3) Energy overhead. Additional energy, called **state-transition energy**, is required to activate and deactivate a power domain. To gain energy saving, the time interval between two power-gating instructions has to exceed certain threshold T_{th}. Previous researchers [4], [6] reported that the state-transition energy is approximately equal to the static energy of keeping a power domain active for 100 cycles and hence $T_{th} \approx$ 100 cycles. The overhead of state-transition energy is one of the major concerns in devising our compiler algorithm.

Besides power-gating control, another challenge in dark silicon era is the programmability over a pool of heterogeneous accelerators. Fully automatic compilation from a general-purpose programming language usually does not generate satisfactory program-to-hardware mapping. As a result, researchers proposed approaches such as code instrumentation [22] and domain-specific languages [23] which have explicit statements to direct the program-to-hardware mapping. Extensive profiling is used to guide the program-to-hardware mapping [22], [24] for embedded applications. Programming a power-gated hardware also encounters similar programmability issue. While the programmability issue and language design are outside the scope of this research, we assume that our compiler algorithm will be deployed in a programming environment with directives for program-to-hardware mapping guided by profiling. We also hope that the proposed theory will affect the programming environment design for accelerators in the future.

B. RELATED WORK: COMPILER-DIRECTED LEAKAGE POWER CONTROL

Various compiler algorithms to control power gating are proposed. The focus is to find out and enlarge the idle period through global instruction scheduling to shut down hardware [25]–[28]. The research trend moves towards practical processors in recent years. Abdel-Majeed et al. enlarges the idle period of functional units in a GPU through dynamic instruction issue policies [29]. Kumar *et al.* proposed algorithms to power off parts of vector lanes in an SIMD accelerator [30]. Aghilinasab *et al.* proposed algorithms to power off vector functional units in a GPU [31]. Considering the high threshold value of T_{th}, Roy *et al.* [6] proposed loop-based control for power gating. Cherupalli *et al.* proposed binary code annotation for power gating control with clustering on logic gates to establish power domains and a power domain may spread across design modules [32]. Power gating may also be applied to routers in network-on-chip (NoC), such as in [33].

The research trend further moves toward quantitative management on power-gated hardware: determining the amount of activated hardware throughout the execution. Power gating on register files and storage elements received attention [34], [35]. Tabkhi and Schirner [36] proposed function-based scheme to control the amount of activated registers. Various researchers proposed some means of *parallelism control* to manage power-gated hardware. Girlado *et al.* [37], [38] proposed designs that dynamically change the instruction-level parallelism (ILP) to execute a program on a VLIW processor. Researchers have also proposed designs that dynamically vary the amount of vector lanes and cores in a GPU [39], [40].

We also studied parallelism control on VLIW-style architectures. Considering that over 60% of the energy dissipated in a VLIW processor spent on register files [41], [42], we proposed the PGRF-VLIW architecture [43] featuring distributed and power-gated register files. Besides the shared register file (SRF) connected to all execution slots, each execution slot is associated with a local register file (LRF). Due to the reduced amount of access ports, a LRF consumes less energy compared to the SRF. Both the LRFs and SRF are partitioned into banks for power gating. A power domain may be (1) a bank in a LRF, (2) a bank in the SRF, or (3) functional units of an execution slot. Parallelism decision affects the energy dissipated on both functional units and register files. The local instruction scheduling algorithm (the DCCS algorithm in [43]) performs operation clustering

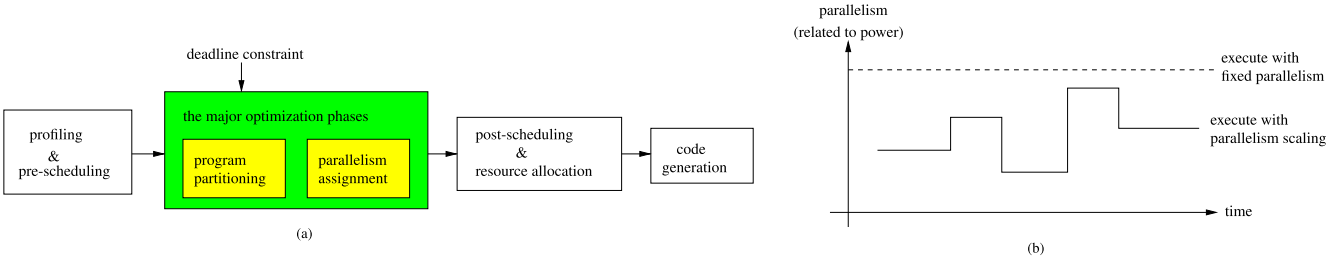

FIGURE 2. Conceptual idea of parallelism scaling. (a) The compilation flow. (b) The execution profile.

to reduce the amount of operand transfers through the SRF. As a result, the amount of cross-slot operand transfers (and hence the energy dissipated on register files) also scales with ILP. Based on the PGRF-VLIW architecture, we further proposed global parallelism adaption heuristic [18] to save energy at peak-performance mode, where each program region is executed with its speedup-saturation parallelism. In this paper, we advance the previous works by taking real-time performance demand to depower more hardware for energy saving.

III. SCENARIO AND MODELING OF PARALLELISM SCALING

The compiler algorithm is to improve energy efficiency of power-gated parallel hardware, aimed at reducing the energy dissipation to satisfy a given performance requirement. Figure 2(a) shows the compilation flow. Assume that the execution parallelism of the hardware can be configured through power gating; Higher execution parallelism gets speedup at the cost of higher power dissipation. The compiler partitions the program into several **power-gating regions** (PGRs); A PGR is a program region to be executed with a fixed power-gating configuration. Each PGR is then assigned the parallelism to execute the program region. The program is then re-scheduled (with resource allocation) according to parallelism decisions on PGRs to determine power gating configurations and the code generation inserts power gating instructions. Consequently, as shown in Figure 2(b), the execution parallelism (and hence power dissipation) changes with time as the execution progress. The optimization problem is to minimize the energy cost subject to the given deadline on the program execution time. Both the computational energy (the energy spent for normal computation) and the state-transition energy (the energy to activate/deactivate power domains) are considered. Later in this section, we formalize the optimization problem.

Figure 3 gives an example of the application scenario. In a multi-media application, the video codec is to be offloaded onto an accelerator with power-gated parallel hardware to achieve the frame rate of 30 frames per second. The compiler takes the deadline constraint, 1/30 seconds to execute the loop body per iteration, from directives. The parallelism scaling compiler inserts power-gating instructions into the loop body to reduce the energy to achieve the frame rate. The compiler

```
///
/// Directive: set performance requirement of 30 frames/sec
///
for (frame_id=0; frame_id<Max_FrameID; frame_id++) {
    F = Get_Frame (frame_id);
    if (Type(F)==I_Frame) {
        Set_PowerGateConfig(X_0);   //compiler generated power-gating
                                    //instruction
        Decode_IFrame(F);
    }
    else {
        Set_PowerGateConfig(X_1);   //compiler generated power-gating
                                    //instruction
        Decode_PFrame(F);
    }
}
```

FIGURE 3. Example of the application scenario.

tries to depower hardware that exceeds performance demand. The compiler optimization relies on profiling information. Such a profiling-based optimization is suitable for applications in embedded systems with stable behavior that can be observed at offline [22], [24].

As the first trial, we experiment the compiler algorithm on the PGRF-VLIW architecture [43], which scales the ILP. In the architecture, power-gating is deployed on execution slots as well as the distributed register files. The execution ILP affects the energy dissipated on both functional units and register files. Note that parallelism scaling can also be applied to traditional VLIW architectures with power gating on functional units only. However, the register file contributes over 60% of the total energy [41], [42] and the energy saving effect is limited without re-designing the architecture.

We now formalize the compiler optimization problem, starting from modeling a power-gated architecture.

A. MODELING A POWER-GATED ARCHITECTURE

A power-gated architecture is modeled as follows. Table 1 lists the notations for the architecture modeling. Assume that the hardware architecture is partitioned into N power domains $\{PD_0, PD_1, PD_2, \ldots, PD_{N-1}\}$. A **power-gating configuration** (**PG-config**) to execute a program region is a 0-1 vector $X = (x_0, x_1, x_2, \ldots, x_{N-1})$, where $x_i = 1$ indicates that the power domain PD_i is ON and $x_i = 0$ indicates that PD_i is OFF. A power domain PD_i has the following attributes:

(1) $e_{st}(PD_i)$: the static power, in units of energy per cycle, of the power domain PD_i,
(2) $e_{act}(PD_i)$: the activation energy to power on the power domain PD_i,

TABLE 1. Notations on modeling a power-gated architecture.

Notation	Description
	Basic attributes of a power domain
PD_i	a power domain
$e_{st}(PD_i)$	static power (energy per cycle) of PD_i
$e_{act}(PD_i)$	energy to activate PD_i
$e_{dact}(PD_i)$	energy to deactivate PD_i
	Vector modeling of power domains
E_{st}	the vector of static power (energy per cycle) for all power domains: $E_{st} = (e_{st}(PD_0), e_{st}(PD_1), e_{st}(PD_2), ..., e_{st}(PD_{N-1}))$
E_{act}	the vector of activation energy for all power domains: $E_{act} = (e_{act}(PD_0), e_{act}(PD_1), e_{act}(PD_2), ..., e_{act}(PD_{N-1}))$
E_{dact}	the vector of deactivation energy for all power domains: $E_{dact} = (e_{dact}(PD_0), e_{dact}(PD_1), e_{dact}(PD_2), ..., e_{dact}(PD_{N-1}))$
X, Y, Z	0-1 vectors of a power-gating configuration (PG-config)
	Power domain attributes by resource type
$e_{st,t}$	the power (static energy per cycle) of a type t hardware component
$e_{act,t}$	the energy to activate a type t hardware component
$e_{dact,t}$	the energy to deactivate a type t hardware component
$R_t(X)$	amount of activated type t hardware in the PG-config X
	Energy attributes
$P_{st}(X)$	the power (static energy per cycle) to execute with PG-config X
$AE(X, Y)$	the activation energy to switch the PG-config from X to Y
$DE(X, Y)$	the deactivation energy to switch the PG-config from X to Y
$E_{str}(X, Y)$	the state-transition energy to switch PG-config from X to Y

(3) $e_{dact}(PD_i)$: the deactivation energy to power off the power domain PD_i.

These energy attributes can be obtained from hardware synthesis. Energy attributes of the whole architecture can also be represented as vectors E_{st}, E_{act}, E_{dact} as stated in Table 1. Refer to [6] for the hardware implementation of the power-gating instruction that controls through such a 0-1 vector X.

The computational energy is estimated as follows. Let X be the PG-config to execute a program region. The static power, in terms of energy per cycle, to execute the program with the PG-config X is the dot-product of E_{st} and X:

$$P_{st}(X) = E_{st} \bullet X = \sum_{PD_i} e_{st}(PD_i) * x_i,$$

which is the sum of the static power of all power-on domains. The computational (static) energy will be $P_{st}(X) * T$ if the processor executes with the PG-config X for T cycles. Multiple power domains may be of the same type of hardware components. An alternative way is to estimate the computational energy from resource requirements to each type of hardware components. Let $R_t(X)$ be the amount of activated type t hardware in PG-config X and $e_{st,t}$ is the static energy per cycle for a type t hardware. The computational energy per cycle for the PG-config X can also be estimated as follows:

$$P_{st}(X) = \sum_t e_{st,t} * R_t(X).$$

The state-transition energy is estimated as follows. Consider the case of switching the PG-config from X to Y. We introduce the notation $||Y - X|| = (z_0, z_1, z_2, \ldots, z_{N-1})$ where

$$z_i = \begin{cases} y_i - x_i & \text{if } y_i - x_i \geq 0 \\ 0 & \text{if } y_i - x_i < 0. \end{cases}$$

($X = (x_0, x_1, \ldots, x_{N-1})$ and $Y = (y_0, y_1, \ldots, y_{N-1})$.) The activation energy of switching the PG-config from X to Y is thus

$$AE(X, Y) = E_{act} \bullet ||Y - X|| = \sum_{PD_i} e_{act}(PD_i) * z_i.$$

Similarly, the deactivation energy of switching the PG-config from X to Y can also be calculated from the dot product:

$$DE(X, Y) = E_{dact} \bullet ||X - Y||.$$

And the state-transition energy of switching from X to Y is the sum of activation and deactivation energy:

$$E_{str}(X, Y) = AE(X, Y) + DE(X, Y).$$

B. MODELING THE COMPILER OPTIMIZATION

Figure 4 illustrates the modeling of the compiler optimization and Table 2 lists the notations for the modeling. The input is the deadline T_{DL} on execution time and a control flow graph (CFG) annotated with profiling and pre-scheduling information to model an application program.

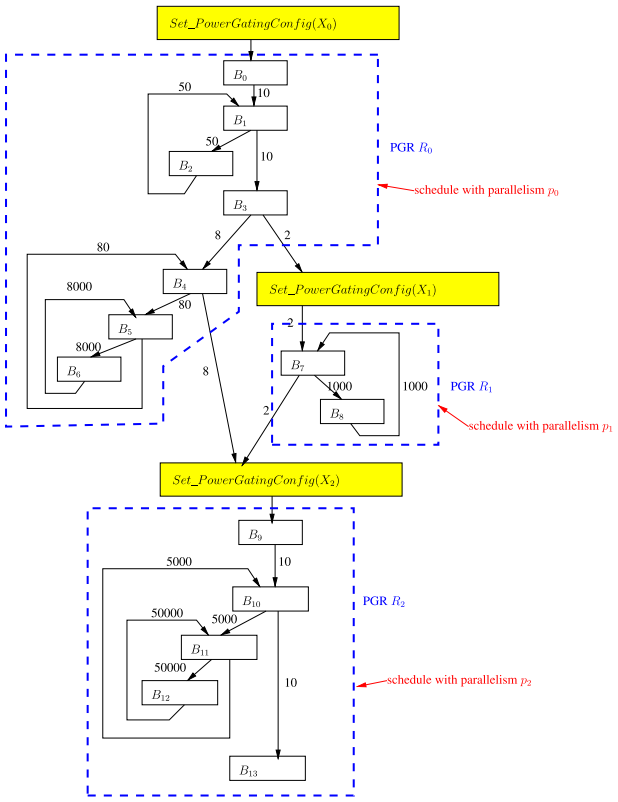

FIGURE 4. Compiler modeling as CFG partitioning and parallelism assignment.

The outcome is a **parallelism scaling solution** (**PS-solution**) $S = (RP, PA)$, which consists of a **region partitioning solution** (**RP-solution**) RP and a **parallelism assignment solution** (**PA-solution**) PA. The RP-solution $RP = \{R_0, R_1, R_2, \ldots, R_{N-1}\}$ is a partitioning on the CFG where each PGR R_i is a connected subgraph of the CFG. The PA-solution $PA = \{p_0, p_1, p_2, \ldots, p_{N-1}\}$ assigns the execution parallelism p_i to each PGR R_i. Through resource allocation, the assigned parallelism p_i determines the power-gating configuration X_i to execute R_i. Power-gating instructions are inserted on transition edges across PGRs. The solution $S = (RP, PA)$ also determines the time $WT(RP, RA)$ and energy cost $WE(RP, PA)$ to execute the program. The optimization problem is to minimize the energy cost $WE(RP, PA)$ subject to the deadline constraint $WT(RP, PA) \leq T_{DL}$ on execution time.

An application program is modeled as a CFG annotated with profiling and pre-scheduling information. (Later in Section VI, the graph-theoretical modeling will be implemented as a cross-procedural optimization.) Profiling annotates the CFG with (1) the execution count $C_{Ex}(B_i)$ on each basic block B_i, and (2) the transition count $C_{tr}(B_i, B_j)$ on each edge (B_i, B_j). (Please refer to [37], [44] for how the compiler profiles an application program to obtain the execution counts.) Local instruction scheduling is performed multiple times on each basic block B_i to obtain the execution cycles per entrance $ETPE(B_i, p)$ for each possible parallelism p.

The time and energy for normal computation determined by a solution $S = (RP, PA)$ are estimated as follows. Assigning parallelism p_i to a PGR R_i determines the cost for normal computation on R_i. The total computational time spent on R_i is estimated from profiling and pre-scheduling information:

$$WT_{cp}(R_i, p_i) = \sum_{B_j \in R_i} C_{Ex}(B_j) * ETPE(B_j, p_i),$$

which is the total cycles to execute all basic blocks $B_j \in R_i$. Note that the program execution may go into R_i multiple times and $WT_{cp}(R_i, p_i)$ counts the total cycles for all entrances. Assigning parallelism p_i to execute a PGR R_i also determines the power-gating configuration $X(R_i, p_i)$, which is a 0-1 vector, to execute R_i. By performing instruction scheduling with resource allocation, the parallelism assignment determines the required amount of hardware components $R_t(R_i, p_i)$ for each hardware type t. (For example, for the PGRF-VLIW architecture [43], performing instruction scheduling and register allocation over R_i with the assigned parallelism p_i determines the amount of of register banks in local and shared register files as well as the amount of functional units to execute R_i.) According to Section III-A, the static power (energy per cycle) to execute R_i is estimated from the resource requirements:

$$P_{st}(R_i, p_i) = \sum_t e_{st,t} * R_t(R_i, p_i).$$

And the total computational energy contributed by R_i can be estimated from the execution time:

$$WE_{cp}(R_i, p_i) = P_{st}(R_i, p_i) * WT_{cp}(R_i, p_i).$$

The total computational cost of a solution $S = (RP, PA)$ is the total cost over all PGRs:

$$WT_{cp}(RP, PA) = \sum_{R_i} WT_{cp}(R_i, p_i),$$

and

$$WE_{cp}(RP, PA) = \sum_{R_i} WE_{cp}(R_i, p_i),$$

where $WT_{cp}(RP, PA)$ and $WE_{cp}(RP, PA)$ is the total computational time and energy, respectively, determined by the solution $S = (RP, PA)$.

During the compiler optimization, the computational costs per entrance to a PGR are also required. The number of times the execution goes into a PGR R_i is

$$C_{Ex}(R_i) = \sum_{R_j \neq R_i} \sum_{(B_k, B_l) \in R_j \times R_i} C_{tr}(B_k, B_l),$$

which is the total transition count for all edges go into R_i. The computational time and energy per entrance to R_i are estimated as follows.

$$ETPE(R_i, p_i) = \frac{WT_{cp}(R_i, p_i)}{C_{Ex}(R_i)}$$

$$EEPE(R_i, p_i) = \frac{WE_{cp}(R_i, p_i)}{C_{Ex}(R_i)}$$

TABLE 2. Notations for modeling the compiler optimization problem.

Notation	Description
CFG model with annotations	
CFG	the control flow graph
$B_i, B_j, ...$	a basic block
$e = (B_i, B_j)$	an edge from B_i to B_j in the CFG
$C_{tr}(B_i, B_j)$	number of transitions from B_i to B_j
$C_{Ex}(B_i)$	execution count of B_i
$ETPE(B_i, p)$	execution time (cycle) per entrance of B_i if scheduled with parallelism p
Parallelism scaling solution and basic attributes	
$S = (RP, PA)$	a parallelism scaling solution (PS-solution)
PGR	power-gating region
$RP = \{R_0, R_1, R_2, ...\}$	a region partitioning solution (RP-solution), where R_i is a PGR in CFG
$PA = \{p_i\}$	a parallelism assignment solution (PA-solution), where p_i is the assigned parallelism to execute R_i
$C_{tr}(R_i, R_j)$	number of times the execution transit from PGR R_i to R_j
$C_{Ex}(R_i)$	number of times the PGR R_i is executed
On computational cost	
$X(R_i, p_i)$	the PG-config (0-1 vector) to execute R_i if R_i is assigned parallelism p_i
$R_t(R_i, p_i)$	required amount of type t hardware components if R_i is assigned parallelism p_i
$P_{st}(R_i, p_i)$	the static power (energy per cycle) to execute R_i with parallelism p_i
$WT_{cp}(R_i, p_i)$	the total computational time spent on R_i if R_i is assigned parallelism p_i
$WE_{cp}(R_i, p_i)$	the total computational (static) energy spent on R_i if R_i is assigned parallelism p_i
$ETPE(R_i, p_i)$	the execution time per entrance to R_i if R_i is executed with parallelism p_i
$EEPE(R_i, p_i)$	the execution (computational) energy per entrance to R_i if R_i is executed with parallelism p_i
$WT_{cp}(RP, PA)$	the total computational time of a solution $S = (RP, PA)$
$WE_{cp}(RP, PA)$	the total computational energy of a solution $S = (RP, PA)$
On state-transition cost	
T_{str}	the time (cycles) to execute a power-gating instruction
$WT_{str}(RP)$	the total state-transition time of the partitioning RP
$WE_{str}(RP, PA)$	the total state-transition energy for the solution $S = (RP, PA)$
On total cost of a PS-solution	
T_{DL}	the deadline constraint on the execution time
$WT(RP, PA)$	the total execution time of a solution $S = (RP, PA)$
$WE(RP, PA)$	the total energy cost of a solution $S = (RP, PA)$

The state-transition overhead of a solution $S = (RP, PA)$ is as follows. For two adjacent PGRs R_i and R_j, the amount of execution transition from R_i to R_j is estimated from the total transition count for all edges going from R_i to R_j:

$$C_{tr}(R_i, R_j) = \sum_{(B_k, B_l) \in R_i \times R_j} C_{tr}(B_k, B_l).$$

We assume that executing the power-gating instruction once spends a constant amount of cycles T_{str}. The time spent for state transition is

$$WT_{str}(RP) = \sum_{R_i \neq R_j} C_{tr}(R_i, R_j) * T_{str}.$$

Note that the state-transition time depends on the partitioning RP only and is independent of the parallelism assignment PA. The solution $S = (RP, PA)$ determines the PG-config $X(R_i, p_i)$ and $X(R_j, p_j)$ for the two PGRs R_i and R_j. Section III-A gives the state-transition energy $E_{str}(X(R_i, p_i), X(R_j, p_j))$ for each time the execution goes from R_i to R_j. Hence we have the total state-transition energy: $WE_{str}(RP, PA) = \sum_{R_i \neq R_j} C_{tr}(R_i, R_j) * E_{str}(X(R_i, p_i), X(R_j, p_j))$.

We summarize the optimization problem as follows. As the input, the application program is modeled as a CFG with profiling and pre-scheduling information annotated. The compiler algorithm determines a PS-solution $S = (RP, PA)$, which consists of the CFG partitioning RP and parallelism assignment PA. The solution determines the total time ($WT(RP, PA)$) and energy ($WE(RP, PA)$) to execute the program, which consists of costs from normal computation and state-transition.

$$WT(RP, PA) = WT_{cp}(RP, PA) + WT_{str}(RP)$$
$$WE(RP, PA) = WE_{cp}(RP, PA) + WE_{str}(RP, PA)$$

Considering both the computational cost and state-transition overhead, the optimization problem is to minimize the energy cost $WE(RP, PA)$ subject to the deadline constraint $WT(RP, PA) \leq T_{DL}$.

IV. OPTIMIZATION STRATEGY

We now start to devise the parallelism scaling algorithm for performance-constrained energy optimization. Taking the deadline constraint, the algorithm seeks for opportunities to depower hardware exceeding performance demand. The motivation toward PCEO will be shown in Section IV-A: program regions differ in the energy efficiency on raising parallelism to earn speedup and tuning parallelism over program regions gains energy saving to meet the given

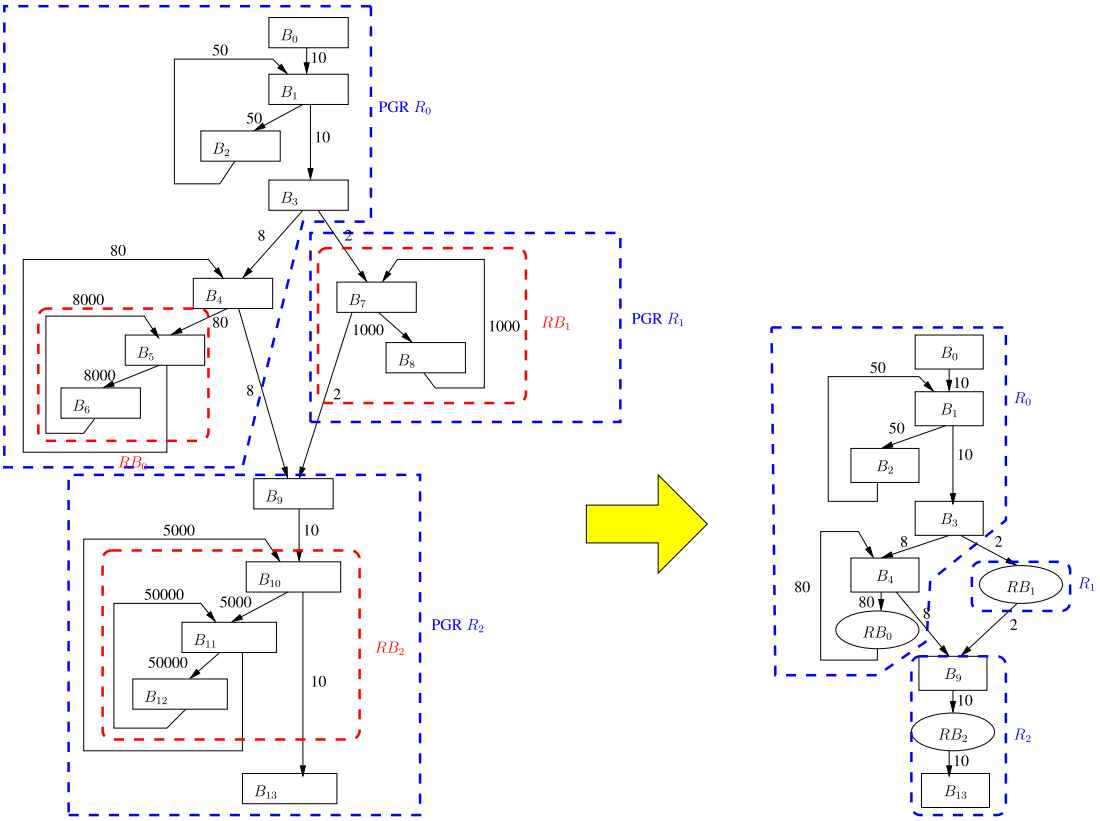

FIGURE 5. Concepts of loop-based program partitioning.

performance demand. However, adapting parallelism for PCEO encounters new difficulties. The compiler has to consider both computational energy and state transition energy. But the two concerns conflict one another: while it is easier to improve resource utilization to save computational energy for small PGRs, a solution with large PGRs saves state-transition energy. Moreover, dealing with the two concerns induces an chicken-egg problem on arranging algorithmic stages. As a result, we propose multi-pass SRE-ED algorithm for PCEO, in which the optimization is directed by the two key quantities *state-returning energy* (SRE) and *energy density* (ED). Here we devise the strategy through introducing the chicken-egg problem.

We begin from the loop-based program partitioning strategy to address the difficulties for PCEO. Figure 5 illustrates the two-stage approach for the program partitioning phase:

- Stage 1: PGR-core identification. This stage identifies loops with sufficiently long **execution time per entrance** (ETPE) $\{RB_0, RB_1, RB_2, \ldots\}$ as PGR cores.
- Stage 2: PGR establishment. This stage expands PGR cores to form PGRs. In the CFG model (the right-hand side of Figure 5), an identified PGR core RB_i is merged as an **R-node**, and ordinary basic blocks in the CFG are called **B-nodes**. A PGR R_i is a connected subgraph having exactly one R-node. The optimization problem in this stage is to minimize the total edge transition count across PGRs subject to the R-node and connectivity constraint.

The reason to adopt the loop-based partitioning strategy comes from the threshold T_{th} on the time interval between two power-gating instructions. It is reported that, to gain energy saving, $T_{th} \approx 100$ in previous works [4], [6]. The most likely program structures to have ETPE>100 cycles are loops. We make each PGR having a loop with $ETPE > T_{th}$ within it.

PCEO induces the chicken-egg problem between the program partitioning and parallelism assignment phase. A straight forward approach is to have a parallelism assignment phase following the program partitioning phase (as the third stage). The task of the program partitioning phase is to control the state-transition overhead (WT_{str} and WE_{str}) by having each PGR has sufficiently long ETPE. But the ETPE of a PGR is determined by the assigned parallelism at a later stage. The parallelism assignment phase is aimed at controlling the computational energy and time (WE_{cp} and WT_{cp}). To limit the complexity, parallelism assignment is made through a series of parallelism tuning watching at computational energy only. However, tuning the parallelism of a PGR R_i not only affects the computational energy of R_i but also affects the state-transition energy to PGRs neighboring to R_i. The parallelism tuning interferes the control of state-transition energy at the program partitioning phase.

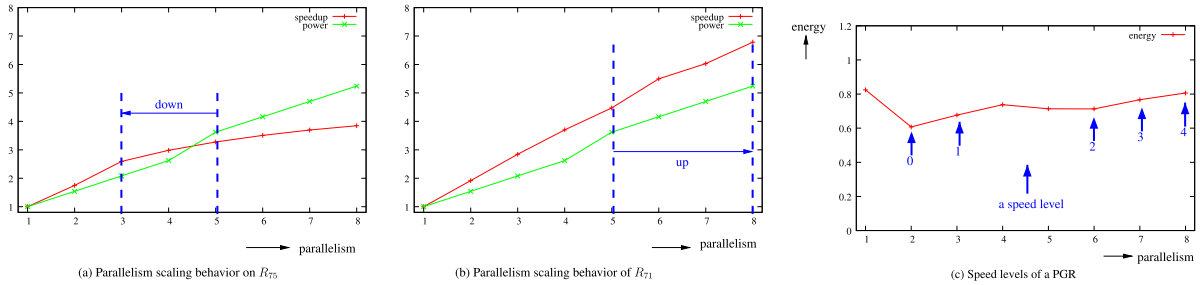

FIGURE 6. Examples of parallelism tuning strategy.

Facing the chicken-egg problem, we propose the multi-pass SRE-ED algorithm for PCEO. Each pass of the algorithm goes through both the program partitioning and the parallelism assignment phase to generate a candidate solution. To balance between computational and state-transition energy, the final outcome is selected from multiple candidate solutions differ in granularity of PGRs. Key issues to realize the idea are (1) parallelism tuning strategy, (2) devise a quantitative measure to control the granularity of PGRs, and (3) control the complexity (amount of passes) of the multi-pass algorithm. To resolve these issues, we start from examining the parallelism tuning phase at Section IV-A and then go back to the program partitioning phase at Section IV-B. The framework of the compiler algorithm for PCEO is given in Section IV-C.

A. PARALLELISM TUNING BY ENERGY DENSITY

We examine the parallelism assignment phase first. Suppose a region partitioning $RP = \{R_0, R_1, \ldots, R_{N-1}\}$ is given and fixed. Parallelism assignment over RP is made through a series of **parallelism tuning** watching at computational energy only. Starting from an initial assignment, the algorithm iteratively tunes up or down the parallelism of some PGR R_i to improve energy efficiency. We use part of the experiment data from MPEG4 encoding as an example to illustrate the motivation. (Refer to Section VII for the experiment method.) Figure 6(a) and (b) show how the static power and speedup scales with ILP for two loops (R_{75} and R_{71}) with ETPE>100 cycles at the peak-performance mode. (The static power is normalized with respect to the power dissipation at ILP = 1.) Both loops have speedup saturated at ILP = 8 but differ on the energy efficiency for high-ILP execution. The loop R_{75} has poor energy efficiency: the slope of speedup gradually reduces for ILP>3 and the power scales up faster than speedup. By contrast, the loop R_{71} has good energy efficiency: the speedup scales linearly with constant slope and grows faster than power. Suppose we have an initial solution that satisfies the performance constraint by setting ILP = 5 for both loops. In this example, we reduce the ILP of R_{75} to 3 to save energy and raise the ILP on R_{71} to 8 to compensate for the lost performance. The tuning achieves the same performance requirement with lower energy cost.

The parallelism tuning works on **speed levels**. The set of speed levels of a PGR R_i is the set of candidate parallelism

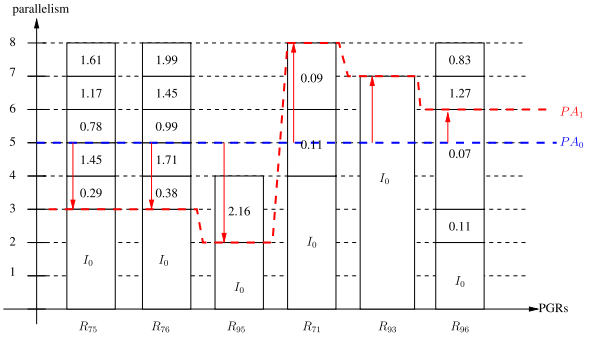

FIGURE 7. Parallelism tuning on item-stack model.

to be decided for R_i. Figure 6(c) gives an example. Observing how the computational energy changes with ILP, speed levels of a PGR are chosen such that energy cost monotonically increases with the candidate ILP. In this example, the possibilities to assign ILP = 4 and 5 are excluded since assigning ILP = 6 is better on both execution time and energy. Speed levels are built by iteratively selecting the next parallelism with minimum energy cost, and the speed-level $l = 0$ is the parallelism of the minimum energy mode.

The parallelism tuning works on the item-stack model as illustrated in Figure 7. Each PGR R_i has a stack of items where an item stands for a speed-level of R_i. The key attribute of a speed-level $l \geq 1$ is **energy density** $\bigtriangleup ED_i(l)$:

$$\bigtriangleup ED_i(l) = \frac{Energy(R_i, l) - Energy(R_i, l-1)}{ExeTime(R_i, l-1) - ExeTime(R_i, l)},$$

which is the average energy cost to reduce one unit of execution time by raising the speed-level of R_i from $(l-1)$ to l. ($Energy(R_i, l)$ and $ExeTime(R_i, l)$ are the computational energy and execution time, respectively, to execute R_i with speed level l.) In Figure 7, we mark the energy density on each item. A PA-solution PA_j is a skyline which cuts through the stack of R_i at the height of the assigned parallelism. Parallelism tuning is to adjust the position where PA_j cuts through a stack. The guideline is to raise parallelism through items with low energy density and down the parallelism through items with high energy density. Furthermore, the selection has to match speed levels and exclude parallelism higher than speedup-saturation points. In Figure 7, PA_1 is a solution obtained by tuning from PA_0 following the guidelines.

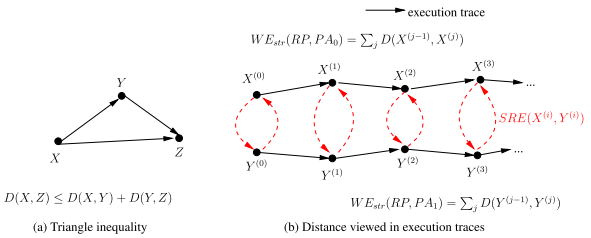

FIGURE 8. Geometric view on state-transition energy.

The chicken-egg problem is that the parallelism tuning may interfere the program partitioning phase on the control of state-transition energy. The parallelism tuning watches at computational energy only and may increase state-transition energy. In the example of Figure 7, the difference on execution ILP between R_{95} and R_{71} is increased after the tuning. This may increase the state-transition energy if the two PGRs are adjacent. We derive a scheme to control the granularity over PGRs to ensure energy saving from parallelism tuning.

B. PROGRAM PARTITIONING BY STATE-RETURNING ENERGY

We now go back to the program partitioning phase and focus on the PGR-core identification stage, which is to control the granularity of PGRs by detecting loops with sufficiently long ETPE as PGR cores. Recall that the parallelism tuning watches at computational energy only and results in estimation error on state-transition energy. To control the granularity of PGRs, the key issue is to assert an upper-bound over the error of state-transition energy resulted from the parallelism tuning. The error upper-bound has to be depending on the candidate PGR (loop) only. The PGR-core identification criteria is then derived from the error upper-bound to ensure that the parallelism tuning will not introduce too much state-transition overhead. We give a short derivation here and readers can refer to Appendix A for detailed mathematical proof behind the derivation.

We devise the error upper-bound through the following topological model over the set of all power-gating configurations. A power-gating configuration X (a 0-1 vector) is a point in the space and the **distance** from a point X to another point Y, denoted $D(X, Y)$, is defined to be the state-transition energy $E_{str}(X, Y)$ to switch the PG-config from X to Y. For any three points X, Y, Z, the following inequality holds:

$$D(X, Z) \le D(X, Y) + D(Y, Z). \tag{1}$$

(Refer to Appendix A for the proof.) Figure 8(a) shows the geometric interpretation of the triangle inequality. A key quantity is the **state-returning energy** (SRE) defined as follows:

$$SRE(X, Y) = D(X, Y) + D(Y, X),$$

which is the energy to switch the PG-config from X to Y and back to X.

An error upper-bound on state-transition energy resulted from parallelism tuning is as follows. Figure 8(b) shows the geometric interpretation. Suppose we have a region partitioning RP which results in the execution trace of PGRs $\{R^{(0)}, R^{(1)}, R^{(2)}, \ldots\}$. We have two PA-solutions PA_0 and PA_1 on the partitioning RP. (One may think that PA_1 is an improved solution tuned from PA_0.) Solutions PA_0 and PA_1 result in execution traces of PG-configs $\{X^{(0)}, X^{(1)}, X^{(2)}, \ldots\}$ and $\{Y^{(0)}, Y^{(1)}, Y^{(2)}, \ldots\}$, respectively. $X^{(j)}$ and $Y^{(j)}$ are two PG-configs applied to the same PGR $R^{(j)}$. The total distance of an execution trace is the total state-transition energy of the underlying solution.

$$WE_{str}(RP, PA_0) = \sum_j D(X^{(j-1)}, X^{(j)})$$
$$WE_{str}(RP, PA_1) = \sum_j D(Y^{(j-1)}, Y^{(j)})$$

Eq. (1) implies the upper-bound over the difference in state-transition energy between the two solutions:

$$|WE_{str}(RP, PA_0) - WE_{str}(RP, PA_1)| \le \sum_j SRE(X^{(j)}, Y^{(j)}). \tag{2}$$

Please refer to Appendix A for the detailed proof of Eq. (2).

Eq. (2) leads to the guideline for PGR granularity control: ensuring that the SRE for each speed level of the PGRs is small compared to the computational energy. (Refer to Appendix A for the rationale from the view point of equation derivations.) The PGR-core identification criteria checks the state-returning energy for each speed level of a candidate loop. A loop is marked as a PGR core if the SRE for each speed level is small compared to its own computational energy. In later sections, we formalize the concepts to devise the algorithm to identify loops as PGR cores.

C. MULTI-PASS ALGORITHM FRAMEWORK FOR PCEO

Algorithm 1 shows the framework of the multi-pass SRE-ED algorithm for performance-constrained energy optimization. Each pass generates a candidate solution $(RP_\epsilon, PA_\epsilon)$ (cf. Line 2 to 7) and the candidate solution with minimum energy cost is selected as the final outcome (cf. Line 8). Each pass goes through three stages. The first stage (Line 3) identifies a set of PGR cores $\{RB_0, RB_1, \ldots, RB_{N-1}\}$ as the seed of the candidate solution $(RP_\epsilon, PA_\epsilon)$. This stage controls the granularity of PGRs with the control variable $\epsilon \in (0, 1)$, which is an upper bound on the error ratio of state-transition energy related to the computational energy. And the error of state-transition energy is calculated from the state-returning energy as devised in Section IV-B. Following the model in Figure 5, the second stage (Line 4) expands PGR cores to form PGRs covering the whole program. The third stage (Line 5) assigns parallelism p_i to each PGR R_i. The parallelism assignment works on the item-stack (speed level) model of Section IV-A. Watching at computational energy

Algorithm 1 Framework of the Multi-Pass SRE-ED Algorithm for Parallelism Scaling

Input: (CFG, T_{DL}, T_{th})
Output: $S = (RP, PA)$

1: initiate candidate solution set $C_S = \emptyset$;
2: **for** each possible ϵ **do**
3: identify a set of PGR-cores $\{RB_0, RB_1, RB_2, \ldots\}$ with control parameters (error ratio) (T_{th}, ϵ);
4: establish PGRs $RP_\epsilon = \{R_0, R_1, R_2, \ldots\}$ from PGR-cores $\{RB_0, RB_1, RB_2, \ldots\}$;
5: assign parallelism $PA_\epsilon = \{p_i\}$ over RP to satisfy deadline T_{DL};
6: $C_S \leftarrow C_S \bigcup \{(RP_\epsilon, PA_\epsilon)\}$;
7: **end for**
8: select the outcome $S = (RP, PA)$:

$$(RP, PA) = \text{argmin}\{WE(RP_\epsilon, PA_\epsilon) | (RP_\epsilon, PA_\epsilon) \in C_S\}$$

only, this stage performs parallelism tuning guided by energy densities.

The theory of state-returning energy in Section IV-B enables the control of the PGR granularity and the algorithm complexity. Estimating the SREs of a candidate region R_i asserts the upper bound on the error of state-transition energy resulted from parallelism tuning. By relating the error upper-bound to R_i's own computational energy, the PGR granularity can be indicated by the *error ratio* ϵ. And ϵ should be controlled within a limited range (0, 1). Our experiment shows that only a limited amount of ϵ values have to be examined to obtain a solution with good energy efficiency. The concept will be formalized in Section V-A.

We elaborate the framework of Algorithm 1 as follows. Section V devise the algorithm on theoretical aspects: assuming the whole program is modeled as a (huge) CFG. Besides realization of SRE-ED strategy, we also make improvements to the PGR establishment stage. The PGR establishment is a N-way multi-cut problem which is proved to be NP-complete [45]. Nevertheless, a CFG resulted from a high-level language program can usually be classified into a limited set of structure types. We propose a polynomial-time algorithm to obtain an optimal partitioning when some well-structuring assumption holds on the CFG. Section VI deals with practical issues to realize the algorithm, where a program contains multiple functions. A cross-procedural optimization algorithm for parallelism scaling is developed. Back-off heuristics are also proposed to deal with situations when the idealized assumptions do not hold.

V. THE SRE-ED ALGORITHM: THEORETICAL ASPECTS

Here we devise the SRE-ED algorithm theoretically with idealized assumptions. Section V-A to V-C elaborate the 3-stage algorithm to establish a solution $(RP_\epsilon, PA_\epsilon)$ for a pass. Section V-D summarizes the development as a multi-pass algorithm.

A. IDENTIFYING PGR CORES WITH GRANULARITY CONTROL

We propose two criteria, the *item-SRE* and *region-SRE* criteria, to check whether a candidate loop R_i is eligible to be a PGR core or not. Quantities involved in the checking are as follows:

1) $T_{cp}(R_i, p_{max})$: the ETPE of R_i when executed with maximum parallelism,
2) $\bigtriangleup E_i(l)$: the increased computational energy per entrance when we switch the speed-level from $(l-1)$ to l for some $l \geq 1$,
3) $E_0(R_i)$: the computational energy per entrance of R_i at speed-level 0,
4) $SRE_i(l)$: the state-returning energy to switch PG-configs between speed level $(l-1)$ and l on R_i.

Criteria 1 (Item-SRE Criteria): R_i is eligible to form a PGR if (1) $T_{cp}(R_i, p_{max}) \geq T_{th}$, and (2) $\frac{SRE_i(l)}{\bigtriangleup E_i(l)} \leq \epsilon$ for each speed level $l \geq 1$. ◇

Criteria 2 (Region-SRE Criteria): R_i is eligible to form a PGR if (1) $T_{cp}(R_i, p_{max}) \geq T_{th}$, and (2) $\frac{SRE_i(l)}{E_0(R_i)} \leq \epsilon$ for each speed level $l \geq 1$ ◇

Granularity control is fully parameterized through two parameters T_{th} and ϵ. Each criteria has two rules. Rule (1) asserts a lower-bound on ETPE such that the energy saved by powering off a hardware does not exceed the additional energy on activation and deactivation. We set T_{th} to be the ratio of activation and deactivation energy to the static energy per cycle. (Note that we can also enlarge T_{th} to tolerate the state-transition time.) Rule (2) asserts the SRE to be within a small portion of the computational energy affected by parallelism tuning. The ratio is controlled by the parameter $\epsilon \in (0, 1)$. Our experiments examine the effect of $\epsilon \in \{0.1, 0.2, 0.3, \ldots, 0.9\}$ and select the solution with minimum energy cost. The criteria are to check whether a loop has a sufficiently large iteration count such that the SRE is small compared to the computational energy. The quantity $SRE_i(l)$ is independent of the iteration count. The quantities $T_{cp}(R_i, p_{max})$, $\bigtriangleup E_i(l)$, and $E_0(R_i)$ are proportional to the iteration count. The criteria will be eventually satisfied if the iteration count exceeds a certain threshold.

Algorithm 2 identifies PGR cores over the loop hierarchy tree (LHT) built for a CFG, where a node L represents a loop (a strong component in the CFG) and a child of L represents an inner loop of L. Algorithm 2 is recursive and examines each loop from the bottom-up of LHT. At Line 11, one may decide to apply Criteria 1 or Criteria 2 to check a loop L if no inner loops of L are marked as PGR cores.

B. PGR COMPOSITION FOR OPTIMAL PARTITIONING

The next stage is to establish PGRs by expanding from the identified PGR cores. Recall Section IV that we are to partition a CFG with R-nodes and B-nodes and trying to find a minimum cut. The N-way minimum cut problem is NP-complete for an arbitrary graph [45]. Nevertheless, a CFG resulted from a high-level language program is ''usually

TABLE 3. Definitions of structural components.

Type	Definition	Program Construct
N-structure	A single-entry region SG is a **single-node structure**, or **N-structure**, if SG contains only one B-node	a basic block
R-structure	A single-entry region SG is a **R-node structure**, or **R-structure**, if SG contains only one R-node	a loop marked as PGR core
S-structure	A single-entry region SG is a **sequential structure**, or **S-structure**, if SG can be decomposed into subgraphs $\{SG_0, SG_1, SG_2, ...SG_{N-1}\}$ such that (1) Each SG_i is a single-entry region and well-structured having the single entry node (head node) HB_i, (2) For each sub-component SG_i where $0 \leq i \leq N-2$, all out-going edges of SG_i goes to HB_{i+1}, and (3) Each out-going edge of SG_{N-1} goes out SG.	a compound statement
L-structure	A single-entry region SG is a **loop structure**, or **L-structure**, if SG can be decomposed into head block HB and a subgraph SG_0 such that (1) HB is the unique entry and exit node of SG, (2) The subgraph SG_0, called the loop body, is a well-structured single-entry region with entry node HB_0, (3) The unique entry edge into SG_0 is (HB, HB_0), and (4) All out-going edges of SG_0 go to HB.	a for or while loop
B-structure	A single-entry region SG is a **branch structure**, or **B-structure**, if SG can be decomposed into head node HB and a set of subgraphs $\{SG_0, SG_1, SG_2, ..., SG_{N-1}\}$ such that (1) The head node HB is the unique entry node of SG, (2) Each sub-component SG_i, called a branch, is a single-entry region which is well-structured and has a unique entry node HB_i, (3) For each sub-component SG_i, the unique incoming edge is (HB, HB_i) and all out-going edges go out of SG.	an if-then-else or switch-case statement
DW-structure	A single-entry region SG is a **do-while structure**, or **DW-structure**, if SG can be decomposed into tail node TB and a subgraph SG_0 such that (1) The tail node TB is the unique exit node of SG, (2) The sub-component SG_0, called the loop body, is a well-structured single-entry region, (3) The unique entry node HB_0 of SG_0 is also the unique entry node HB of SG, (4) All out-going edges of SG_0 go to TB, and (5) There is an edge (TB, HB_0).	a do-while statement

Algorithm 2 *RMark_CoreLoop*

Input: a node L in the LHT
Return: amount of PGR cores identified in the subtree of L

```
{Recursion into each sub-loop}
Cnt ← 0
for each sub-loop L_i of L do
    Cnt ← Cnt + RMark_CoreLoop(L_i);
end for

{terminate if there are sub-loops marked as PGR-core}
if Cnt > 0 then return Cnt;

{check loop L with the criteria}
if L satisfies the PGR core criteria then
    mark the program region of L as a PGR core;
    return 1;
else
    return 0;
end if
```

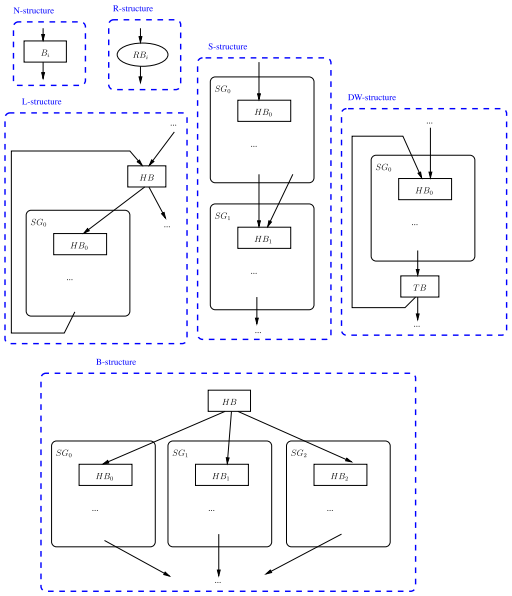

FIGURE 9. Patterns of structural components.

well-structured''. Here we propose a polynomial-time algorithm to obtain the optimal RP-solution for such a well-structured CFG. Brief description of the PGR composition algorithm is given here and the details are at Appendix B.

A CFG is **well structured** in the sense that the CFG can be re-organized as a hierarchy of structural components. A compiler can partition the CFG into a hierarchy of **single-entry regions**, where each region is a subgraph of the CFG with one entrance block [46]. We further observed that single-entry regions resulted from a high-level language program usually fall into a limited set of structure types. A **structural component** is a single-entry region that matches one of the structure-type classification rules in Table 3. Figure 9 shows the graphical patterns for each structure type. The bottom levels are structures having only one node: a **N-structure** for a B-node and an **R-structure** for an R-node (a loop marked as a PGR core). In an intermediate level, a structural component *SG* has a head node *HB*, the unique entry node, and a set of sub-components $\{SG_0, SG_1, SG_2, \ldots\}$, where each SG_i is also a single-entry region. The structures are formed from typical program constructs such as a for loop, a while loop, a switch-case or an if-then-else statement. The hierarchy is represented as a **structural component tree** (SCT), in which a node is a structural component *SG* and a child of *SG* is one of its sub-components. Figure 10 gives an example: the program at left-hand side is transformed to the hierarchy of single-entry regions (by LLVM v2.9 [44]) at the right-hand side. The for loop forms the L-structure SG_1, which has a head node and a loop body SG_3 as a S-structure. In the loop body SG_3, the if-then-else statement forms the B-structure SG_4.

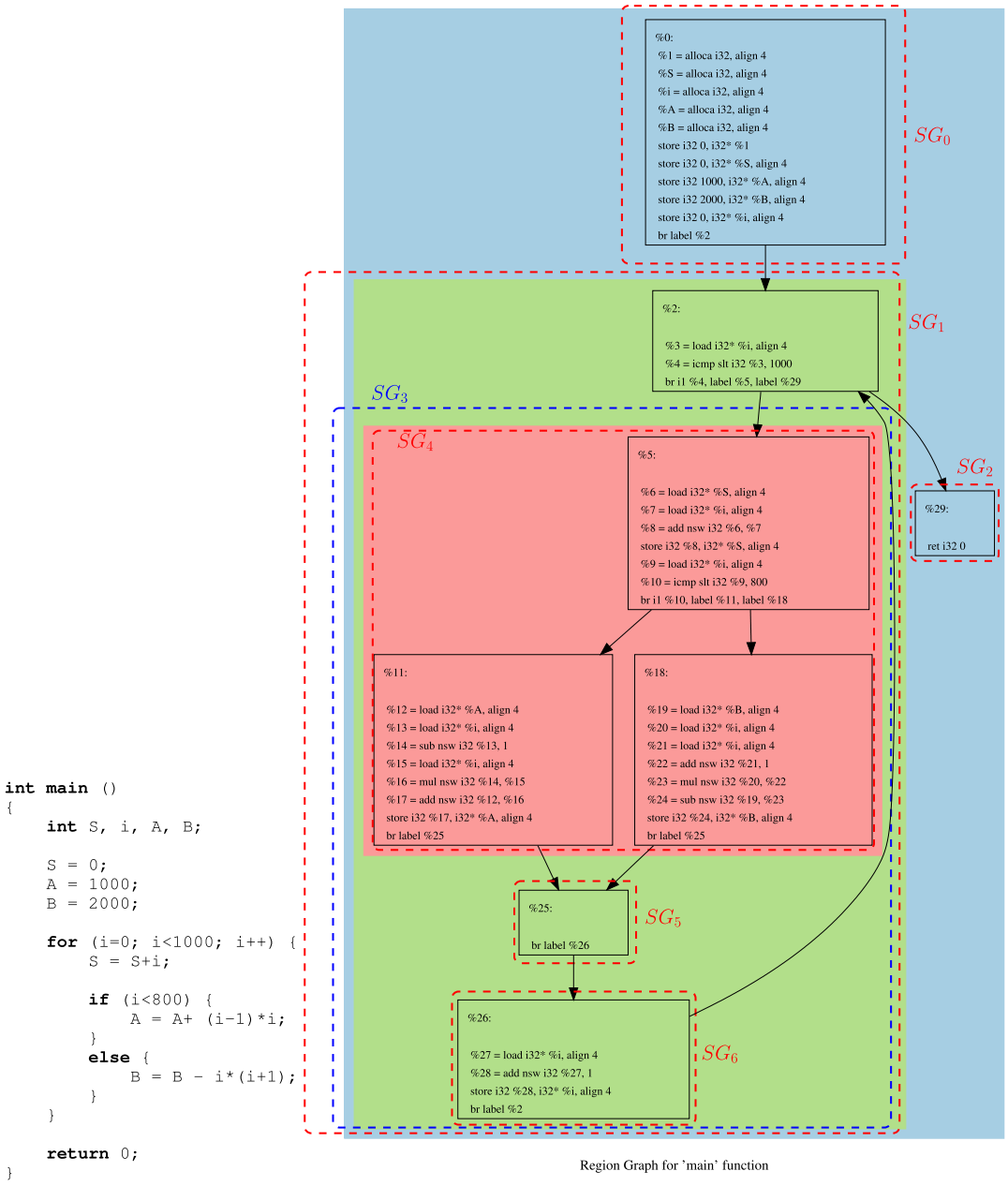

FIGURE 10. Sample code to show structuring hierarchy.

1) BASIC CONCEPTS OF PGR COMPOSITION

Our algorithm builds RP-solutions bottom-up with a SCT. Consider a structural component SG having sub-components $\{SG_0, SG_1, SG_2, \ldots\}$. A solution RP over SG is "composed from" a set of **partial solutions** $\{RP_0, RP_1, RP_2, \ldots\}$, where each RP_i is a RP-solution over the sub-component SG_i. In the composed solution RP, a PGR R_j is either a PGR in some RP_i or built by merging PGRs from multiple partial solutions. In the latter case, such R_j spreads across multiple sub-components.

An RP-solution RP over a structural component SG is a partitioning over nodes in SG with two additional pseudo nodes: PN_s at the start side and PN_e at the end side. Each of PN_s and PN_e can be assigned to be either R-type or B-type. Let R_s be the PGR containing PN_s and R_e be the one containing PN_e. RP is **extend-out style** (**EO-style**) at the start side if PN_s is B-type, which means that R_s (rooted at some R-node in SG) is to be extended out and merge with other PGR outside SG. RP is **extend-in style** (**EI-style**) at the start side if PN_s is R-type and R_s has nodes other than PN_s. In this case, a PGR outside SG (rooted at an R-node also outside SG) is to be extended into SG and merge with nodes in R_s. RP is **closed style** (**C-style**) at the start side if PN_s is the unique nodes in R_s. Similarly, the end side of RP can also be

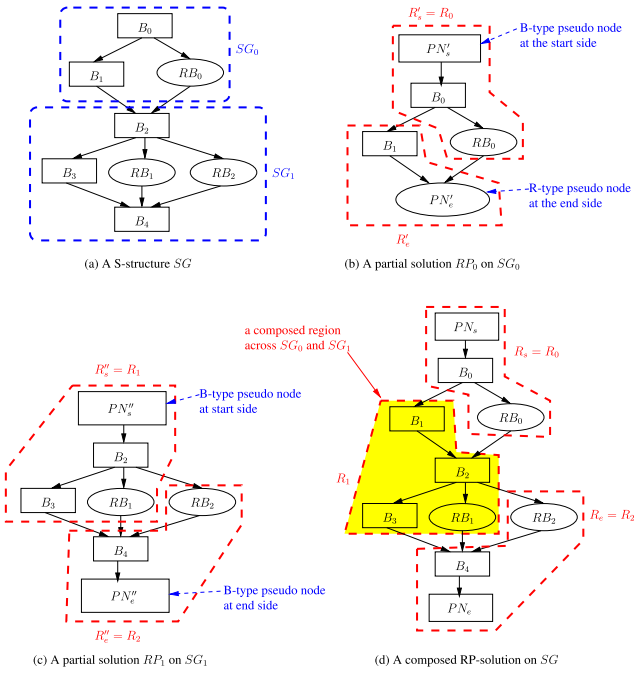

FIGURE 11. Example: PGR composition on S-structure.

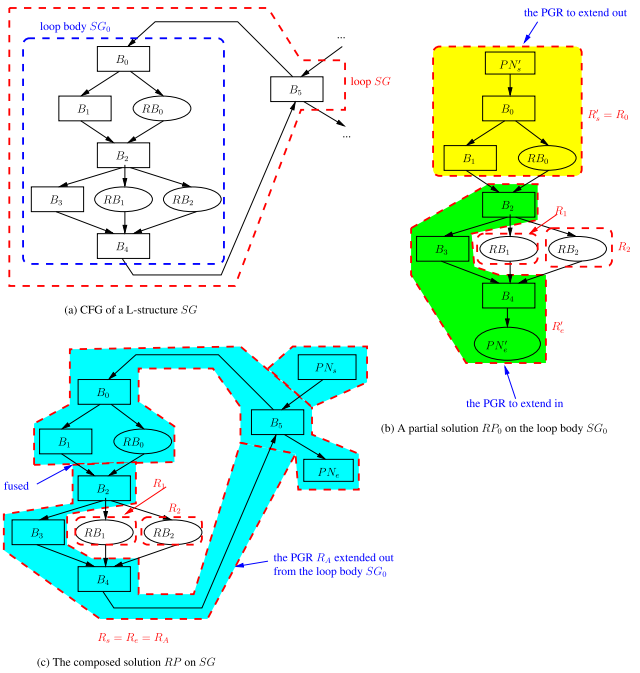

FIGURE 12. Example: compose a DSEO solution on L-structure.

classified into one of EO/EI/C-style depending on the type of PN_e and R_e.

Figure 11 illustrates how PGR composition works. Imagine that a PGR is "grown" from an R-node as the root and gradually includes B-nodes into the PGR. Figure 11(a) is a S-structure SG having two sub-components SG_0 and SG_1. Figure 11(b) shows a partial solution RP_0 on SG_0, which is EI-type at the end side. Figure 11(c) is a partial solution RP_1 over SG_1, and is of EO-type on the start side. RP_0 and RP_1 are eligible to compose a new solution since the EO/EI type matches on the contacting side. Figure 11(d) shows the solution RP over SG, which is composed from RP_0 and RP_1. The solution RP contains PGRs in RP_0 and RP_1. The two distinguished PGRs R'_e and R''_s are merged to form a PGR R_1 spread across SG_0 and SG_1. R_1 is connected and has exactly one R-node RB_1.

2) RULE-BASED PGR COMPOSITION ALGORITHM

RP-solutions over a structural component are classified into 12 classes. For a solution RP with $R_s \neq R_e$, the type code $X - Y$ denotes that RP is X-style at the start side and Y-style at its end side, where X and Y are one of the direction code EO, EI, C. For the example in Figure 11, the partial solution RP_0 is EO-EI type and the composed solution RP is EO-EO type. For RP-solutions that places the two pseudo nodes in the same PGR $R_{se} = R_s = R_e$, we make the three distinct types: (1) **DSEO type**: RP is **double-sided extend-out** if R_{se} contains B-type pseudo nodes on both sides. (2) **FET type**: RP is **forward extend through** type if R_{se} has a R-type pseudo node at the start side and a B-type pseudo node at the end side. (3) **BET type**: RP is **backward extend through** type if R_{se} has a B-type pseudo node at the start side and a R-type pseudo node at the end side.

RP-solutions are built from case-by-case composition rules. For each structure type and target solution type, a composition rule is established to specify how such a RP-solution is composed from partial solutions. A rule has two parts: (1) specifying all available type combinations to select partial solutions from sub-components, and (2) rules to compose PGRs by merging and keeping PGRs in partial solutions. There are 4*12 = 48 rules, and the complete rule table is in Appendix B. Table 4 lists part of the rule table to illustrate how the composition rules work.

Examples of PGR composition using the rule table are as follows. Figure 11 is an example of the first case in Table 4: composing an EO-EO solution on a S-structure with two sub-components. The example selects partial solutions in the type combination (EO-EI, EO-EO) listed in the table. Figure 12 shows an example of the second case in Table 4: composing a DSEO solution on a L-structure SG. A partial solution on the loop body is eligible to build the target type if it has exactly one PGR in $\{R_s, R_e\}$ for extending out. The type selection rule enumerates all type combinations satisfying the constraint. In Figure 12(b), we select a partial solution in EO-EI type. The DSEO region R_{se} is built as follows: R_{se} is initially rooted at R'_s on the start side of the loop body, extended out to cover $\{B_5, PN_s, PN_e\}$ of the loop, and then extended back into SG_0 to cover R'_e. This results in the solution in Figure 12(c). PGRs may be fused after composition. This affects cost estimation for optimality and distinguished types are made for the case that $R_s = R_e$.

Algorithm 3 shows the framework to compose RP-solutions. The algorithm is recursive and RP-solutions are built bottom-up from the SCT. For a structural component SG, at most 12 RP-solutions are built for SG: one **representative**

TABLE 4. Example of PGR composition rules.

Case	Structure type	Target solution type	Rule to select partial solutions	Rule to compose the target solution
1	S-structure	EO-EO	Available combination of partial solutions $(RP_0, RP1)$ from sub-components (SG_0, SG_1) are • (EO-EI, EO-EO) • (EO-EI, DSEO) • (BET, EO-EO) • (EO-EO, EI-EO) • (EO-EO, FET) • (DSEO, EI-EO) • (EO-C, C-EO)	Build PGRs in RP as follows: (1) merge R'_e of RP_0 with R''_s of RP_1 (2) keep remaining PGRs in RP_0 and RP_1
2	L-structure	DSEO	Select partial solution RP_0 of the loop body SG_0 from one of the following classes: • DSEO • EO-EI • EO-C • EI-EO • C-EO	Build PGRs in RP as follows: (1) set $R_s = R_e = R'_s \bigcup R'_e \bigcup \{HB, BS_s, BS_e\}$ (2) keep remaining PGRs in RP_0
...				

Algorithm 3 Recursive Algorithm to Compose Representative RP-Solutions

Input: SG

Output: $\{RP_T(SG)\}$

if SG is a R-structure or a N-structure **then**
  compose all representative solutions for SG as shown in Figure 13;
  return;
else
  for each sub-component SG_i of SG **do**
    perform Algorithm 3 to build representative solutions $\{RP_T(SG_i)\}$ for SG_i;
  end for
end if

for each solution type T **do**
  initiate the candidate set $C_{RP} \leftarrow \emptyset$;
  for each type combination $(T_0, T_1, \ldots, T_{n-1})$ on partial solutions for type T **do**
    if the set of partial solutions $\{RP_{T_i}(SG_i)\}$ exists **then**
      compose type-T solution RP_c of SG from $\{RP_{T_i}(SG_i)\}$;
      $C_{RP} \leftarrow C_{RP} \bigcup \{RP_c\}$;
    end if
  end for
  pick the representative solution

$$RP_T(SG) = \text{argmin}\{C_{str}(RP_c) | RP_c \in C_{RP}\};$$

end for

solution $RP_T(SG)$ for each solution type T. For the bottom level of SCT, Figure 13 enumerates all solutions on a N- and R-structure. In an intermediate level, the solution $RP_T(SG)$ is built by composing representative solutions from its sub-components. The algorithm examines each possible type combination $(T_0, T_1, T_2, \ldots, T_{n-1})$ of partial solutions for

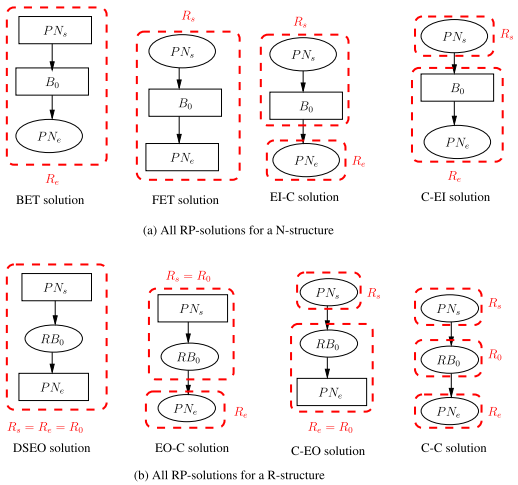

FIGURE 13. Representative solutions for single-node structures.

the target type T. The solution $RP_{T_i}(SG_i)$ is selected if type-T_i partial solution from SG_i is needed. A candidate solution RP_c is built if all partial solutions in the set $\{RP_{T_i}(SG_i)\}$ are available. The candidate solution with minimum transition cost $(C_{str}(RP_c))$ is picked as $RP_T(SG)$ for the target type. We prove that the algorithm generates $RP_T(SG)$ when the set of type-T solutions on SG is not empty. Moreover, we prove that $RP_T(SG)$ has minimum transition cost over all type-T solutions on SG. At the root of this SCT, the representative C-C solution is selected as the outcome. Appendix B gives the detailed proof.

C. PARALLELISM ASSIGNMENT

The last stage is to find a PA-solution $PA = \{p_i\}$. The objective is to minimize the computational energy $WE_{cp}(RP, PA)$ subject to the deadline constraint $WT_{cp}(RP, PA) \leq T'_{DL}$ on computation time, where T'_{DL} is calculated from the deadline T_{DL} with state-transition time excluded. The time needed for state-transition is determined by RP and is independent of the parallelism assignment. Algorithm 4 is the proposed

Algorithm 4 GPED Algorithm

Input: $(RP = \{R_i\}, T'_{DL})$
Output: $PA = \{p_i\}$

1: {Initialize}
2: $I_{PA} \leftarrow \emptyset$ and $I_{ready} \leftarrow \emptyset$;
3: $(WT_{cp}, WE_{cp}) \leftarrow$ the total computation time and energy of assigning all PGRs with speed-level 0;
4: **for** each PGR R_i **do**
5: **if** $I_{i,1}$ exists **then**
6: $I_{ready} \leftarrow I_{ready} \bigcup \{I_{i,1}\}$;
7: **end if**
8: **end for**
9:
10: {Iterative packing to establish I_{PA}}
11: **while** $WT_{cp} > T'_{DL}$ and $I_{ready} \neq \emptyset$ **do**
12: remove $I_{i,j}$ with minimum $\bigtriangleup ED(I_{i,j})$ from I_{ready};
13: $I_{PA} \leftarrow I_{PA} \bigcup \{I_{i,j}\}$;
14: $WT_{cp} \leftarrow WT_{cp} - \bigtriangleup WT(I_{i,j})$;
15: $WE_{cp} \leftarrow WE_{cp} + \bigtriangleup WE(I_{i,j})$;
16: **if** $I_{i,j+1}$ exists **then**
17: $I_{ready} \leftarrow I_{ready} \bigcup \{I_{i,j+1}\}$;
18: **end if**
19: **end while**
20:
21: **return** solution PA built from I_{PA};

heuristic, named greedy packing by energy density (GPED). Parallelism assignment is treated as an item packing problem and an item $I_{i,j}$ stands for the speed-level j of a PGR R_i. Figure 14 illustrates how the algorithm works on the item-stack model using the same example on Figure 7. The parallelism tuning starts from an initial solution PA_0, which assigns all PGRs with speed-level 0. The tuning iteratively packs items into a set I_{PA}, which is the set of items below the skyline of a solution PA. Packing $I_{i,j}$ into I_{PA} is to raise the parallelism of R_i from speed-level $(j-1)$ to j. The effect is to reduce the computation time by $\bigtriangleup WT(I_{i,j})$ at the cost of additional energy $\bigtriangleup WE(I_{i,j})$. The policy is to select the ready item $I_{i,j}$ with minimum energy density: $\bigtriangleup ED(I_{i,j}) = \frac{\bigtriangleup WE(I_{i,j})}{\bigtriangleup WT(I_{i,j})}$. (An item $I_{i,j}$ is ready if $I_{i,j-1}$ is already in I_{PA}.) The procedure terminates when the performance constraint is satisfied.

D. SUMMARY: THE SRE-ED ALGORITHM

Algorithm 5 finds the parallelism scaling solution, which is elaborated from the framework at Algorithm 1. Two approaches, the item-SRE and region-SRE approach, share the same framework depending on which criteria at Section V-A is used at Line 3. This is a multi-pass algorithm and a pass builds a candidate solution $S_\epsilon = (RP_\epsilon, PA_\epsilon)$ from PGR-cores identified with control parameter (T_{th}, ϵ). The parameter T_{th} is fixed to the overhead ratio: $T_{th} =$(activation plus deactivation energy)/(static energy per cycle). In our experiments, we set $\epsilon \in \{0.10, 0.20, 0.30, \ldots, 0.90\}$. The candidate solution with minimum energy is selected as the outcome.

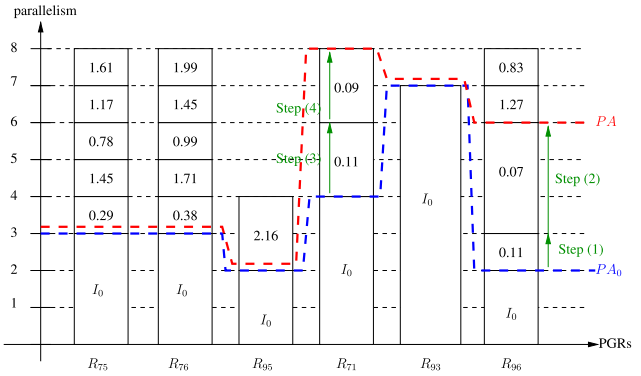

FIGURE 14. Running example of GPED algorithm.

Algorithm 5 Algorithm to Find a Parallelism Scaling Solution for PCEO

Input: (CFG, T_{DL}, T_{th})
Output: $S = (RP, PA)$

1: initiate candidate solution set $C_S = \emptyset$;
2: **for** each possible ϵ **do**
3: perform Algorithm 2 to identify a set of PGR-cores $\{RB_0, RB_1, RB_2, \ldots\}$ with control parameters (T_{th}, ϵ);
4: perform Algorithm 3 to establish PGRs $RP_\epsilon = \{R_0, R_1, R_2, \ldots\}$ from PGR-cores $\{RB_0, RB_1, RB_2, \ldots\}$;
5: perform Algorithm 4 to decide parallelism $PA_\epsilon = \{p_i\}$ over RP for deadline T_{DL};
6: $C_S \leftarrow C_S \bigcup \{(RP_\epsilon, PA_\epsilon)\}$;
7: **end for**
8: select the outcome $S = (RP, PA)$:

$$(RP, PA) = \text{argmin}\{WE(RP_\epsilon, PA_\epsilon) | (RP_\epsilon, PA_\epsilon) \in C_S\}$$

VI. PRACTICAL REALIZATION OF SRE-ED ALGORITHM

Two issues must be dealt with to realize Algorithm 5 for practical use: (1) a back-off heuristic for program partitioning in case that the CFG is not well structured, and (2) cross-procedural partitioning such that a PGR may spread across multiple functions. We briefly describe the policies here and the details are in Appendix C.

A. BACK-OFF HEURISTIC FOR ARBITRARY CFG STRUCTURES

The back-off heuristic for PGR composition is as follows. A single-entry region not fitting to any type in Table 3 is marked as an **irregular structure** (**I-structure**). The PGR composition still builds representative RP-solutions bottom-up with the SCT. RP-solutions over an I-structure is built by greedy expansion through the edge with maximum state-transition count.

B. PARALLELISM SCALING AS CROSS-PROCEDURAL OPTIMIZATION

The second issue is to build cross-procedural program partitioning. Modeling the whole program as a CFG is sound

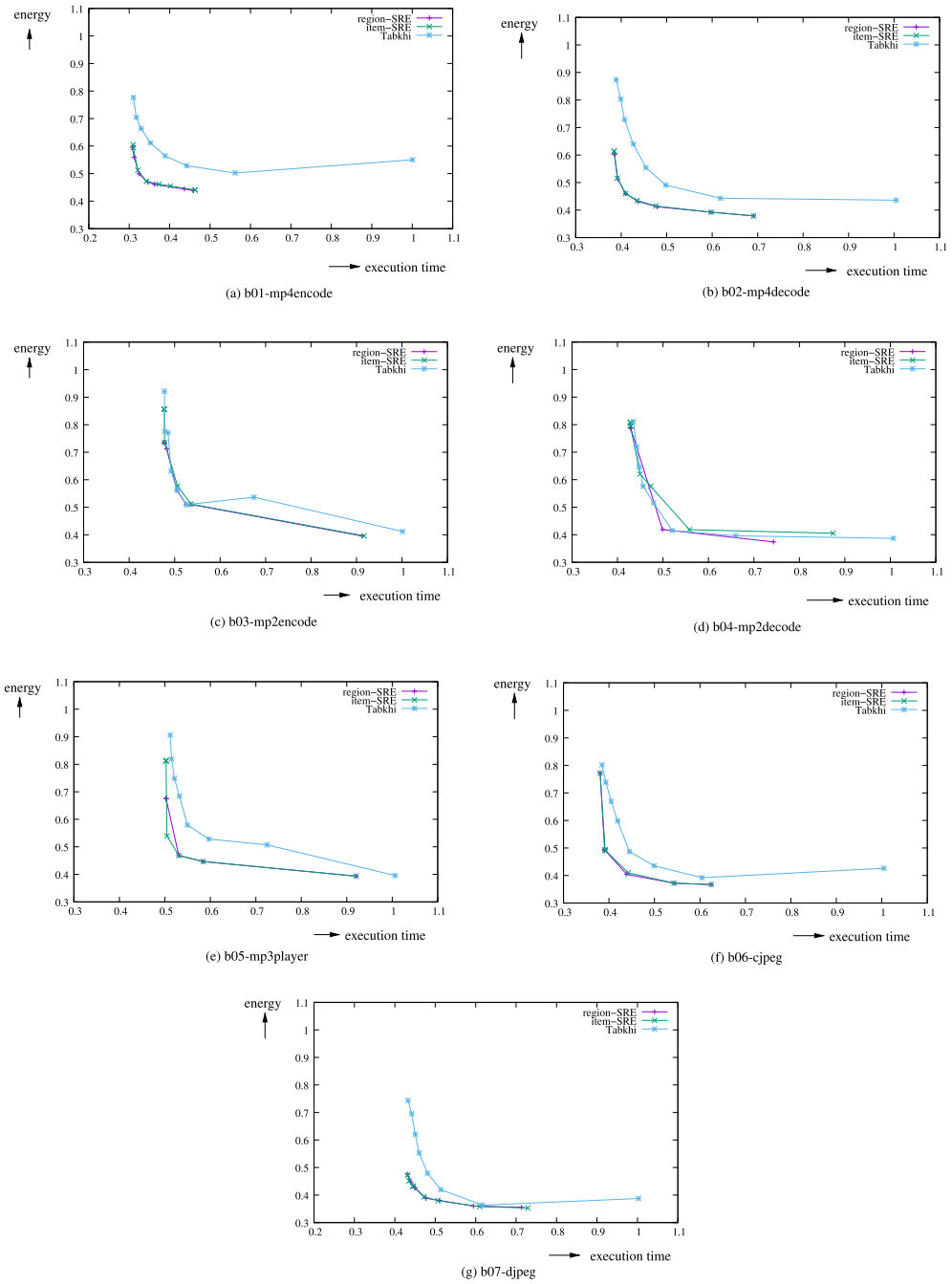

FIGURE 15. Evaluation results with OVR = 100 on the E-T space.

in theory but not realistic in practice. Usually an application program has multiple functions, and a CFG generated by the compiler platform (such as LLVM [44]) covers only a single function. Expanding all functions to build a (huge) CFG is not realistic. Moreover, there may be small functions with very low ETPE that are not eligible to form any PGRs, and a PGR may need to spread across multiple functions.

Cross-procedural program partitioning is realized as follows. Algorithms 2 and 3 are implemented to work on a patched CFG covering a single function. The CFG is patched with **function structures** (**F-structures**) to link information to callees. PS-solutions are obtained by Algorithm 5 with Lines 3 and 4 work on the call graph. Both the PGR-core identification and PGR-composition stage examines the call graph (with strong components merged as a single node) in reverse topological order. Upon examining a function, Algorithms 2 and 3 work on a CFG with F-structures and the resulting PGR may spread across multiple functions. The details are in Appendix C.

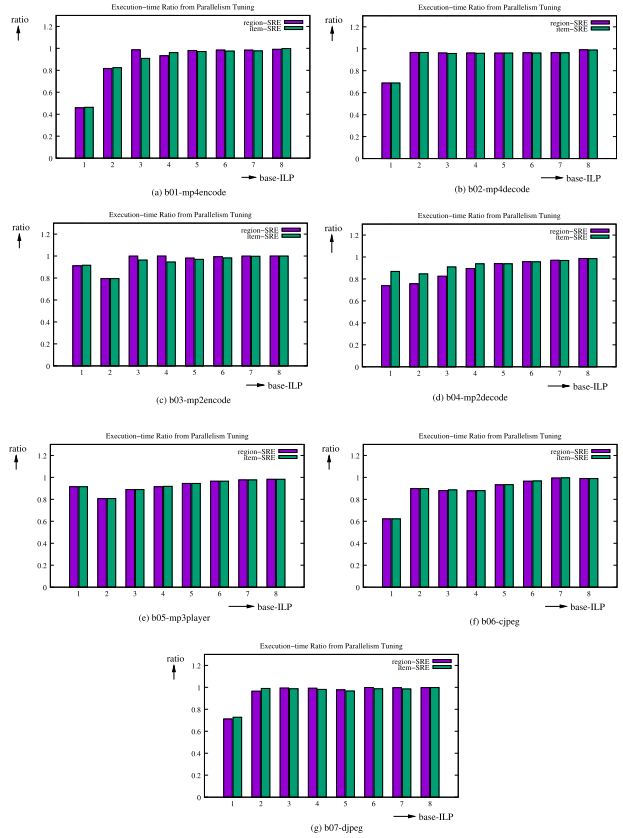

FIGURE 16. Ratio of execution time resulted from parallelism tuning.

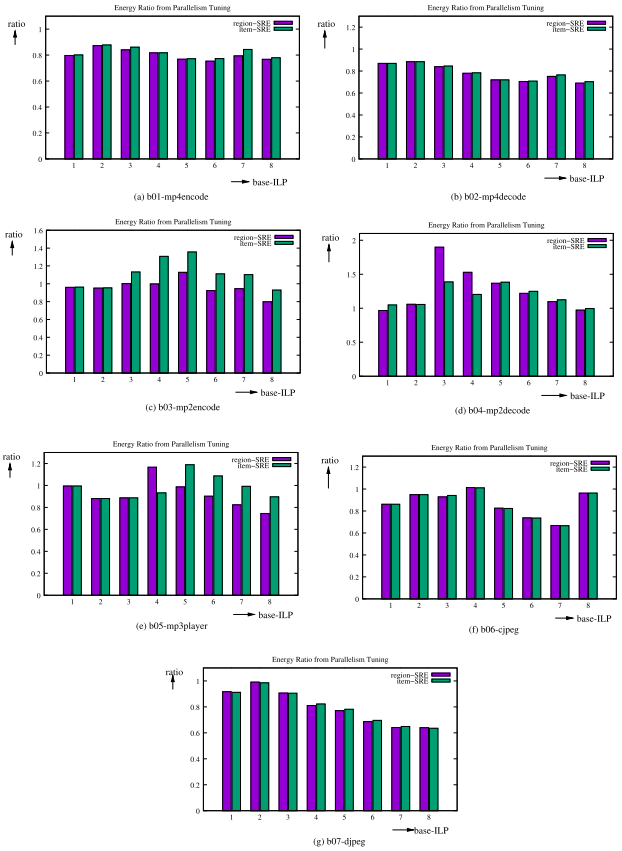

FIGURE 17. Ratio of energy resulted from parallelism tuning.

VII. EVALUATION

The evaluation on energy efficiency is conducted using the Denbench benchmark suite [47]. The proposed algorithm is implemented on the LLVM compiler platform (version 2.9) [44] to obtain PS-solutions. An energy model is established through logic synthesis with SAED 28/32 nm [48] cell library to obtain the static energy over functional units and register files of the PGRF-VLIW architecture [43]. Here we present analysis data to draw the major conclusions. More data for insight analysis can be found in Appendix D.

The evaluation is on an 8-issue PGRF-VLIW architecture, which contains a shared register file (SRF) connected to all execution slots; each execution slot has its own local register file (LRF). The SRF has 16 read ports, 8 write ports, and 8 banks of registers with 4 registers per bank for power gating. A LRF has 2 read ports, 1 write port, and 4 banks of registers with 4 registers per bank. (The feasibility in terms of implementation overhead was justified in [43].) The VLIW architecture has 8 homogeneous integer execution slots, and 2 of the execution slots can execute load/store operations. Each execution slot is an individual power domain. We take an LLVM-IR operation as a machine operation, which matches standard RISC instruction set.

Static power is obtained by synthesizing functional units and register files with the design constraint of 1 GHz clock frequency. All shared and local register files are implemented in Verilog for synthesis. We synthesize the ALU, multiplier, and divisor from RISC-V Rocket core [49] to obtain parameters of an execution slot. We adopt the design proposed in [6], in which executing a power-gating instruction costs only one cycle. We introduce the parameter *OVR* to indicate the overhead ratio: the activation and deactivation energy are calculated as the static energy per cycle times *OVR*. According to previous works [4], [6], we analyze the energy efficiency assuming $OVR = 100$. Various physical design factors may affect OVR [4] and there are other researchers report OVR around 10 to 20 [39]. We also evaluate our approach for *OVR* ranging from 10 to 1000.

We compare our approaches to two baseline approaches:

- Baseline 1: the traditional VLIW architecture without power gating, which has a shared register file connected to all execution slots. Application programs are scheduled with fixed ILP = 8.
- Baseline 2: the Tabkhi's approach to manage power-gated hardware [36]. A function forms a PGR and a uniform parallelism is assigned to all PGRs. To the best of our knowledge, the Tabkhi's work is the most relevant state-of-art work on quantitative management over power-gated hardware resources.

A. ENERGY EFFICIENCY OF SCALING PARALLELISM

Evaluation results are presented as solution series projected onto the energy-time space (E-T space). Figure 15 shows the

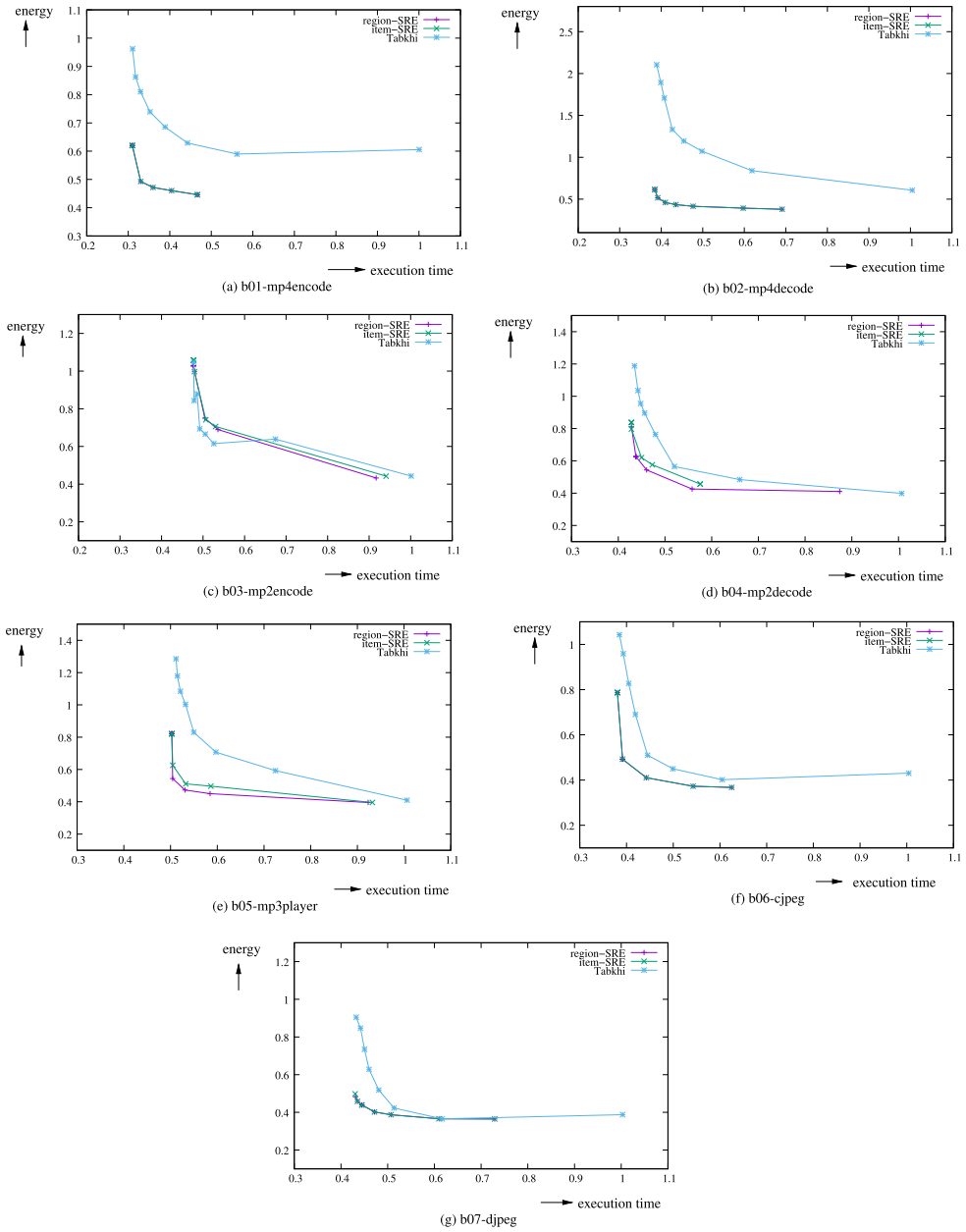

FIGURE 18. Solution series on E-T space for OVR = 1000.

results for OVR = 100. A PS-solution S is projected to the point $(WT(S), WE(S))$. Time and energy are normalized:

- 1 unit of the time is the time to execute the application with one execution slot without power-gating,
- 1 unit of energy is the total energy to execute the application on traditional VLIW architecture without power gating (Baseline 1).

A solution series is represented by a curve on the E-T space. The baseline curve "Tabkhi" is the solution series $\{S_1, S_2, \ldots, S_8\}$ where each S_k is resulted from the Baseline 2 and scheduled with the uniform parallelism k. We take execution time from the baseline series as a set of deadline constraints to generate improved solutions from our algorithms. The curve "region-SRE" is the solution series $\{S'_1, S'_2, \ldots, S'_8\}$ generated by the region-SRE approach, where the solution S'_k is generated by setting deadline constraint $T_{DL} = WT(S_k)$. We call S'_k the solution obtained by parallelism tuning from the **base-ILP** k. Similarly, the "item-SRE" curve is generated by the item-SRE approach with $T_{DL} = WT(S_k)$ for each base-ILP k. Sweeping from right to left, the curve of a solution series shows how the energy cost scales up as performance demand rises. The bottom-right corner stands for the minimum energy mode and the peak-performance mode is at the top-left corner.

Each of the power-gating approaches has significant energy saving in peak-performance mode: each curve lies under the horizontal line of energy = 1, the energy cost of the non-power-gated approach. Compared to Baseline 1, our approaches (both region-SRE and item-SRE) save 15% to 53% of the static energy in the peak-performance mode. Compared to Baseline 2, our approaches reduce 23% to 32% of the energy cost in peak-performance mode except for "b04-mp2encode" and "b06-cjpeg". On "b04-mp2encode" and "b06-cjpeg", our approaches save 5% and 8% of baseline energy, respectively.

Overall, our approaches reduce the energy required to meet performance demand in PCEO mode compared to Baseline 2. The curves of region-SRE and item-SRE are lower than "Tabkhi", except for the two programs "b03-mp2encode" and "b04-mp2decode". Figure 16 and 17 shows an alternative view to the results: the ratio of execution time ($\frac{WT(S'_k)}{WT(S_k)}$) and energy ($\frac{WE(S'_k)}{WE(S_k)}$) obtained from parallelism tuning for each base-ILP k. Our parallelism tuning satisfies the performance demand for each tuning case: the execution-time ratio is under 1.00 in Figure 16. Figure 17 shows that the energy saving effect becomes obvious when we raise the performance demand. For high performance demand with base-ILP$\geq$5, our approaches save 20% to 30% of the energy required to meet the same performance demand. For most of the tuning cases, the region-SRE and item-SRE approach have comparable efficiency. However, "b03-mp2encode" and "b05-mp3player" shows that the region-SRE approach has a more stable optimization effect compared to the item-SRE approach. When performance demand is reduced (for base-ILP$\leq$4), the advantage of our approaches is to reduce the execution time without increasing energy cost. Observed from Figure 16, the execution time is significantly lower than demand when base-ILP is 1 or 2.

The energy saving comes from exploiting the program behavior on the distribution of workload and energy densities. Refer to Appendix D for the analysis. The analysis shows that (1) core loops with sufficiently long ETPE occupies most of the workload, and (2) such loops have high variance on speedup-saturation parallelism/power and energy densities. For applications with such program behaviors, we can expect energy saving from parallelism scaling if the state-transition overhead is zero. The SRE-criteria serves as a filter to filter out core loops by the error ratio ϵ, which relates the state-transition overhead to the saving on computational energy. We apply multiple filters (ϵ values) to find a balance point between computational energy and state-transition overhead. The reduction on total energy indicates that such balance points do exist.

The evaluation reveals the room to save static energy by PCEO. Raising the demand from minimum-energy mode to peak-performance mode produces speedup values ranging from 1.49x to 2.04x. Moreover, 47% to 73% of the energy cost of peak-performance mode is reduced if we lower the demand to minimum-energy mode. Executing in peak-performance mode is relatively energy inefficient. In Figure 15, the curves of the solution series look like the letter "L". This means that, when pushing the performance demand near peak-performance, an exorbitant energy cost is paid just to gain minuscule saving in execution time. This encourages the application developer to do solution space exploration for PCEO rather than executing in peak-performance mode as in our previous work [18]. Considering all the evaluation results, we recommend the developer to do performance-constrained energy optimization using the region-SRE approach.

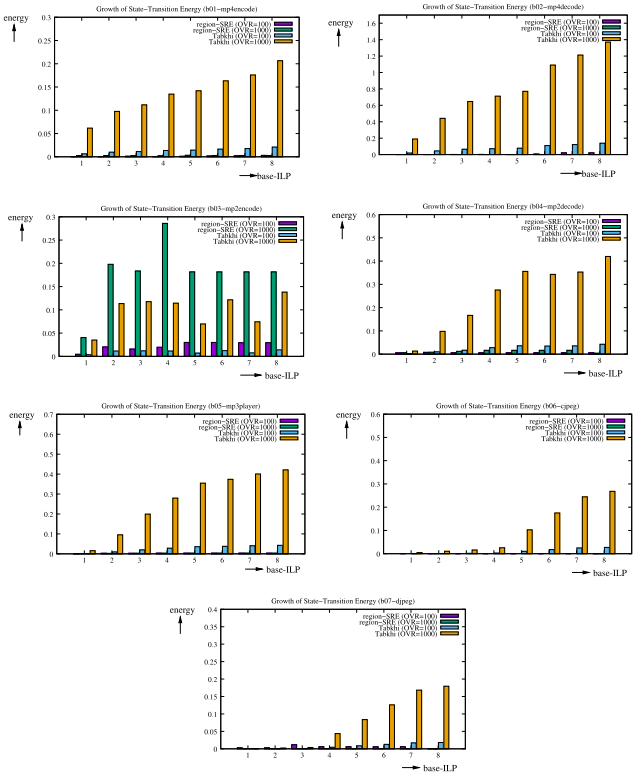

FIGURE 19. State-transition energy of region-SRE approach.

B. ADAPTION FOR POWER GATING OVERHEAD

We evaluated the energy efficiency for $OVR \in \{10, 50, 100, 500, 1000\}$ to show that our approach is suitable for various circuit technologies that varies on power gating overhead. High OVR results in difficulty gaining energy savings through power gating. A search of the literature revealed no articles that reported OVR greater than 1000 [4], [6], [39], [40]. Here we present the results for OVR = 1000. The complete results are in Appendix D.

Figure 18 shows the solution series when OVR = 1000. The region-SRE and item-SRE curves are still below the "Tabkhi" curve except for the benchmark program b03-mp2encode. For several benchmark programs, the Tabhki approach fails to save energy and has energy cost greater than 1.00. Curves of our approaches are still below the horizontal line of energy = 1.00 for most of the benchmark programs. Figure 19 shows the growth of the state-transition energy of the region-SRE approach when OVR increased

from 100 to 1000. (The item-SRE results are in Appendix D.) Our PGR-core identification criteria is still effective to adapt the PGR granularity for the state-transition overhead except for the program b03-mp2encode. For b03-mp2encode, our approach fails to find a balance point between computational energy and state-transition overhead. The success of parallelism scaling relies on the PGR granularity control to balance between computational energy and state-transition overhead.

VIII. CONCLUSION AND FUTURE WORK

This paper proposes the parallelism scaling approach to improve energy efficiency through power gating. The compiler algorithm attempts to depower hardware exceeding performance requirements and improve the efficiency to trade speedup with energy cost. The development establishes the mathematical programming theory as a general framework for parallelism control:

1) We developed a fully parameterized PGR granularity control scheme such that the program partitioning can adapt for power gating overhead. With the PGR-granularity control scheme, the multi-pass algorithm finds a balance between computational energy and state-transition overhead for performance-constrained energy optimization.
2) We proposed the PGR composition algorithm to obtain the optimal program partitioning in polynomial time. The N-way multi-cut problem is NP-complete for arbitrary graphs but partitioning the CFG of a high-level language program is not so hard. The optimal partitioning can be obtained from rule-based composition since there are only limited set of structure types.
3) We proposed a model and a heuristic to decide execution parallelism for performance-constrained energy optimization. Success of the parallelism tuning relies on the PGR granularity control to limit the state-transition overhead.

Evaluation over a VLIW-style architecture shows 20% to 30% energy saving to meet performance demand compared to the state-of-art approach [36]. Our approach also has significant energy saving at the peak-performance mode. Compared to the previous parallelism control scheme [18] at peak-performance mode, the evaluation reveals a significant room to save energy through parallelism control with performance demand. We recommend developers to do solution space exploration to find a sweet spot between performance and energy cost.

The proposed parallelism scaling algorithm may be improved from several aspects. The multi-pass algorithm to find a parallelism scaling solution is of high complexity. A future research direction is to seek for low-complexity algorithm that gets satisfactory energy efficiency. On the other aspects, how parallel instructions are exploited may also affect the efficiency of parallelism scaling. Experimenting the parallelism scaling framework with more instruction scheduling approaches, such as software pipelining on loops, is another future work.

The major future work is to experiment the theory with other styles of accelerators having parallel hardware. Unlike programming a general purpose CPU, programming an accelerator needs extensive works on profiling and instrumentation to direct how a program is offloaded and realized [22]. The parallelism scaling is a general framework to do solution space space exploration for accelerator programming. The theory can be applied to other styles of power-gated parallel hardware as long as the execution parallelism determines the speed and power consumption. We recommend to redesign the microarchitecture (as in the case of [43]) such that most of the processor's power is affected by parallelism control. Recently, several VLIW processors are enhanced with vector or SIMD instructions to support data-level parallelism (DLP) [7], [14]–[17]. The VLIW architecture with DLP support has more parallel hardware and is our next target to experiment the parallelism scaling theory. There are also other architectures having parallel hardware, such as GPUs or multi-core processors. These parallel architectures are also potential applications of the parallelism scaling theory.

REFERENCES

[1] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE J. Solid-State Circuits*, vol. SSC-9, no. 5, pp. 256–268, Oct. 1974.
[2] M. T. Bohr and I. A. Young, "CMOS scaling trends and beyond," *IEEE Micro*, vol. 37, no. 6, pp. 20–29, Nov. 2017.
[3] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Power limitations and dark silicon challenge the future of multicore," *ACM Trans. Comput. Syst.*, vol. 30, no. 3, Aug. 2012, Art. no. 11.
[4] Y. Shin, J. Seomun, K.-M. Choi, and T. Sakurai, "Power gating: Circuits, design methodologies, and best practice for standard-cell VLSI designs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 15, no. 4, pp. 1–37, Sep. 2010.
[5] M. Keating, *Low Power Methodology Manual for System-on-Chip Design*. New York, NY, USA: Springer, 2007.
[6] S. Roy, N. Ranganathan, and S. Katkoori, "A framework for power–gating functional units in embedded microprocessors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 11, pp. 1640–1649, Nov. 2009.
[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. San Mateo, CA, USA: Morgan Kaufmann, 2018.
[8] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *Proc. 23rd Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*. ACM: New York, NY, USA, 2018, pp. 751–766.
[9] Freescale Semiconductor, *Tuning C Code for StarCore-Based Digital Signal Processors*, document AN3357, 2008.
[10] *Getting Started With Blackfin Processors*, Analog Devices, Norwood, MA, USA, 2010.
[11] *Tms320c6455 Fixed-Point Digital Signal Processor*, Texas Instruments, Dallas, TX, USA, 2005.
[12] *Highly Integrated Programmable System-On-Chip*, Philips, Amsterdam, The Netherlands, 2011.
[13] St Nomadik, "St Nomadik multimedia processor," STMicroelectronics, Geneva, Switzerland, Tech. Rep., 2011.
[14] AnandTech, "The qualcomm snapdragon 855 pre-dive: Going into detail on 2019's flagship Android SoC," Future Plc, London, U.K., Tech. Rep., 2018.
[15] *CEVA-XM6*, CEVA, Baar, Switzerland, 2018.
[16] *The Cadence Tensilica Vision DSP*, Cadence Des. Syst., San Jose, CA, USA, 2019.
[17] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O'riordan, and V. Toma, "Always-on vision processing unit for mobile applications," *IEEE Micro*, vol. 35, no. 2, pp. 56–66, Mar. 2015.

[18] Y. Tong, W. Zhang, Y.-C. Ma, Y. Liu, Y. Liang, T. Zhang, and H. Luo, "Compiler–guided parallelism adaption based on application partition for power–gated ILP processor," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 4, pp. 1329–1341, Apr. 2017.
[19] International Technology Roadmap for Semiconductors, "Industry technology roadmap for semiconductors," Tech. Rep., 2010.
[20] M. B. Taylor, "A landscape of the new dark silicon design regime," *IEEE Micro*, vol. 33, no. 5, pp. 8–19, Sep. 2013.
[21] N. S. Kim, D. Chen, J. Xiong, and W.-M.-W. Hwu, "Heterogeneous computing meets near–memory acceleration and high–level synthesis in the post–Moore era," *IEEE Micro*, vol. 37, no. 4, pp. 10–18, Aug. 2017.
[22] J. M. P. Cardoso, J. G. F. Coutinho, and P. C. Diniz, *Embedded Computing for High Performance: Efficient Mapping of Computations Using Customization, Code Transformations, and Compilation*. San Mateo, CA, USA: Morgan Kaufmann, 2017.
[23] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Halide: Decoupling algorithms from schedules for high-performance image processing," *Commun. ACM*, vol. 61, no. 1, pp. 106–115, Dec. 2017.
[24] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools*. Amsterdam, The Netherlands: Elsevier, 2005.
[25] H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Adapting instruction level parallelism for optimizing leakage in VLIW architectures," in *Proc. ACM SIGPLAN Conf. Lang., Compiler, Tool Embedded Syst.* New York, NY, USA: ACM, 2003, pp. 275–283.
[26] Y.-P. You, C. Lee, and J. K. Lee, "Compilers for leakage power reduction," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, no. 1, pp. 147–164, Jan. 2006.
[27] Y.-P. You, C.-W. Huang, and J. K. Lee, "Compilation for compact power-gating controls," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 4, Sep. 2007, Art. no. 51.
[28] M. Wang, Y. Wang, D. Liu, Z. Qin, and Z. Shao, "Compiler-assisted leakage-aware loop scheduling for embedded VLIW DSP processors," *J. Syst. Softw.*, vol. 83, no. 5, pp. 772–785, May 2010.
[29] M. Abdel-Majeed, D. Wong, and M. Annavaram, "Warped gates: Gating aware scheduling and power gating for GPGPUs," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2013, pp. 111–122.
[30] R. Kumar, A. Martínez, and A. González, "Efficient power gating of SIMD accelerators through dynamic selective devectorization in an HW/SW codesigned environment," *ACM Trans. Archit. Code Optim.*, vol. 11, Oct. 2014, Art. no. 25.
[31] H. Aghilinasab, M. Sadrosadati, M. H. Samavatian, and H. Sarbazi-Azad, "Reducing power consumption of GPGPUs through instruction reordering," in *Proc. Int. Symp. Low Power Electron. Design (ISLPED)* New York, NY, USA: ACM, 2016, pp. 356–361.
[32] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori, "Enabling effective module–oblivious power gating for embedded processors," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 157–168.
[33] L. Chen and T. M. Pinkston, "NoRD: Node–router decoupling for effective power-gating of on–chip routers," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2012, pp. 270–281.
[34] D. Atienza, P. Raghavan, J. L. Ayala, G. D. Micheli, F. Catthoor, D. Verkest, and M. López-Vallejo, "Joint hardware–software leakage minimization approach for the register file of VLIW embedded architectures," *Integration*, vol. 41, no. 1, pp. 38–48, 2008.
[35] S. Roy, N. Ranganathan, and S. Katkoori, "State–retentive power gating of register files in multicore processors featuring multithreaded in–order cores," *IEEE Trans. Comput.*, vol. 60, no. 11, pp. 1547–1560, Nov. 2011.
[36] H. Tabkhi and G. Schirner, "Application–guided power gating reducing register file static power," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 12, pp. 2513–2526, Dec. 2014.
[37] J. S. P. Giraldo, A. L. Sartor, L. Carro, S. Wong, and A. C. S. Beck, "Evaluation of energy savings on a VLIW processor through dynamic issue-width adaptation," in *Proc. Int. Symp. Rapid Syst. Prototyping (RSP)*, Oct. 2015, pp. 11–17.
[38] J. S. P. Giraldo, L. Carro, S. Wong, and A. C. S. Beck, "Leveraging compiler support on VLIW processors for efficient power gating," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2016, pp. 502–507.
[39] M. Abdel-Majeed, D. Wong, J. Kuang, and M. Annavaram, "Origami: Folding warps for energy efficient GPUS," in *Proc. Int. Conf. Supercomput. (ICS)* New York, NY, USA: ACM, 2016, Art. no. 41.
[40] K. Dev, S. Reda, I. Paul, W. Huang, and W. Burleson, "Workload–aware power gating design and run–time management for massively parallel GPGPUs," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2016, pp. 242–247.
[41] H. Wang, L.-S. Peh, and S. Malik, "A technology-aware and energy-oriented topology exploration for on-chip networks," in *Proc. Conf. Design, Autom. Test Eur. (DATE)*. Washington, DC, USA: IEEE Computer Society, Apr. 2005, pp. 1238–1243.
[42] N. Magen, A. Kolodny, U. Weiser, and N. Shamir, "Interconnect-power dissipation in a microprocessor," in *Proc. Int. Workshop Syst. Level Interconnect Predict. (SLIP)* New York, NY, USA: ACM, 2004, pp. 7–13.
[43] Z. Liang, W. Zhang, and Y.-C. Ma, "Deadline–constrained clustered scheduling for VLIW architectures using power–gated register files," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 2, pp. 1–26, Jul. 2014.
[44] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim., Feedback-Directed Runtime Optim. (CGO)*. Washington, DC, USA: IEEE Computer Society, 2004, p. 75.
[45] D. P. Williamson and D. B. Shmosys, *The Design Approximation Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 2011.
[46] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools, 2/e*. Reading, MA, USA: Addison-Wesley, 2007.
[47] *Denbench: An EEMBC Benchmark*, EEMBC, Hillsboro, OR, USA, 2018.
[48] *Teaching Resources for IC Design*, Synopsys, Mountain View, CA, USA, 2019.
[49] *RISC-V Cores*, RISC-V Found., 2019.

YUNG-CHENG MA received the B.S. and Ph.D. degrees in computer science and information engineering from National Chiao-Tung University, Hsinchu, Taiwan, in 1994 and 2002, respectively. He was a Hardware Engineer with the Computers and Communication Laboratory, Industry Technology Research Institute, Hsinchu, where he led a group to develop embedded VLIW DSP processors. In 2008, he joined Chang-Gung University, Taoyuan, Taiwan, where he is currently an Assistant Professor with the Department of Computer Science and Information Engineering. His current research interests include computer architecture, parallel and distributed systems, energy efficient systems design, and domain-specific architectures.