Iteration Interleaving–Based SIMD Lane Partition

YAOHUA WANG, DONG WANG, SHUMING CHEN, ZONGLIN LIU,
SHENGGANG CHEN, XIAOWEN CHEN, and XU ZHOU, National University of Defense
Technology P.R. China

The efficacy of single instruction, multiple data (SIMD) architectures is limited when handling divergent control flows. This circumstance results in SIMD fragments using only a subset of the available lanes. We propose an iteration interleaving–based SIMD lane partition (IISLP) architecture that interleaves the execution of consecutive iterations and dynamically partitions SIMD lanes into branch paths with comparable execution time. The benefits are twofold: SIMD fragments under divergent branches can execute in parallel, and the pathology of fragment starvation can also be well eliminated. Our experiments show that IISLP doubles the performance of a baseline mechanism and provides a speedup of 28% versus instruction shuffle.

CCS Concepts: • **Computer systems organization** → **Single instruction, multiple data**; • **Hardware** → Application specific processors

Additional Key Words and Phrases: SIMD, iteration interleaving, vector iteration, SIMD lane partition, instruction shuffle

ACM Reference Format:

Yaohua Wang, Dong Wang, Shuming Chen, Zonglin Liu, Shenggang Chen, Xiaowen Chen, and Xu Zhou. 2015. Iteration interleaving–based SIMD lane partition. *ACM Trans. Archit. Code Optim.* 12, 4, Article 58 (December 2015), 18 pages.
DOI: <http://dx.doi.org/10.1145/2847253>

1. INTRODUCTION

Single instruction, multiple data (SIMD) architectures can effectively exploit data-level parallelism (DLP) while reducing hardware costs in data-independent structures. SIMD capability is widely adopted in both commercial processor products and academic prototypes, such as GPU architectures [NVIDIA 2008, 2009, 2012], Cell Broadband Engine [Chen et al. 2007], and Many Integrated Core [Seiler et al. 2008]. Some academic prototypes seek to improve the performance or utilization of SIMD architectures. They include the FT-Matrix [Chen et al. 2014], vector-thread (VT) processors [Krashinsky et al. 2004], stream processors (Imagine [Khailany et al. 2001], FT-64 [Yang et al. 2007]), vector lane threading (VLT) [Rivoire et al. 2006], AnySP [Woh et al. 2010], and Maven [Lee et al. 2011].

This work is supported by the Research Project of National University of Defense Technology under grant GC-14-06-02 and the National Science Foundation of China under grants 61402493 and 61433007.

This article is an original, previously unpublished, paper.

Authors' addresses: Y. Wang, S. Chen, Z. Liu, S. Chen, X. Chen, and X. Zhou, College of Computer Science, National University of Defense Technology, Changsha, Hunan Province, China, 410073; emails: nudtyh@gmail.com (primary), yhwang@nudt.edu.cn (secondary), smchen@nudt.edu.cn, zllnudt@126.com, shgchen@nudt.edu.cn, xwchen@nudt.edu.cn, zhouxu@nudt.edu.cn; D. Wang, Department of Science, National University of Defense Technology, Changsha, Hunan Province, China, 410073; email: nudtjum@163.com.

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.

© 2015 ACM 1544-3566/2015/12-ART58 \$15.00

DOI: <http://dx.doi.org/10.1145/2847253>

Fig. 1. An unbalanced branch paths structure is shown (a), with the corresponding execution scenario of IS and GI schemes illustrated in (b) and (c), respectively. Black dashed arrows represent idle SIMD lanes. GI executes different branch paths sequentially, whereas IS dynamically partitions SIMD lanes among branch paths. However, the execution time of the IS scheme is determined by the longest branch path, causing fragment starvation pathology. A possible optimization for fragment starvation is shown (d), which partitions SIMD lanes into branch paths with comparable execution time through iteration interleaving.

However, the performance of SIMD architectures can be restricted by divergent control flows, upon which SIMD lanes executing in lock-step would branch into different branch paths. Constrained under the single instruction structure, different branch paths have to be executed sequentially, leading to active execution of only a subset of SIMD lanes, which we refer to as an SIMD fragment. Since SIMD architectures tend to integrate more and more lanes, handling SIMD fragments becomes increasingly important for performance scaling. This problem has received considerable attention in SIMD architecture designs. Examples include the commonly used guarded instruction (GI) mechanism [Bouknight et al. 1972], dynamic warp formation (DWF) [Fung et al. 2007, 2009], and SIMD lane partition schemes like instruction shuffle (IS) [Wang et al. 2012; Wang et al. 2013]. Each design employs different strategies.

Among the preceding solutions, IS dynamically partitions SIMD lanes among different branch paths and captures a significant fraction of the benefits of multiple instruction, multiple data (MIMD) hardware on SIMD processors. However, the benefit of the IS scheme is greatly affected by the unbalanced branch paths structure that stem from the inherent nature of data-parallel applications. Figure 1(a) shows an unbalanced divergent control flow structure that helps to illustrate such inefficiency. The execution time (marked in blocks) of each branch path is quite different: seven for A-B-D-H, nine for A-B-E-H, and three for A-C-H. The corresponding execution scenario of two consecutive iterations is shown in Figure 1(b). The execution of GI is also illustrated (Figure 1(c)) for comparison.

As shown in Figure 1(c), GI lets SIMD fragments executed sequentially upon branches. Numerous SIMD lanes, marked with black dashed arrows, are wasted. IS partitions SIMD lanes dynamically and facilitates the concurrent execution of multiple SIMD fragments, recovering most of the performance loss by GI. However, the execution

time of each iteration is determined by the longest branch path (e.g., A-B-E-H), resulting in wasted SIMD lanes for fragments under short branch paths (e.g., A-C-H). We call this pathology *fragment starvation*, which greatly limits the benefit of IS. Worse performance is expected for applications with even unbalanced branch structures.

Although such fragment starvation decreases the efficiency of each iteration, there are opportunities to improve the efficiency by considering consecutive iterations. Figure 1(d) provides an example. By interleaving the execution of two consecutive iterations, fragments under paths A-B-D-H and A-B-E-H are scheduled in the first iteration, leaving fragments A-C-H in the second iteration. Consequently, wasted SIMD lanes caused by fragment starvation are greatly reduced.

Based on the preceding insight, we propose an iteration interleaving–based SIMD lane partition (IISLP) mechanism, which helps to eliminate the fragment starvation pathology while maintaining the key benefits of SIMD lane partition. The key idea of IISLP is that through interleaving of consecutive iterations, SIMD fragments under comparable branch path execution time can be rescheduled to execute together. This greatly helps to improve the performance of applications that suffer from divergent control flows on SIMD architectures. We summarize our main contributions as follows:

- We identify fragment starvation pathology as a significant challenge that limits the efficiency of existing SIMD lane partition schemes upon unbalanced branch path structures. We propose the idea of categorizing branch paths according to their execution time.
- We introduce the concept of IISLP, in which SIMD fragments under the same category of branch paths are rescheduled to execute together. We develop the IISLP microarchitecture and carry out the corresponding register-transfer–level (RTL) implementation based on our in-house SIMD prototype.
- We quantitatively evaluate IISLP, which doubles the performance of GI and provides an additional performance boost of 28% over existing SIMD lane partition schemes. We compare IISLP with related architectures like MIMD and DWF. We also reveal that a better performance gain can be expected with a larger amount of register files (RFs) in each SIMD lane.

2. BASELINE SIMD ARCHITECTURE

In this section, we introduce the SIMD prototype that we use to demonstrate the IISLP architecture. A brief description of the IS microarchitecture is also given.

2.1. Overview of SIMD Architecture

Our baseline SIMD prototype (Figure 2(a)) consists of a scalar control unit (SCU), SIMD unit, and a memory subsystem. The SIMD unit is tightly coupled with the SCU to reduce data transfer time to and from the SCU. The same philosophy is employed in Intel’s MIC and AMD’s Fusion. The SCU fetches instructions from the I-cache, executes scalar instructions, and loads SIMD instructions. The SIMD unit has multiple lanes executing in a lock-step manner. In our current design, four RFs exist in each lane. Thus, the execution contexts of four iterations can be maintained, and SIMD lanes are time shared among multiple iterations to tolerate long latency operations. The memory subsystem contains the instruction cache and data cache for the SCU and a private multibanked scratchpad memory for the SIMD unit. A DMA engine supports efficient nonblocking data transfers between the D-cache and the scratchpad memory.

2.2. IS Microarchitecture

As shown in Figure 2(a), the IS component serves as an extension of the instruction loading for SIMD unit. The cornerstone is composed of the instruction buffer array (IBA), instruction shuffle unit (ISU), and pending lane buffer (PLB). To achieve

Fig. 2. Block diagram of the baseline SIMD architecture and its IS component (a). Execution procedure of IS, in which SIMD lanes are dynamically partitioned among branch paths and different branch paths are executed concurrently (b).

dynamic partition of SIMD lanes upon divergent control flows, the code is first statically analyzed. Each basic instruction block, defined as the sequence of instructions between branch targets, is scheduled to one of the instruction buffers at compile time. The target address of branch instructions is transformed to a form (referred to as PC for clarity) that combines the instruction buffer number and offset within the buffer.

During execution, the ISU between the IBA and SIMD lanes is dynamically reconfigured according to the branch outcome in each lane. Thus, each lane can connect to the instruction buffer that corresponds to its branch path and execute concurrently. As for the code in Figure 1(a), Figure 2(b) illustrates the corresponding execution procedure. At first, all SIMD lanes connect to B0. At the end of block A, the connection pattern of the ISU is dynamically changed, and SIMD lanes under branch paths B and C connect to B0 and B1, respectively. Both SIMD fragments can run concurrently. The PLB is used to suspend the execution of SIMD lanes upon instruction buffer confliction and resume the execution of these SIMD lanes as soon as the target buffer becomes idle.

To maintain efficiency of conventional SIMD execution and reduce buffer confliction, synchronization is necessary at the end of each iteration. Due to this synchronization, iterations' execution time is determined by the longest branch path. For applications with unbalanced branch paths structure, a large amount of execution bubbles will show up in SIMD fragments under shorter branch paths, causing severe fragment starvation problems. This can greatly affect the benefits of IS.

3. IISLP ARCHITECTURE

The key idea of our IISLP architecture is to interleave the execution of consecutive iterations after branch divergence so that SIMD fragments with comparable branch

Fig. 3. The block diagram of IISLP. In addition to the IBA and ISU, IISLP architecture also introduces a schedule unit, which contains distributed tags, a status table, and reschedule and reload control modules. The distributed tags contain execution contexts, including branch types, timestamps, and PC values. The status table facilitates the rescheduling process by managing information of distributed tags. Reschedule and control modules maintain the status table and form fragment starvation–eliminated iterations.

path execution time can be dynamically rescheduled to execute together. As a result, fragment starvation is eliminated and the benefit of the IS scheme is well maintained.

3.1. Overall Design

Figure 3 is a block diagram of the IISLP architecture. In addition to the IBA and ISU, the IISLP architecture also introduces a schedule unit. Next we discuss how these components work together with the example shown in Figure 1(a).

Application code is first statically analyzed, and branch paths are divided into two types according to their path length (execution time). Branch paths of the same type have relatively similar length. This can be simply achieved by the k -means cluster algorithm, with k set to 2. The k -means cluster is a well-studied algorithm that aims to partition n observations into k clusters, in which each observation belongs to the cluster with the nearest mean. For Figure 1(a), the first type corresponds to paths B-D and B-E, which we refer to as the long type. The second type corresponds to path C, which we refer to as the short type. The branch path type is encoded into a special instruction, which is transformed from the original branch divergence instruction, and determines the types of both taken and untaken branch paths. The code of different branch paths is then loaded into different buffers in the IBA.

For the example in Figure 1(a), Figure 4 shows the overall execution process of IISLP. Figure 5 illustrates the corresponding context of the status table and distributed tags (only branch path type) in detail. For description simplicity, we refer an N -iteration bundle that can be executed concurrently on an N -wide (8-wide in the figure) SIMD architecture as a vector iteration (VI). To distinguish, we refer to single iterations in a VI as per-lane iterations. Each SIMD lane contains four sets of RFs so that the context of four consecutive VIs can be maintained. The status table in Figure 5 is set to contain

Fig. 4. The overall execution process of IISLP. Four consecutive VIs are suspended one after another upon the branch divergence point, then per-lane iterations are rescheduled according to their branch type. The rescheduling process helps to generate an S-VI, containing per-lane iterations of the same branch type. The execution of S-VI can well eliminate the fragment starvation problem. When S-VI finishes its execution, a new VI is reloaded to the vacancies left by the last S-VI, then the whole execution is ready for rescheduling again.

two rows for two distinctive branch path types, and the distributed tags are illustrated in the 4*8 grid defined by the RF and SIMD lane.

As shown in Figure 4, VIs are suspended upon the branch divergent point, which is at the end of block A in Figure 1(a). Branch path types (L for long and S for short), PC values, and timestamps of each per-lane iteration are registered in distributed tags according to the branch resolution (shown in Figure 5(a)). For clarity, PC values and timestamp are not shown. The status table collects indexes of both types in each lane under a timestamped priority. A “null” is marked when the exact branch path type in that lane does not exist. If no null exists in a status table row, the flag bit of that row is set to 1.

After the suspension of four consecutive VIs, the rescheduling process begins, in which the execution of VIs is interleaved. Per-lane iterations with the same branch path type are rescheduled to form a synthetic vector iteration (S-VI). With the help of the status table, rescheduling can be simply done by selecting per-lane iterations according to the indexes in the row whose flag bit is 1. As shown in Figure 5(a), the flag bit of the L row is 1, and an L-type S-VI can be formed according to the indexes in that row (01, 01, 01, 01, 00, 00, 00, 00), which corresponds to per-lane iterations originally from VI (2, 2, 2, 2, 1, 1, 1, 1), respectively.

As shown in Figure 4, the rescheduling process is followed by the execution of the newly formed S-VI. Although per-lane iterations in the S-VI are of the same branch path type, they are not necessarily the same branch path. Thus, during the execution of S-VI, the whole execution mode is turned into the SIMD lane partition manner, in which multiple SIMD fragments under comparable branch paths run concurrently. The status table and distributed tags are also updated accordingly during this execution process (shown in Figure 5(b)).

When the S-VI finishes its execution, a new VI (VI 5) is loaded and suspended upon branch divergence. The reload control module facilitates the storing of corresponding context into the vacancies left by the last S-VI. The status table is updated accordingly, looking for opportunities to eliminate the original null items. As shown in Figure 5(c),

Fig. 5. The context of status table and distributed tags (only branch type) before and after the rescheduling process. (a) The status table collects indexes of each branch type (L for long and S for short) in each lane right after the suspension of a VI. If no null exists, the flag bit F is set to 1, indicating an S-VI of the same branch path type (L type) can be formed. (b) After the rescheduling process, the status table and distributed tags are updated accordingly. (c) A new VI is loaded, filling vacancies left by the last S-VI; the status table will also be updated accordingly; and the whole architecture then gets ready for rescheduling again. (d) An S-type S-VI is formed.

the new VI is suspended with the branch path type registered in grid vacancies left by the last S-VI, and the original null items in the status table (L 2, L 6, L 7) are also updated. After this stage, the rescheduling process forms an S-type S-VI (shown in Figure 5(d)). The reload, suspension, and rescheduling process will repeat until the end of execution. The rest of this section describes details of the IISLP components.

3.2. Hardware Components

This section describes details of each IISLP component. To make this article self-contained, we briefly introduce the IS component.

IS component. This component contains two parts: the IBA and ISU. The IBA is a set of single-ported instruction buffers coupled with decoders, providing instruction streams to SIMD lanes that fall into different branch paths. In the rare case where the absolute code size of a kernel is larger than the total size of the IBA, the kernel can

Fig. 6. Block diagram of the schedule unit. It contains four parts: reschedule control, status table, reload control, and distributed tags. The distributed tags are shown along with RFs in each SIMD lane. Timestamps, branch path types (both indicated by T), and PC values of suspended per-lane iterations are recorded in the tags.

be explicitly segmented into multiple smaller kernels by the programmer. Moreover, proper instruction buffer size from profiling of target applications can well eliminate such issues. The ISU is a crossbar-based structure and supports flexible connection patterns between IBA and SIMD lanes. The connection patterns can be dynamically configured according to the branch outcome in each SIMD lane.

We remove the PLB structure and merge the branch path at software level to avoid instruction buffer confliction. This path merge has equal performance effect as the original PLB operation while eliminating the hardware complexity of the PLB. Another reason for this lies in the fact that the IISLP scheme executes not all branch paths concurrently but only branch paths of the same type. This helps to decrease the number of concurrent instruction flows, and thus instruction buffer conflictions are reduced accordingly.

Schedule unit. The schedule unit is shared by all SIMD lanes and is used to interleave the execution of suspended VI. As shown in Figure 6, the schedule unit is composed of four parts: the status table, the reschedule and reload control modules, and distributed tags.

The status table has two rows, each corresponding to a branch type (L and S). Each table row contains N indexes and one flag, where N is equal to the SIMD width. The index is the RF set number occupied by a per-lane iteration, which has the exact branch path type. When all indexes in a row are not null, the flag is set to 1, indicating that an S-VI can be formed.

Distributed tags are illustrated along with each RF in SIMD lanes. Tag content includes branch types (T), timestamps (T), and PC values for each per-lane iteration. The branch path type records the runtime branch path type (i.e., L or S), whereas PC contains its corresponding ISU configuration, which is used to connect SIMD lanes

with corresponding instruction buffers in the IBA. Timestamps record the time when a per-lane iteration is suspended.

The main function of the reschedule control module includes two parts: S-VI construction and update of the status table. The reschedule control module constructs S-VIs according to the indexes of the status table. The construction policy is as follows. The status table row with a flag bit set to 1 will be selected. If both flags of the status table are 1, then the row that is different from the last selection will be chosen. This is to keep all per-lane iterations running in a relatively synchronized manner. It is also possible, however, that flags of both L and S rows are 0. In this circumstance, the L row will be selected, and those null items in the L row will be filled by corresponding items in the S row. Although this will form an S-VI of different branch path types, compared to leaving null items unfilled, the benefit of the IS scheme can be well maintained.

Update of the status table happens in two circumstances. One is upon the suspension of a VI, during which the schedule control module checks if the newly suspended VI contains branch path types that can fill the original null items of the table row. The other is after the construction of the S-VI, in which the schedule unit first clears the corresponding table row and then refills the table row in a timestamped policy according to distributed tags. The latter type of update operations can always be done concurrently with the execution of an S-VI, introducing no extra cycle cost.

The reload control module improved the original VI load with the ability of loading per-lane iterations into dedicated vacancies left by the last S-VI rather than into the same set of RFs. This can be fulfilled by simply introducing a MUX to each SIMD lane (shown in Figure 6).

Overall, the extra time cost of rescheduling lies in the formation of an S-VI and the update process upon VI suspension. As described earlier, both processes need only simple logic. Besides the rescheduling, extra logic for reloading is simple as well. In our current design, we merge them with existing pipeline stages, and no extra pipeline cycles are introduced.

3.3. Barrier Suspension Point

As described previously, rescheduling happens only after VIs are suspended upon the branch divergence point, and the branch path types of all per-lane iterations should be determined after suspension. To distinguish this special branch divergence point, we call it the *suspension point*. However, for applications with nested branches, the suspension point, which determines the type of descendant branch paths, can be located in a nested branch path. As shown in Figure 7, branch paths B-D, B-E, and C-G belong to the long type, whereas branch path C-F belongs to the short type. Branch divergence point X, located at the end of block C, is the so called suspension point, as no matter which branch path a per-lane iteration executes, its branch path type can be determined right after X.

To synchronously suspend per-lane iterations in a VI, we introduce the barrier suspension point, which is inserted in other branch paths at the same execution cycle with the suspension point, helping to guarantee that the branch path types of per-lane iterations in a VI can be well determined after suspension. For the example shown in Figure 7, barrier suspension points Y and Z are inserted into branch paths B-D and B-E, respectively, at cycle 3, which is the exact execution cycle of X. In the corresponding execution process, all per-lane iterations (0 to 7) will be suspended synchronously at the end of cycle 3 with branch path types determined. As in our SIMD prototype, the execution cycle of the whole instruction set is fixed; both the location of suspension point and insertion of barrier suspension points can be done at compile time.

Fig. 7. Control flow structure with barrier suspension points and corresponding execution process.

3.4. Instruction Set Architecture

To efficiently support the execution process of the IISLP architecture, we introduce four instructions for the SCU: *Fetch*, *SIS*, *Sus*, and *Inqs*. We also introduce two instructions for SIMD lanes: *Synch* and *Stop*. Compared to the IS architecture, *Sus* is the only instruction that we introduced additionally.

Fetch is used by the SCU to asynchronously load instructions to the IBA. It has two operands: *addr* and *size*. The *addr* is the start address of instructions. The *size* indicates the prefetching size. *SIS* enables SIMD lanes to fetch instructions from the IBA through the ISU rather than from the SCU so that SIMD lanes under different branch paths can obtain instructions concurrently. *Sus* is the suspension instruction, which is transformed from the original branch instruction. Aside from the PC value of the taken path, *Sus* also has two additional operands, which indicate the types of both taken and untaken branch paths. When used as barrier suspension instructions, the branch type of the untaken path is set to null. *Sus* also triggers the rescheduling process when the execution context of four VIs exist. *Synch* is used to synchronize the execution of all SIMD lanes. *Stop* sets the state of all SIMD lanes to idle. *Inqs* is executed in the SCU, which waits for all SIMD lanes to become idle, and switches the entire execution mode back to the traditional SIMD mode, in which the SCU loads instructions for SIMD lanes.

Figure 8(b) illustrates the use of these instructions for a slice of conventional branch divergence code shown in Figure 8(a). At first, the SCU executes *Fetch* to prefetch instructions. To hide the prefetching overhead, *Fetch* can be placed well before *SIS*. *SIS* then enables the SIMD lane partition mode. The original branch instruction is transformed into the *Sus* instruction, which contains the target PC (Buff1, C) and branch path types (L, S) of both the taken and untaken path. *Sus* triggers the rescheduling process followed by the execution of S-VI. All SIMD lanes then synchronize before block D. In the end, SIMD lanes execute *Stop* to set the state to idle. Then the SCU executes *Inqs*, turning the whole execution mode into the conventional SIMD manner.

Fig. 8. Example code of the IISLP architecture.

3.5. Combination with Horizontal Permutation

Some parallel applications exhibit a regular branch resolution pattern in each VI [Rhu and Erez 2013a]. For instance, the first SIMD lanes may always fall into the branch paths of long type, whereas other SIMD lanes correspond to the short type. Although this regular pattern may reduce the amount of compactable threads in GPUs, it could limit the benefit of IISLP, as the number of S-VIs containing different types of per-lane iterations can increase. Rhu and Erez [2013a] proposed balanced horizontal permutations to break the regular branch resolution pattern in GPUs. Although IISLP is orthogonal to the horizontal permutation scheme, a proper combination with such permutation can provide much more robust performance gains.

To achieve this type of combination, we apply additional data swizzle operations to refine the mapping between per-lane iteration identifiers and SIMD lanes. Swizzle parameters are generated with Rhu’s balanced permutation algorithm, which guarantees that the branch resolution can be refined into a more balanced distribution pattern. As data swizzle operations, which help SIMD lanes communicate with each other under the control of swizzle parameters, are commonly supported in SIMD architectures [NVIDIA 2012; Chen et al. 2014], no extra hardware support is needed.

3.6. Hardware Implementation

We have implemented the RTL modules for the key components of IISLP in Verilog. The Synopsys Design Compiler is used to synthesize the implementations in 40nm technology at 700MHz. The IBA has four instruction buffers and decoders, and the ISU, supporting 4 to 16 flexible connections, consumes about 276,537 μm^2 and 18,342 μm^2 , respectively, and the schedule unit accounts for 11423 μm^2 , which is about 3.9% of the original IS component. We have also integrated these modules into our in-house SIMD prototype design, which is oriented for the embedded data-parallel processing domain and derived from the FT-Matrix chip [Chen et al. 2014]. The prototype then becomes an instance of the IISLP architecture. The prototype works at 700MHz with an overall area of about 25mm². Considering such a core area, the overall area overhead of IISLP can be omitted.

Table I. Parameters for Baseline SIMD Architecture

Scalar Control Unit (SCU)	
I-cache	32KB, 2-set associative
Scalar data path (SCU)	Function unit: MAC, ALU, SHIFT, Load/Store, Branch Register file: 32-entry 64 bit
D-cache	8KB, 2-set associative Cache line: 128 bits
SIMD Unit	
Number of SIMD lanes (SIMD width)	16
Function units in each lane	MAC, ALU, SHIFT, Load/Store*2
Register file in each lane	4 sets of 32-entry 64bit
Local Memory	
Memory type	Scratchpad memory
Number of banks	16
Bandwidth per bank	32Gbps
Storage per bank	64KB
DMA bandwidth	512Gbps

4. METHODOLOGY

We have developed a detailed cycle-accurate microarchitectural simulator. The simulator is validated against our prototype chip. Baseline architecture parameters are summarized in Table I. In the baseline SIMD architecture, interleaving of VIs is supported as four sets of RFs are maintained. SIMD unit simulation is combined with a cycle-driven memory system simulation. The divergent control flow problem is handled with the GI scheme. The idea of a divergence stack, which is always used in the GPU for branch reconvergence, is not supported currently. However, the idea of IISLP does not preclude the use of such a structure.

Based on the preceding simulator, we integrate the components of the IISLP micro architecture. The IBA has four instruction buffers with corresponding decoders, and each contains a maximum of 1,024 instructions. The ISU consists of 16 4:1 configurable MUXes. The status table has two table lines, each with 16 indexes and one flag. The size of distributed tags is set to $16(\text{SIMD width}) * 4(\text{sets of RFs}) * 20(\text{tag content})$ bits. Reload and rescheduling logic is also included. For comparison, we set the IS architecture as the one without a status table, distributed tags, and reload and rescheduling logic; architecture parameters for other parts are kept the same as IISLP.

DWF [Fung et al. 2007, 2009] also adopts the idea of rescheduling iterations in GPUs; the key difference between DWF and IISLP is that DWF regroups identical branch paths rather than comparable paths. To understand such a difference, we conduct the comparison with DWF. As our hardware and simulation infrastructure are quite different from GPU architectures, we quantitatively analyze the performance benefit of regrouping only identical branch paths in the IISLP microarchitecture, which we refer to as a DWF-like scheme. We have also qualitatively analyzed possible performance benefits of IISLP over DWF, when adopted in GPU architectures.

Besides the DWF scheme, the concurrent execution of multiple branch paths makes the IISLP architecture an intermediate between SIMD and MIMD architectures. To fully understand the performance benefit, we also conduct the performance comparison with the MIMD architecture. For the MIMD architecture, we modify the baseline SIMD architecture with an additional instruction cache, instruction fetch, and issue unit for each SIMD lane. Thus, multiple SIMD lanes can execute in a MIMD manner.

Application code is handwritten in assembly using the prototype instruction set. This approach limited the number of benchmarks we could consider; however, an extended C compiler is under development. Our current benchmarks cover the areas of video

Table II. Benchmark Description

Benchmark	Description	Parameters
De-blocking filter	Filter to smooth the sharp edges	16*16MB 640*480 frame
Black_Jack	Simulation of blackjack card game	52-card deck per data path
Collision Detect	Intersection detection for 2D objects	1,024 objects
Line Drawing	Bresenham's algorithm	Average 10-pixel lines
Bitonic	Bitonic sort algorithm	Randomly generated 0, 1 sequence
Odd_Even	Odd-even mergesort algorithm	Average sequence length 256
BFS	Breadth-first search	1 million vertex random graph, degree: 6

Fig. 9. Overall performance gain.

processing [Suhring 2015], physics simulation [OPCODE 2003], and computer graphics [He et al. 2006]. A detailed description is provided in Table II.

5. PERFORMANCE EVALUATION

Now we present the performance gains from our experiments; GI serves as the baseline performance. To fully understand the performance benefit of IISLP, comparison with IISLP with horizontal permutation (IISLP+Perm), IS, DWF-like, and MIMD architecture are carried out. Moreover, we evaluate the effect of varying the number of RF sets. The impact on memory systems is also analyzed.

5.1. Overall Performance Gain

Figure 9 shows the performance gain of IISLP, IISLP+Perm, IS, DWF-like, and MIMD architecture relative to that of the baseline GI scheme. Overall, IISLP is 28% faster than IS and about two times faster than the GI scheme. Much of its performance benefits are attributed to conquering the sequential execution of different branch paths, which leads to very low SIMD efficiency on baseline SIMD architectures. Although IS can achieve speedups on these benchmarks as well, it also exhibits inefficiency for benchmarks that have unbalanced branch paths (De-blocking, Black_Jack, Bitonic, Odd_Even, and BFS), lowering the overall speedup to 74%. IISLP recovers from most of this inefficiency, especially for benchmarks with a limited number of branch paths (Bitonic, Odd_Even, and BFS). For these applications, the rescheduling result leads to

SV-Iter with the same branch path. Thus, the execution bubbles are reduced as much as possible, leading to high overall efficiency. However, there can still be execution bubbles, which come from the fact that the S type is used to compensate the L-type per-lane iterations when both flag bits in the status table are 0.

For applications with an intensive number of branch paths (Black_Jack and De-blocking), barrier suspension points are inserted due to the nested branch structure; branch paths merging is also adopted to fit the number of instruction buffers. This can change the original balance among branch paths, leading to relatively balanced structure with small performance gain. Benchmarks that have a balanced branch paths structure (Collision Detect and Line Drawing) are not significantly affected by fragment starvation, and hence we do not anticipate significant benefits from IISLP. In fact, for those original balanced benchmarks, the schedule unit in IISLP can be set to be bypassed.

As Line Drawing, Bitonic, and Odd_Even contain relatively regular branch resolution patterns, the performance of IISLP+Perm surpasses that of IISLP. This is because the horizontal permutation within each VI breaks the branch resolution, resulting in a more balanced distribution pattern that provides better rescheduling resources for the formation of an S-VI. On average, 20% of S-VIs are transformed from mixed type to single type, greatly reducing execution bubbles. For other applications with a relatively balanced branch resolution distribution, IISLP+Perm does not show obvious performance gains. Collision Detect exhibits even worse performance due to those extra swizzle operations. Although the combination with horizontal permutation can be done flexibly at software level, the benefit of this combination depends on the branch resolution characteristic of applications, which can be pre-exploited by programmers.

As shown in Figure 9, the performance of the DWF-like scheme is lower than IISLP for all applications while surpassing IS for Bitonic, Odd_Even, and BFS. This result seems a little contradictory to one's intuition, because theoretically, DWF-like regroups only identical branch paths together so that pipeline bubbles can be reduced as much as possible, and the fragment starvation pathology should be extinct. However, after an in-depth exploration of our experiment, we find that different from GPU architectures, in which a large amount of VI (called *warps* in the GPU) contexts are maintained, our SIMD architecture only holds the context of four VIs. This makes it difficult to group enough per-lane iterations under identical branch paths into a full VI, causing wasted SIMD lanes in newly formed VIs. Thus, the potential of DWF is greatly limited.

This is why applications (De-blocking and Black_Jack) with intensive branch path structures show much lower performance than that of IS. The same reason stands for balanced branch path applications (Collision Detect and Line Drawing). For applications (Bitonic, Odd_Even, and BFS) with a limited number of unbalanced branch paths, pipeline bubbles caused by wasted SIMD lanes become smaller than the one caused by fragment starvation, which is why DWF-like surpasses IS. However, the performance of DWF-like is still lower than IISLP, because in circumstances where the number of per-lane iterations under the same type is not enough to form a full VI, IISLP not only combines the same type of branch paths but also different types of branch paths. Consequently, more pipeline bubbles can be reduced.

Compared to our baseline architecture, the large amount of warp contexts in GPUs provides much more grouping resources for DWF. However, recent research [Rhu and Erez 2013b] has pointed out that even in GPUs, the potential of TBC (an advanced version of DWF) is still limited due to the insufficient amount of qualified threads. Whereas Rhu's method broke such a limitation by horizontal permutation of threads in each warp (VI), the IISLP serves as an orthogonal solution by vertically interleaving the execution of threads among different warps. This can also be adopted in GPUs.

Fig. 10. The effect of increasing RFs.

MIMD achieves the highest performance gains due to its flexibility in branch divergence handling, whereas IISLP greatly approaches the performance of the MIMD architecture. The performance gap between IISLP and MIMD lies in the fact that rescheduling in IISLP can only compact branch paths of the same type, and execution bubbles still exist within the each type. In addition, it can happen that S-type branch paths are used to complement L-type branch paths when the flag bit of both status table rows are 0. This is the case for applications with a limited number of branch paths (Bitonic, Odd_Even, and BFS). Although the performance of IISLP architecture is relatively a bit lower than that of the MIMD one, IISLP maintains the major efficiency of SIMD architecture in Instruction Fetch, Decode, and Issue (IFDI). The main hardware cost of IISLP includes the IBA and ISU, whereas the MIMD architecture needs another 15 IFDI sets. In our current prototype design, the IFDI part consumes about 3.87% of the overall SIMD core area. Thus, 15 IFDI sets can be quite costly. Moreover, instead of four instruction buffers in the IBA, the MIMD architecture also needs 16 instruction caches, which adds additional hardware cost and scalability problems.

5.2. The Effect of Increasing RFs

To provide a zero overhead context switch between VIs, a large number of RF sets are being adopted in SIMD architectures like GPUs. In fact, SIMD architectures tend to integrate even more RFs [Jing et al. 2013]. To understand the benefit of IISLP for a larger number of RF sets, we have varied the number of RF sets in each SIMD lane from 4 to 16. The corresponding performance is shown in Figure 10.

With an increased number of RFs, Bitonic, Odd_Even, and BFS achieve higher performance due to the increasing number of suspended VIs, which provide better reschedule opportunities to form more balanced S-VIs. Collision Detect and Line Drawing do not show noticeable performance gains due to their original balanced characteristic. For branch path-intensive applications (De-blocking and Black_Jack), each branch type (L and S) contains multiple branch paths. This helps to increase the occurrences of both branch types in each VI, which in turn provides more reschedule candidates with even a small number of RFs. Thus, increasing the number of RFs does not show significant performance gain. For these applications, classifying the branch paths into more fine-grained types can be a possible way to increase performance when more RFs are adopted. However, this will also introduce additional hardware cost in the schedule unit. Overall, with a larger number of RF sets, better performance can be expected.

5.3. Impact on the Memory System

Although IISLP itself does not disturb the memory-coalescing capability, the formation of S-VIs interleaves the execution of per-lane iterations from different VIs. This can degrade memory access behavior. Among the evaluated benchmarks, those exhibiting a substantial increase in performance show a noticeable increase in accesses to the scratchpad memory, with an average increase of about 1.8 times. Fortunately, the increase comes mainly from the unaligned store operations, as the alignment of load operations is well maintained. Unlike the load operations, which can stall the pipeline, the store operations are done in a back-end manner. Thus, extra cost can always be concealed in the computation process. The impact on overall performance is negligible. However, the unaligned store operations can have a side effect on the overall power consumption. Proposing a write buffer to realign the store operations can help, and we leave this to our future work.

6. RELATED WORK

This work has improved the IS mechanism [Wang et al. 2012; Wang et al. 2013] by interleaving the execution of consecutive VIs so that SIMD lanes can be dynamically partitioned into branch paths with comparable length. Related techniques are VLT [Rivoire et al. 2006] and VT [Krashinsky et al. 2004]. VLT partitions SIMD lanes statically, and it is not able to dynamically adjust the partition according to branch divergence, which is one of the major causes of SIMD performance limitations. VT added separate instruction caches to each SIMD lane, guaranteeing concurrent execution of different SIMD fragments. However, the scalar control processor can become a bottleneck of instruction bandwidth in VT when different instructions are demanded by SIMD lanes. Another drawback of this approach is the scalability problem and significant area overhead caused by the per-lane instruction cache structure. In fact, the successor [Lee et al. 2011] of the VT architecture has removed such per-lane cache structure.

Beside the preceding two schemes, there are several approaches for the branch divergence problem. These approaches can be divided into two categories. One is the branch merging execution scheme, which lets SIMD lanes under different control flows run sequentially after the branch divergence point and then merge together to run concurrently after the branch convergence point. Examples include the MAVEN [Lee et al. 2011] architecture, which adopts a pending vector fragment buffer (PVFB). Each time after the branch divergence point, SIMD lanes under the fall-through path continue their execution while SIMD lanes under the taken path are suspended into the PVFB and executed later. The branch merging scheme improves the performance of the GI scheme [Bouknight et al. 1972]. However, SIMD lanes are still underutilized between the divergence and convergence points of application kernels.

Another is the branch paths interleaving scheme, which enables concurrent scheduling of SIMD lanes under different branch paths so that during cycles when SIMD lanes under one branch path are waiting for long latency events, other SIMD lanes can still make progress. DWS [Meng et al. 2010] interleaves the execution of SIMD lanes under two branch paths while taking into account the divergence in SIMD memory access latencies. This results in improved memory latency hiding and memory-level parallelism. The dual-path model [Rhu and Erez 2013b] improves DWS with optimal reconvergence of branch paths at the immediate postdominator [Fung et al. 2007], achieving a high SIMD lane utilization rate. The multipath [EITantawy et al. 2014] scheme further extends the dual-path model by interleaving the execution of SIMD lanes under multiple branch paths while the optimal reconvergence is still well maintained. This is fulfilled with two separate tables for tracking the execution of multiple branch paths and their reconvergence points separately. Although subsets of SIMD lanes under different branch paths can be scheduled concurrently, only one

subset under the same branch path can be executed at a time, whereas all SIMD lanes under different branch paths can be executed concurrently in IISLP.

The third approach is the dynamic regrouping method, which regroupes datasets or control flows at runtime so that the newly formed group of datasets or control flows become suitable to be executed on SIMD architectures. Conditional stream operations [Kapasi et al. 2000] regroup the datasets into subsets with identical control flow. Thus, a kernel with divergent control flow is broken into multiple kernels connected with interkernel buffers. Each newly formed kernel has regular control flows. DWF [Fung et al. 2007, 2009], TBC [Fung et al. 2011], LWM [Narasiman et al. 2011], and CAPRI [Rhu and Erez 2012] adopt the idea of regrouping control flows, where threads from different warps after the branch divergence point are regrouped. Thus, the newly formed warps are suitable for executing on SIMD architectures. The key difference between IISLP and the regrouping method is that we regroup similar control flows together rather than identical ones. This helps to main a high regrouping efficiency with reasonable hardware cost. The SIMD lane permutation scheme [Rhu and Erez 2013a] improved the regrouping efficiency of TBC by interleaving the execution of SIMD fragments in a horizontal direction, which is orthogonal to our approach, as we interleaved the execution of SIMD fragments in a vertical direction.

7. CONCLUSION

This article proposes an IISLP architecture that interleaves the execution of consecutive VIs after branch divergence so that SIMD fragments with comparable branch path length can be dynamically rescheduled to form S-VIs. Compared to original VIs, these synthetic ones can substantially reduce pipeline bubbles. We find that our proposal addresses some key challenges of IS [Wang et al. 2012, 2013] while maintaining its ability in dynamic SIMD lane partition upon divergent branches. Our evaluation quantifies that IISLP achieves an overall 28% speedup over the IS mechanism for a set of divergent applications. Comparisons with DWF and MIMD are also carried out. In the future, we plan to investigate the realigning of store operations. Besides that, more comprehensive compiler support will also be investigated for the IISLP architecture.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Chengkun Wu, Kai Zhang, and Xi Ning of National University of Defense Technology for providing valuable suggestions about the microarchitecture.

REFERENCES

- W. Bouknight, S. Denenberg, D. McIntyre, J. Randall, A. Sameh, and D. Slotnick. 1972. The Illiac IV system. *Proceedings of the IEEE* 60, 4, 369–388.
- S. Chen, Y. Wang, S. Liu, J. Wan, H. Chen, H. Liu, K. Zhang, X. Liu, and X. Ning. 2014. FT-Matrix: A coordination-aware architecture for signal processing. *IEEE MICRO* 34, 6, 64–73.
- T. Chen, R. Raghavan, and J. Dale. 2007. Cell broadband engine architecture and its first implementation a performance view. *IBM Journal of Research and Development* 51, 559–572.
- A. ElTantawy, J. Wenjie Ma, M. O'Connor, and T. M. Aamodt. 2014. A scalable multi-path microarchitecture for efficient GPU control flow. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computing Architecture (HPCA-20)*. 248–259.
- W. W. L. Fung and T. M. Aamodt. 2011. Thread block compaction for efficient SIMT control flow. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*. 25–36.
- W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. *IEEE MICRO* 40, 407–420.
- W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. 2009. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM Transactions on Architecture and Code Optimization* 6, 2, Article No. 7.
- Q. He. 2006. *The Principle of the Computer Graphics*. Tsinghua University Press, Beijing, China.

- N. Jing, Y. Shen, Y. Lu, S. Ganapathy, Z. Mao, M. Guo, R. Canal, and X. Liang. 2013. An energy-efficient and scalable eDRAM-based register file architecture for GPGPU. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. 344–355.
- U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany. 2000. Efficient conditional operations for data-parallel architectures. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-33)*. 159–170.
- B. Khailany, W. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, and A. Chang. 2001. Imagine: Media processing with streams. *IEEE Micro* 21, 2, 35–46.
- B. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casner, and K. Asanovic. 2004. The vector-thread architecture. *IEEE Micro* 24, 6, 84–90.
- Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanovic. 2011. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. 129–140.
- J. Meng, D. Tarjan, and K. Skadron. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA'10)*. 235–246.
- V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. 308–317.
- NVIDIA Corporation. 2008. GeForce Gtx 280 Specifications. Available at <http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-280/specifications>.
- NVIDIA Corporation. 2009. Nvidia's Next Generation CUDA Compute Architecture: Fermi. Available at <http://www.nvidia.com>.
- NVIDIA Corporation. 2012. Nvidia's Next Generation CUDA Compute Architecture: Kepler GK110. Available at <http://www.nvidia.com>.
- OPCODE. 2003. OPCODE Optimized Collision Detection Library (OPCODE). Retrieved December 7, 2015, from <http://www.codercorner.com/Opcode.htm>.
- M. Rhu and M. Erez. 2012. CAPRI: Prediction of compaction-adequacy for handling control-divergence in GPGPU architectures. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA'12)*.
- M. Rhu and M. Erez. 2013a. Maximizing SIMD resource utilization in GPGPUs with SIMD lane permutation. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. 356–367.
- M. Rhu and M. Erez. 2013b. The dual-path execution model for efficient GPU control flow. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. 591–602.
- S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis. 2006. Vector lane threading. In *Proceedings of the International Conference on Parallel Processing (ICPP'06)*. 55–64.
- L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. 2008. Larrabee: A many-core x86 architecture for visual computing. In *Proceedings of ACM SIGGRAPH 2008 Papers (SIGGRAPH'08)*. 1–15.
- K. Suhring. 2015 H.264 Joint Model (JM)-h.264/AVC Reference Software. Available at <http://iphome.hhi.de/suehring/ttml/>.
- Y. Wang, S. Chen, J. Wan, J. Meng, K. Zhang, W. Liu, and X. Ning. 2013. A multiple SIMD, multiple data (MSMD) architecture: Parallel execution of dynamic and static SIMD fragments. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. 603–614.
- Y. Wang, S. Chen, K. Zhang, J. Wan, X. Chen, H. Chen, and H. Wang. 2012. Instruction shuffle: Achieving MIMD-like performance on SIMD architectures. *IEEE Computer Architecture Letters* 11, 2, 37–40.
- M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. 2010. AnySP: Anytime anywhere anyway signal processing. *IEEE Micro* 30, 1, 81–91.
- X. Yang, X. Yan, Z. Xing, Y. Deng, J. Jiang, and Y. Zhang. 2007. A 64-bit stream processor architecture for scientific applications. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. 210–219.

Received July 2015; revised October 2015; accepted November 2015