Deadline-Constrained Clustered Scheduling for VLIW Architectures using Power-Gated Register Files

ZHIBIN LIANG and WEI ZHANG, Tianjin University
YUNG-CHENG MA, Chang-Gung University

Designing energy-efficient Digital Signal Processor (DSP) cores has become a key concern in embedded systems development. This paper proposes an energy-proportional computing scheme for Very Long Instruction Word (VLIW) architectures. To make the processor power scales with adapted parallelism, we propose incorporating distributed Power-Gated Register Files (PGRF) into VLIW to achieve a PGRF-VLIW architecture. For energy efficiency, we also propose an instruction scheduling algorithm called the Deadline-Constrained Clustered Scheduling (DCCS) algorithm. The algorithm clusters the data dependence graph to reduce data transfer energy and makes optimal use of low-powered local registers for tree-structured data dependence graphs. The results of evaluations conducted using the MiBench and DSPstone benchmark suites substantiate the expected power saving and scaling effects.

Categories and Subject Descriptors: C.0 [Computer Systems Organization]: VLIW Architectures

General Terms: Design, Algorithms, Performance, Low-Power

Additional Key Words and Phrases: instruction scheduling, energy-proportional computing, MTCMOS power-gating, distributed register files architecture

ACM Reference Format:

Zhibin Liang, Wei Zhang, and Yung-Cheng Ma. 2014. Deadline-constrained clustered scheduling for VLIW architectures using power-gated register files. *ACM Trans. Architect. Code Optim.* 11, 2, Article 20 (June 2014), 26 pages.

DOI: <http://dx.doi.org/10.1145/2632218>

1. INTRODUCTION

Digital Signal Processors (DSPs) with Very Long Instruction Word (VLIW) architectures are widely used in embedded systems and multimedia systems-on-chip (SoCs) [Philips 2011; Texas Instruments 2011]. Assisted by the compiler to exploit instruction-level parallelism, a VLIW DSP has multiple execution slots to provide high-performance arithmetic computing. Many semiconductor manufacturers provide VLIW DSP cores for embedded SoCs [Freescale Semiconductor 2008; Texas Instruments 2005]. However, a wide-issue VLIW architecture increases processor power and has a significant impact on battery life, system density, cooling cost, and system reliability. On the other hand, in the era of deep submicron semiconductor fabrication, designers face new challenges such as reducing the exponentially increasing leakage

This work is supported by the High-Speed Intelligent Communication Research Center of Chang-Gung University and the CGU research funding under grant BMRPB88.

Authors' addresses: Z. Liang and W. Zhang, School of Electronic Information Engineering, Tianjin University, Tianjin 300072, People's Republic of China; email: {liangzb; tjuzhangwei}@tju.edu.cn; Y.-C. Ma (corresponding author), Department of Computer Science and Information Engineering, Chang-Gung University, Kwei-Shan, Taoyuan, Taiwan; email: ycma@mail.cgu.edu.tw.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2014 ACM 1544-3566/2014/06-ART20 \$15.00

DOI: <http://dx.doi.org/10.1145/2632218>

power [Industry Technology Roadmap for Semiconductors 2010]. This article investigates energy-efficient VLIW architectures for deep submicron processes using emerging technologies.

Main-stream VLIW DSPs are in some means of distributed register files architecture. Researchers report that the register file power accounts for a large proportion of the total processor power [Balfour et al. 2008; Dally et al. 2008]. The power dissipated in the global register file of an N -issue VLIW processor is on the order of $O(N^3)$ [Rixner et al. 2000]. Major VLIW DSPs employ some forms of distributed register files architectures to reduce the register file power [Freescalse Semiconductor 2008; Lin et al. 2008; Texas Instruments 2005]. Various researchers have also proposed a variety of compiler optimization technologies for distributed register files architectures [Akturan and Jacome 2001; Aleta et al. 2007; Qian et al. 2002a, 2002b; Zalamea et al. 2001].

In recent years, energy-proportional computing [Barroso and Holzle 2007; Cameron 2010] has been proposed as a means of improving energy efficiency. An energy-proportional processor has various speed modes to adapt to applications with diverse performance requirements. Power consumption is reduced while working in a lower speed mode. The philosophy is to run the processor just fast enough to meet an application's need. A Dynamic Voltage Scaling (DVS) [Corporation 2000; Devices 2001; Fleischmann 2001] processor is a typical energy-proportional processor. However, although the DVS scheme successfully reduces the dynamic power, designers are facing new challenges arising from the advancements in semiconductor manufacturing processes.

One of the challenges mentioned earlier is that the leakage power increases exponentially and becomes comparable to the switching power [Industry Technology Roadmap for Semiconductors 2010]. To cope with the increasing leakage power, Multiple Threshold-Voltage CMOS (MTCMOS) power-gating technologies are proposed [Shin et al. 2010]. Various researchers have proposed compiler-assisted power-gating control to reduce the power dissipation of instruction-level parallel processors [Kim et al. 2003; Wang et al. 2010; You et al. 2006, 2007]. Although power reduction on functional units is widely discussed, we focus on reducing the power dissipated in register files by means of power gating.

In this article, we propose a novel approach—energy-proportional computing on VLIW architectures. Utilizing MTCMOS power-gating technologies, the design objective is to make the processor power scale with adapted parallelism. We propose incorporating distributed Power-Gated Register Files (PGRF) with VLIW to achieve a proposed PGRF-VLIW architecture that reduces the power dissipation on functional units as well as register files. The power efficiency of the architecture relies on compiler instruction scheduling to make efficient use of local registers and reduce power-hungry data transfers across execution slots. Our major contribution is our proposed Deadline-Constrained Clustered Scheduling (DCCS) algorithm for energy-aware instruction scheduling. The algorithm is motivated by the observation that a DSP application is formed primarily by equation evaluations and has tree-like data dependence structures. The DCCS algorithm clusters a tree-like data dependence graph to reduce cross-slot data dependencies. Moreover, for a tree-structured data dependence graph, we prove that the DCCS algorithm generates optimal use of local registers. The results of evaluations conducted using the MiBench [Guthaus et al. 2001] and DSP-stone [Zivojnovic et al. 1994] benchmark suites verify that our approach is effective.

This article is organized as follows. Section 2 overviews background information on VLIW processors and power-gating technologies. The main concept underlying the proposed approach is given in Section 3. Sections 4 and 5 are devoted to the DCCS algorithm. Section 6 discusses practical issues on applying the proposed approach. Finally, evaluation results are discussed in Section 7, and conclusions given in Section 8.

2. BACKGROUND

This section overviews the two diverse branches underpinning our research about power gating on VLIW-DSP cores with distributed register files architectures.

2.1. Trends in Energy-Efficient VLIW Architectures Design

Most commercial VLIW DSPs employ distributed register files architectures to reduce power dissipation. The trend comes from the energy complexity of a register file [Rixner et al. 2000]. In a distributed register files architecture, the register set is partitioned into pieces, each of which is connected to a limited number of functional units. The Texas Instruments C6000-series DSP [Texas Instruments 2005] is an eight-issue VLIW processor with two homogeneous clusters. Functional units are organized into two identical clusters, and each cluster has a register file connecting to all functional units in the cluster. The Freescale Starcore DSP [Freescale Semiconductor 2008] is a six-issue VLIW processor with heterogeneous cluster architecture. Functional units are grouped into two clusters: one cluster for two-issue memory operations and the other for four-issue arithmetic operations. The PAC-DSP core provided by the Industry Technology Research Institute (ITRI) [Lin et al. 2008] is also a clustered architecture using a ping-pong register file design—a register file with reconfigurable connection to functional units—for intercluster data exchange with a limited amount of access ports.

The key issue in developing a VLIW architecture is to design the instruction scheduling algorithm for the compiler. Since a distributed register files architecture results in data-transfer limitations across execution slots and may degrade performance, researchers have proposed a variety of instruction scheduling schemes aimed at high-performance execution under the data-transfer limitations. Terechko et al. [2007] established a comprehensive theory of code optimization for clustered VLIW architectures. The theory defines various intercluster communication models and proposes scheduling algorithms for each of them. A DSP core is mainly used to process streamed data with a kernel loop. Various researchers have proposed software pipelining algorithms to improve the performance of clustered VLIW architectures [Akturan and Jacome 2001; Qian et al. 2002a, 2002b; Zalamea et al. 2001]. This article redefines the instruction scheduling problem from an alternative viewpoint: deadline-constrained energy optimization for energy-proportional computing.

In the drive to design compilers for architectures with distributed register files, previous researchers [Cai et al. 2008; Codina et al. 2007] proposed graph clustering schemes for instruction scheduling. A clustered architecture induces additional latency in intercluster data transfer communications. To reduce program execution time, the Data Dependence Graph (DDG) is partitioned into virtual clusters such that the communication cost is reduced without affecting parallelism exploitation. The virtual clusters are formed by repeatedly examining critical paths of the DDG. The virtual clusters are then mapped to physical clusters with an as soon as possible (ASAP) scheduling policy. This article revisits the graph clustering problem from an alternative viewpoint: saving the data transfer energy on a shared register file.

2.2. Compiler-Assisted MTCMOS Power Gating

2.2.1. MTCMOS Circuitry for Leakage Power Reduction. Progress in semiconductor manufacturing has given rise to a new challenge: how to reduce leakage power [Industry Technology Roadmap for Semiconductors 2010]. MTCMOS circuitry [Shin et al. 2010] with compiler-assisted power gating has been proposed to overcome this challenge, realizing that MTCMOS power gating induces overhead and results in constraints for architecture-level design. The first constraint comes from time and energy overhead for transition between active and sleep modes. To prevent the circuit being destroyed by a sudden rush of current, a gradual wake-up scheme has to be applied. To save energy,

Fig. 1. (a) The PGRF-VLIW processor architecture. (b) The power-gated register file.

power gating is only enabled when the idle period exceeds a certain threshold. The second constraint comes from the area overhead. An isolation cell has to be inserted on each net across power domains to protect the circuit. As a result, power domain partitioning has become a key issue in architecture-level design.

2.2.2. Compiler-Directed Power-Gating Control. A power-gated architecture requires compiler support to exploit its energy efficiency. The compiler analyzes the parallelism in the program and inserts power-gating instructions to turn on/off hardware for program execution. Various researchers have been working on this issue. Kim et al. [2003] proposed a global scheduling algorithm to shut down idle hardware without affecting program execution time. The algorithm traverses the control flow graph to estimate the required parallelism: the ratio of the number of operations to the schedule makespan. Hardware exceeding the estimated parallelism is shut down via power gating. You et al. [2006, 2007] proposed an instruction scheduling algorithm to reduce power-gating overhead. Their proposed algorithm, called “sink-and-hoist,” traverses the control flow graph to estimate the lifetime of functional units. Instruction scheduling is then performed to assemble idle periods of functional units and pack multiple power-gating instructions into one. The algorithm not only reduces the amount of power-gating instructions but also improves the energy efficiency. Wang et al. [2010] studied loop scheduling for a VLIW processor with power-gating control. Rotation scheduling over a kernel loop is performed to build an ASAP schedule with minimum parallelism. As a result, power consumption is reduced while the processor performance is retained.

All the works just cited consider power gating on functional units only. However, researchers report that the register file power occupies a larger portion of a processor’s power pie chart [Dally et al. 2008]. Therefore, we propose power-gated architecture and compiler support aimed at reducing the power dissipated in register files.

3. DESIGN OVERVIEW: THE PGRF-VLIW ARCHITECTURE WITH COMPILER SUPPORT

Our objective is to minimize the power dissipated in both functional units and register files. The main theme is energy-proportional parallel computing: adapting processor power by scaling the instruction-level parallelism. Power gating is deployed on both functional units and register files. The compiler has directives for a user to specify program execution deadline over a code fragment. To insert power-gating instructions, the compiler analyzes the parallelism required and optimizes hardware resource usage subject to the execution deadline constraint. The remaining sections of this article are devoted to the joint effort of hardware and software design.

3.1. The PGRF-VLIW architecture

The PGRF-VLIW architecture is shown in Figure 1(a). An instruction packet flows through the fetch and decode units, and operations are dispatched to multiple execution

Fig. 2. (a) Code optimization flow. (b) Example of a data dependence graph (DDG).

slots. Like the traditional architecture, a Shared Register File (SRF) is connected to all execution slots for data transfer across execution slots. Each execution slot has a Local Register File (LRF) that is accessible only by functional units in the execution slot. Power gating is applied to execution slots as well as register files. Each execution slot is an individual power domain. Moreover, as shown in Figure 1(b), each of the local register files and the shared register file is partitioned into banks with individual power gating. Register banks not allocated with operands are shut down during execution.

The architecture is designed to extend the overall power-scaling range of the processor. Each execution slot is an individual power domain, and hence power dissipation on the functional units scales with parallelism. The energy complexity of register files [Rixner et al. 2000] leads to the use of LRFs: an LRF has fewer access ports and hence consumes less power. The architecture relies on the compiler to direct most of the operand accesses to LRFs. Moreover, it is believed that data transfer through SRF increases as parallelism increases. Consequently, we expect power dissipation on both functional units and register files to scale with changing parallelism.

3.2. Compiler Support for the PGRF-VLIW Processor

The architecture relies on the compiler to exploit its energy efficiency. Power reduction is achieved through instruction scheduling and register allocation to allocate operations and operands onto the architecture. The optimization problem is to minimize the processor power subject to the user-specified program execution deadline. Instruction scheduling imposes constraints on register allocation and hence plays a key role in optimizing the power of register files. We focus on the local instruction scheduling that is the theoretical foundation underpinning the success of the architecture.

3.2.1. Overall Compiler Optimization Flow. Figure 2(a) shows the code optimization flow. The compiler front-end translates the high-level language source code into an Intermediate Representation (IR)—a series of machine-level operations in Static Single Assignment (SSA) form. Operations in IR work on pseudo data operands called *temporaries*. A spill stage inserts load/store operations into the IR such that the shared register file is able to store all register operands. Instruction scheduling is performed to assign when (cycle) and where (execution slot) each operation is to be executed. The instruction scheduling sets lifetime and positioning constraints to allocate temporaries. Constrained by the instruction scheduling outcome, the register allocation maps temporaries to registers distributed on LRFs and SRF. Finally, the code-generation stage generates the parallel assembly code. In addition, power-gating instructions are inserted to turn off unused execution slots and register banks during code generation.

Fig. 3. Example of a DDG and two different schedule charts for the example.

In practical use, the proposed scheduling algorithm cooperates with global optimization technologies (such as trace scheduling [Aho et al. 2007]) to deal with real applications. A global optimizer generates segments of straight programs (such as traces) to feed into the proposed instruction scheduler for energy optimization.

3.2.2. Energy-Aware Instruction Scheduling Guidelines. In the instruction scheduling process, the program IR is modeled as a Data Dependence Graph (DDG). Figure 2(b) shows an example. A series of operations I_i s (operating on temporaries t_j s) on the left-hand side is modeled as the DDG on the right-hand side. A node of the DDG represents an operation, and a directed edge indicates the producer-consumer relation of a temporary.

The outcome of the instruction scheduling process is a schedule chart that is viewed as a layout of the DDG. An example schedule chart for the DDG in Figure 3(a) is shown in Figure 3(b). In a schedule chart, a row corresponds to an execution slot, and a column corresponds to a cycle. An operation I_i at position $(Slot_k, \tau)$ indicates that I_i is to be executed by execution slot $Slot_k$ at cycle τ . A cross-slot DDG edge (unbroken lines in Figure 3(b)) indicates that a temporary is transferred across execution slots. This kind of cross-slot dependence implies that the underlying temporary has to be allocated to the power-hungry SRF. Conversely, an intraslot DDG edge (the broken lines in Figure 3(b)) indicates that there is a chance to transfer operands through low-powered LRFs.

The guidelines for deadline-constrained energy-aware scheduling are (1) to minimize the hardware parallelism to meet the deadline constraint and (2) to reduce the number of cross-slot DDG edges to direct more temporary accesses to the LRFs. We use the DDG in Figure 3(a) to illustrate the guidelines. If the deadline constraint is set to four cycles, then Figure 3(c) shows a preferred schedule compared to Figure 3(b). The schedule in Figure 3(c) meets the deadline with only two execution slots and hence consumes less computational energy. Moreover, the schedule in Figure 3(c) has fewer cross-slot DDG edges and directs more accessed temporaries to low-powered LRFs.

3.2.3. Motivating Idea Underlying the Proposed DCCS Algorithm. We propose our DCCS algorithm along the just noted guidelines. The algorithm is motivated from the observation that most DSP applications consist of equation evaluations, and the DDG can be viewed as an inverted tree. The DCCS algorithm partitions a tree-like DDG into clusters: each cluster is a tree, and all operations in a cluster are scheduled in the same execution slot. Such a clustering sets most of the edges in the DDG as intraslot edges. In addition, we propose an algorithm to obtain a sequential schedule aimed at minimizing the register pressure within an execution slot. The proposed sequential scheduling algorithm has been proved to minimize the number of local registers required when the DDG is a tree. The final schedule for all parallel execution slots is established by controlling the clustering granularity with the deadline constraint.

Fig. 4. (a) Example of a tree-structured DDG. (b) Example of a sequential schedule.

Table I. Notations for Sequential Scheduling

Notation	Description
T	An arbitrary tree-structured DDG
$root(T)$	The unique sink node of T
$DQ(T) = (\{T_1, T_2, \dots, T_n\}, T_R)$	Decomposition of T into parallel branches $T_1, T_2, \dots, T_n$, and the branch T_R containing $root(T)$
Q	An arbitrary feasible sequence for a DDG
$SSQ(T')$	The subsequence of Q containing only vertices in subtree T'
$RQ(T')$	The time range over which vertices in subtree T' are spread
$RP(Q)$	Register pressure of a feasible sequence Q
$RP^*(T)$	Register pressure of the optimal sequencing for DDG T
$Q_1 \ll Q_2$	The sequence constructed from appending Q_2 to the tail of Q_1

4. SEQUENTIAL SCHEDULING TO MINIMIZE REGISTER REQUIREMENTS

4.1. Modeling the Sequential Scheduling

We first propose the High Register Pressure First (HRPF) algorithm for sequential scheduling within an execution slot. Figure 4(a) illustrates a tree-structured DDG T with accessing temporary t_i s marked with edges and shows our terminologies. In this example, each operation (vertex) has a one-cycle latency. For a path from vertex I_i to I_j , we say I_i is an **ancestor** of I_j and I_j is a **descendant** of I_i . A vertex having no ancestors in T is called a **source node**, and a vertex having no descendants is called a **sink node**. For a tree T , there are at least one source node and one unique sink node called the **root** of T . Table I gives our notations associated with sequential scheduling. Figure 4(a) also illustrates our decomposition scheme $DQ(T) = (\{T_1, T_2, \dots\}, T_R)$ on a tree T . Suppose that T is not a path graph. (In a path graph, all vertices are in a simple path [Gross and Yellen 2006].) Backtracking from the root, we obtain the first vertex u , with at least two immediate ancestors, as the **split point**. Decomposing from the split point, we obtain a set of parallel branches $\{T_1, T_2, \dots\}$ and the subtree T_R covering the path from u to the root. Each pair of branches has no vertices in common.

A feasible sequential schedule is a time mapping from a topological order of the subgraph. Any topological order of vertices in T is a **feasible sequence** for the scheduling. To construct a sequential schedule, the sequence is mapped to a linear timeline with the first operation assigned to cycle 1. We build ASAP schedules, and each operation is mapped to the earliest available time satisfying dependence constraints. Thus, the operation sequencing uniquely determines the sequential schedule. In the latter discussion, we use the two terms “feasible sequence” and “feasible sequential schedule” interchangeably. Figure 4(b) illustrates a feasible sequential schedule. Our theory is about how to decompose a feasible sequence. Consider a feasible sequence Q for a tree

Fig. 5. Example of HRPF scheduling.

T . Let T' be a subtree of T and Q' be the subsequence of Q consisting of only vertices in T' . The sequence Q' is also a feasible sequence for subtree T' . Notations $SS_Q(T')$ and $R_Q(T')$ denote the subsequence and time range for a subtree T' in Q , respectively. For the example in Figure 4, we have $SS_Q(T_1) = \{I_2, I_1, I_5, I_6, I_8\}$, and $R_Q(T_1) = [1, 6]$.

The scheduling determines the register requirement of a program. The **live range** of a temporary t_i is the time range in which some register has to be occupied by the value of t_i . A schedule defines live ranges for all temporaries. The live range of t_i starts from the time the value is generated by the producer operation and ends at the time the last operation reading t_i is scheduled. In this example, all operations have one-cycle latency to produce the outcome. The **register pressure** of a schedule is the maximum number of simultaneously alive temporaries. With perfect register allocation, the register pressure is the required number of registers. The lower half of Figure 4(b) shows the live ranges, for which we have $RP(Q) = 3$.

We can now define the optimization problem. Given a tree-structured DDG T , we build a topological order Q of T to find a feasible sequence with minimum register pressure. In general, finding an optimal schedule is NP-complete [Garey and Johnson 1979]. Nevertheless, we devise a polynomial-time algorithm that is guaranteed to find an optimal solution when the DDG is a tree.

4.2. The HRPF Sequencing Algorithm

Algorithm 1 presents the sequential scheduling algorithm. The algorithm builds a feasible sequence recursively by decomposing the tree T into subgraphs. The HRPF policy is applied to cascade sequential schedules for each subgraph. The recursion terminates when a path graph is reached. Figure 5 shows an example. By applying the HRPF policy, the temporary t_4 carrying the result of subtree T_1 does not increase the register pressure in the final result.

4.3. Exactness of the HRPF Algorithm

We now prove that our proposed algorithm generates an optimal sequence for any tree T when each operation has a latency of one cycle.

LEMMA 4.1. *Let Q be any feasible sequence for a tree-structured DDG T and T' be a subtree of T . For any time point $\tau \in R_Q(T')$, except for the first cycle of $R_Q(T')$, there exists a temporary t_j defined in T' that is alive at time point τ .*

PROOF. Let I_1 be the first operation of T' appearing in Q and I_n be the root of T' . Vertex I_1 is a source node of T' and I_n is the last operation in $SS_Q(T')$. There exists a path γ from I_1 to I_n in T' and, say, consists of the sequence of vertices $\{I_1, I_2, \dots, I_n\}$. For any time point $\tau \in R_Q(T')$, except for the first cycle of $R_Q(T')$, there exists a vertex

Fig. 6. Pattern associated with the proof of Lemma 4.2.

ALGORITHM 1: HRPF Sequencing**Input:** Tree-structured DDG T .**Output:** A topological order Q of T .

```

1: if  $T$  is a path graph then
2:   Let  $Q$  be the unique topological order of  $T$ ; Return  $Q$ ;
3: end if
4: Backtracking from  $root(T)$  to find the first vertex  $u$  with multiple immediate ancestors;
5: Decompose  $T$  as  $DQ(T) = (\{T_1, T_2, \dots, T_n\}, T_R)$  from the split point  $u$ ;
6: for each branch  $T_j$  do
7:   Construct sequence  $Q_j$  for each branch  $T_j$  using Algorithm HRPF_Sequencing;
8: end for
9: Let  $QS = \{Q_1, Q_2, \dots, Q_n\}$ ; //exclude  $Q_R$  for  $T_R$  that contains  $root(T)$ 
10: while  $QS$  is not empty do
11:   Remove  $Q_j$  with the largest  $RP(Q_j)$  from  $QS$ ;
12:    $Q = Q \ll Q_j$ ;
13: end while
14:  $Q = Q \ll Q_R$ ; Return  $Q$ ;

```

I_i in γ such that τ falls after the scheduled time of I_i and is less than or equal to the schedule time of I_{i+1} . The temporary defined by I_i is alive at time τ . $\square$

LEMMA 4.2. Consider a feasible sequence Q for a tree T with $DQ(T) = (\{T_1, T_2, \dots, T_n\}, T_R)$. Sort parallel branches $\{T_1, T_2, \dots, T_n\}$ in the order that the first operation of T_j appears in Q . Then, $RP(Q) \geq \max_{1 \leq j \leq n} \{RP(SS_Q(T_j)) + (j - 1)\}$.

PROOF. Pick an arbitrary branch T_j and the time point $\tau \in R_Q(T_j)$ with maximum number of alive temporaries in T_j . We assert that each branch T_i with $i < j$ contributes an additional temporary t that is alive at time τ . Consider all possible cases on the intersection of $R_Q(T_i)$ and $R_Q(T_j)$ w.r.t. τ . (Figure 6 depicts all the possible cases.) In the first case (Figure 6(a)), τ falls before the end of T_i . Lemma 4.1 states that there is at least one additional temporary t_a defined by T_i that is alive at time τ . In the second case (Figure 6(b)), τ falls after the end of T_i , the temporary t_b carrying T_i 's result to T_R is alive at time τ . Considering all branches T_i s appearing before T_j , we have $RP(Q) \geq RP(SS_Q(T_j)) + (j - 1)$, and this lemma follows. $\square$

We draw the relation of optimal register pressures between a tree and its subtree.

LEMMA 4.3. Let T be a tree-structured DDG with decomposition $DQ(T) = (\{T'_1, T'_2, \dots, T'_n\}, T_R)$, where the parallel branches are ordered such that $RP^*(T'_1) \geq RP^*(T'_2) \geq \dots \geq RP^*(T'_n)$. Then, $RP^*(T) \geq \max_{1 \leq j \leq n} \{RP^*(T'_j) + (j - 1)\}$.

ALGORITHM 2: Multipass DCCS**Input:** A data dependence graph DDG and a deadline T .**Output:** A schedule chart SC .

- 1: Build schedules $\{SC_1, SC_2, \dots, SC_m\}$ of DDG with $DCCS$ algorithm for each possible parallelism;
- 2: Return the SC_k with minimum parallelism that satisfies the deadline T ;

ALGORITHM 3: Deadline-Constrained Clustered Scheduling (DCCS)**Input:** A DDG and parallelism M .**Output:** A schedule SC .

- 1: Clustering Stage: Establish clusters $\{S_i\}$ and dependence graph SDG over the clusters;
- 2: Partitioning-to-Fit Stage: Refine the clustering $\{S_i\}$ and SDG to meet the critical path deadline CPD ;
- 3: Allocation Stage: Build the schedule SC by allocating nodes of SDG to execution slots;
- 4: Return SC ;

PROOF. Let Q^* be an optimal sequence for T and consider an arbitrary branch T'_j . In the case where the appearance order of T'_j is not less than j , Lemma 4.2 implies that $RP(Q^*) \geq RP(SS_{Q^*}(T'_j)) + (j - 1)$. Consider the case in which the appearance order of T'_j is less than j . There exists a branch T'_i , with $i < j$ (and hence $RP^*(T'_i) \geq RP^*(T'_j)$), such that the appearance order of T'_i is not less than j . Lemma 4.2 implies that $RP(Q^*) \geq RP(SS_{Q^*}(T'_i)) + (j - 1)$. Hence, we have $RP^*(T) \geq RP^*(T'_j) + (j - 1)$ for each branch T'_j , and this lemma follows. $\square$

Finally, we prove that the HRPF sequencing algorithm is exact.

THEOREM 4.4. *Suppose each operation has one-cycle latency. The HRPF algorithm generates an optimal sequence for any tree-structured DDG .*

PROOF. An optimal sequence is obtained for a path graph because there is only one choice. For a tree T with decomposition $DQ(T) = (\{T_1, T_2, \dots, T_n\}, T_R)$, we prove the theorem by induction on the number of decompositions to simplify to path graphs. Suppose that the algorithm obtains optimal sequence Q_j^* for each branch T_j . Sort the parallel branches in nonincreasing order of the optimal register pressure and consider the sequence Q generated by the HRPF algorithm. Note that the sorted order of T_j s is also the appearance order of all branches in Q . In sequence Q , each branch preceding T_j contributes one result-carrying temporary that is alive throughout $R_Q(T_j)$. Hence, we have $RP(Q) = \max_{1 \leq j \leq n} \{RP^*(T_j) + (j - 1)\}$, and Lemma 4.3 states the optimality. $\square$

5. THE DCCS ALGORITHM FOR SCHEDULING PARALLEL OPERATIONS

On the basis of the HRPF sequencing algorithm for a single execution slot, we propose energy-aware instruction scheduling for parallel execution slots. The top-level framework is shown in Algorithm 2, which executes the DCCS algorithm multiple times to select a schedule that meets the deadline with minimum parallelism.

The core of the scheduling is the DCCS algorithm for fixed parallelism. Algorithm 3 gives the framework: a clustering-and-allocation approach. A cluster S_i is a tree-structured subgraph of the DDG to be scheduled in a single execution slot using the HRPF sequencing algorithm. Data dependencies among clusters are represented by a **Subgraph Dependence Graph** (SDG). The first stage, the clustering stage, gives an initial clustering on DDG . Intended to reduce accesses to SRF, this stage tries to

maximize the cluster size. The second stage, the partitioning-to-fit stage, adjusts the clustering granularity by imposing the critical path deadline CPD over SDG. We estimate $CPD = (\text{total execution time of all operations})/M$, where M is the amount of active execution slots or the hardware parallelism. As the parallelism M is increased in the multipass algorithm, CPD is reduced and forces the parallel portions in SDG to increase with finer clustering granularity. Finally, the allocation stage allocates nodes of SDG onto parallel execution slots.

5.1. Clustering Stage

The clustering stage groups operations and builds the SDG. Following the clustering stage, each cluster S_i becomes a tree-structured subgraph in the DDG. Moreover, the clustering stage also identifies clusters that can be executed in parallel to utilize multiple execution slots. Data dependencies among clusters are represented by SDG: a vertex represents a cluster S_i , and an edge (S_i, S_j) indicates that the output temporary of S_i is read by some operation in S_j .

ALGORITHM 4: Clustering

Input: A DDG G .

Output: Clusters $\{S_i\}$ and SDG H

```

1: Initialization: Create a cluster  $S_i$  for each operation  $I_i$  and mark  $S_i$  as open;
2: Assign levels to each operation  $I_i$ ;
3: for each level  $m$  in increasing order do
4:   for each operation  $I_i$  in level  $m$  do
5:     /* Step 1: Decisions to merge ancestors with  $I_i$  */
6:     if all ancestors of  $I_i$  are in open clusters then
7:       Merge  $I_i$  with all ancestor clusters;
8:     else if one of  $I_i$ 's ancestors is in a closed cluster then
9:       /*  $I_i$  is an half-open cluster */
10:      Keep  $I_i$  as an individual cluster;
11:      Close all ancestor clusters of  $I_i$ ;
12:     else if one of  $I_i$ 's ancestors is in a half-open cluster then
13:       Merge  $I_i$  with all ancestor clusters which is half-open;
14:       Close each ancestor cluster of  $I_i$  which is not merged with  $I_i$ ;
15:     end if
16:
17:     /* Step 2: Closing decision */
18:     if  $out\_degree(I_i) = 0$  then
19:       Close the cluster containing  $I_i$ ;
20:     else if  $out\_degree(I_i) = 1$  then
21:       Keep current cluster of  $I_i$  in present state;
22:     else if  $out\_degree(I_i) > 1$  then
23:       Close the cluster containing  $I_i$ ;
24:       Mark all descendant clusters of  $I_i$  as half-open;
25:     end if
26:   end for
27: end for
28: Build SDG  $H$  for dependencies over clusters and return  $H$ ;

```

Algorithm 4 describes the clustering algorithm. At initialization, each operation is in a separate cluster, and the algorithm tries to merge clusters by examining operations on a level-by-level basis. The level of an operation refers to the length of the longest path from a source node. Merging decision on an operation I_i is determined by the states of its ancestor clusters (cf. Lines 5–15). Clusters are classified into open, closed, and half-open states. A cluster is **open** if it can be further merged with other clusters.

Fig. 7. Running example of the clustering stage.

No further merging is performed on a **closed** cluster. A **half-open** cluster begins a new session to search for a tree-structured portion. Only downward merging is allowed on a half-open cluster. An operation I_i with multiple outgoing edges triggers a series of cluster closings (cf. Lines 22–24) by setting its descendants half-open. A cluster is closed if it cannot be merged with its descendants (cf. Lines 9 and 14).

Figure 7 shows a running example of the clustering stage. In the snapshot depicted in Figure 7(a), open clusters are merged to form larger trees. For example, the operation I_{16} is merged with its open ancestor clusters $\{S_a, S_b\}$. In Figure 7(b), a series of closing decisions are triggered by I_{18} , which has more than one outgoing edge. Both I_{19} and I_{20} are marked as being in half-open clusters, and the cluster S_3 is closed at I_{18} . It is possible to execute I_{19} and I_{20} in parallel, and the decision is left to the allocation stage. The clustering stage keeps merging the half-open cluster $\{I_{19}\}$ with its descendants $\{I_{21}, I_{22}\}$ to form a larger cluster, Figure 7(d). The clusters S_1 and S_2 are closed at I_{16} and I_{17} , respectively, after finding they are not merged with their descendants. As a result, the clustering stage establishes sequential execution clusters and cluster sets $(\{S_1, S_2, S_3\}$ and $\{S_4, S_5\})$ that may be executed in parallel.

5.2. The Partitioning-to-Fit Stage

This stage refines the clustering at Stage 1 to meet the deadline constraint. Recall that a cluster is allocated in one execution slot. The clustering reduces the software parallelism, and oversized clusters may cause the execution deadline to be missed. This stage partitions oversized clusters to increase software parallelism according to the CPD determined from the hardware parallelism.

We examine the critical path of SDG to refine the clustering. Each node and each edge in SDG is associated with a weight to indicate the execution time. The weight of an edge is the latency for the producer operation to generate the underlying temporary. The weight of a node S_i , denoted $ET(S_i)$, is the makespan obtained from applying the HRPF algorithm to schedule the cluster S_i . The length of a path P , denoted ET_P , is the total weight for all nodes and edges in P . The quantity ET_P indicates the execution time lower bound to execute operations in P for any feasible schedule of SDG. The

ALGORITHM 5: Partitioning-to-Fit**Input:** A *SDG* and a *CPD*.**Output:** An updated *SDG*.

- 1: Find the critical path *CP* in *SDG*;
- 2: **while** $ET_{CP} > CPD$ and *CP* is not composed of path graphs **do**
- 3: Find S_k : The node with maximum $ET(S_i)$ in *CP* and S_k is not a path graph;
- 4: Decompose S_k : $DQ(S_k) = (\{T_1, T_2, T_3, \dots, T_n\}, T_R)$;
- 5: Update *SDG*: Replace S_k with vertices $\{T_1, T_2, \dots, T_n, T_R\}$ and corresponding dependent edges;
- 6: Recompute the critical path *CP* for *SDG*;
- 7: **end while**
- 8: Return *SDG*;

ALGORITHM 6: Allocation**Input:** An *SDG* and hardware parallelism *M*.**Output:** A schedule chart *SC*.

- 1: Initialize *SC* to be empty;
- 2: Establish a topological order *SS* of *SDG* with high-register-pressure-first policy;
- 3: **for** each node S_j in the order of *SS* **do**
- 4: Establish sequential schedule *IS* for the cluster S_j using *HRPF_Sequencing* algorithm;
- 5: Find the earliest available cycle τ satisfying all dependencies for S_j ;
- 6: **for** $i = 0$ to $M - 1$ **do**
- 7: Compute affinity AF_i : the number of DDG edges connecting operations in $Slot_i$ to operations in S_j ;
- 8: **end for**
- 9: Select $Slot_k$: The slot with maximum affinity that is empty after cycle τ ;
- 10: Update *SC* by scheduling operations in S_j onto $Slot_k$ in the order of *IS* and starting from cycle τ ;
- 11: **end for**
- 12: Return schedule chart *SC*;

critical path of *SDG*, denoted *CP*, is the longest path in *SDG*. The partitioning criteria are set to keep the length of the critical path within the required deadline *CPD*.

Algorithm 5 outlines the partitioning algorithm. The algorithm iteratively refines the *SDG* to meet the deadline *CPD*. At each iteration, the node S_k with highest sequential execution time in *CP* is decomposed to reduce the length of the critical path. The selected node S_k represents a cluster of tree-structured DDGs, and we use the same decomposition scheme presented in Section 4.1. The execution time is reduced by overlapping the execution of parallel branches.

5.3. Allocation Stage

Finally, we complete the scheduling with the allocation stage. This stage allocates nodes in *SDG* to a schedule chart *SC*, where a node is a tree-structured subgraph that is allocated in one execution slot. Allocation guidelines are (1) to build an ASAP schedule that tries to shorten the makespan of the whole *SDG*, (2) to reduce the amount of cross-slot dependencies that results in operand accesses through the high-powered SRF, and (3) to reduce the register pressure resulting from the schedule *SC*.

Algorithm 6 gives an outline of the allocation algorithm. A topological order *SS* over nodes in *SDG* is established by depth-first traversal from sink nodes by treating each edge in reverse direction [Cormen et al. 2009]. To select the next node to append to *SS*, we break the tie using the HRPF policy. Each cluster S_j is then allocated to a row of empty positions in the schedule chart *SC* following the ASAP guideline: a cluster

Fig. 8. Example of partition-to-fit and allocation stages.

starts from the earliest available cycle τ that satisfies all dependence constraints. In cases where there are multiple empty slots at cycle τ , we break the tie using the affinity measure AF_i , which counts DDG edges connecting the cluster to an execution slot. Note that the depth-first-style topological order also helps to reduce cross-slot dependencies. With restriction on hardware parallelism, some tree-structured portion of the SDG will be allocated in a single execution slot.

Figure 8 illustrates how the partition-to-fit and allocation stages work. After the clustering in Figure 7, instruction scheduling is now reduced to schedule the SDG in Figure 8(a). Figures 8(b) and (c) illustrate how the partition-to-fit stage adjusts the clustering granularity using *CPD*. Suppose that the SDG fails to satisfy *CPD*. The cluster with the longest execution time (S_2) is decomposed and parallelized to shorten the makespan. Figure 8(d) gives a snapshot in the allocation stage. The scheduling sequence is $SS = \{S_{2.1}, S_{2.2}, S_{2.3}, S_1, S_3, S_4, S_5\}$. To schedule S_5 , the dependence from S_3 sets the earliest available cycle $\tau = m + 2$. There are three available slots ($Slot_2$, $Slot_3$, and $Slot_4$) at cycle τ . The affinity, contributed by edge (S_3, S_5) , leads S_5 to $Slot_4$.

6. PRACTICAL USE OF THE DCCS ALGORITHM

The DCCS algorithm serves as the core theory for energy optimization and can cooperate with various existing technologies. To implement power-gating instructions, we may deploy Sleep Control Registers (SCR) to store power configurations of all power domains [Roy et al. 2009]. Energy optimization on register files is realized through a compiler platform with two-pass scheduling [Chang et al. 1995]. The DCCS algorithm serves as the prescheduling stage to set allocation constraints for register allocation. After register allocation, power-gating instructions are inserted in the postscheduling stage to ensure correct execution. In cooperation with window scheduling [Muchnick 1997], the DCCS algorithm can be utilized for software pipelining. In window scheduling, the DDG is partitioned into virtual pipeline stages for overlapped execution of loop iterations. The DCCS algorithm is applied to schedule each virtual stage for energy optimization. The required initiation interval for software pipelining now serves as the deadline constraint to the DCCS algorithm.

The DCCS algorithm can also be adapted to various architecture features. For operations with variable latencies, the allocation stage calculates the earliest available time to schedule a cluster considering the operation latencies. The earliest available time of an operation is the time that all source operands are ready, considering the latencies from all dependencies. The HRPF sequential scheduling also takes the operation latencies to build the timed schedule. Most of the VLIW-DSP cores have heterogeneous execution slots. To support heterogeneous slots, we modify the closing rules of the clustering stage such that a cluster of operations fits into some execution slot types. The allocation stage then affirms the feasibility of assigning a cluster to an execution slot.

Special care should be paid for long-latency operations, especially for memory operations. The DCCS algorithm is designed to optimize for short-latency operations,

Fig. 9. Flow of the evaluation process.

typically for operations in integer arithmetic units. In Section 4.3, the assumption that each operation has one-cycle latency is necessary for the HRPF algorithm to obtain an optimal solution. With long-latency operations, one can build counterexamples to Lemma 4.2 and 4.3 and show that a better schedule is obtained by interleaving operations from two different branches. The optimization quality of the DCCS algorithm degrades when there are long-latency operations. Most of DSP processors have only integer units, and real-number arithmetics are realized with fixed-point encoding [Freescall Semiconductor 2008; Analog Devices 2010; Texas Instruments 2005]. Long-latency operations are primarily memory operations executed on dedicated load/store slots with internal ALUs for address calculation. Similar to the adaption for heterogeneous execution slots, we suggest modifying the closing rule of Algorithm 4 such that a cluster is closed immediately whenever a memory operation is included. This makes the memory operation the last operation in the cluster, and the HRPF algorithm still works on a sequence of short-latency integer operations. Moreover, the modified closing rule enables the allocation stage to utilize the delay slots between the underlying memory operation and the operations consuming the memory operand.

The proposed approach can also be applied to clustered hardware architectures [Terechko and Corporaal 2007], with the addition of LRFs in each execution slot and the deployment of power gating. Instruction scheduling to support a clustered architecture has two phases [Trimaran 2007]. The first phase partitions the DDG into subgraphs, where each subgraph is assigned to a hardware cluster. The second phase schedules each subgraph onto parallel execution slots connected to a shared register file in a cluster. The DCCS algorithm serves as the scheduling algorithm for the second phase and can cooperate with various cluster assignment algorithms [Terechko and Corporaal 2007] in the first phase.

7. EVALUATION OF THE DCCS ALGORITHM

This section discusses the evaluation results of our approach conducted on power efficiency. The static power consumed by the functional units in proportion to the parallelism is trivial. Our evaluation focuses on the power dissipated on register files.

We also compare the performance of the DCCS algorithm to Cai's algorithm [Cai et al. 2008]. Whereas our algorithm adjusts the parallelism and clustering granularity by the intended performance, Cai's algorithm clusters the DDG to reduce the communication cost without affecting parallelism exploitation. The comparison indicates how we can balance between parallelism exploitation and power dissipation.

7.1. Evaluation Scheme

Figure 9 shows the flow of the evaluation process. We utilized benchmark programs from the MiBench [Guthaus et al. 2001] and DSPstone [Zivojnovic et al. 1994] suites. The LLVM compiler [Lattner and Adve 2004] was used to generate the DDG and the temporaries set. Code optimization worked on the LLVM IR and loop unrolling performed to exploit parallelism. The DCCS algorithm was implemented to build optimized schedule charts. The schedules were fed to the register allocator using the Chaitin graph coloring algorithm [Chaitin 1982]. The register allocator allocated as many local temporaries as possible on the LRFs. In cases where an LRF did not have sufficient space, local temporaries with the lowest access counts were sacrificed.

Table II. Evaluated Configuration and the Estimated Chip Area Overhead

Configuration code	Bank size in LRF	Bank size in SRF	Overhead ratio
(1,1)-Ideal	1	1	—
(4,4)	4	4	16%
(4,8)	4	8	8%
(8,8)	8	8	7%

Table III. Parameters of the Power Model

Notation	Description
m	The hardware parallelism (amount of active execution slots)
Power parameters obtained from CACTI	
$\hat{e}_{st}$	Static energy per cycle of a bank in SRF
$\hat{e}_{rw}$	Dynamic energy per read/write to/from SRF
e_{st}	Static energy per cycle of a bank in LRF
e_{rw}	Dynamic energy per read/write to/from LRF
Parameters from compiler outcome	
$LB(m)$	Number of active banks in all LRFs (when parallelism is m)
$SB(m)$	Number of active banks in SRF (when parallelism is m)
$SR(m)$	Ratio of total accesses directed to SRF (when parallelism is m)
A	Total number of register operand accesses
$ET(m)$	Execution time (cycle) of a program
$SP(m)$	The speedup of a program
$SA(m)$	Ratio of register accesses directed to SRF
Calculated power dissipation	
$P_{st}(m)$	Static power dissipated on all register files (when IPC is m)
$P_{dyn}(m)$	Dynamic power dissipated on all register files (when IPC is m)
$P_{st,iso}$	Static power dissipated on all isolation cells
$P_{dyn,iso}(m)$	Dynamic power dissipated on all isolation cells (when IPC is m)
$P_{st,SRF}$	Static power of the register file for the SRF-only architecture without power gating
$P_{dyn,SRF}$	Dynamic power of the register file for the SRF-only architecture without power gating

Table II shows the evaluated bank size configurations and the chip area overhead. Configuration code (m, n) signifies an architecture in which a bank of LRFs has m registers and a bank of SRFs has n registers. A special configuration, coded “(1,1)-Ideal,” indicates the ideal power saving effect achievable if VLSI researchers keep reducing the power-gating overhead. This configuration represents an architecture in which each register is power gated individually without any overhead.

7.1.2. Power Model Used in the Evaluation. We computed power dissipation using the compiler outcome and the CACTI power model. Table III shows all the related parameters. CACTI is an SRAM model and provides the (1) static power, (2) dynamic energy per access, and (3) access delay of an SRAM block. The SRAM configuration includes the (1) manufacturing process, (2) number of access ports, (3) number of storage cells, and (4) width per access (equivalent to register width). A bank of PGRF is modeled as an SRAM block, and power parameters e_{rw} , e_{st} , $\hat{e}_{rw}$, and $\hat{e}_{st}$ are obtained. For our proposed architecture, the power formulas with the isolation cells overhead considered are:

$$P_{st}(m) = LB(m) * e_{st} + SB(m) * \hat{e}_{st} + P_{st,iso},$$

$$P_{dyn}(m) = \frac{1}{T} * A * ((1 - SR(m)) * e_{rw} + SR(m) * \hat{e}_{rw}) + P_{dyn,iso}(m),$$

Fig. 11. Speedup achieved by the DCCS algorithm.

Fig. 12. Speedup achieved by Cai's algorithm.

Fig. 13. SRF access ratio for the DCCS algorithm.

where $P_{st,iso}$ is approximated using the static power of a 1-port SRAM block with an equal number of storage cells and $P_{dyn,iso}(m)$ is approximated using a 1-port single-register SRAM block as the energy-per-transfer of an isolation cell. We also computed the power dissipation of the SRF-only architecture to compare with the traditional design. Detailed formulations of $P_{st,SRF}$ and $P_{dyn,SRF}$ are omitted for space constraints.

Note that the evaluation overestimates the energy overhead on isolation cells. Energy from decoding logic, which is not consumed by isolation cells, is calculated in our approximation scheme. Even with the overestimated overhead, our result still shows significant power savings from the power-gated architecture. Experiment reports on validating the power model can be found on our website [DCCS 2013].

7.2. Code Optimization Effect from the Compiler

To understand the effect the compiler has on power efficiency, we investigated how register file usage scales with parallelism and cross-referenced it with the speedup.

7.2.1. The Speedup. Figure 11 shows the speedup scaling with parallelism m . The benchmark programs are classified into three classes: (1) the high-parallelism class (Figure 11(a)), which has almost linear speedup up to the maximum parallelism; (2) the middle-parallelism class (Figure 11(b)), which gets saturated above $m = 7$; and (3) the low-parallelism class for remaining benchmark programs. Figure 12 shows the speedup achieved by Cai's algorithm. Both algorithms achieve comparable speedup on most of the benchmark programs. Cai's algorithm achieves a better speedup for a set of benchmark programs in the low-parallelism class.

7.2.2. SRF Access Ratio. Figure 13 shows, for the DCCS algorithm, the ratio of total register accesses sent to the SRF. The DCCS algorithm directs more than 40% of register accesses to low-powered LRFs in order to conserve power. High- and

Fig. 14. SRF access ratio for Cai's algorithm.

Fig. 15. Number of active SRF registers resulting from the DCCS algorithm.

middle-parallelism benchmark programs can be divided into two sets: the first set has $SA(m)$ and scales up with parallelism, and the second set has low and saturated $SA(m)$. The benchmark program “13_fir2dim” exemplifies the first set, and the behavior matched our expectations: the dynamic power scaled up with increased parallelism. The second set of benchmark programs has a different behavior: the ratio to access SRF is low and virtually fixed. The benchmark “03_susan_smoothing” exemplifies this set. Note that each benchmark program in the high- and middle-parallelism classes obtains speedup increments with increased parallelism. The saturated $SA(m)$ indicates that the DCCS algorithm has a good clustering effect to parallelize the application without inducing much cross-slot dependencies. For the second set, we pay only additional energy cost on functional units to gain better performance.

Figure 14 shows the SRF access ratio for Cai's algorithm. Compared to Cai's algorithm, the DCCS algorithm lowers the SRF access ratio by at least 5%, except for “23_pbmsrch,” where both algorithms achieve very low SRF access ratios. For 18 out of the 28 benchmark programs, additional saving by the DCCS algorithm is more than 15%. Cross-referenced with the speedup, the DCCS algorithm provides a better balance between parallelism exploitation and reduction of cross-slot dependencies.

7.2.3. Register File Usage. Figure 15 illustrates how the number of active SRF registers scales with parallelism. The result shows the need for power-gating of the register files: benchmark programs have diverse behaviors on register requirements. Although some benchmark programs (such as “11_convolution”) use all 32 registers, other benchmark programs (such as “22_sha”) use only a small portion of SRF. Power gating on register files is required in order for a single DSP core to keep good energy efficiency over diverse applications. The second observation is on how the static power will scale with parallelism. Similar to the observation of Figure 13, high- and middle-parallelism benchmarks are further classified into two sets: the set with SRF usage that scales up with parallelism and the set with saturated scaling curves. The classification indicates whether the static power of SRF will scale with increased parallelism or not. Compared to Figure 13, Figure 15 has fewer saturated curves. Scaling of static power is more sensitive than that of dynamic power. Figure 16 shows the results for Cai's algorithm. Compared to Cai's algorithm, the DCCS algorithm reduces the register pressure by at least 20% for 19 out of the 28 benchmark programs. Furthermore, the DCCS algorithm is more effective on the high-parallelism class.

Fig. 16. Number of active SRF registers resulting from Cai's algorithm.

Fig. 17. Scaling of LRF pressure for the DCCS algorithm.

Fig. 18. Scaling of LRF pressure for Cai's algorithm.

Figure 17 shows the scaling of LRF pressure. The **LRF pressure** refers to the maximum number of active LRF registers among all the activated execution slots. As expected, the required number of LRF registers per execution slot scales down as parallelism increases. With high-parallelism execution, an execution slot needs only four to eight local registers. Figure 18 shows the results for Cai's algorithm. For most of the benchmark programs, the DCCS algorithm has comparable LRF pressure. The curves for SRF access and usage imply that the DCCS algorithm allocates more temporaries on LRFs. The comparable LRF pressure comes from the effect of HRPF sequencing to reduce the register pressure.

7.3. Power Efficiency of Register Files

We also evaluated the power scaling functions on static and dynamic power

$$PSL_{st}(m) = \frac{P_{st}(m)}{P_{SRF,st}}, PSL_{dyn}(m) = \frac{P_{dyn}(m)}{P_{SRF,dyn}},$$

to justify the expected effects. The power-scaling functions present the power dissipation normalized to the traditional SRF-only architecture. We evaluated the configurations in Table II by considering overhead coming from isolation cells. The power scaling resulting from Cai's algorithm is also presented to compare the clustering effect.

Figures 19 through 21 present the static power scaling $PSL_{st}(m)$, classified into high-parallelism, mid-parallelism, and low-parallelism classes. The results show that our approach saves more than 40% of the static power compared to the SRF-only architecture. The results also suggest finer grained power-domain partitioning on the register files: the most power efficient configuration is (4, 4). The normalized static power of a benchmark is either saturated under 0.2 or scales up with parallelism. For

Fig. 19. Scaling of static power for high-parallelism benchmarks.

Fig. 20. Scaling of static power for middle-parallelism benchmarks.

most of the high- and middle-parallelism benchmarks, the overall trend is for the static power to scale up with increased parallelism. However, this trend is not monotonic: there are several local minimums and local maximums in each power-scaling curve. The imperfect scaling implies that the compiler needs a precise power model considering both functional units and register files to discover proper parallelism.

In Figures 19 through 21, we also compared the static power dissipation to that of Cai's algorithm on the configuration (4, 4) (the curve titled "(4, 4)_Cai"), which also represents the best power saving for Cai's algorithm. For 7 out of the 28 benchmark programs, both the DCCS and Cai's algorithm had PSL values that were saturated under 0.2. The most obvious advantage of the DCCS algorithm appears with the benchmark programs "01_fft_float," "02_fht," "03_susan_smoothing," and "10_encrypt." For these benchmark programs, the DCCS algorithm causes the PSL value to become saturated under 0.2, whereas Cai's algorithm results in high and scaled-up power dissipation.

Fig. 21. Scaling of static power for low-parallelism benchmarks.

Fig. 22. Scaling of dynamic power for high-parallelism benchmarks.

For the remaining benchmark programs, the DCCS algorithm has an apparently lower PSL curve and is approximately 5–20% lower than Cai’s algorithm.

Figures 22 through 24 present the scaling curves of the normalized dynamic power. The overall trend is for the normalized dynamic power to either saturate under 0.6 or scale up with increased parallelism. The only exception occurs for “28_startup,” which has saturated dynamic power over 0.8. The results show that dynamic power is not sensitive to power-domain (bank) partitioning. From the perspective of conservation of dynamic power, the design benefits from LRFs with reduced access ports. Compared to Cai’s algorithm, the DCCS algorithm consumes less dynamic power. For 18 out of the 28 programs, the DCCS algorithm saves at least 10% of the dynamic power.

Fig. 23. Scaling of dynamic power for middle-parallelism benchmarks.

Fig. 24. Scaling of dynamic power for low-parallelism benchmarks.

8. CONCLUSION

This article investigated energy-proportional computing on power-gated VLIW architectures. Although previous researchers focused on power gating the functional units, our contribution saves the power dissipated on register files. The success relies on the DCCS algorithm for instruction scheduling to reduce the register pressure and cross-slot data transfer. The results of evaluations conducted confirm the expected effects as follows: (1) Compared to the traditional SRF-only architecture without power gating, the PGRF-VLIW architecture saves approximately 20% to 50% of register file power even with maximum parallelism. (2) As an overall trend, the register file power scales up with increased parallelism. The scaling range occupies approximately 20% to 35% of the dynamic power and 35% to 65% of the static power.

The most important finding is that the power dissipated on register files can be saved and scaled through instruction scheduling with parallelism adaption. The compiler outcome shows the need to apply power gating on register files: application programs have diverse storage and parallelism requirements. Although this work establishes the architecture with theory on local code optimization, future work will enhance the compiler for practical use. Loop and global code optimizations, such as software pipelining, are planned for the next stage in our drive toward energy-proportional computing.

REFERENCES

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jefferey D. Ullman. 2007. *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison Wesley.
- Cagdas Akturan and Margarida F. Jacome. 2001. CALiBeR: A software pipelining algorithm for clustered embedded VLIW processors. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'01)*. IEEE Press, Piscataway, NJ, 112–118.
- Alex Aleta, Josep M. Codina, Antonio Gonzalez, and David Kaeli. 2007. Heterogeneous clustered VLIW microarchitectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*. IEEE Computer Society, Washington, DC, 354–366. DOI: <http://dx.doi.org/10.1109/CGO.2007.15>
- Analog Devices. 2010. Getting started with Blackfin processors. (2010).
- J. Balfour, W. J. Dally, D. Black-Schaffer, V. Parikh, and JongSoo Park. 2008. An energy-efficient processor architecture for embedded systems. *Computer Architecture Letters* 7, 1 (Jan.-June 2008), 29–32. DOI: <http://dx.doi.org/10.1109/L-CA.2008.1>
- L. A. Barroso and U. Holzle. 2007. The case for energy-proportional computing. *Computer* 40, 12 (Dec. 2007), 33–37. DOI: <http://dx.doi.org/10.1109/MC.2007.443>
- Qiong Cai, J. M. Codina, J. Gonzalez, and A. Gonzalez. 2008. A software-hardware hybrid steering mechanism for clustered microarchitectures. In 1–12. DOI: <http://dx.doi.org/10.1109/IPDPS.2008.4536229>
- Kirk W. Cameron. 2010. The challenges of energy-proportional computing. *Computer* 43, 5 (May 2010), 82–83. DOI: <http://dx.doi.org/10.1109/MC.2010.145>
- G. J. Chaitin. 1982. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN'82)*. ACM, New York, 98–105. DOI: <http://dx.doi.org/10.1145/800230.806984>
- P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. -M. W. Hwu. 1995. The importance of prepass code scheduling for superscalar and superpipelined processors. *IEEE Transactions on Computers* 44, 3 (1995), 353–370. DOI: <http://dx.doi.org/10.1109/12.372029>
- Josep M. Codina, Jesus Sanchez, and Antonio Gonzalez. 2007. Virtual cluster scheduling through the scheduling graph. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*. IEEE Computer Society, Washington, DC, 89–101. DOI: <http://dx.doi.org/10.1109/CGO.2007.39>
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*, 3rd ed. MIT Press.
- Intel Corporation. 2000. The Intel Xscale microarchitecture.
- W. J. Dally, J. Balfour, D. Black-Schaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield. 2008. Efficient embedded computing. *Computer* 41, 7 (July 2008), 27–32. DOI: <http://dx.doi.org/10.1109/MC.2008.224>
- DCCS. 2013. Complementary material for DCCS algorithm. Available at <http://163.25.101.87/wiki/doku.php?id=research:pgvliw:dccs>.
- Advanced Micro Devices. 2001. Mobile AMD Athlon 4 processor model 6 CPGA data sheet.
- M. Fleischmann. 2001. Longrun power management: dynamic power management for Crusoe processors. *AMD Technical Report 24319*.
- Freescale Semiconductor. 2008. Tuning C code for StarCore-based digital signal processors. (2008).
- Michael R. Garey and David S. Johnson. 1979. *Computer and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman.
- Jonathan L. Gross and Jay Yellen. 2006. *Graph Theory and Its Applications*. Chapman and Hall/CRC.
- M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 2001 IEEE*

- International Workshop on Workload Characterization (WWC-4)*. 3–14. DOI: <http://dx.doi.org/10.1109/WWC.2001.990739>
- Industry Technology Roadmap for Semiconductors. 2010. ITRS Report. Available at <http://www.itrs.net/>.
- H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. 2003. Adapting instruction level parallelism for optimizing leakage in VLIW architectures. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tools for Embedded Systems (LCTES'03)*. ACM, New York, 275–283. DOI: <http://dx.doi.org/10.1145/780732.780770>
- Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO'04)*. IEEE Computer Society, Washington, DC, 75.
- Yung-Chia Lin, Chia Han Lu, Chung-Ju Wu, Chung-Lin Tang, Yi-Ping You, Ya-Chaio Moo, and Jenq-Kuen Lee. 2008. Effective code generation for distributed and ping-pong register files: A case study on PAC VLIW DSP cores. *Journal of Signal Processing Systems* 51 (June 2008), 269–288. Issue 3. DOI: <http://dx.doi.org/10.1007/s11265-007-0059-4>
- Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufman.
- Philips. 2011. Philips nexperia highly integrated programmable system-on-chip. Available at <http://www.semiconductors.philips.com/products/nexperia>.
- Yi Qian, Steve Carr, and Philip Sweany. 2002a. Loop fusion for clustered VLIW architectures. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems (LCTES/SCOPES'02)*. ACM, New York, 112–119. DOI: <http://dx.doi.org/10.1145/513829.513850>
- Yi Qian, Steve Carr, and Philip H. Sweany. 2002b. Optimizing loop performance for clustered VLIW Architectures. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*. IEEE Computer Society, Washington, DC, 271–280.
- S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens. 2000. Register organization for media processing. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA-6)*. 375–386. DOI: <http://dx.doi.org/10.1109/HPCA.2000.824366>
- S. Roy, N. Ranganathan, and S. Katkooi. 2009. A framework for power-gating functional units in embedded microprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 11 (2009), 1640–1649. DOI: <http://dx.doi.org/10.1109/TVLSI.2008.2005774>
- Youngsoo Shin, Jun Seomun, Kyu-Myung Choi, and Takayasu Sakurai. 2010. Power gating: Circuits, design methodologies, and best practice for standard-cell VLSI designs. *ACM Transactions on Design Automation of Electronic Systems* 15, 4, Article 28, 37 pages. DOI: <http://dx.doi.org/10.1145/1835420.1835421>
- A. S. Terechko and H. Corporaal. 2007. Inter-cluster communication in VLIW architectures. *ACM Transactions on Architecture Code Optimization* 4, 2 (2007), 11. DOI: <http://dx.doi.org/10.1145/1250727.1250731>
- Texas Instruments. 2005. TMS320C6455 Fixed-point Digital Signal Processor. Available at <http://www.ti.com/product/tms320c6455>.
- Texas Instruments. 2011. OMAP 5 Mobile Applications Platform. Available at <http://www.ti.com/pdfs/wtbu/SWCT010.pdf>.
- Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. 2008. CACTI 5.1. *HP Laboratories Technical Report HPL-2008-20*.
- Trimaran. 2007. Trimaran 4.0 Manual. Available at <http://www.trimaran.org/documentation.shtml>.
- Meng Wang, Yi Wang, Duo Liu, Zhiwei Qin, and Zili Shao. 2010. Compiler-assisted leakage-aware loop scheduling for embedded VLIW DSP processors. *Journal of Systems and Software* 83, 5 (2010), 772–785. DOI: <http://dx.doi.org/DOI: 10.1016/j.jss.2009.11.727>
- Hyo-Sig Won, Kyo-Sun Kim, Kwang-Ok Jeong, Ki-Tae Park, Kyu-Myung Choi, and Jeong-Taek Kong. 2003. An MTCMOS design methodology and its application to mobile computing. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design (ISLPED'03)*. 110–115. DOI: <http://dx.doi.org/10.1109/LPE.2003.1231845>
- Yi-Ping You, Chung-Wen Huang, and Jenq Kuen Lee. 2007. Compilation for compact power-gating controls. *ACM Transactions on Design Automation of Electronic Systems* 12, 4 (2007), 51. DOI: <http://dx.doi.org/10.1145/1278349.1278364>
- Yi-Ping You, Chingren Lee, and Jenq Kuen Lee. 2006. Compilers for leakage power reduction. *ACM Transactions on Design Automation of Electronic Systems* 11, 1 (2006), 147–164. DOI: <http://dx.doi.org/10.1145/1124713.1124723>

- Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. 2001. Modulo scheduling with integrated register spilling for clustered VLIW architectures. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture(MICRO'34)*. IEEE Computer Society, Washington, DC, 160–169. <http://dl.acm.org/citation.cfm?id=563998.564020>.
- Vojin Zivojnovic, J. Martinez Velarde, Christian Schlager, and Heinrich Meyr. 1994. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of International Conference on Signal Processing Applications and Technology*, Vol. 94, 715–720.

Received May 2013; revised November 2013; accepted January 2014