Energy-Aware Code Motion for GPU Shader Processors

YI-PING YOU and SHEN-HONG WANG, National Chiao Tung University

Graphics processing units (GPUs) are now being widely adopted in system-on-a-chip designs, and they are often used in embedded systems for manipulating computer graphics or even for general-purpose computation. Energy management is of concern to both hardware and software designers. In this article, we present an *energy-aware code-motion* framework for a compiler to generate concentrated accesses to input and output (I/O) buffers inside a GPU. Our solution attempts to gather the I/O buffer accesses into clusters, thereby extending the time period during which the I/O buffers are clock or power gated. We performed experiments in which the energy consumption was simulated by incorporating our compiler-analysis and code-motion framework into an in-house compiler tool. The experimental results demonstrated that our mechanisms were effective in reducing the energy consumption of the shader processor by an average of 13.1% and decreasing the energy-delay product by 2.2%.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms: Algorithms, Experimentation, Languages

Additional Key Words and Phrases: Compilers for low power, shader processors, energy management

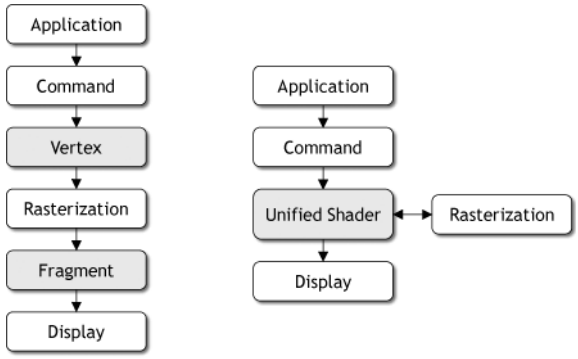
ACM Reference Format:
You, Y.-P. and Wang, S.-H. 2013. Energy-aware code motion for GPU shader processors. ACM Trans. Embedd. Comput. Syst. 13, 3, Article 49 (December 2013), 24 pages.
DOI: http://dx.doi.org/10.1145/2539036.2539045

1. INTRODUCTION

The traditional fixed-function pipeline in 3D graphics rendering has evolved into a programmable pipeline which enables customizations of vertex and pixel/fragment processing and allows developers to write their own shader programs for various types of special effects. This programmable pipeline was derived by introducing vertex and pixel/fragment shader processors with shader models, as shown in Figure 1, to formulate 3D graphics. A *unified shader model*, in which both vertex and pixel/fragment shader programs run on the same processors, as shown in Figure 1(b), allows for more flexible use of the graphics hardware and takes over the task of load balancing from graphics programmers. The model has been widely used in modern GPUs (graphics processing units) along with graphics libraries, such as OpenGL [Khronos Group 2011] or Direct3D [Microsoft 2008], to drive the underlying graphics hardware. OpenGL and Direct3D also provide high-level shader languages for programming customized shading effects. OpenGL ES 2.0 [Khronos Group 2010], which is adapted to embedded platforms, was recently publicly released and is supported by various mobile devices,

This study was partially supported by the National Science Council of Taiwan under grants NSC-97-2218-E-009-043-MY3 and NSC-100-2218-E-009-011-MY3 and by the Institute for Information Industry.
Authors' addresses: Y.-P. You and S.-H. Wang, Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan; corresponding author's email: ypyou@cs.nctu.edu.tw.

DOI: http://dx.doi.org/10.1145/2539036.2539045



(a) Fixed shader model. (b) Unified shader model.

Fig. 1. Programmable graphics pipelines.

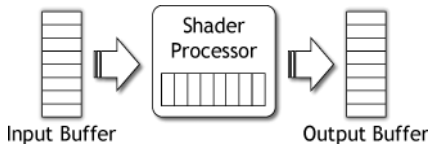


Fig. 2. A shader processor with input/output buffers.

such as the iPhone, iPad [Apple 2011], Android platform [Google 2011], and Palm webOS [Palm 2011]. However, the battery life is often crucially important in embedded systems, and therefore energy consumption must be considered in either hardware or software designs, or both.

A shader processor works similarly to stream processing. As shown in Figure 2, the shader processor reads input data (e.g., uniforms, attributes, and varyings) from an input buffer, performs calculations in general-purpose or temporary registers (according to the shader program), and then writes the results into an output buffer. The cycle repeats until all input data have been processed. The aim of shader processing is to keep the shader processors busy in order to achieve high performance. However, for most of the time during shader processing, the input and output buffers (I/O buffers) stay inactive but powered-up—they are usually accessed only at the beginning and end of the process, respectively—and hence become a major source of leakage energy dissipation in GPUs. Therefore, in this article, we focus on reducing the energy consumption of the I/O buffers via clock or power gating. A naïve method of saving energy is to promote all input data from the input buffer to the temporary registers at the beginning of the program execution so that the input buffer can be turned off for the rest of the program execution, while demoting the results from the temporary registers to the output buffer immediately before the program terminates in order to keep the output buffer off for as long as possible. Unfortunately, both approaches have the side effect of a high register pressure on the shader processors and consequently can impair the speed and size of the compiled code, or even prevent program execution—some shader-processor designs inhibit memory accesses and therefore registers are not allowed to be spilled to memory. This side effect may also counteract the benefit of I/O buffer management in terms of reducing the energy consumption.

In this article, we propose an *energy-aware code-motion* framework for a compiler to address the issue of high register pressures due to register promotion and demotion. Rather than grouping all the I/O buffer accesses around the beginning and end

of the program execution, we gather these accesses into multiple clusters in order to avoid register spillage as well as to reduce the amount of clock- or power-gating controls. We have performed simulation experiments of the energy consumption in our framework using an in-house compiler. The experimental results demonstrate that our mechanisms are effective in markedly reducing the energy consumption relative to naïve methods, with only minor effects on performance. Our energy-aware code-motion framework reduced the energy consumption of the shader processor by an average of 13.1% compared with the no-clock-gating method. The energy-delay product was 2.2% lower for the proposed framework than for the no-clock-gating method, which shows that the proposed framework is effective in reducing the energy consumption without sacrificing too much performance.

The remainder of this article is organized as follows: Section 2 discusses related work on low-power GPU designs. Section 3 describes a machine architecture for the target platform. Section 4 presents the energy-aware code-motion framework. Section 5 describes the experimental results of our study. Finally, conclusions about our work are drawn in Section 6.

2. RELATED WORK

Energy- and power-aware designs for GPUs have received little attention in the literature, especially from the perspective of software techniques. Mochocki et al. proposed an online, signature-based estimation technique for predicting 3D graphics workloads and used it as a basis for dynamic voltage and frequency scaling of the GPU [2006]. Silpa et al. introduced a code transformation scheme that partitions vertex shaders into position-variant and position-invariant shaders in order to reduce the power consumption associated with geometry processing [2009]. Hong and Kim developed an analytical model for predicting both the performance and power consumption of GPU applications in order to manage the number of active cores in GPU architectures [2010]. Their work revealed the energy efficiencies of memory-bandwidth-limited applications. Wang et al. presented a hardware-based, predictive shutdown mechanism for GPU shader processors that reduces the power consumed by components of a system [2009]. In contrast to the these previous approaches, our framework uses software techniques to reschedule instructions and insert power- or clock-gating operations at compile time.

Cooper et al. proposed a compilation approach for transforming several costly calculations (e.g., array-addressing expressions used in loops) into less-expensive ones so as to reduce the number of execution cycles and thus the energy consumption [2001]. Mahjur et al. proposed several instruction scheduling methods to eliminate unnecessary instructions whose results are not used by other instructions in order to reduce the code size, execution time, and energy consumption [2008]. These approaches are performance-oriented optimizations, with any reduction in energy consumption merely representing a by-product. Recent studies have attempted to reduce the leakage power consumption using integrated architecture and compiler power-gating mechanisms [You et al. 2002, 2005, 2006, 2007; Rele et al. 2002; Dropsho et al. 2002; Yang et al. 2002; Zhang et al. 2003]. These approaches involve compilers inserting instructions into programs to shut down and wake up components whenever appropriate, based on a dataflow or profiling analysis. Although the results obtained in these studies indicated that the described approaches were effective, none of them considered transforming a program code by instruction scheduling into one in which the utilization of the system components is concentrated so as to avoid frequent clock- or power-gating controls. Roy et al. explored several existing compiler optimizations for enhancing opportunities to deactivate functional units [2009], but those optimizations were tailored for performance enhancement rather than for reducing the energy consumption. In contrast, our proposed energy-aware code-motion framework for a compiler solution directly

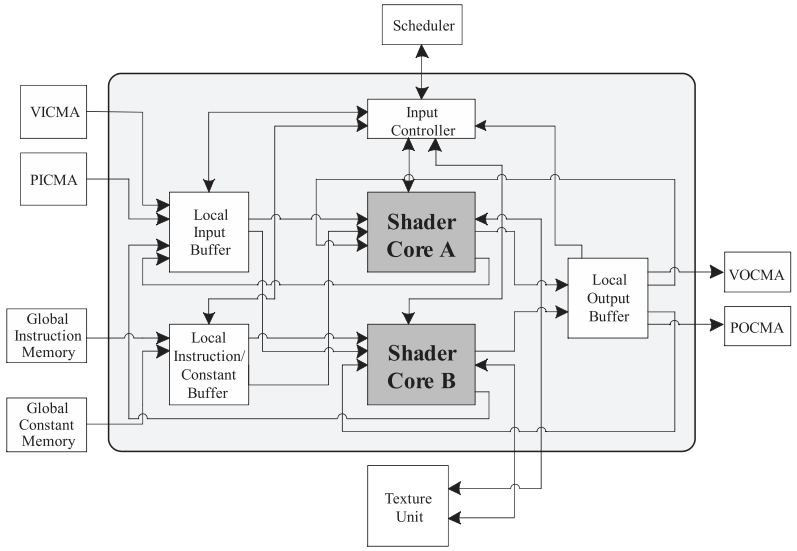


Fig. 3. Block diagram of the Cyclone shader architecture.

addresses the energy consumption of the I/O buffers of GPUs by aggressively gathering I/O buffer accesses and using clock or power gating to turn off the I/O buffers when they are idle.

3. UNIFIED SHADER ARCHITECTURE

The Cyclone GPU[1], the target architecture in this study, is a processor comprising multiple cores specifically designed to deliver the performance required for state-of-the-art mobile graphics system [Ko et al. 2011]. It runs at a frequency of 200 MHz with a 10kB cell-based SRAM and fully meets the OpenGL-ES 2.0 specification. The shader cluster in the Cyclone GPU differs from early shader architectures that use different instruction sets for vertex and pixel shaders. The shader cluster in the Cyclone GPU is a unified architecture that is consistent with the vertex and pixel shaders of the ESSL specification [Khronos Group 2009].

The Cyclone GPU also uses a central task scheduler to manage all computation resources in order to efficiently achieve scalability. Figure 3 shows the block diagram of the shader cluster. There are two shader cores inside which originated from Chien et al.'s designs [2008], and three local buffers shared by the shader cores to reduce the amount of internal memory required. The input controller controls the entire shader cluster by receiving requests from the task scheduler and sending commands to each module. These two shader cores have a unified structure such that either vertex or pixel shader code can be processed.

The input controller receives requests from the task scheduler and sends required control signals to each module. It first informs the input buffer and instruction/constant buffer to fetch required attributes/varyings, constant/uniform, and shader code. When these two buffers finish data fetching, the input controller sends out a start signal to initiate the shader core. After all of the data are written out, the input controller returns a finish signal to the task scheduler and waits for the next request. The input buffer is used to store input data of the shader core. Because the shader cores do not

[1]The released Cyclone GPU open-source package includes RTL modeling, drivers, an ESSL compiler, and a simulator. It can be obtained by contacting I-Ting Lin <itlin@nmi.iii.org.tw>.

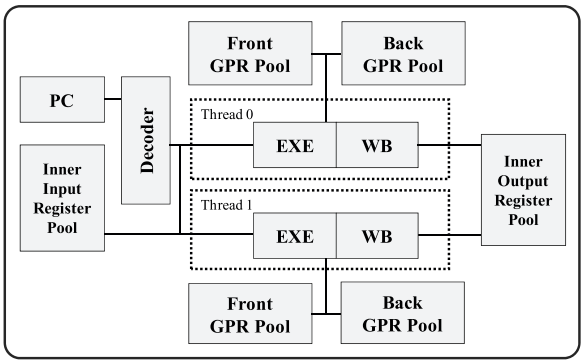


Fig. 4. Block diagram of the Cyclone shader core.

have load/store instructions, all of the required data must be fetched to local (temporary) registers to speed up the computation. The output buffer is used to store the computation results of the shader core. When a shader program is finished, the output buffer writes out the results to an external buffer.

Figure 4 shows the block diagram of the Cyclone shader core. Each Cyclone shader core is an RISC with a five-stage pipeline VLIW architecture. The shader core is capable of processing two 32-bit instruction words in parallel, assuming neither data nor structure dependency between these two words. Concerning computation efficiency and the fact that there is no data dependency and synchronization across shader cores, load/store instructions and memory access stage are not considered in the architecture design. There are eight read-only input buffer registers, eight readable/writable output buffer registers, and eight readable/writable temporary registers. The power requirements of the I/O buffers represented 9.1% and 15.3% of the total, respectively. The detailed hardware specification is discussed in Section 5.

Recently, Chang et al. proposed a variant of the Cyclone GPU that contains eight shader cores to deliver energy-efficient and high-performance graphics rendering for mobile multimedia applications [2011].

4. ENERGY-AWARE CODE-MOTION FRAMEWORK

A naïve way to address the problem of applying clock or power gating to I/O buffers for energy saving is to relocate all input data from the input buffer to the temporary registers at the early stage of the program execution and hold all output results in the temporary registers until the end of the execution so as to keep the I/O buffers inactive during the process. However, this type of approach extends the live range of both input and output variables and thus increases the register pressure of the temporary registers, which may decrease the overall performance in terms of the speed, code size, and energy consumption due to register spillage.

For this reason, we present a compiler framework for code motion of I/O buffer accesses that also considers register pressure in order to reduce the energy consumed in the I/O buffers. The primary concept of this framework is to gather I/O buffer accesses into clusters on the premise that the register pressure does not exceed the maximum operating pressure, that is, all variables can be allocated to physical registers without causing a spill, so that the I/O buffers remain inactive for long periods of time. More specifically, the framework clusters the I/O buffer accesses by moving input buffer accesses forward and by postponing output buffer accesses. Figure 5 presents the compiler flow of the proposed energy-aware code-motion framework. In the framework, step I inserts data-transfer operations (DTOs) in order to resolve data-dependency

Energy-Aware Code-Motion Framework

Input: A shader program.
Output: The shader program with clustered I/O buffer access.

I. Perform *Insertion of Data-Transfer Operations*.
II. Perform *Local Code Motion*.
III. Perform *Global Code Motion*.

Fig. 5. The energy-aware code-motion framework.

issues, step II attempts to cluster I/O buffer accesses within basic blocks, and step III involves performing a dataflow analysis and clustering I/O buffer accesses across basic blocks, producing the code-motion results.

In the following sections, we explain in detail how the framework works, though we elaborate only from the view of managing the input buffer, since the concept of code motion for output buffer accesses is similar. The energy-aware code motion for the output buffer is discussed briefly in Section 4.4.

4.1. Insertion of Data-Transfer Operations

The framework is based on the idea of advancing accesses to the input buffer so as to extend its idle period. A simple way to move an access forward is to advance the operation (or instruction), say *A*, which accesses the input buffer. However, such motion might involve more than one operation being advanced, since all of the operations that produce the data required by operation *A* must be executed prior to *A*. In an extreme case, the program remains the same after code motion in that there is a chain of data dependencies for operation *A*, and all of the operations prior to *A* result in a cascading effect of code motion in order to ensure the program correctness.

In view of this, we insert a DTO that relocates input data from the input buffer to a temporary register before an input buffer access and replace the input buffer access by the access to the temporary register. A DTO could be simply a *move* instruction, such as a `VMOV` (vector move) instruction in our target Cyclone architecture. With the data-transfer process, we can advance an input buffer access (i.e., a move instruction) without significantly altering the program code. Figure 6 illustrates an example of the insertion of DTOs. Given a basic block containing three input buffer accesses, as shown in Figure 6(a), where operands starting with "`v`" and "`t`" refer to an input buffer access and a temporary register, respectively, and "`FADD t0, t0, v3`" performs a floating-point addition of `t0` and `v3` and places the result in `t0`, we insert a `VMOV` instruction for each of the input buffer accesses and change the input buffer accesses to temporary registers accordingly, as shown in Figure 6(b).

4.2. Local Code Motion

We propose a *local code-motion* algorithm to cluster the DTOs inserted in Section 4.1 within each basic block. The algorithm aims to bring forward the DTOs with register pressure in mind so as to concentrate input buffer accesses. In other words, a single DTO is moved ahead to be clustered with other DTOs if the code motion does not result in immoderate use of temporary registers; otherwise, the DTO stays where it is. In addition, DTOs that transfer the same data from the input buffer tend to be clustered together first so that those DTOs could be merged into a single operation in order to reduce the overhead of the execution performance as well as the code size.

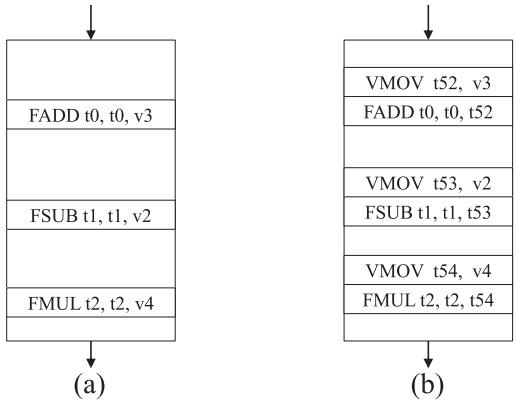


Fig. 6. An illustration of the insertion of data transfer operations: (a) a basic block containing three input buffer accesses and (b) the basic block after inserting data-transfer operations.

Algorithm 1 outlines the pseudocode for performing local code motion. Given a basic block, we first eliminate redundant DTOs by traversing all DTOs in a backward manner (lines 2–8). Suppose there is a pair of DTOs, say I_i and I_{i-1}, where I_i and I_{i-1} have the same source operand and I_{i-1} is the last DTO prior to I_i. I_i is merged with I_{i-1} by eliminating I_i and replacing all of the usages of the definition of I_i with the target operand of I_j if $\mathsf{CMT}(I_j, I_i)$ is greater than zero, where $\mathsf{CMT}(I_j, I_i)$ returns the *code-motion tunnel* between I_{i-1} and I_i (i.e., the number of unoccupied registers between I_{i-1} and I_i). I_i can be moved ahead to the position following I_{i-1} only when there is an unoccupied register to be allocated for the extended live range of the target operand of I_i. The code-motion tunnel suggests how many DTOs could be moved across it.

ALGORITHM 1: Local Code Motion

Input : A shader program with inserted data-transfer operations.
Output: The shader program with clustered data-transfer operations within basic blocks.

```
1  foreach basic block b ∈ input program do
2      foreach data transfer operation I_i ∈ b in the reverse order of their appearance do
3          if there exists an I_{i-1} that is the last one prior to I_i and I_i and I_{i-1} have the same
           source operand then
4              if CMT(I_{i-1}, I_i) > 0 then
5                  Eliminate I_i and replace all of the usages of the definition of I_i with the
                   target operand of I_{i-1};
6              end
7          end
8      end
9      Each data transfer operation I_i ∈ b forms a cluster C_i;
10     foreach cluster C_i ∈ b in the reverse order of their appearance do
11         if there exists a C_{i-1} that is the last cluster prior to C_i then
12             if CMT(C_{i-1}, C_i) ≥ |C_i| then
13                 Move C_i to the position following C_{i-1} and combine C_i and C_{i-1} into a
                   new C_{i-1};
14             end
15         end
16     end
17 end
```

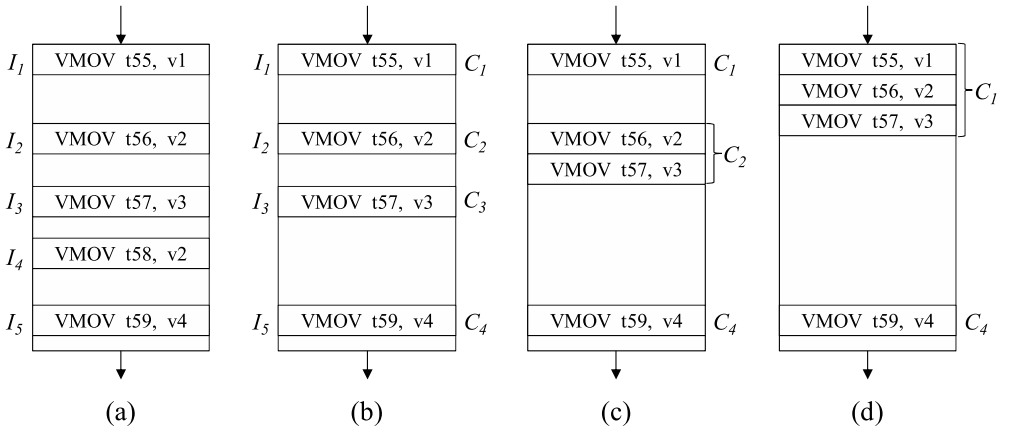


Fig. 7. An illustration of local code motion: (a) a basic block with four data-transfer operations inside, (b) the basic block after eliminating redundant data-transfer operations, (c) the basic block after the iterations of local code motion, and (d) the basic block after the local code motion.

Following the elimination of redundant DTOs, we implement an iterative code-motion method from the bottom to the top of the basic block while treating a cluster as the smallest unit for code motion (lines 9–16). Initially, each DTO forms a single cluster. If a cluster, C_i, could be moved ahead to the position following the last cluster (i.e., C_{i-1}) prior to C_i without exceeding the maximum operating register pressure (i.e., C_i can pass through the code-motion tunnel between C_i and C_{i-1}), C_i is combined with C_{i-1} into a new C_{i-1}. In other words, in line 12, $\mathsf{CMT}(C_{i-1}, C_i) \geq |C_i|$ could be interpreted as the number of unoccupied registers between the last DTO of C_{i-1} and the first DTO of C_i being greater than or equal to the number of DTOs in C_i so that the live range of the target of all of the DTOs in C_i could be extended to the end of C_{i-1}.

It is worth mentioning that local code motion is performed so as to move a cluster (instead of a DTO) ahead so that it is combined with another cluster. Moving a DTO seems to be more intuitive than moving a cluster, but we do not benefit by moving a DTO ahead unless there is no other input buffer access around the DTO. Recall that a DTO actually contains an input buffer access—it transfers a data value from the input buffer to a temporary register. If there are other DTOs not being forwarded, the input buffer must remain alive and cannot be shut down. Besides, moving a DTO forward so that it is clustered with other DTOs would increase the size of the cluster that might be moved across basic blocks via subsequent global code motion. For these reasons, it is better to move a cluster than a DTO.

Figure 7 illustrates an example of local code motion. Figures 7(a) and 7(b) elaborate an example of the first half of Algorithm 1: elimination of redundant DTOs. Given a basic block with five DTOs, we traverse the block backward and meet a DTO, I_4 (I_5 is the first DTO we meet, but it is not a candidate to eliminate according to the rules given next). We then examine whether I_4 could be eliminated by inspecting in a backward direction whether there is a DTO that is prior to I_4 and transfers the same data (`v2` in this case) from the input buffer as I_4, and learn that I_4 might be eliminated due to the prior DTO, I_2. Next, we check on whether there is an unoccupied register that could be used to extend the live range of `t58` (the target operand of I_4) by the condition of $\mathsf{CMT}(I_2, I_4) > 0$; this indicates that I_4 should be eliminated and the consequent `t58` should be renamed as `t56`, the target operand of I_2. Figure 7(b) shows the result after repeating this procedure for each DTO.

Figures 7(b) to 7(d) provide an example of the second half of Algorithm 1. Initially, each I_i forms a C_i, which is tagged on the right side of Figure 7(b). We traverse the block backward and meet a cluster, C_4, and another cluster, C_3, prior to C_4. We then examine whether C_4 could be moved ahead to the position following C_3 by determining whether the inequality $\mathsf{CMT}(C_3, C_4) \geq |C_4|$ holds; in other words, we examine whether there are unoccupied registers between C_3 and C_4, and whether there are more of them than the number of DTOs in C_4. If this is not the case, we continue to process the next cluster, C_3. Similarly, we determine whether the inequality $\mathsf{CMT}(C_2, C_3) \geq |C_3|$ applies. If this is the case, we combine C_2 and C_3 into a new C_2, as shown in Figure 7(c). The process is continued in the same manner until all clusters have been traversed, at which point we have the result of the local code motion, as shown in Figure 7(d).

4.3. Global Code Motion

Local code motion clusters input buffer instructions within each basic block and could reduce the power switching overhead when clock- or power-gating mechanisms are applied, but there remain some cases that local code motion cannot handle. If a source program consists of many small basic blocks and contains complex control flows, little advantage is gained from local code motion, because there is a low probability of DTOs having been clustered within basic blocks. In this case, we need a more powerful scheme, *global code motion*, to move DTOs or DTO clusters across basic blocks. More specifically, clusters that are located at the beginning of each basic block after local code motion are the ones to be relocated in global code motion, since clusters other than those at the top of a basic block can no longer be moved forward due to the register pressure reaching its maximum operating limit.

We propose a backward dataflow analysis method, called *cluster placement analysis*, to determine the possible locations for placing the DTO clusters while considering register pressure, and perform a global code motion of clusters based on the information gathered by the dataflow analysis. The algorithm of cluster placement analysis is listed in Algorithm 2, where EXIT is the exit block of a control-flow graph. The algorithm works similarly to a traditional dataflow analysis, which refers to a group of techniques that derive information about the flow of data along program execution paths [Aho et al. 2006]. In our proposed dataflow analysis, we associate each program point with a dataflow value that represents a set of clusters that could be placed at that point and denote the dataflow value immediately before and immediately after each basic block b by $\mathsf{CLUSTER}_{in}(b)$ and $\mathsf{CLUSTER}_{out}(b)$, respectively. The transfer equations, which define the relationship between the dataflow values before and after a basic block, and

ALGORITHM 2: Cluster Placement Analysis

Input : A shader program after local code motion.
Output: Possible locations for placing the DTO clusters.

1 $\mathsf{CLUSTER}_{out}(\mathsf{EXIT}) = \emptyset$;
2 **foreach** *basic block* $b \in$ *input program other than* EXIT **do**
3 $\quad \mathsf{CLUSTER}_{gen}(b) = \{c \mid c$ is the cluster at the beginning of $b\}$;
4 $\quad \mathsf{CLUSTER}_{in}(b) = \emptyset$;
5 **end**
6 **while** *changes to any* $\mathsf{CLUSTER}_{in}$ *occur* **do**
7 $\quad$ **foreach** *basic block* $b \in$ *input program other than* EXIT **do**
8 $\quad\quad \mathsf{CLUSTER}_{out}(b) = \bigcup_{cs \in CandidateSucc(b)} \mathsf{CLUSTER}_{in}(cs)$;
9 $\quad\quad \mathsf{CLUSTER}_{in}(b) = \mathsf{CLUSTER}_{gen}(b) \cup \mathsf{CLUSTER}_{out}(b)$;
10 $\quad$ **end**
11 **end**

predicates of the control-flow equations for collecting cluster placement information are given as follows.

—$\mathsf{CLUSTER}_{gen}(b)$ is a set with only one element, which is the cluster at the beginning of basic block b. $\mathsf{CLUSTER}_{gen}(b)$ represents the cluster that is initially placed at the beginning of basic block b and might "flow" to other basic blocks.

—$\mathsf{CLUSTER}_{out}(b)$ is a set of clusters that are from the successors of basic block b and can be moved to the end of block b.

$$\mathsf{CLUSTER}_{out}(b) = \bigcup_{cs \in CandidateSucc(b)} \mathsf{CLUSTER}_{in}(cs), \tag{1}$$

where $CandidateSucc(b)$ basically indicates a set of successors of block b in which all of the clusters at the top of them can pass through the code-motion tunnel of block b, that is, the clusters can be hoisted to the top of block b. $\mathsf{CLUSTER}_{out}(b)$ basically holds clusters that flow from the successors of block b and can be moved to the beginning of block b. Moreover, $\mathsf{CLUSTER}_{out}(b)$ excludes the clusters that belong to a set of clusters at the top of a successor of block b when the other clusters in the set cannot be moved. This exclusion eliminates the unnecessary code motion of clusters—the input buffer must remain powered-up if the other clusters are not brought forward as well—and thus decreases the likelihood of an increase in register pressure. Therefore, we define $CandidateSucc(b)$ as follows.

$$CandidateSucc(b) = \begin{cases} Succ(b), \text{ if } \| \bigcup_{s \in Succ(b)} \mathsf{CLUSTER}_{in}(s) \| \leq \mathsf{CMT}(b), \\ \{s\}, \text{ where } s \in Succ(b) \text{ and} \\ \qquad \| \mathsf{CLUSTER}_{in}(s) \| \leq \mathsf{CMT}(b), \text{ otherwise}, \end{cases} \tag{2}$$

where $Succ(b)$ is the set of successor program blocks of block b, $\|X\|$ returns the number of usages of the input buffer registers in the set X, and $\mathsf{CMT}(b)$ is the code-motion tunnel in block b. $\mathsf{CMT}(b)$ conceptually represents the number of unoccupied registers between the first and last operations (instructions) of block b. Specifically, $CandidateSucc(b)$ is identical to $Succ(b)$ if the number of usages of the input buffer registers in $\mathsf{CLUSTER}_{in}(s)$ for each successor s of block b is less than or equal to the code-motion tunnel of block b; otherwise, it is a singleton set in which is one of the successors, s, of block b, and $\|\mathsf{CLUSTER}_{in}(s)\|$ is either less than or equal to $\mathsf{CMT}(b)$ or the empty set. In the former case, Equation (1) transfers all of the dataflow values at the points immediately before the successors of b to the point immediately after b. In the latter case, suppose that there is a block b with two successors (a basic block has at most two successors) and that the clusters at the top of the two successors (one cluster in each successor) are able to be moved individually to the top of b, that is, only one of the two clusters can pass through the code-motion tunnel of block b. There should be a policy to determine shifting which of the two clusters would make the most profit for global code motion. In general, path profiling information would help in choosing an appropriate cluster to shift, with the cluster in a highly skewed branch preferentially moved forward so as to be merged with other clusters. However, feedback compilation is not practicable in many cases, so we propose two alternatives—*large-cluster-favorable* and *small-cluster-favorable* code motion—to determine which cluster to select for code motion when two clusters are able to pass through a tunnel but only one of them is allowed to do so. As implied by the names, the larger and smaller clusters are respectively chosen in the large- and small-cluster-favorable approaches. An alternative, naïve approach, called *zero-or-all* code motion, is to move no clusters if not all of them can pass through the tunnel. These approaches are evaluated and discussed in Section 5.

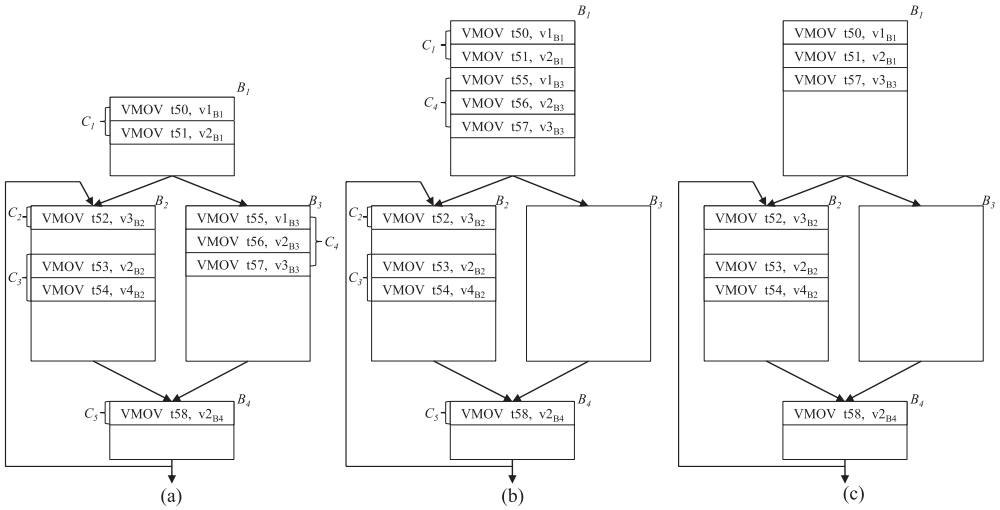


Fig. 8. An illustration of global code motion: (a) a control-flow graph before global code motion, (b) the control-flow graph after global code motion without redundancy elimination, and (c) the control-flow graph after global code motion.

—$\mathsf{CLUSTER}_{in}(b)$ is a set of clusters that can be moved to the beginning of block b.

$$\mathsf{CLUSTER}_{in}(b) = \mathsf{CLUSTER}_{gen}(b) \cup \mathsf{CLUSTER}_{out}(b). \tag{3}$$

Basically, two kinds of clusters are included in the set of $\mathsf{CLUSTER}_{in}(b)$: (1) the cluster that are originally located at the beginning of block b after local code motion, namely, the cluster in $\mathsf{CLUSTER}_{gen}(b)$, and (2) those that flow from $\mathsf{CLUSTER}_{out}(b)$.

We now present a running example to illustrate how the analysis works. Given a control-flow graph, as shown in Figure 8(a), we can determine where DTO clusters can be placed by performing cluster placement analysis. Suppose that steps I and II in the energy-aware code-motion framework (Figure 5) have been performed and the static register pressure information, $\mathsf{CMT}(b)$, has been calculated. In this example, it is found that clusters C_1, C_2, C_4, and C_5 are located at the beginning of B_1 to B_4, respectively. By the definition of $\mathsf{CLUSTER}_{gen}(b)$, which is a set of clusters at the beginning of block b, we have $\mathsf{CLUSTER}_{gen}(B_1) = \{C_1\}$, $\mathsf{CLUSTER}_{gen}(B_2) = \{C_2\}$, $\mathsf{CLUSTER}_{gen}(B_3) = \{C_4\}$, and $\mathsf{CLUSTER}_{gen}(B_4) = \{C_5\}$. We iteratively perform the analysis backward from block B_4 according to Algorithm 2 until $\mathsf{CLUSTER}_{in}(b)$ and $\mathsf{CLUSTER}_{out}(b)$ converge for each basic block b. Tables I and II give the computation results of the first and second iterations, respectively, and Table III lists the computation results of the third and fourth iterations (the algorithm converges in the fourth iteration). Note that a superscript following $\mathsf{CLUSTER}_{out}(b)$ represents the number of usages of input buffer registers in the DTOs of those clusters in $\mathsf{CLUSTER}_{out}(b)$.

While $\mathsf{CLUSTER}_{in}(b)$ indicates the set of clusters that can be advanced to the beginning of block b, the analysis results still cannot be used directly to move clusters, since some clusters may be placed in multiple locations; however, $\mathsf{CLUSTER}_{in}(b)$ provides hints about where to place the clusters. Algorithm 3 outlines the pseudocode for global code motion. We first perform cluster placement analysis to gather the $\mathsf{CLUSTER}_{in}(b)$ information for each block b and then determine the appropriate position for each cluster (lines 2–15). The principle of the determination process is to advance a cluster as far as possible away from its located block so that it has a higher probability of being merged or clustered, and thereby to extend the inactive periods of the input buffer. In

Table I. Results of Cluster Placement Analysis after the First Iteration

Block	CMT(b)	CLUSTER$_{gen}$(b)	CLUSTER$_{in}$(b)	CLUSTER$_{out}$(b)
B_1	3	$\{C_1\}$	$\{C_1, C_2, C_4, C_5\}$	$\{C_2, C_4, C_5\}^3$
B_2	1	$\{C_2\}$	$\{C_2, C_5\}$	$\{C_5\}^1$
B_3	2	$\{C_4\}$	$\{C_4, C_5\}$	$\{C_5\}^1$
B_4	2	$\{C_5\}$	$\{C_5\}$	$\emptyset$

Table II. Results of Cluster Placement Analysis after the Second Iteration

Block	CMT(b)	CLUSTER$_{gen}$(b)	CLUSTER$_{in}$(b)	CLUSTER$_{out}$(b)
B_1	3	$\{C_1\}$	$\{C_1, C_2, C_4, C_5\}$	$\{C_2, C_4, C_5\}^3$
B_2	1	$\{C_2\}$	$\{C_2\}$	$\emptyset$
B_3	2	$\{C_4\}$	$\{C_2, C_4, C_5\}$	$\{C_2, C_5\}^2$
B_4	2	$\{C_5\}$	$\{C_2, C_5\}$	$\{C_2, C_5\}^2$

Table III. Results of Cluster Placement Analysis after the Third and Forth Iterations

Block	CMT(b)	CLUSTER$_{gen}$(b)	CLUSTER$_{in}$(b)	CLUSTER$_{out}$(b)
B_1	3	$\{C_1\}$	$\{C_1, C_2, C_4, C_5\}$	$\{C_2, C_4, C_5\}^3$
B_2	1	$\{C_2\}$	$\{C_2\}$	$\emptyset$
B_3	2	$\{C_4\}$	$\{C_2, C_4, C_5\}$	$\{C_2, C_5\}^2$
B_4	2	$\{C_5\}$	$\{C_2, C_5\}$	$\{C_2\}^1$

general, if a cluster could be placed at both block b and block p (which is one of b's predecessors), it will eventually be placed at block p. However, if block b has at least two predecessors and the cluster cannot pass through one of them, no code motion should be performed in order to ensure program correctness. Moreover, even if the cluster can pass through all of them, placing the cluster at all of its predecessors (named *cluster-duplication-allowed* code motion) might increase the code size. Therefore, we propose an alternative approach, called *cluster-duplication-forbidden* code motion, in which a cluster c in block b can be moved to block d only if block d is a strict dominator[2] of block b, and cluster c can pass through all of the blocks on the paths from block b to block d. In other words, $c \in \text{CLUSTER}_{in}(d)$ and $c \in \text{CLUSTER}_{in}(t)$, where t is a block on the path between blocks d and b, that is, $t \in \{t \mid d \in S.Dom(t)$ and $b \in P.Dom(t)\}$, where $d \in S.Dom(t)$ indicates block d is a strict dominator of block t and $b \in P.Dom(t)$ represents that block b is a postdominator[2] of block t. In addition, the correctness of the program execution is guaranteed by not moving clusters in a loop kernel to the blocks outside the loop. Hence a cluster c at the beginning of b can be moved to block d only if there is no back edge from any of the descendant blocks of b, $Desc(b)$, to block t, which is located on the path between blocks d and b. In other words, cluster c can be moved to block d only if $Pred(t) \cap Desc(b) = \emptyset$, where $Pred(t)$ represents the set of predecessor blocks of t. We keep the placement information in GCM(b) for each block b, which represents the set of clusters that should be placed at the beginning of b and is initially the empty set. Accordingly, cluster c is put into GCM(d) if c can be moved to the beginning of block d and the preceding constraints are met. Next, the clusters are moved according to the placement information

[2]Block M is a strict dominator of block N if every path from the entry that reaches block N has to pass through block M, where $M \neq N$, while block N is a postdominator of block M if every path from block M to the exit has to pass through block N.

Table IV. Placement Information GCM

Block	$\mathsf{CLUSTER}_{in}(b)$	$S.Dom(b)$	$P.Dom(b)$	$Desc(b)$	$\mathsf{GCM}(b)$
B_1	$\{C_1, C_2, C_4, C_5\}$	$\emptyset$	$\{B_4\}$	$\{B_2, B_3, B_4\}$	$\{C_4\}$
B_2	$\{C_2\}$	$\{B_1\}$	$\{B_4\}$	$\{B_2, B_4\}$	$\emptyset$
B_3	$\{C_2, C_4, C_5\}$	$\{B_1\}$	$\{B_4\}$	$\{B_2, B_4\}$	$\emptyset$
B_4	$\{C_2, C_5\}$	$\{B_1\}$	$\{B_4\}$	$\{B_2, B_4\}$	$\emptyset$

ALGORITHM 3: Global Code Motion

Input : A shader program after local code motion.
Output: The shader program with clustered data-transfer operations across basic blocks.

```
1  Perform Cluster Placement Analysis;
2  foreach basic block b ∈ input program do
3  |  GCM(b) = ∅;
4  end
5  foreach cluster c where c is at the beginning of basic block b; that is, c = CLUSTER_gen(b) do
6  |  foreach basic block d ∈ S.Dom(b) in the order of their appearance and c ⊆ CLUSTER_in(d) do
7  |  |  foreach basic block t ∈ {t | d ∈ S.Dom(t) and b ∈ P.Dom(t)} do
8  |  |  |  if c ⊊ CLUSTER_in(t) or Pred(t) ∩ Desc(b) ≠ ∅ then
9  |  |  |  |  continue 6;
10 |  |  |  end
11 |  |  end
12 |  |  GCM(d) = GCM(d) ∪ c;
13 |  |  break;
14 |  end
15 end
16 foreach basic block b ∈ input program do
17 |  Perform code motion according to GCM(b);
18 end
19 foreach basic block b ∈ input program do
20 |  foreach data-transfer operation I_i clustered at the beginning of b in the reverse order of their appearance do
21 |  |  if there exists an I_{i-1} that is the last one prior to I_i and I_i and I_{i-1} have the same source operand then
22 |  |  |  Eliminate I_i and replace all of the usages of the definition of I_i with the target operand of I_{i-1};
23 |  |  end
24 |  end
25 end
```

(lines 16–18), and finally, the redundant DTOs are eliminated, as in local code motion (lines 19–25).

Continuing with the running example in Figure 8(a), the calculation results of $\mathsf{GCM}(b)$ are given in Table IV. Figure 8(b) illustrates the control-flow graph after global code motion according to the $\mathsf{GCM}(b)$ information. Figure 8(c) presents the final results of the global code motion after eliminating redundant DTOs.

4.4. Energy-Aware Code Motion for Output Buffers

The only difference between the concepts of code motion for output buffer and input buffer accesses is the direction of motion. In the case of the output buffer, we prefer

to demote the results from the temporary registers to the output buffer immediately before the program terminates so that the output buffer can be kept off for as long as possible. As illustrated in Figure 5, three phases of actions are performed: insertion of DTOs, local code motion, and global code motion. Unlike in code motion for the input buffer, we insert a DTO that relocates output data from a temporary register to the output buffer after each output buffer access, replace the output buffer access by the access to the temporary register, and then move the DTOs backward according to local and global code motions so as to extend the idle period of the output buffer. The concept of local code motion for the output buffer involves bringing backward the DTOs (based on consideration of the register pressure) within basic blocks so as to concentrate output buffer accesses, whereas we bring DTOs forward for the input buffer, as described in Algorithm 1. There is no need to eliminate redundant DTOs in this case, because each output buffer register is generally written to once only. Algorithm 3 is also used for global code motion of the output buffer with some minor revisions with regard to functions, such as dominators/postdominators, predecessors/successors, and descendants/ancestors, being exchanged and the backward cluster placement analysis defined in Algorithm 2 being modified into a forward analysis.

5. EXPERIMENTAL RESULTS AND DISCUSSION

5.1. Experimental Setup

We used a shader core in the Cyclone GPU, which runs at a frequency of 200 MHz with a 10kB cell-based SRAM with hardware clock-gating control, as described in Figure 3, as the target architecture for our experiments. The proposed energy-aware code-motion framework was incorporated into an in-house compiler that was developed based on the Vincent 3D Rendering Library [Vincent Pervasive Media Technologies 2011] by the Institute for Information Industry (III), Taiwan, prior to the phase of register allocation, and it was evaluated by a cycle-approximate simulator. Nine common shader programs (which are provided by III) were evaluated. Table V summarizes the characteristics of each benchmarks, which includes both fragment and vertex shader programs. The second column of the table presents the execution time ratios of the fragment and vertex shader programs, the third and fourth columns indicate the proportions of the input-buffer-access cycles to the execution cycles of fragment/vertex shader programs and the proportions of the output-buffer-access cycles to the execution cycles, respectively, and the fifth column gives the numbers of registers required. In general, the lesser the register requirement and the lower the proportion of input or output buffer accesses in a program, the higher the reduction in energy reduction that might be achieved, since the proposed framework is likely to cluster these input/output buffer accesses at the beginning/end of the program.

The target architecture contains an eight-entry input buffer, an eight-entry output buffer, and eight temporary registers in the shader core, with both the output buffer and the temporary registers having read/write accesses, and the input buffer being read-only. Since the target shader core has no load/store instructions, the compiler does not generate memory spill code, but the readable/writable output buffer may be used for storing spilled variables if none of the temporary registers are available. Therefore, programs with register requirements higher than eight registers will contain extra output buffer accesses due to spilling. Unfortunately, such an implementation makes the proposed framework ineffective in managing the output buffer for certain shader programs, since it changes the behavior of output buffer accesses after those accesses have been clustered.

Table VI presents the dynamic power consumption of the input buffer, output buffer, and other components of a Cyclone shader core given by the Synopsys Design

Table V. Benchmark Characteristics

Benchmark name		Execution time ratio	Proportion of input-buffer-access cycles*		Proportion of output-buffer-access cycles*		Register requirement	Description
*Blur*0	frag.†	90.6%	11.2%	10.9%	0.9%	3.1%	3	Blur rendering (high float precision)
	vert.‡	9.4%	8.3%		25.0%		3	
*Blur*1	frag.	90.6%	11.2%	10.9%	0.9%	3.1%	3	Blur rendering (medium float precision)
	vert.	9.4%	8.3%		25.0%		3	
HGaussianBlur	frag.	82.8%	26.0%	26.7%	1.0%	6.0%	3	Horizontal Gaussian blur rendering
	vert.	17.2%	30.0%		30.0%		2	
NormalMap	frag.	27.7%	8.9%	10.4%	1.8%	2.5%	6	Adding detail to a flat surface
	vert.	72.3%	11.0%		2.7%		8	
POM	frag.	95.1%	2.9%	3.5%	21.5%	20.9%	13	Parallax occlusion mapping
	vert.	4.9%	14.9%		8.5%		6	
SimpleTexLight	frag.	43.9%	19.1%	16.1%	1.5%	5.2%	6	Texture and light rendering
	vert.	56.1%	13.8%		8.0%		7	
SimpleTex	frag.	69.5%	10.2%	11.7%	0.7%	3.6%	8	Texture rendering
	vert.	30.5%	15.0%		10.0%		7	
VGaussianBlur	frag.	83.9%	24.0%	15.0%	1.0%	5.6%	3	Vertical Gaussian blur rendering
	vert.	16.1%	30.0%		30.0%		2	
WaterWave	frag.	93.8%	3.6%	6.2%	0.6%	1.7%	5	Water wave rendering
	vert.	6.2%	45.5%		18.2%		3	

*The proportion of the input-/output-buffer-access cycles to the execution cycles of the fragment and/or vertex shader programs.

†frag. is abbreviation of fragment shader program.

‡vert. is abbreviation of vertex shader program.

Table VI. Dynamic Power Consumption in a Cyclone Shader Core

Component	Power Consumption (unit)	Power Breakdown
Input Buffer	3.28	9.1%
Output Buffer	5.51	15.3%
Others	27.20	75.6%

Compiler® [Synopsys 2009]. The energy overhead of hardware clock gating can be ignored, since only an extra `AND` gate is needed to implement the clock-gating mechanism, but an extra cycle is required to turn on the gated clock.

5.2. Evaluations

The results presented in the following sections are evaluated with three configurations: performing the proposed energy-aware code-motion framework when (1) only the input buffer, (2) only the output buffer, and (3) both of the I/O buffers are considered to be clock gated. Sections 5.2.1, 5.2.2, and 5.2.3 discuss the evaluation results, respectively.

5.2.1. Applying the Proposed Framework to the Input Buffer. Figures 9–11 give the compilation and simulation results of the energy-aware code-motion framework when clock gating is considered during the use of the input buffer and compare the results from five types of experiments: (1) no clock-gating mechanism (used as the baseline), (2) naïve clock gating without code motion, (3) clock gating with local code motion, (4) clock gating with both local and global code motions, and (5) clock gating with naïve

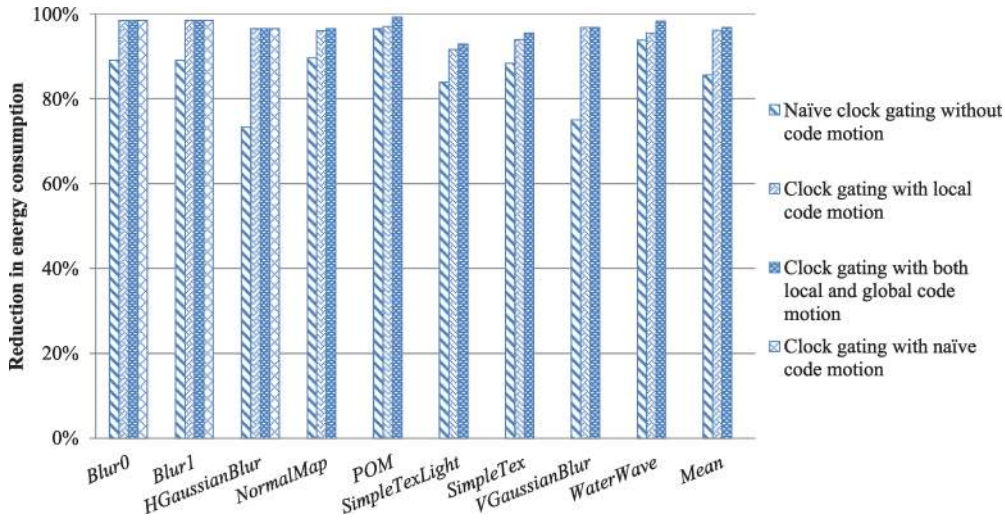


Fig. 9. Reduction in energy consumption of the input buffer when the input buffer is considered to be clock gated.

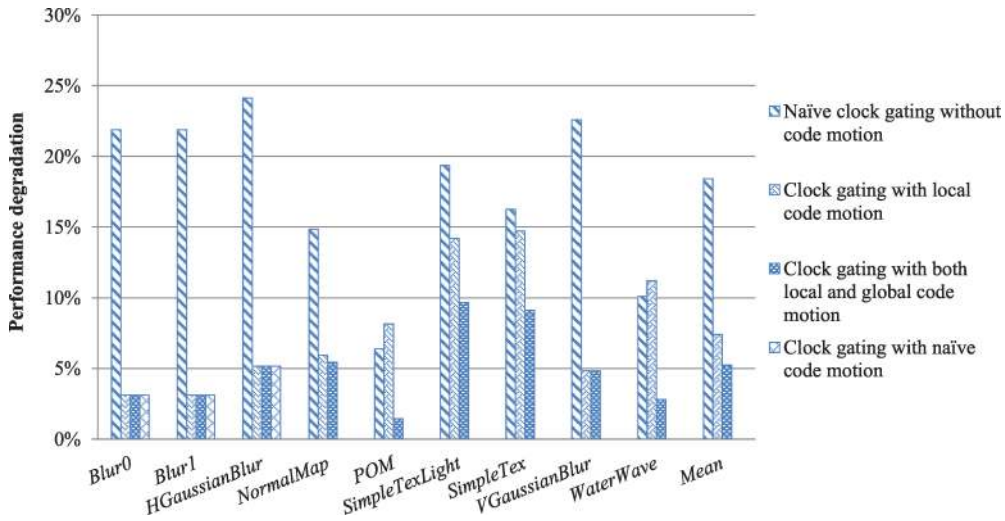


Fig. 10. Performance degradation when the input buffer is considered to be clock gated.

code motion. Naïve clock gating means that the input buffer is clock gated immediately after the buffer is accessed and its gated clock is turned on before an access, and hence suffers from frequent clock gating; whereas naïve code motion refers to relocating all input data from the input buffer to the temporary registers at the beginning of the program execution.

Figure 9 illustrates the reduction in energy consumption of the input buffer for various programs, each of which includes both fragment and vertex shader programs. The figure shows that the method of naïve clock gating, clock gating with local code motion, clock gating with both local and global code motions reduced the energy consumption of the input buffer by averages of 85.6%, 96.1%, and 96.8%, respectively, relative to the conventional no-clock-gating method. The reductions made by naïve clock gating are reversely proportional to the proportion of the input-buffer-access cycles of programs

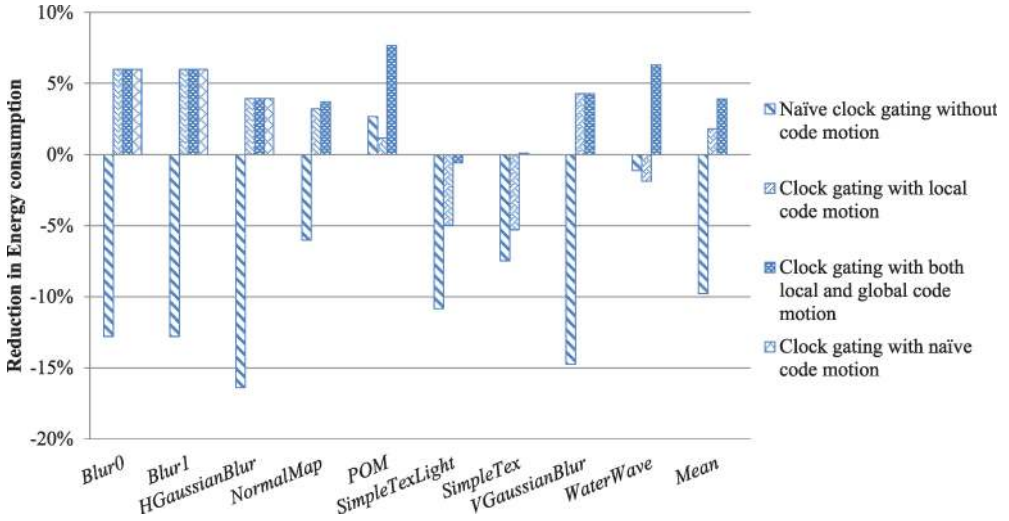


Fig. 11. Reduction in energy consumption of the entire shader processor when the input buffer is considered to be clock gated.

(i.e., column three of Table V). Clock gating with both local and global code motions reduced the energy consumption by the greatest amount because it clusters—and also eliminates—DTOs (the accesses to the input buffer) across basic blocks so as to keep the buffer clocked-off for as long as possible and reduce the amount of clock gating. Clock gating with naïve code motion was only applicable to the first three benchmarks whose register pressure is sufficiently low to allow the promotion of input data from the input buffer to the temporary registers at the beginning of the programs, so the fourth bar is absent for the other benchmarks. Recall that the Cyclone shader cores have no load/store instructions, and thus spilling a register to memory does not apply, and the execution of a shader program will fail if the register allocator runs out of registers. Although the method of clock gating with naïve code motion should be the simplest approach and provide the greatest reduction in energy consumption (if it is applicable), the other proposed methods actually work just as well.

Although the naïve clock-gating method provided acceptable results in terms of the reduction in energy consumption of the input buffer, frequent clock gating significantly slowed down program execution and increased the energy consumption of the other parts of the shader processor, which counteracted the benefit. Figure 10 presents the performance degradation due to the inserted DTOs and the clock-gating delay. Without appropriate management of the input buffer accesses, the performance degradation was up to 24.1% (18.4% on average) for naïve clock gating. When we applied the proposed energy-aware code-motion framework, the performance degradation was on average reduced to 7.4% and 5.3% using the local and global code-motion schemes, respectively. Local code motion worked as well as both local and global code motions for most of the programs containing a single basic block, but did not work well for *POM*, *SimpleTexLight*, *SimpleTex*, and *WaterWave*, which are characterized by small basic blocks and complex control flows. When using only local code motion, *POM*, *SimpleTexLight*, *SimpleTex*, and *WaterWave* suffered from the inserted DTOs, especially within loops, and some of them took longer to execute, even compared with naïve clock gating.

Figure 11 shows the reduction in energy consumption for the entire shader processor, based on the data in Table VI. The energy consumption of the entire shader processor was reduced by −9.8%, 1.8%, and 3.9% for naïve clock gating, clock gating with local

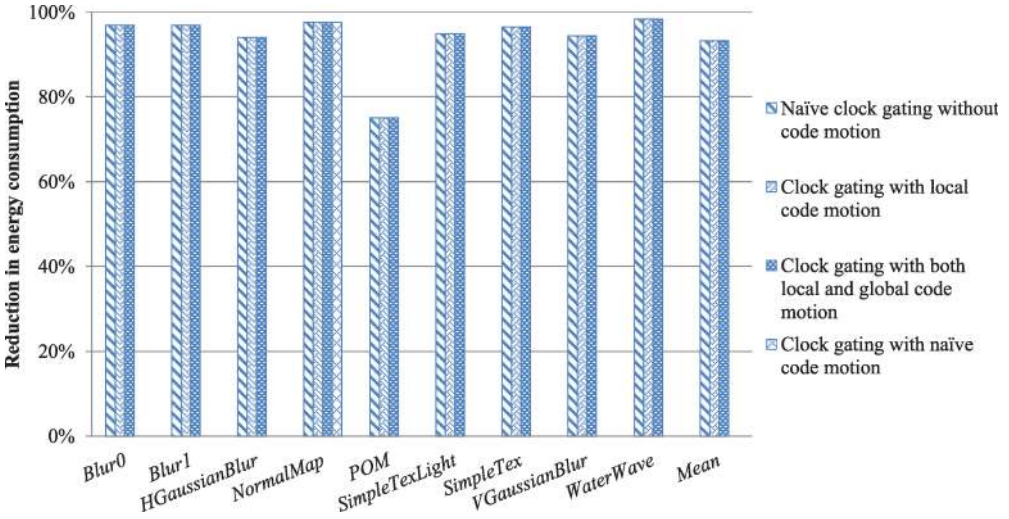


Fig. 12. Reduction in energy consumption of the output buffer when the output buffer is considered to be clock gated.

code motion, and clock gating with both local and global code motions, respectively, relative to the no-clock-gating method. Again, local code motion had a negative effect on the energy consumption for *POM* and *WaterWave* compared with naïve clock gating due to their inclusion of small basic blocks and complex control flows. Clock gating with both local and global code motions failed to reduce the energy consumption of the entire shader processor for *SimpleTexLight* and *SimpleTex* (with an increase of 0.6% and a reduction of 0.1%, respectively), since the high register pressure and the slightly high proportion of input buffer accesses of these programs disallowed the code-motion scheme to cluster all of the input buffer accesses, meaning that these programs still suffered from frequent clock gating although the amount of clock gating was reduced.

In Section 4.3, we propose several strategies for global code motion: (1) cluster-duplication-allowed and (2) cluster-duplication-forbidden code motions for deciding how a cluster is moved across a joint point of programs where two branches meet. We also propose (1) large-cluster-favorable, (2) small-cluster-favorable, and (3) zero-or-all code motions for determining which cluster to move across a branch point of programs when two clusters (one in each branch path) could be moved across the branch point individually. The strategies basically produced the same outcome with regard to both the reduction in energy consumption and the performance degradation for the evaluated benchmarks (with average less than 3% of variation). Comparing the strategies for joint-point code motion, cluster-duplication-allowed code motion performed slightly better than cluster-duplication-forbidden code motion, since the former provided more opportunities for clusters to be merged with other clusters, allowing more DTOs to be eliminated. As for the strategies for branch-point code motion, zero-or-all code motion performed the worst in most cases, since it did not allow even partial code motion of clusters and thus lost the opportunity for a movable cluster to be merged with other clusters, while large-cluster-favorable and small-cluster-favorable code motions performed about the same.

5.2.2. Applying the Proposed Framework to the Output Buffer. Figures 12–14 illustrate the reduction in energy consumption of the output buffer, the performance degradation, and the reduction in energy consumption of the entire shader processor, respectively,

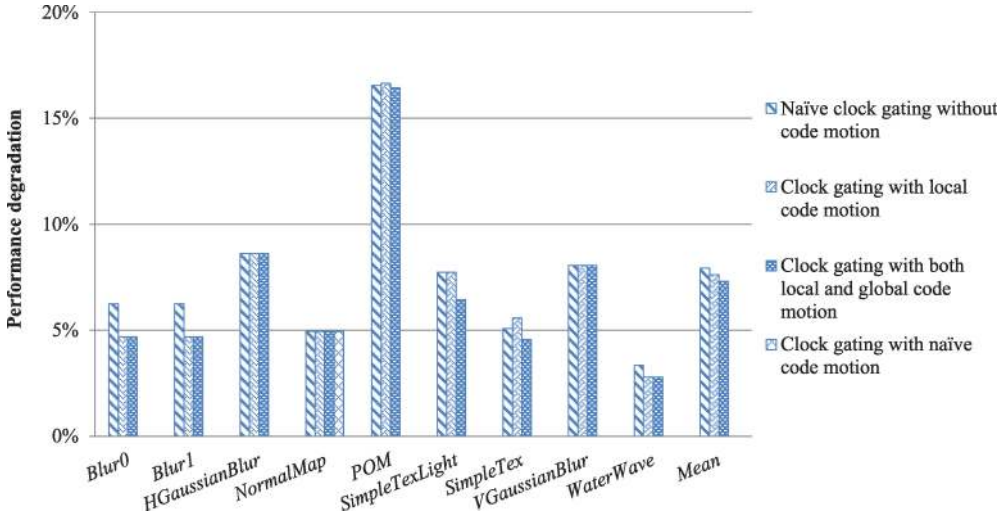


Fig. 13. Performance degradation when the output buffer is considered to be clock gated.

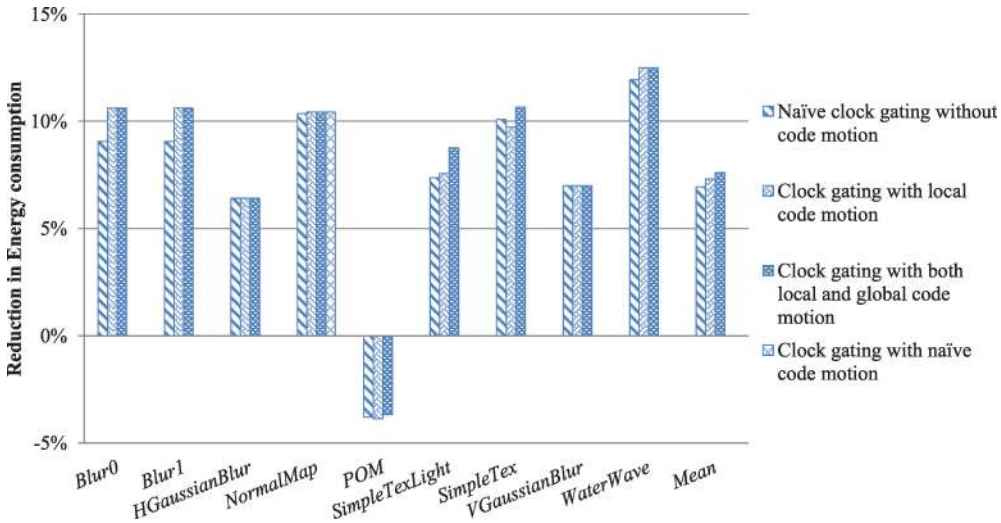


Fig. 14. Reduction in energy consumption of the entire shader processor when the output buffer is considered to be clock gated.

obtained when applying the reverse code-motion framework described in Section 4.4 for various programs, each of which includes both fragment and vertex shader programs. These figures show that the proposed framework was effective in reducing the energy consumption of the entire shader processor for most programs (with an average of 7.6% reduction and an average of 7.3% of performance degradation), but it is only slightly more effective than the naïve clock-gating method (which has an average of 6.9% reduction in energy consumption of the entire shader processor and an average of 7.9% of performance degradation). This slight difference is attributed to the proportion of the output buffer accesses of these programs being mostly less than or equal to 6% (as shown in Table V) and the code-motion schemes having few chances to gather DTOs or even eliminate redundant DTOs; this also explains naïve clock gating, clock gating

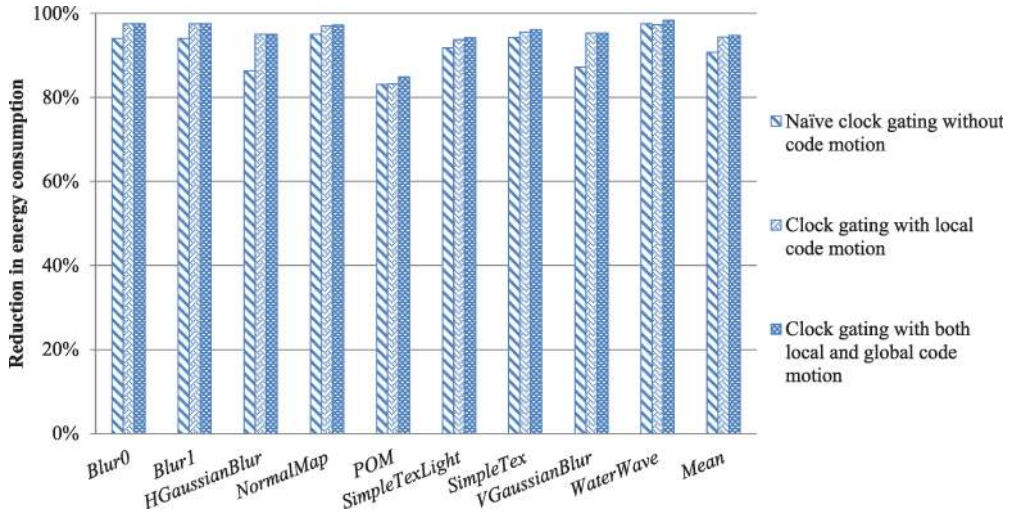


Fig. 15. Reduction in energy consumption of the I/O buffers when the I/O buffers are considered to be clock gated.

with local code motion, and clock gating with both local and global code motion having the same reduction in energy consumption of the output buffer for each program, as shown in Figure 12.

It is noteworthy that the proposed framework increased the average energy consumption of the entire shader processor by 8.1% for vertex shader programs, since some of these vertex shader programs are so small (relative to fragment programs) that the additional DTOs and the clock-gating operations inserted by the code-motion framework have a negative impact on energy consumption. Nevertheless, the reduction in energy consumption of the entire shader processor was 7.6% on average when considering both fragment and vertex shader programs.

However, *POM* behaved extremely badly, with 3.7% of increase in energy consumption of the entire shader processor and 16.4% of performance degradation. This negative effect is attributed to the access behavior of the output buffer in *POM* being more complex than that of other programs: the proportion of the output buffer accesses is 20.9% and the number of register requirement is 13, which exceeds the number of temporary registers (which is eight). In addition to the accesses triggered by programmers, the output buffer might be used for spilling registers due to the lack of load/store operations in the target shader processors. Furthermore, according to OpenGL ES 2.0 specification, programmers are allowed to use the output buffer in the same manner as the use of the temporary registers. Unfortunately, these kinds of output buffer access are very difficult to barely move. Therefore, the output buffer is often accessed randomly and frequently so that the recurring clock gating negates energy-efficiency efforts, especially for programs with a high register pressure. The phenomenon will eventually diminish as the number of temporary registers increases. We performed another experiment with 16 temporary registers in the shader core and obtained 12.5% reduction in energy consumption of the entire shader processor for *POM*.

5.2.3. Applying the Proposed Framework to Both Input and Output Buffers. Figures 15–18 give the reduction in energy consumption of the I/O buffers, the performance degradation, the reduction in energy consumption of the entire shader processor, and the energy-delay product for various programs when clock gating is considered during the use of I/O buffers, respectively.

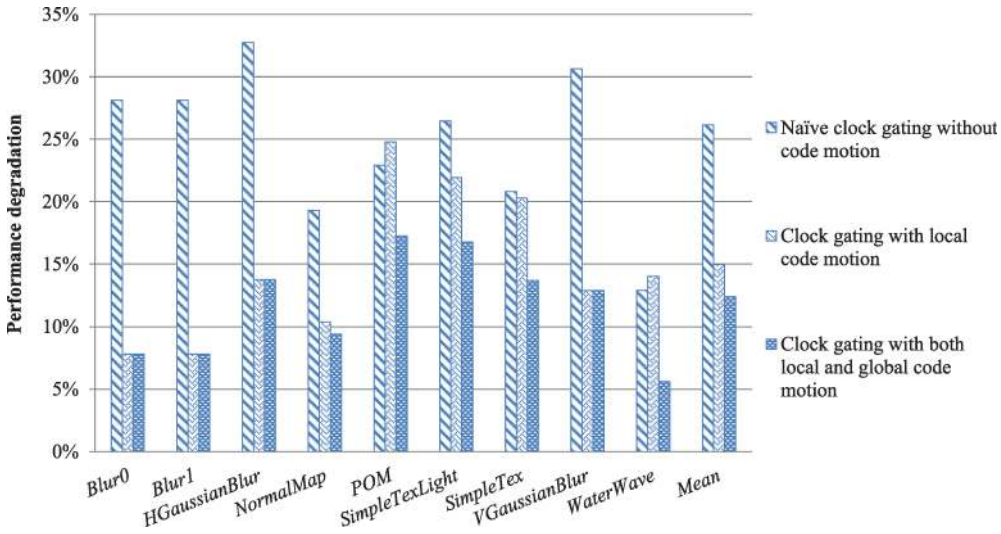


Fig. 16. Performance degradation when the I/O buffers are considered to be clock gated.

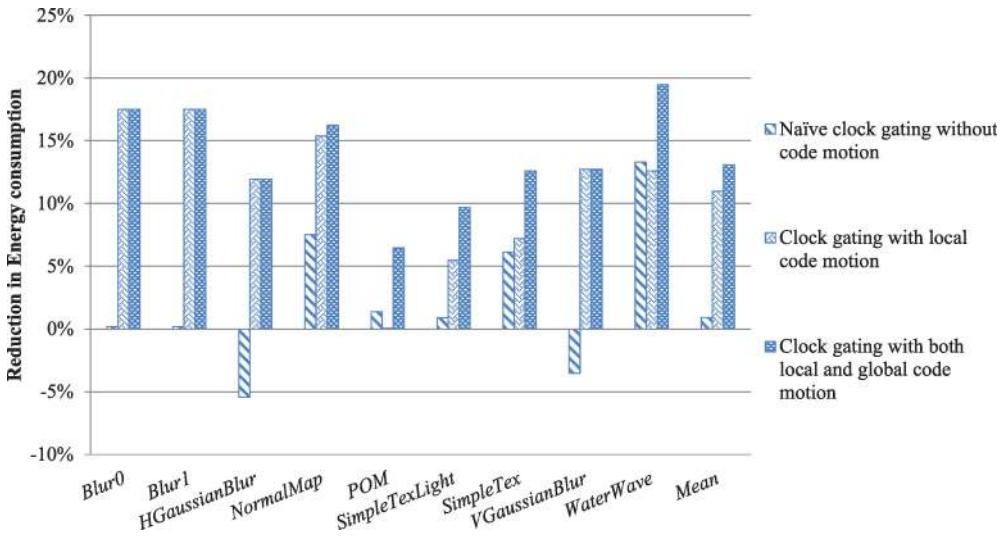


Fig. 17. Reduction in energy consumption of the entire shader processor when the I/O buffers are considered to be clock gated.

Figure 15 shows that the method of naïve clock gating, clock gating with local code motion, clock gating with both local and global code motions reduced the energy consumption of the I/O buffers by averages of 90.7%, 94.3%, and 94.7%, respectively, while Figure 16 shows that these methods increased the number of execution cycles by averages of 26.1%, 15.0%, and 12.4%, respectively. Figure 17 indicates that the energy-aware code motion framework was effective in reducing the energy consumption of the entire shader processor, with an average of 13.1% reduction, for applying the framework to both of the I/O buffers, whereas naïve clock gating and clock gating with local code motion reduced the energy consumption by averages of 0.9% and 11.0%, respectively. It can be seen that the proposed method has a synergistic effect on the

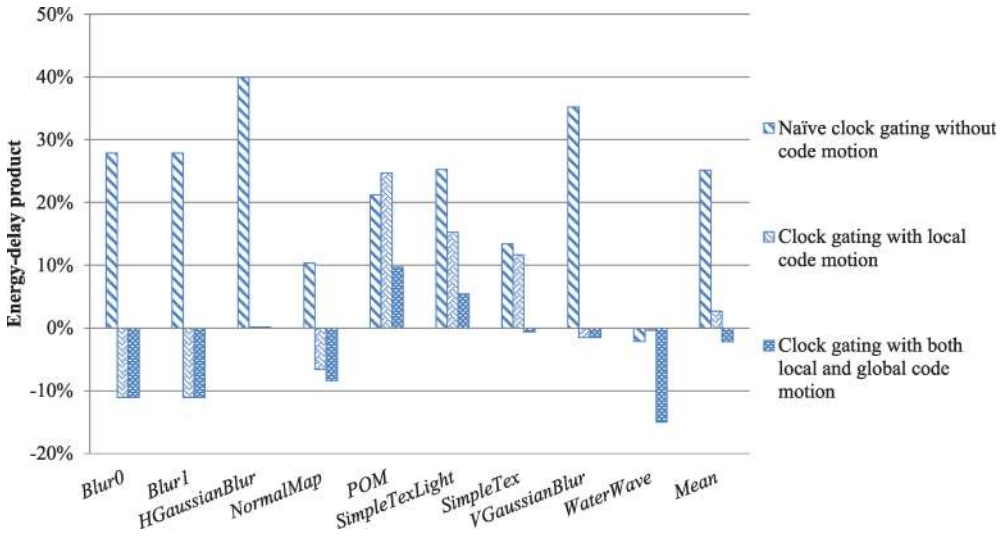


Fig. 18. Energy-delay product in terms of the entire shader processor when the I/O buffers are considered to be clock gated.

combination of applying it individually to the input and output buffers for most programs. This effect can be explained by the fact that for most shader programs, the live intervals of input variables (residing in the input buffer) usually do not overlap with those of output variables (residing in the output buffer); hence, promoting data in the input buffer and demoting data in the output buffer to the temporary registers do not interfere with each other.

Figure 18 illustrates the energy-delay product relative to the no-clock-gating method. The figure shows that the proposed framework has a positive effect on reducing the energy-delay product by an average of 2.2%, whereas naïve clock gating and clock gating with local code motion increased the energy-delay product by averages of 25.1% and 2.7%, respectively. The proposed framework had a negative impact on *POM* and *SimpleTexLight* (with an increase of 9.7% and 5.5%, respectively), since *POM* and *SimpleTexLight* are characterized by small basic blocks and complex control flows, as described in Section 5.2.1, and *POM* has complex output buffer accesses, as described in Section 5.2.2. However, when the number of the temporary registers is increased to 16, the proposed framework reduced the energy-delay product by 18.7% for *POM*. With the 16-temporary-register configuration, the reduction in energy-delay product is 4.8% on average for applying the framework to both of the I/O buffers. However, for most programs, the results with the 16-temporary-register configuration are worse than those with the 8-temporary-register configuration. This is attributed to the increased size of temporary registers also consuming energy per se, and in our experiment, it was assumed that eight temporary registers consume the same energy as an eight-entry output buffer, since they are both readable and writable.

All in all, the proposed energy-aware code-motion framework was found to be effective in reducing the energy consumption of the entire shader processor and the overall energy-delay product. The framework reduced the energy consumption of the entire shader processor by averages of 3.9%, 7.6%, and 13.1%, and reduced the energy-delay product by averages of 3.0%, 0.7%, and 2.2% for applying the framework to the input buffer, output buffer, and both of the I/O buffers, respectively. In the framework, local code motion is suitable for handling programs with large basic blocks, and combining

global code motion with local code motion can further handle those with small basic blocks or complex control flows.

6. CONCLUSIONS AND FUTURE WORK

Herein, we have proposed an energy-aware code-motion framework for a compiler solution to reduce the energy consumption of buffers and memory in stream-processing-like architectures. Our experiments have demonstrated that the proposed energy-aware code-motion framework improves the energy consumption and performance: the energy consumptions of the entire shader processor were reduced from −9.8% (by naïve clock gating) to 3.9% and 6.9% to 7.6% compared with the no-clock-gating method when we applied the proposed framework to the input buffer and output buffer, respectively, while the performance degradation was reduced from 18.4% to 5.3% and 8.0% to 7.3%. The results can be further improved if we apply the proposed framework to both of the I/O buffers: an average of 13.1% of reduction in energy consumption of the shader processor was achieved, whereas the energy-delay product was reduced by 2.2%.

We are currently in the process of incorporating single-static-assignment forms into our framework to simplify the data-dependency problem so as to increase the efficiency of clustering buffer-access instructions. The experiments were performed only with clock-gating mechanisms, and we expect that our approach will be even more beneficial if it is applied to system-on-a-chip platforms with power-gating controls that reduce both dynamic and static energy consumption.

The proposed framework could also be applied to compilers for embedded CPUs or any other platforms as long as the execution model of such a platform is similar to the stream-processing model (in which the processing unit reads input data from an input buffer, performs calculations, and writes the results into an output buffer) and the processing unit is equipped with power control systems for the I/O buffers. Conceptually, data caches in CPUs could be treated as a joint I/O buffer, that is, the data caches are both an input buffer and an output buffer. With our proposed scheme, accesses to the data caches would be clustered, with data reads and writes forming distinct groups, and hence the data caches may be forced to enter a low-power mode when they are not accessed. However, general-purpose programs may require more accesses to data caches than graphics-processing programs to I/O buffers, making the clusters less concentrated. We think this might be an interesting issue and leave it for future study.

REFERENCES

AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. 2006. *Compilers: Principles, Techniques, and Tools* 2nd Ed. Prentice Hall.

APPLE 2011. OpenGL ES on iOS. `http://developer.apple.com/iphone/library/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/OpenGLESontheiPhone/OpenGLESontheiPhone.html\#//apple_ref/doc/uid/TP40008793-CH101-SW1`.

CHANG, C.-M., CHEN, Y.-J., LU, Y.-C., LIN, C.-Y., CHEN, L.-G., AND CHIEN, S.-Y. 2011. A 172.6mW 43.8GFLOPS energy-efficient scalable eight-core 3D graphics processor for mobile multimedia applications. In *Proceedings of the IEEE Asian Solid-State Circuits Conference (A-SSCC'11)*. 405–408.

CHIEN, S.-Y., TSAO, Y.-M., CHANG, C.-H., AND LIN, Y.-C. 2008. An 8.6mW 25Mvertices/s 400-MFLOPS 800-MOPS 8.91mm2 multimedia stream processor core for mobile applications. *IEEE J. Solid-State Circuits 43,* 9, 2025–2035.

COOPER, K. D., SIMPSON, L. T., AND VICK, C. A. 2001. Operator strength reduction. *ACM Trans. Program. Lang. Syst. 23,* 5, 603–625.

DROPSHO, S., KURSUN, V., ALBONESI, D. H., DWARKADAS, S., AND FRIEDMAN, E. G. 2002. Managing static leakage energy in microprocessor functional units. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO'02)*. IEEE Computer Society Press, 321–332.

GOOGLE. 2011. Android 3.2 platform. `http://developer.android.com/sdk/android-3.2.html`.

HONG, S. AND KIM, H. 2010. An integrated GPU power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. 280–289.

KHRONOS GROUP. 2010. OpenGL ES 2.0 specification. http://www.khronos.org/opengles/2_X/.

KHRONOS GROUP. 2011. OpenGL. http://www.opengl.org/.

KHRONOS GROUP. 2009. The OpenGL ES Shading Language. http://www.khronos.org/opengles/sdk/docs/manglsl/.

KO, M.-Y., LIN, I.-T., LEE, S.-Y., LYU, Z.-H., CHANG, C.-M., AND CHENG, Y.-J. 2011. Cyclone—A GPU IP designed for embedded 3D games. In *Proceedings of the 24th Conference on Computer Vision, Graphics, and Image Processing (CVGIP'11)*. SS1–1–4.

MAHJUR, A., TAGHIZADEH, M., AND JAHANGIR, A.-H. 2008. Lazy instruction scheduling: Keeping performance, reducing power. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'08)*. 375–380.

MICROSOFT. 2008. DirectX common-shader core (directx hlsl). http://www.microsoft.com/windows/directx/default.mspx.

MOCHOCKI, B. C., LAHIRI, K., CADAMBI, S., AND HU, X. S. 2006. Signature-based workload estimation for mobile 3D graphics. In *Proceedings of the 43rd Annual Design Automation Conference (DAC'06)*. 592–597.

PALM. 2011. webOS developer center. https://developer.palm.com/.

RELE, S., PANDE, S., ONDER, S., AND GUPTA, R. 2002. Optimizing static power dissipation by functional units in superscalar processors. In *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*. 261–275.

ROY, S., RANGANATHAN, N., AND KATKOORI, S. 2009. Exploring compiler optimizations for enhancing power gating. In *Proceedings of the IEEE International Symposium on Circuit and Systems (ISCAS'09)*. 1004–1007.

SILPA, B., VEMURI, K. S., AND PANDA, P. R. 2009. Adaptive partitioning of vertex shader for low power high performance geometry engine. In *Advances in Visual Computing*. Lecture Notes in Computer Science, vol. 5875, Springer, Berlin, 111–124.

SYNOPSYS. 2009. Design Compiler. http://www.synopsys.com/.

VINCENT PERVASIVE MEDIA TECHNOLOGIES. 2011. Vincent 3D rendering library—open source graphics libraries for mobile and embedded devices. http://www.vincent3d.com/software/software.html.

WANG, P.-H., CHEN, Y.-M., YANG, C.-L., AND CHENG, Y.-J. 2009. A predictive shutdown technique for GPU shader processors. *IEEE Comput. Architect. Lett. 8,* 1, 9–12.

YANG, H., GOVINDARAJAN, R., GAO, G. R., CAI, G., AND HU, Z. 2002. Exploiting schedule slacks for rate-optimal power-minimum software pipelining. In *Proceedings of the 3rd Workshop on Compilers and Operating Systems for Low Power (COLP'02)*.

YOU, Y.-P., HUANG, C.-W., AND LEE, J. K. 2005. A Sink-N-Hoist framework for leakage power reduction. In *Proceedings of the ACM International Conference on Embedded Software (EMSOFT'05)*. 124–133.

YOU, Y.-P., HUANG, C.-W., AND LEE, J. K. 2007. Compilation for compact power-gating controls. *ACM Trans. Des. Autom. Electron. Syst. 12,* 4, 51.

YOU, Y.-P., LEE, C., AND LEE, J. K. 2002. Compiler analysis and supports for leakage power reduction on microprocessors. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC'02)*. Lecture Notes in Computer Science, vol. 2481, Springer Verlag, Berlin, 63–73.

YOU, Y.-P., LEE, C., AND LEE, J. K. 2006. Compilers for leakage power reduction. *ACM Trans. Des. Autom. Electron. Syst. 11,* 1, 147–164.

ZHANG, W., KANDEMIR, M. T., VIJAYKRISHNAN, N., IRWIN, M. J., AND DE, V. 2003. Compiler support for reducing leakage energy consumption. In *Proceedings of the 6th Design Automation and Test in Europe Conference (DATE'03)*. 1146–1147.

Received September 2011; revised April, September 2012; accepted December 2012