A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Sy
vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

Predictable GPU Wavefront Splitting for Safety-Critical Systems

ARTEM KLASHTORNY, University of Waterloo, Canada
ZHUANHAO WU, University of Waterloo, Canada
ANIRUDH MOHAN KAUSHIK, Intel of Canada, Canada
HIREN PATEL, University of Waterloo, Canada

We present a predictable wavefront splitting (PWS) technique for graphics processing units (GPUs). PWS improves the performance of GPU applications by reducing the impact of branch divergence while ensuring that worst-case execution time (WCET) estimates can be computed. This makes PWS an appropriate technique to use in safety-critical applications, such as autonomous driving systems, avionics, and space, that require strict temporal guarantees. In developing PWS on an AMD-based GPU, we propose microarchitectural enhancements to the GPU, and a compiler pass that eliminates branch serializations to reduce the WCET of a wavefront. Our analysis of PWS exhibits a performance improvement of 11% over existing architectures with a lower WCET than prior works in wavefront splitting.

CCS Concepts: • **Computer systems organization → Real-time system architecture**.

Additional Key Words and Phrases: GPU, safety-critical systems

ACM Reference Format:
Artem Klashtorny, Zhuanhao Wu, Anirudh Mohan Kaushik, and Hiren Patel. 2023. Predictable GPU Wavefront Splitting for Safety-Critical Systems. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2023), 25 pages. https://doi.org/10.1145/3609102

1 INTRODUCTION

Graphics processing units (GPUs) are becoming increasingly common computing platforms used in safety-critical systems. GPUs offer high performance for workloads such as vision processing and machine-learning algorithms. Self-driving automotive and advanced assisted driving systems are examples that employ GPUs in their deployments to process these algorithms. Existing compute platforms such as Tesla's Autopilot system-on-chip (SoC) [5] and NVIDIA's Jetson SoC [4] exploit the GPU for assisted driving features.

Employing GPUs in safety-critical systems continues to be a difficult challenge. This is because safety-critical systems require certain tasks to be guaranteed to meet certain temporal requirements [21], and providing these guarantees is extremely difficult given modern GPU microarchitectures. The temporal guarantees are typically computed via analysis tools that produce the

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES), 2023

Authors' addresses: Artem Klashtorny, aklashto@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, Ontario, Canada, N2L 3G1; Zhuanhao Wu, z284wu@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, Ontario, Canada, N2L 3G1; Anirudh Mohan Kaushik, anirudh.kaushik@intel.com, Intel of Canada, Toronto, Ontario, Canada; Hiren Patel, hdpatel@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, Ontario, Canada, N2L 3G1.

1539-9087/2023/1-ART1 $15.00
https://doi.org/10.1145/3609102

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

worst-case execution time (WCET) the kernel takes to execute. Producing such analyses requires a detailed understanding of the GPU microarchitecture and its delays. Since most commercial off-the-shelf (COTS) GPUs are poorly documented or closed-source, developing such WCET analysis tools is difficult [16, 21]. To address this issue, there have been efforts to reverse-engineer COTS GPUs and to propose techniques that make them more suitable for safety-critical systems [4, 6, 12, 16]. Others have proposed new GPU architectures that are specifically designed to support static WCET analysis [17, 20].

Despite the progress of GPU research for safety-critical applications, these works do not address the impact of branch divergence on the WCET. Normally, a GPU executes multiple threads, known as work-items, grouped into units called wavefronts. Work-items in a wavefront execute the same instruction simultaneously in lockstep on vectorized data. If the GPU program has conditional logic, then work-items must execute distinct sets of instructions at runtime [1]. To avoid breaking up the lockstep execution of work-items, GPUs serialize the execution of the branching paths, reducing the performance of the GPU. In the worst-case, all branches that are data-dependent can diverge at runtime. Thus, branch divergence also present a significant challenge for safety-critical systems. Prior works propose techniques that address branch divergence, including dynamic wavefront splitting [9, 10, 15, 18], in which the GPU breaks up the lockstep wavefront execution to avoid serialization. However, these techniques target improving average-case performance rather than reducing the WCET.

In this work, we present a GPU architecture and a compiler pass that together allow for predictable wavefront splitting (PWS). There are three key challenges this work addresses. The first challenge is identifying when to perform wavefront split operations to compute a low WCET. The second challenge is scheduling the execution of the split wavefronts (SWFs). The third challenge involves providing provable lower WCETs compared to existing GPU architectures and prior works through compiler adjustments and WCET analysis. Our novel contributions are as follows.

(1) We introduce two new GPU ISA instructions, `split` and `merge`, which allow the GPU programmer to indicate when the GPU will split wavefronts.
(2) We propose a GPU microarchitecture that adds execution hardware that is explicitly used for split wavefronts. This guarantees that split wavefronts can execute in parallel.
(3) We propose a compiler pass that prunes control-flow graphs to eliminate branch serialization and reduce the WCET of a wavefront.
(4) We extend an existing WCET analysis to support PWS, and our results show that the WCET of a GPU kernel can be lowered.

The remainder of this paper is organized as follows. Section 2 introduces the relevant background on GPU hardware and the corresponding programming model. Section 3 identifies the problems with existing implementations of wavefront splitting in safety-critical systems and motivates the need for the main contributions of PWS. Section 4 illustrates the architectural details that enable contributions (1) to (3). Section 5 presents a WCET analysis of a GPU kernel executing using PWS and compares it to prior techniques. Section 6 outlines the related works in wavefront splitting and the use of GPUs in safety-critical systems. Section 7 demonstrates the benefits of PWS using data collected from a set of benchmarks executed using a GPU simulator.

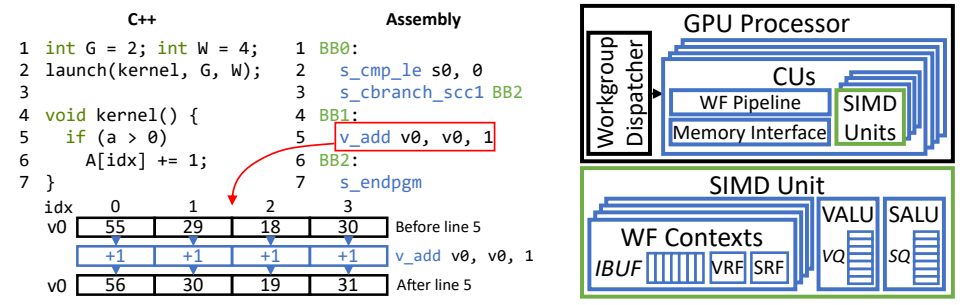
2 BACKGROUND

The GPU has a programming model component and a hardware component, both of which are fundamental to our work. We describe both in this section.

Programming model. A GPU executes a program called a kernel. The kernel implements multi-threaded functions in which each thread is called a *work-item* (WI). Work-items execute each kernel

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

instruction in parallel on distinct data values. This is known as single instruction multiple data (SIMD) execution [1]. Figure 1a demonstrates SIMD execution using a simple kernel that performs addition. We present the kernel as both C++ and assembly code in Figure 1a, where `idx` is the work-item index. For simplicity, we assume data from variable `a` is stored in register `s0` and the data `A[idx]` from array `A` is stored in register `v0`. Notice that all work-items perform the same addition operation on the elements of the vector register `v0` simultaneously.



(a) A GPU kernel and associated execution of a vector instruction.

(b) Our GPU processor model.

Fig. 1. A very simple example GPU kernel execution and the GPU hardware model.

When deploying a kernel for execution on the GPU, the programmer specifies N_{WIWG}, the number of work-items grouped into a unit called a *workgroup* (WG), as well as the total number of workgroups, G. Figure 1a uses the variables `G` and `W` to represent these parameters. All WIs within a WG share execution resources such as synchronization barriers and scratchpad memory, which is called the local data share (LDS). The GPU may not have enough hardware resources to execute all work-items specified by the programmer in parallel. Thus, it dispatches the workgroups by dividing them into units of N_{WIWF} work-items called a *wavefront* (WF). The GPU performs scheduling and SIMD execution using these wavefronts.

Hardware model. We base our GPU's hardware model on AMD's GPU implementation. We show the most relevant parts of the hardware in Figure 1b [2, 3].

Compute units. A compute unit (CU) is a GPU hardware unit that has N_{SIMD} SIMD vector execution units, a wavefront pipeline, and a memory interface. We assume the GPU contains N_{CU} CUs.

Workgroup dispatcher. When the GPU receives a request to execute a kernel, the workgroup dispatcher distributes the workgroups to the CUs. Then, the dispatcher divides the work-items in the workgroup into wavefronts on the selected CU. Note that multiple wavefronts may be required to completely execute all work-items in a workgroup. The dispatcher continues dispatching workgroups to CUs until there are no more CUs remaining with enough resources to execute additional wavefronts. Wavefronts from the same workgroup are guaranteed to be dispatched to the same CU. These wavefronts can synchronize with each other and share data.

Wavefront context. The CU maintains a state for each of its wavefronts, which we call a *wavefront context* (WF context). The WF context consists of a vector register file (VRF), a scalar register file (SRF), and an instruction buffer (IBUF). Each register in the VRF is a vector in which each element, or *lane*, corresponds to a single work-item index, similar to the example vector register in

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

Figure 1a. Each wavefront also has a special register called the *execute mask*. The mask dictates the work-items in a wavefront that execute a given instruction. Each bit of the register corresponds to a work-item index. If the bit is 1, the work-item is active and executes the given instruction; if it is 0, the work-item is dormant and executes a `NOP`. The wavefront uses the VRF to manage data and status individually for each work-item index in conjunction with the VALU in the WF unit. Similarly, the wavefront uses the SRF for single operations for the whole wavefront. The scalar instructions in Figure 1a show the use of scalar registers for constants and the results of comparisons.

Wavefront scheduling. The CU wavefront pipeline is responsible for fetching and decoding instructions for WF contexts and for scheduling WF execution on the SIMD units. The wavefront pipeline will fetch and decode instructions within each WF context's instruction buffers. If the instruction at the front of the buffer is free of dependencies, the CU schedules the WF for execution. It adds any vector instructions to a queue for the VALU (VQ) and scalar instructions to a queue for the SALU (SQ). If the SIMD executes a memory instruction, it will compute the address and then the CU adds the memory request to a queue for the memory interface.

SIMD units. A SIMD unit enables lockstep SIMD execution of a wavefront. A SIMD unit consists of a vector ALU (VALU) and a scalar ALU (SALU). The VALU has N_{WIWF} parallel ALUs that allow for N_{WIWF} parallel integer or floating point operations to occur. It enables SIMD operation of a wavefront on vectorized data by executing instructions in parallel across all work-items, as shown in Figure 1a with the `v_add` instruction. Conversely, the SALU performs operations that are not vector instructions, such as unconditional branches and synchronization barriers. In the simple kernel from Figure 1a, the assembly instructions with the prefix `s_` correspond to instructions that run on the SALU. Each SIMD contains a set of N_{WFC} WF contexts; WF contexts execute exclusively on their own SIMD.

3 MOTIVATION

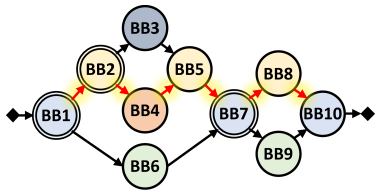
We motivate the need for a predictable wavefront splitting (PWS) GPU by starting with explaining the impact of branch divergence in GPUs on predictability, and briefly reviewing the pertinent state-of-the-art approaches and the issues that they pose for predictability. Then, we describe our insights in designing PWS.

Branch divergence. Branch divergence occurs when conditional statements in the kernel cause the control flow of the work-items within a wavefront to execute distinct paths. This results in significant performance degradation [11]. Branch divergence also lengthens the WCET of a kernel because it serializes the execution of the divergent paths. Given that real-world workloads are subject to branch divergent instructions, there have been several solutions that address the issue of branch divergence [11, 15]. All prior efforts primarily focus on improving average-case performance. In this work, we believe we are the first to investigate the predictability of such solutions.

Consider the example kernel in Figure 2b that has four work-items with indices zero to three. The program accesses this index using `idx` in the conditions of the `if` statements. The kernel's control-flow graph (CFG) in Figure 2a has vertices that are basic blocks (denoted *BBx*). The edges in the CFG represent the possible work-item execution orderings between the basic blocks. We show the value of array `A` in Figure 2b.

Figure 2c shows the execution of a wavefront. All four WIs in the wavefront execute *BB*1 at time 0. Note that solid arrows indicate active WIs and empty arrows indicate inactive or stalled WIs. The `if` condition reads contents of array `A` by using each WI's respective index to determine the next basic block to execute. WIs whose `A[idx]` value is greater than 25 execute *BB*2 to *BB*5 (yellow

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

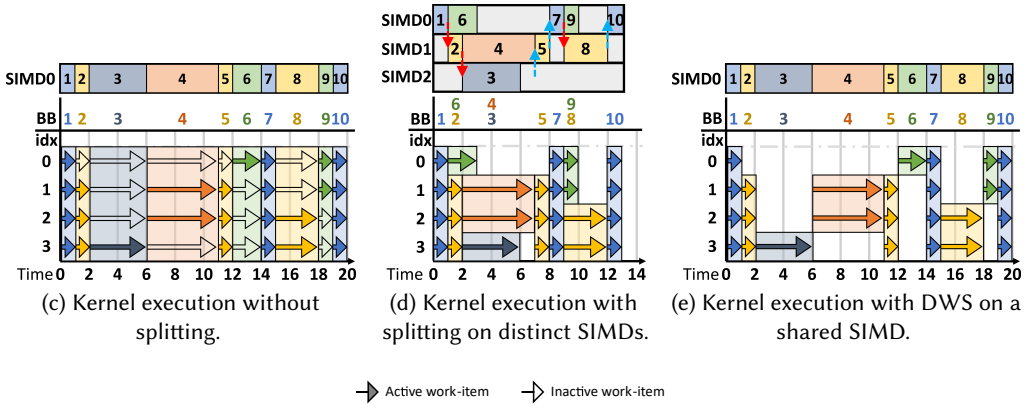


BB	1	2	3	4	5	6	7	8	9	10
WCET	1	1	4	5	1	2	1	3	1	1

(a) Kernel CFG and associated WCETs for each BB.

```
1  // A = {17, 29, 52, 80};
2  // BB1
3  if (A[idx] > 25) {
4      // BB2
5      if (A[idx] > 75) {
6          // BB3
7      } else {
8          // BB4
9      }
10     // BB5
11 } else {
12     // BB6
13 }
14 // BB7
15 if (A[idx] > 50) {
16     // BB8
17 } else {
18     // BB9
19 }
20 // BB10
```

(b) C++ code representing the kernel.



(c) Kernel execution without splitting.

(d) Kernel execution with splitting on distinct SIMDs.

(e) Kernel execution with DWS on a shared SIMD.

Fig. 2. Control-flow graph depicting branch divergence with nesting and sequences of branches with corresponding C++ code, with associated execution flow of the kernel. Execution of WFs and SWFs is temporally depicted on the timeline with the hardware allocation to SIMD units below.

in Figure 2a) and all other work-items execute *BB*6 (green in Figure 2a). Hence, work-items with indices 1, 2, and 3 execute *BB*2 to *BB*5, and 0 executes *BB*6. The SIMD execution handles each instance of branch divergence by executing both paths serially. Hence, at time 1, WIs 1, 2, and 3 execute (solid arrow), while WI 0 is stalled. The WIs executing *BB*2 encounter another branch where WIs with index 3 execute *BB*3 (in navy blue) and WIs with indices 1 and 2 execute *BB*4 (in orange). Then, WIs 1 to 3 all become active again and execute *BB*5. Upon completion of *BB*5, WI 0 executes *BB*6, while all other WIs are stalled. Next, all WIs execute *BB*7. In a similar behaviour to the earlier branches, the WIs take different paths through *BB*8 and *BB*9 based on whether the data value is greater than 50. Finally, all WIs execute *BB*10. Note that the actual path executed depends on the work-item index and the input data. Notice that the entire execution uses only one SIMD, and divergent paths have to serially use SIMD0.

Dynamic wavefront splitting. One approach to address branch divergence proposed by recent works is by using *dynamic wavefront splitting* (DWS) [9, 10, 15, 18]. DWS divides work-items that take divergent paths into separate schedulable wavefronts. We call these split-wavefronts (SWFs). The DWS implementations in all prior works identify, create, and execute SWFs dynamically at run-time. For a diverging branch, the wavefront splits into two where the work-items executing one path form one SWF, and those executing the other path form another SWF. For the example

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

execution in Figure 2d, at time 1, the `if` statement diverges resulting in two SWFs: one executes *BB*2 on work-items with indices 1 through 3 and the second SWF executes *BB*6 on work-item 0. These SWFs can execute in parallel given the availability of SIMD units. *BB*2 executes on SIMD0 and *BB*6 on SIMD1. As the execution continues and completes, we note that it takes 13 units of time to complete the wavefront's execution using DWS (as shown in Figure 2d), which is less than the 20 cycles required to complete the execution without splitting (shown in Figure 2c).

Issues in adopting DWS for safety-critical systems. Prior works that deployed DWS only concentrated on improving average-case performance [9, 10, 15, 18]. To the best of our knowledge, there are no efforts in evaluating DWS for safety-critical systems. Safety-critical systems have the additional requirement that the deployed applications meet safety certification standards such as those in [8, 14]. A central requirement in these standards is to produce worst-case execution time (WCET) estimates for critical tasks. For our illustrative example, we assume that the WCET of each basic block is known as shown in the table in Figure 2a. Hence, the WCET of the kernel on the original GPU (without DWS) in Figure 2c is 20; simply the sum of all basic block WCETs. This is because all basic blocks execute and no two blocks can execute in parallel.

With DWS, it would appear that the latency for kernel in Figure 2 is 13, as seen in Figure 2d. However, the WCET is larger, and ends up being 20, as shown in Figure 2e. We identify two main issues that cause the WCET of DWS to be the same as the original architecture without splitting.

Issue 1: Run-time decision on WF splitting of branches. The DWS hardware determines whether a branch diverges at run-time. Hence, it is possible that all branches, no branches, or some number in between, in the kernel diverge. The WCET analysis needs to assume that all branches diverge, and that each wavefront executing the branch creates an SWF. This is because whether a branch diverges is a run-time property and multiple executions of the kernel may trigger different branches to diverge. Since DWS creates a separate schedulable SWF on a divergent branch, which we call a *split point*, the newly created SWF may suffer interference from all other WFs in the WG. This is the key reason the WCET increases. For the example in Figure 2a, suppose that the GPU must select only one split point on either *BB*1 or *BB*2. If it splits the wavefront at *BB*1, then *BB*3 and *BB*4 must be serialized. If it splits at *BB*2, then *BB*6 must be serialized with *BB*2 to *BB*5. Since *BB*6 has a lower WCET than *BB*3 or *BB*4, there is a significant timing penalty when choosing to split at *BB*1. With DWS, the GPU selects at run-time where to split; hence, the WCET of the WF must account for the split point that incurs the largest execution latency.

Solution 1: Statically specify branches to create SWFs. We introduce instructions in the instruction-set architecture (ISA) that allows a GPU programmer to specify the branches to trigger the creation of SWFs. This enables WCET analyses to statically assess occupancy of SWFs. Note that selecting the branches that result in the lowest WCET is in itself a problem that we do not explore in this paper, and reserve it for future work. By allowing the GPU programmer to statically mark branches for splitting, we can guarantee the selection of split points. When the GPU splits a wavefront, each SWF takes one of two paths of a branch. Since we can determine statically which branch nodes in the CFG will be split points, we can predict branch paths each SWF will not take. To aid the WCET analysis, we present a compiler pass that prunes the kernel CFG to correspond to the worst-case path an SWF can execute. This reduces the WCET of a WI.

Issue 2: SIMD units are shared between SWFs and WFs. Prior works use the same SIMD units to execute SWFs; making no distinction between an SWF and WF executing on a SIMD unit. While this allows sharing of SIMD units across SWFs and WFs, it results in a large WCET. Consider encountering a divergent branch with no available SIMD units to execute the corresponding SWF.

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

This results in serially executing the SWF, as in Figure 2c. One prior work executes SWFs in parallel on distinct execution units [9]; however, this work is limited to a maximum of two SWFs in parallel and limits the capacity for wavefronts. Consequently, SWFs must compete for the same WF contexts as wavefronts that have not split. A WCET analysis of this execution must consider the worst-case, which is the serial execution of all SWFs and a reduced simultaneous capacity for wavefronts on SIMD units.

Solution 2: Dedicated SIMD units. We propose providing hardware compute units dedicated to the execution of SWFs called *split SIMD units* (SpSIMD units or SpSIMDs). These SpSIMDs are available in addition to the existing SIMD units of the GPU, but they are reserved only for SWF execution. At a divergent branch, one SWF executes on the SpSIMD while the other path executes on the original hardware unit. These solutions to the challenges comprise our architectural implementation, which we call *predictable wavefront splitting* (PWS).

4 PWS DESIGN

PWS proposes architectural modifications to the GPU hardware and the ISA. PWS also uses a compiler pass to transform the kernel CFG into a version for PWS. This CFG allows us to reduce the WCET for each SWF executing the kernel. We discuss these architectural modifications next.

4.1 Architectural modifications for PWS

We add two key architectural features to enable PWS: (1) an instruction that allows GPU programmers to specify split points in kernels, and (2) a set of dedicated SpSIMD units for SWFs to execute in parallel.

Statically specifying split points for SWFs. We extend the GPU ISA with two instructions that allow the programmer to statically select split and merge points within the kernel, which we call `split` and `merge`. The `split` instruction informs the GPU to split a wavefront into two SWFs. The `merge` instruction indicates that two SWFs should merge back into a single wavefront. We present the encoding for these instructions in Figure 3a.

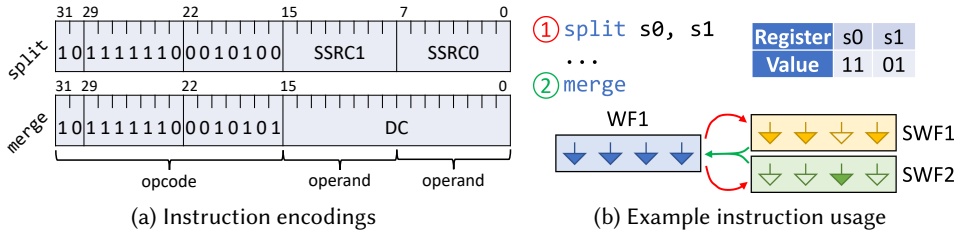


Fig. 3. `split` and `merge` instruction encodings

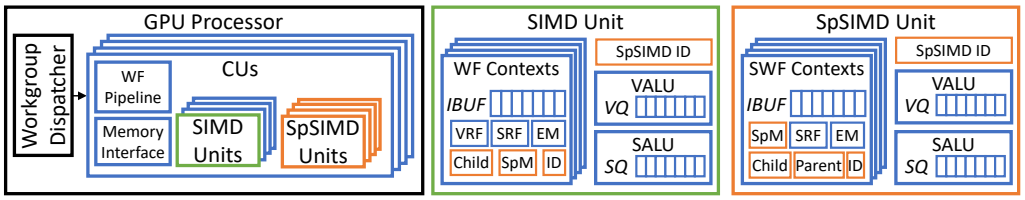
The `split` and `merge` instructions are 32 bits each. For `split`, bits 0 to 7 and 8 to 15 represent two scalar register source operands. Bits 16 to 31 represent the instruction opcode. The scalar registers are each 32 bits wide; we use each bit to indicate whether a work-item is active or inactive in the resulting SWFs. Having two source operands allows PWS to support wavefronts with $N_{WIWF} \leq 64$, an upper bound which supports all recent AMD GPU architectures [2, 3]. The `merge` instruction also consists of 16 opcode bits, while the remaining bits are unused.

The GPU programmer can insert `split` between any two consecutive instructions in the program. During execution, a `split` before a branch instruction causes PWS to split the WF into two SWFs. A

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

`split` in other locations is removed by our compiler. Figure 3b shows an example using `split`. In the example, we assume that each wavefront has four work-items and scalar registers are two bits each. We show how the GPU combines the two 2-bit scalar registers to track the active work-items in a wavefront of 4 work-items. When the wavefront executes `split`, it uses `s0` and `s1` to indicate the work-items in the resulting SWFs. Notice that `s0` corresponds to whether the first two work-items are in SWF1 while `s1` corresponds to the last two work-items. The wavefront sets the work-items in SWF2 based on the negation of these bits. When the two SWFs finish executing each divergent branch, they execute a basic block known as the post-dominator (the join point for the branch). After the post-dominator, PWS merges the two SWFs via the `merge` instruction so that downstream branches could split again and reuse the SWF resources. Our compiler pass automatically inserts these `merge` instructions. In Figure 3b, the merge instruction regroups SWF1 and SWF2 to WF1, in which all work-items are active.

Dedicated hardware for SWFs. In PWS, we add GPU hardware execution resources that are exclusively provisioned for SWFs as shown in Figure 4. We add SpSIMD units, which are SIMD units with added registers to the WF contexts to accommodate SWFs. We call WF contexts on SpSIMD units SWF contexts. This added hardware allows PWS to guarantee parallel execution of SWFs.



(a) GPU processor model (b) SIMD unit model (c) SpSIMD unit model

Fig. 4. Our implementation of PWS hardware

SpSIMD units. For each SIMD unit in the baseline architecture, PWS adds $\mathcal{S}$ SpSIMDs and a uniquely-valued read-only identifier register (SpSIMDID). We permit each wavefront to split up to a maximum of $\mathcal{S}$ times. GPU microarchitects can select $\mathcal{S}$ as a design-time parameter based on their resource constraints.

SWF contexts. Each SpSIMD unit contains an SWF context, which is a modified WF context that enables the GPU to maintain state for SWFs. An SWF context contains a SRF along with new registers called the *split mask* (SpM) and *parent*, as well as a stack of *children*. We also add the SpM and child stack to the WF context; this allows WF contexts to accommodate SWFs. Finally, we add uniquely-valued read-only identifier registers (ID) that number each WF and SWF context to preserve relationships between SWFs.

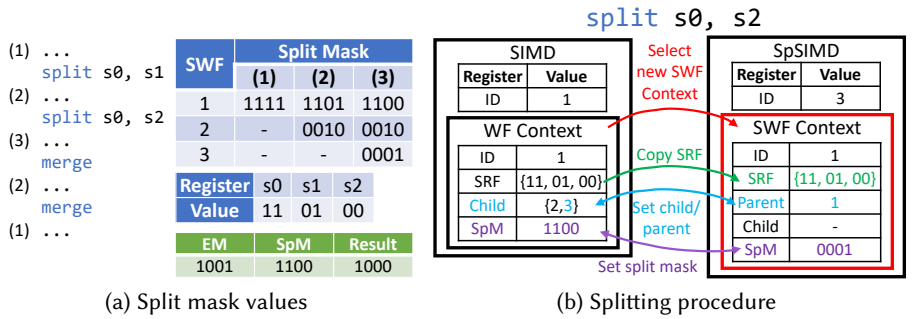
Recall that each wavefront has an execute mask (EM) that specifies the currently active work-items in the wavefront. The execute mask changes each time the wavefront executes a vector comparison for a branch instruction. Unlike the execute mask, the split mask stores the work-items that are included in the SWF and does not change until a subsequent `split` or `merge` instruction. When executing, the wavefront can determine whether a work-item is active if the bits in the execute mask and split mask are both set. We show an example of this behaviour in the green table in Figure 5a. If the execution mask is 1001 and the split mask is 1100, only the first work-item will

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

execute because it is set in both masks. When a wavefront splits into two SWFs, the collections of set bits in each of their split masks are mutually exclusive, as seen in Figure 3b. This means that each work-item will execute on independent SWFs. Thus, the two SWFs can share the same VRF in the SIMD unit, reducing the amount of hardware overhead needed.

Within a SIMD or SpSIMD, the WF or SWF contexts are numbered from 1 to N_{WFC} using the ID register. SWFs with the same ID correspond to the same source wavefront before splitting. We use the parent register and child stack to keep track of this relationship. The parent register represents the SpSIMDID of the wavefront that created the SWF. The top of the child stack represents the SpSIMDID of the most recent SWF that was created by this SWF, if one exists. A single wavefront may split up to $\mathcal{S}$ times; thus, the child stack has a depth of $\mathcal{S}$.

`split` *and* `merge` *semantics in hardware.* Figure 5 shows an example of the split procedure on the PWS hardware. In this example, we assume that the GPU has one SIMD with one WF context and two SpSIMDs with one SWF context each. This configuration allows one wavefront to split up to two times. Figure 5a presents a sample kernel that splits and merges twice. It also presents a table of split masks to supplement the kernel. The table shows the split masks for each of the three SWFs at different points in the kernel execution, which are numbered to correspond to the regions numbered in the kernel code. The next table shows the scalar register values used by the `split` instructions. Figure 5b shows the PWS procedure for the second `split` instruction. The tables in each block show a selection of important registers and their corresponding values. Note that one of the SpSIMDs is omitted from the diagram.



(a) Split mask values (b) Splitting procedure

Fig. 5. A simple example to demonstrate the hardware semantics of the `split` instruction

When a wavefront encounters a `split` instruction, it performs the following procedure. One SWF will occupy the original context, so the GPU allocates one unused SWF context for the second SWF. An SWF context is unused if its parent register has not been set. The SWF context must have the same ID as the original WF context. Notice how in Figure 5b the context ID is 1 for both SWFs. Next, the GPU copies the SRF contents to the new SWF context, sets the parent register, and pushes the ID onto the child stack of the original context. In the example, SWF1 is a parent of SWF3 and SWF3 is a child of SWF1.

Next, the GPU sets the split masks for both SWFs. The split mask of one SWF is set to the bitwise AND of the previous split mask and the split mask provided by the `split` instruction. In the example, we show how SWF1 splits into SWF1 and SWF3 at the second `split`. The split mask of SWF1 starts as 1101 and the split mask provided by the instruction is 1100. Hence, the new split mask of SWF1 is $1101 \wedge 1100 = 1100$. The split mask of the second SWF is set to the bitwise AND

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

of the previous split mask and the negation of the `split` mask. Hence, the new split mask for SWF3 is $1101 \wedge \neg 1100 = 0001$.

Finally, the new SWF context is ready to enter the SpSIMD execution pipeline, at which point the GPU transitions the wavefront to the running state. Effectively, this mechanism ensures that one SWF executes the `if` path and the other SWF executes the `else` path of a branch. Whenever an SWF is ready to execute, all of the SWFs that came from the original parent will execute in parallel on the separate SpSIMD units, unless they are stalled on a memory instruction or synchronization barrier. In Figure 5, SWF1, SWF2, and SWF3 will execute in parallel on their respective SpSIMDs. Note that since an SpSIMD unit also has a child stack, it can also support splitting into further SWFs.

When the SWF encounters the `merge`, it is ready to merge with its parent or child wavefront. If the SWF has child SWFs, then it must merge with the SWF at the top of its child stack; otherwise, the SWF merges with its parent wavefront. First, the GPU stalls execution until the other SWF reaches the merge point. Once both SWFs are synchronized at the same instruction, the GPU merges them. Scalar registers are used for wavefront status, so there is no need to copy these register contents from one SWF to another. The two SWFs share a VRF, so there is nothing to be done to it. The GPU chooses the parent of the SWF contexts as a target for the merged WF or SWF. The target context updates the split mask to be the bitwise OR of the two SWF split masks. In Figure 5a, this behaviour corresponds with moving from column (3) to (2) and (2) to (1). When moving from column (3) to (2), SWF1 and SWF3 merge, resulting in a split mask of $1100 \vee 0001 = 1101$; SWF2 does not change. After the two SWFs merge, the GPU clears the remaining SWF context for reuse by a subsequent split.

4.2 Compiler flow

In this section, we describe our compiler pass that reduces the WCET of the kernel's execution on PWS by pruning kernel CFGs. Figure 6 shows the steps we add to the compiler flow to enable PWS and its analysis. The "Insert `merge`" step corresponds to identifying the branch post-dominators and adding a `merge`. In this step, we also remove any redundant `split` or `merge` instructions.

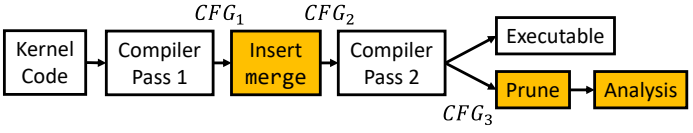


Fig. 6. Block diagram of the compiler flow, with added components highlighted

We use implicit path enumeration (IPET) [22] to compute the WCET of a wavefront, E_{WFi}, where i is an identifier for each wavefront. IPET computes the WCET using an integer linear programming formulation with the execution count and WCETs of the basic blocks from the CFG. Although prior research in using IPET for GPUs [6] can be directly applied to PWS, we find that the computed WCETs are unduly pessimistic. The key reason for this pessimism is that prior approaches do not consider parallel execution of diverging branches, which PWS promotes. Hence, prior analyses compute as though the divergent branches are serialized. We lower the WCET for PWS by providing IPET with a pruned wavefront CFG that incorporates the static identification of splits and the parallel execution of SWFs. This allows us to reuse IPET formulation as in prior works [6].

Key intuition in lowering WCET for PWS. To handle branch divergence, the GPU compiler introduces a sequence of instructions that serializes the execution of divergent paths. We call such a sequence of instructions a branch serialization block (BSB). We show a canonical CFG representation, C_{conv},

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

in Figure 7a, and the CFG generated by the GPU compiler, C_{BSB}, in Figure 7b. We represent BSBs using square nodes in the compiled CFG. The divergent branch in both CFGs is $BB1$, and the two branching paths are shown as basic blocks $BB2$ and $BB3$. However, the CFG generated by the GPU compiler in Figure 7b inserts a BSB for serialization.

The CFG executes on the GPU as follows. When a wavefront reaches the divergent branch $BB1$, the wavefront evaluates the condition for the branch and accordingly sets the active work-items. If no work-items need to execute $BB2$, the wavefront may skip $BB2$ and proceed directly to the BSB. Otherwise, the entire wavefront proceeds to execute $BB2$. Recall that a wavefront can have multiple work-items, and only active work-items execute instructions in $BB2$. Once $BB2$ completes, the execution reaches the BSB. At this point, if there were work-items that had to execute $BB3$ instead of $BB2$, then these work-items become active and the entire wavefront executes $BB3$. Otherwise, the wavefront skips $BB3$ and proceeds directly to $BB4$. Once again, only the active work-items execute instructions when executing $BB3$. Note that the wavefront cannot skip both $BB2$ and $BB3$ because all work-items must execute one of the two paths. We highlight the three possible execution paths in Figure 7c in violet, red, and blue, respectively. Due to the insertion of the BSB, this mechanism enables the CFG to illustrate how a wavefront serializes branch execution.

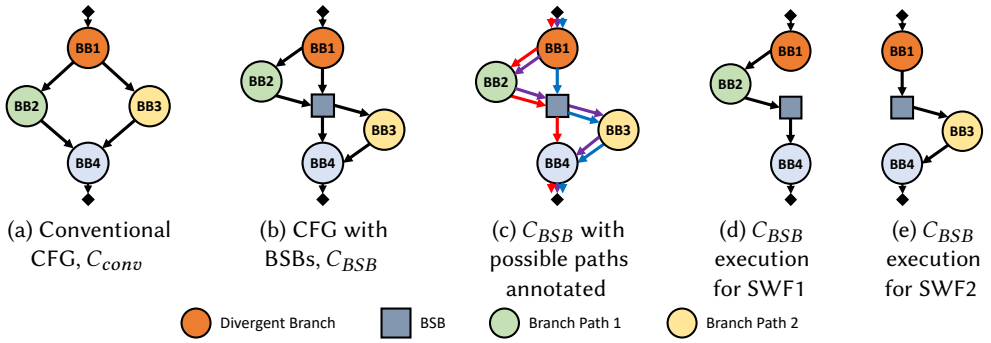


Fig. 7. Conventional CFG compared to a GPU CFG with encoded branch serialization

PWS uses the branch serialization mechanism to skip basic blocks. Consider again the example in Figure 7 executed by a wavefront that splits at $BB1$. When the wavefront reaches $BB1$, it creates a new SWF. All work-items that execute $BB2$ constitute one SWF and the remaining work-items that execute $BB3$ constitute the other SWF. Consider the first SWF, SWF1, that only consists of work-items that execute $BB2$. At $BB1$, it will proceed to $BB2$ because its work-items must execute $BB2$. Next, it proceeds to the BSB. Since all of the work-items executed $BB2$, none of them must execute $BB3$. Hence, the SWF proceeds to $BB4$. Thus, this SWF is guaranteed to execute the path $BB1 \rightarrow BB2 \rightarrow BB4$, highlighted in Figure 7d. By a similar process, the second SWF, SWF2, is guaranteed to execute the path $BB1 \rightarrow BB3 \rightarrow BB4$ in Figure 7e because all of its work-items execute $BB3$ and not $BB2$.

SWFs use the wavefront branch serialization mechanism to skip basic blocks that their work-items do not need to execute. In doing so, SWFs guarantee that a branch will not be serialized and the SWF will only execute one of the two paths. Hence, the worst-case execution time analysis can omit some paths in the CFG for SWFs. By eliminating branch serialization in the CFG, we reduce the resulting wavefront WCET, E_{WFi}. Therefore, PWS can reduce E_{WFi}. For the IPET analysis to consider this behaviour, we prune the kernel's CFG to match. For example, removing the edge between the BSB and $BB3$ before solving for the WCET of the CFG gives E_{WF}^{SWF1} because it disallows

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

execution of $BB3$ in the worst case. Applying this CFG pruning to each split point allows us to determine E_{WFi} for each SWF.

Pruning algorithm. We split our pruning algorithm into two parts and present them in Algorithms 1 and 2. SELECT takes the kernel CFG, C_{BSB}, and the total number of hardware SWF contexts per wavefront, $\mathcal{S}$, as input parameters. Since each SWF has at least one basic block that it does not execute, we can remove an edge leading to that block. SELECT outputs a set of branch nodes for which to remove edges. PRUNE uses this set of branch nodes to construct an SWF CFG with some edges removed to correspond to the WCET of an SWF. We assume that the compiler rejects kernels with `goto` statements and performs loop unrolling to reduce the CFG to simple sequences and branches. Each algorithm iterates through the vertices of C_{BSB}. Thus, both algorithms run in $O(|V|)$ time, assuming the input CFG is in topological order.

Algorithm 1: Select split point branches given CFG

```
1  SELECT(C_BSB = (V, E), S)
2      Let P ⊂ V be the branch nodes marked for splitting
3      Let swfCount ← 0 be a counter for the number of allocated SWFs so far
4      Let SB ← ∅ be a set of the currently allocated branch nodes for splitting
5      foreach u ∈ P in topological order do
6          if ∃v ∈ SB such that PARENT(u) = PARENT(v) then
7              SB ← SB ∪ {u}
8          else if swfCount < S then
9              SB ← SB ∪ {u}
10             swfCount ← swfCount + 1
11     return SB
```

We aid the explanation of Algorithms 1 and 2 using the example CFG in Figure 8, assuming that $\mathcal{S} = 2$. We assume that SELECT has access to the set of vertices in the CFG that are branch nodes marked for splitting, denoted as P. Figure 8 highlights these nodes in orange and the lists out the nodes in P.

Algorithms 1 and 2 also rely on some functions which they use as subroutines. Given a branch node or BSB u, we use the subroutine RECONV(u) to compute the basic block at which the two branching paths reconverge. In Figure 8, RECONV(1) = RECONV(12) = 18 and RECONV(2) = RECONV(4) = 6. Next, SUCCBB(u) determines the successor to a node u that is not a BSB or a reconvergence point. Similarly, BSB(u) determines the successor to a node u that is a BSB. These relationships are shown for node 1 in Figure 8. We also use a function called PARENT to determine the outer branch or BSB that is a dominator of a given node. Note that a dominator of node u is a node v such that every path to node u first passes through v.

Definition 4.1. The function PARENT(u) is defined as

$$\text{PARENT}(u) = v, \quad v < u < \text{RECONV}(v),\ u, v \in B \tag{1}$$

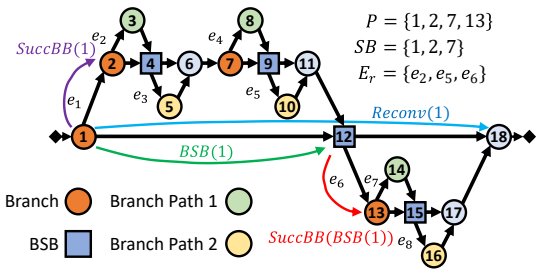
where B is the set of all branch nodes and BSBs and the inequality is defined with respect to the topological ordering of the CFG.

In Figure 8, PARENT(2) = PARENT(7) = 1, PARENT(13) = 12, and PARENT(1) = 0. The function allows the algorithm to determine when two branches are in sequence with each other, such as nodes 2 and 7.

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

Algorithm 2: Prune CFG for the SWF with the largest WCET

```
1  PRUNE(C_BSB = (V, E), SB)
2      Let B ⊂ V be the set of all branch nodes
3      Let L ⊂ V be the set of all BSBs
4      Let E_r ← ∅ be the set of edges to remove
5      Let F be a numerical array of size |V| + 1
6      F[|V| + 1] ← 0
7      foreach u ∈ V in reverse topological order do
8          if u ∈ B then
9              F[SUCCBB(u)] ← F[SUCCBB(u)] − F[BSB(u)] + F[RECONV(u)]
10             if u ∈ SB then
11                 F[u] ← max(F[SUCCBB(u)], F[BSB(u)])
12                 if F[SuccBB(u)] < F[BSB(u)] then
13                     E_r ← E_r ∪ {u → SUCCBB(u)}
14                 else
15                     E_r ← E_r ∪ {BSB(u) → SUCCBB(BSB(u))}
16             else
17                 F[u] ← F[SUCCBB(u)] + F[BSB(u)] − F[RECONV(u)]
18             F[u] ← F[u] + e^u_bb + e^{BSB(u)}_bb
19         else if u ∉ L then
20             F[u] ← F[SUCCBB(u)] + e^u_bb
21         else
22             F[u] ← F[SUCCBB(u)]
23     return C_out ← (V, E \ E_r)
```



u	e^u_{bb}	F[u]
1	6	37
2	4	30
3	3	34
4	2	37
5	6	37
6	4	31
7	4	29
8	5	24
9	1	23

u	e^u_{bb}	F[u]
10	4	23
11	3	19
12	3	16
13	1	16
14	1	12
15	1	11
16	2	11
17	5	9
18	4	4

Fig. 8. A sample CFG illustrating the components of Algorithm 1

The algorithms proceed as follows. First, SELECT iterates through each member, u, of the set P in topological order. If there are unallocated SWF contexts available, it will select the branch node for splitting. Notice that branch node 13 is marked for splitting by the GPU programmer but not selected by the algorithm. This is because there are not enough SWF contexts ($\mathcal{S} = 2$) to support it by the time the algorithm reaches the node. When a branch node is selected for splitting, it will add the node to the set SB.

Another way a branch node may be selected for splitting is if it can reuse an SWF context and the corresponding SpSIMD unit. Recall that we allow SWFs to merge once they have reached their post-dominator. The process of merging frees an SWF context, which enables another branch to

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

use the SWF context. Consequently, if there are two branch nodes in sequence with each other, they do not need to use SWF contexts simultaneously and can both be allocated for splitting. For example, branch nodes 2 and 7 are in sequence with each other and may both use the same SWF context for splitting. The algorithm handles this case for a branch node, u, by checking if one of the branch nodes in SB already contains a branch node, v, where $\textsc{Parent}(u) = \textsc{Parent}(v)$. The hardware execution shown in Figure 2d demonstrates how PWS reuses the SpSIMD unit to split two different branch nodes. In that example, the split operation at $BB7$ reuses the SWF hardware on SpSIMD1.

Once Select completes, it returns a set of branch nodes that PWS will choose as split points, SB. If $|SB| < \mathcal{S}$, we can determine that there will be $N_{unused} = (\mathcal{S} - |SB|)$ unused SpSIMD units per SIMD unit. Using $|SB|$, PWS can provision some of the idle units as SIMD units with a corresponding number of $|SB|$ SpSIMD units each. The new total number of SIMD units is given by

$$N_{SIMD}^{total} = N_{SIMD} + \left\lfloor \frac{N_{unused} \times N_{SIMD}}{|SB| + 1} \right\rfloor \tag{2}$$

which allows every SIMD unit to have $|SB|$ SpSIMD units associated with it. For example, consider a CU that has four SIMD units with three SpSIMDs allocated per SIMD unit (i.e. $\mathcal{S} = 3$). If Algorithm 1 outputs SB such that $|SB| = 1$, then two SpSIMDs will be idle per SIMD unit. These idle SpSIMDs can be re-provisioned into four SIMDs and four SpSIMDs, for a total of eight SIMD units and eight SpSIMD units on the CU. Using this static information about the number of split operations required by the kernel, PWS can determine how many idle SpSIMDs there will be to reuse and how to re-purpose them.

Using the output set SB from Select, Prune removes edges from C_{BSB}. At every split point in SB, each SWF will skip one of the two branch paths. Prune chooses which of these two paths has the smaller total WCET and removes an edge leading to this path to indicate that the SWF will not execute it. Prune keeps track of the WCET from each node to the end of the CFG using array F. Figure 8 shows the values in F after Prune executes given e_{bb}^{u} for each node u.

The algorithm proceeds through each node in C_{BSB} in reverse topological order, with an extra element $F[|V| + 1] = 0$. At each node u, it adds the value e_{bb}^{u} to the total from the node's successor. For branch nodes, it compares the value in F for each of the two branching paths. If the branch node is not in SB, this means that the wavefront will not split at this branch node and the GPU will serialize the execution of the two branch paths. Hence, the contribution to the overall WCET will be the sum of the two branch path execution times. Therefore, the algorithm updates $F[u]$ to this sum. For example, since branch node 13 is not in SB, $F[13] = F[14] + F[16]$. If the branch node is in SB, then the algorithm updates the value of $F[u]$ to the maximum of the two branch path executions. In the example, $F[2] = \max(F[3], F[5])$. When $u \in SB$, Prune removes the first edge to one of the two branch paths corresponding to the branch with the smaller execution time. For example, Prune removes e_2 because $F[3] < F[5]$. It adds each edge to be removed to set E_r. Once Prune completes its pass through C_{BSB}, it outputs $C_{out} = (V, E \setminus E_r)$ with these edges pruned. Figure 8 lists all edges in E_r.

Note that the choice of split points in the kernel can affect the WCET and is an interesting optimization problem which we do not explore in this paper; we reserve it for future work. Nevertheless, PWS provides a mechanism for marking split points and guaranteeing that SWFs execute in parallel in the worst case. Given this information, Select and Prune can reduce the WCET of a kernel CFG for a wavefront.

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

5 WORST-CASE EXECUTION TIME ANALYSIS

There are two parts to the WCET analysis of a kernel: (1) computing the WCET of a wavefront, and (2) using the WCET of a wavefront to compute the kernel's WCET. We extend an existing hybrid WCET analysis to support the proposed GPU architecture with PWS [6]. We summarize the symbols we use for this analysis in Table 1.

Table 1. System model symbols

(a) Hardware constructs

Name	Description
N_{CU}	# CUs on GPU
N_{SIMD}	# SIMDs per CU
N_{WIWF}	# work-items per WF
N_{WFC}	# WF contexts per SIMD
$\mathcal{S}$	# SpSIMDs per WF

(b) Software constructs

Name	Description
N_{WIWG}	# WIs per WG
G	# WGs
E_{kernel}	WCET of the kernel
E_{WFi}	WCET of WF i
e^j_{bb}	WCET of basic block j

Inputs to the analysis. To compute the WCET of a kernel, we assume the following inputs.

(1) The number of CUs, N_{CU}, SIMDs per CU, N_{SIMD}, and wavefront contexts per SIMD, N_{WFC}. These are properties of the GPU's hardware implementation. We assume that $N_{CU} = N_{SIMD} = 4$, based on recent AMD architectures [2, 3].
(2) The kernel launch parameters: the number of workgroups, G, and the number of work-items per workgroup, N_{WIWG}. These are provided by the GPU programmer; all kernels have them.
(3) The WCET of every basic block in the kernel, e^j_{bb}, where j indicates the basic block identifier. These can be obtained by performing static timing analysis [12] or by using measurement-based analysis approaches [6].
(4) The initialization delay, d_i, suffered by wavefront i. The initialization delay is defined as the delay the first instruction in the workgroup experiences from the time the workgroup is dispatched to the execution of the first instruction. As with e^j_{bb}, d_i values can be determined through static timing analysis or measurement-based analysis.
(5) The worst-case delay involved in splitting, e_s, and merging SWFs, e_m. We determine these constant values from our design.

WCET of a wavefront. To determine E_{WF}, we use Algorithms 1 and 2. Using each CFG in the output set D, we construct the flow constraints for a set of IPET optimization problems. We use an ILP solver to solve these problems, giving E_{WFi} for each SWF. We call the largest of these values E_{WF}, plus the cost of splitting and merging each of the SWFs, e_s and e_m, multiplied by the number of SWFs, $|SB|$, allocated by Algorithm 1.

Lemma 5.1. *The WCET of a single wavefront, E_{WF}, is given by*

$$E_{WF} = IPET(C_{out}) + |SB| \times (e_s + e_m), \tag{3}$$
$$\text{where } SB = \textsc{Select}(C, \mathcal{S}) \text{ and } C_{out} = \textsc{Prune}(C, SB)$$

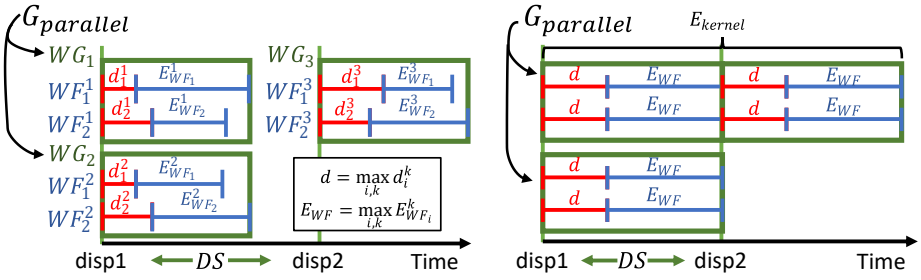
Proof. To prove Lemma 5.1, we separately show that each term in the expression for E_{WF} is an upper bound. The first term uses the output CFG from Prune, C_{out}, as an input to the IPET formulation. By removing the edges corresponding to the shorter branch path for each branch in SB, we ensure that the execution path of the SWF executing C_{out} is a maximum. Finally, each SWF

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

has split up to $|SB|$ times, meaning that the total cost of splitting and merging will be no more than $|SB|$ multiplied by the cost to split and merge once. Therefore, each term is an upper bound on the execution time of a wavefront and Lemma 5.1 holds. □

When the first of two SWFs reaches a merge point, it must stall its execution. This stalling behaviour does not make any additional impact on the WCET given in Lemma 5.1. This is because we compute the analytical WCET based on the SWF that takes the longest to execute each branch. Recall that we can do this because we guarantee that diverging branches execute in parallel. The faster SWF at each branch cannot stall past the point when the slower SWF reaches the merge point. Algorithms 1 and 2 output C_{out}, which represents the execution of an SWF that executes the path with the larger execution time each of the split points. This guarantees that the SWF that executes C_{out} takes the longest possible execution time and will not need to stall to wait for its parent or child SWF.

WCET of a kernel using PWS. Using the individual wavefront WCET, E_{WF}, we compute the WCET of the entire kernel on the GPU, E_{kernel}. We use Figure 9 to illustrate the components of E_{kernel}. Figure 9a shows a sample execution of three workgroups that have two wavefronts each. Recall that the GPU workgroup dispatcher distributes workgroups to available WF contexts on GPU hardware. We refer to the number of workgroups the GPU can dispatch simultaneously as $G_{parallel}$. The GPU used in the execution in Figure 9a has enough resources to execute two workgroups in parallel. In the diagram, each row WF_i^k represents the i-th wavefront in the k-th workgroup. The workgroups are outlined and labelled in green boxes. Within a workgroup box, each wavefront execution timeline consists of the initialization delay, d_i^k, and the wavefront execution time, $E_{WF}{}_i^k$. Notice that in the sample execution, there are a variety of execution times for d_i^k and $E_{WF}{}_i^k$.



(a) Sample execution of a kernel (b) Sample kernel worst case

Fig. 9. A sample GPU kernel execution timeline and its worst case

Figure 9b illustrates how we use the sample kernel execution to compute the WCET of the kernel. For this example, we use the largest initialization delay from all wavefronts, and the largest wavefront execution time, d and E_{WF}, respectively. At the start of the timeline, the GPU dispatches a set of $G_{parallel}$ workgroups to the WF contexts. When a workgroup completes execution, the GPU may dispatch a new workgroup. In the worst case, every wavefront in each workgroup is guaranteed to complete execution after $d + E_{WF}$, which allows the GPU to dispatch another set of $G_{parallel}$ workgroups. We refer to these dispatch points in time with the prefix *disp*; there are DS such points and the GPU needs to dispatch a set of $G_{parallel}$ workgroups DS times to execute all G workgroups. The kernel's WCET, E_{kernel}, reaches its endpoint when all workgroups finish executing the kernel.

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

Workgroup serialization. In Figure 9b, $G_{parallel}$ workgroups can execute in parallel. To determine E_{kernel}, we derive how many kernel dispatches, DS, are needed to execute all G workgroups specified by the programmer. First, we define how many workgroups can run simultaneously on GPU hardware. There are a total of $N_{CU} \times N_{SIMD}^{total} \times N_{WFC}$ WF contexts available at dispatch. Recall that we defined N_{SIMD}^{total} in Equation 2 as the total number of SIMD units after re-provisioning all extraneous SpSIMD units. Thus, there are $N_{CU} \times N_{SIMD}^{total} \times N_{WFC} \times N_{WIWF}$ work-items across all WF contexts. Each workgroup is defined as having N_{WIWG} work-items, so the hardware work-items can be divided into groups of N_{WIWG} work-items.

Definition 5.2. The number of workgroups that can execute simultaneously in parallel on GPU hardware is given by

$$G_{parallel} = \left\lfloor \frac{N_{CU} \times N_{SIMD}^{total} \times N_{WFC} \times N_{WIWF}}{N_{WIWG}} \right\rfloor \tag{4}$$

Next we define the total number of workgroup dispatches in the worst case.

Definition 5.3. The total dispatch count in the worst case, DS, is given by

$$DS = \left\lceil \frac{G}{G_{parallel}} \right\rceil \tag{5}$$

Critical instance. Next, we establish the critical instance. This instance relies on conditions for E_{WF}, d, DS, and $\mathcal{S}$.

Lemma 5.4. *A kernel exhibits its WCET when (1) all wavefronts split exhibit the wavefront WCET, E_{WF}, (2) each workgroup experiences the worst-case initialization delay, d, and (3) a series of DS workgroups executes serially.*

Proof. For (1), a workgroup cannot complete execution until the final wavefront is done; thus, if the final wavefront takes less time than E_{WF}, the next workgroup can start earlier. Hence, E_{kernel} will be reduced. By Lemma 5.1, the wavefront execution time also cannot exceed E_{WF}. Thus, the final wavefront of all workgroups must have an execution time of E_{WF} in the worst case. Similarly, for (2), if any workgroup exhibits less than the worst-case initialization delay, then there is a reduction in E_{kernel}. Thus, each workgroup must exhibit the d in the worst case. Finally, (3) holds by Definition 5.3. □

E_{kernel} with PWS. In the PWS implementation, each SWF can run in parallel with the original wavefront. As a result, they are guaranteed to be able to execute in parallel in the worst case.

Lemma 5.5 (PWS). *The WCET of executing a workload with PWS is given by*

$$E_{kernel}^{PWS} = DS \times \left(d + E_{WF}^{PWS}\right) \tag{6}$$

Proof. The WCET of each workgroup, of which DS execute serially, is characterized by the final wavefront to complete execution. By definition, this wavefront will start execution with a delay of d from the time the GPU dispatches the workgroup. Given the availability of $\mathcal{S}$ SpSIMDs dedicated to SWFs, we can guarantee that the $\mathcal{S}$ SWFs that were created from the same source wavefront will execute in parallel with the source wavefront. From Lemmas 5.1 and 5.4, the WCET of the final wavefront to complete execution is E_{WF}^{PWS}. □

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

5.1 E_{kernel} with DWS

Lemma 5.6. *The worst-case wavefront execution times for PWS and DWS are related by*

$$E_{WF}^{PWS} \leq E_{WF}^{DWS} \tag{7}$$

Proof. Given $\mathcal{S}$, the number of SWFs is the same for both DWS and PWS in the worst case. Thus, the values for each e_{bb}^{j} are the same between the two mechanisms. Each CFG contains a set B of branch nodes. Since branches are not marked for splitting in DWS, the GPU may arbitrarily select any $\mathcal{S}$ branches from B for splitting. In the worst case, DWS selects the branches that result in the largest WCET. On the other hand, Select will select branch nodes from $P \subseteq B$ for PWS. The set P may not contain all nodes that result in the largest WCET. If P does not contain all such nodes, then $E_{WF}^{PWS} < E_{WF}^{DWS}$; otherwise, $E_{WF}^{PWS} = E_{WF}^{DWS}$. Therefore, Lemma 5.6 holds. □

In the case of DWS, we have no guarantee about where wavefronts will split, even if $\mathcal{S}$ is given. Without this guarantee, the worst-case analysis for DWS must assume that it will split at the worst choices of $\mathcal{S}$ branch nodes. Furthermore, the absence of hardware specifically provisioned for SWFs means the GPU must serialize the execution of all SWFs. Note that, if $\mathcal{S} = 0$, E_{kernel} is the WCET of the kernel without splitting.

Lemma 5.7 (DWS). *The WCET of executing a workload with SWFs but without special hardware dedicated to SWFs is given by*

$$E_{kernel}^{DWS} = DS \times \left(d + (\mathcal{S} + 1) \times E_{WF}^{DWS}\right) \tag{8}$$

where E_{WF}^{DWS} is the E_{WF} computed under DWS.

Proof. The WCET of each workgroup, of which DS execute serially, is characterized by the final wavefront to complete execution. By definition, this wavefront will start execution with a delay of d from the time the GPU dispatches the workgroup. When the GPU creates SWFs, there is no guarantee that the SWFs execute in parallel. Therefore, the SWFs execute serially in the worst case. Since SWFs compete for the same hardware as other WFs, there is no guarantee that they will execute in parallel. Furthermore, each wavefront will split into $\mathcal{S}$ SWFs. Hence, the WCET of each wavefront corresponds to its own execution time plus the sum of each of the executions of the $\mathcal{S}$ SWFs it creates. □

Lemma 5.8. *The worst-case kernel execution times for PWS and DWS are related by*

$$E_{kernel}^{PWS} \leq E_{kernel}^{DWS} \tag{9}$$

Proof. By Lemma 5.6, $E_{WF}^{PWS} \leq E_{WF}^{DWS}$. Observing Lemmas 5.5 and 5.7, the key difference between the expressions for E_{kernel}^{PWS} and E_{kernel}^{DWS} is the multiplication of E_{WF}^{DWS} by $\mathcal{S}$. Using these two results, $E_{kernel}^{PWS} \leq E_{kernel}^{DWS}$. □

This result emphasizes the increase in the WCET when the GPU hardware does not support parallel hardware for SWFs. However, the architectural changes to the GPU introduced by PWS mitigate this issue and guarantee parallel execution. PWS provisions extra SpSIMD units for each existing SIMD unit. The number of SpSIMD units per SIMD unit is statically known and constant. Hence, $\mathcal{S}$ SWFs can execute on SIMD and SpSIMD units in parallel. Using static knowledge of the number of SpSIMD units, we can guarantee these SWFs execute in parallel. This is not the case for DWS; if the GPU programmer uses DWS, diverging branches executed by SWFs are serialized in the worst case. This is the reason for the result in Lemma 5.8.

To illustrate the result in Lemma 5.8, we revisit the simple example CFG in Figure 8. We can use this example to illustrate how PWS results in a lower WCET than DWS. We assume that the

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

e^j_{bb} values are kept the same as in the table in Figure 8 and that $SB = \{1, 2, 7\}$ and $\mathcal{S} = 2$ as before. For simplicity, we assume that d, e_s, e_m, and DS are all equal to 1. In the case of DWS, there is no guarantee that SWFs will execute in parallel because there are no additional SpSIMD units. Hence, E^{DWS}_{WF} is the sum of all basic block execution times and the costs to split and merge SWFs. Using this E^{DWS}_{WF}, we compute E^{DWS}_{kernel} using Lemma 5.7, resulting in an overall WCET of 192. Conversely, the PWS analysis starts with a pruned CFG with edges e_2, e_5, and e_6 removed. By Lemmas 5.1 and 5.5, $E^{PWS}_{WF} = 48$ and $E^{PWS}_{kernel} = 49$. We summarize these results in Table 2, along with a comparison to the case where splitting is not enabled, which is the way existing GPU architectures operate.

Table 2. Analytical WCET values for No Splitting, DWS, and PWS based on the example in Figure 8.

S	No Splitting	DWS	PWS	WCET reduction under PWS over No Splitting	WCET reduction under PWS over DWS
1	60	127	52	13%	59%
2	60	192	49	18%	74%
3	60	245	47	22%	81%

It is clear that the E^{PWS}_{kernel} is considerably lower than E^{DWS}_{kernel}, by as much as 81% in the given example.

6 RELATED WORK

Dynamic wavefront splitting. There have been several prior works that address branch divergence through wavefront splitting. Meng et al. proposed the first dynamic wavefront splitting method to reduce branch divergence [15]. When a wavefront encounters branch divergence, it creates two SWFs, where each SWF is a separate schedulable entity equivalent to a wavefront. If the SWF encounters a stall, the remaining work-items in another SWF can still be scheduled to execute because they may be free of dependencies. This behaviour can provide a performance gain over not splitting in the case where all wavefronts would be stalled executing a serialized branch.

Rhu and Erez [18] improve on DWS with a new reconvergence data structure to allow splits to merge at the immediate post-dominator of a branch. Brunie et al. propose allowing work-items from the same wavefront to execute distinct instructions simultaneously by doubling the number of work-items in a wavefront while cutting the number of wavefronts in half [9].

A more recent work by Damani et al. explores the hardware requirements for implementing wavefront splitting [10]. This work also separates the work-items within a wavefront, but performs scheduling at the work-item level, in addition to the wavefront level. The status of each work-item is tracked using a finite state machine and the status information is kept in a status table. All work-items which are ready constitute an SWF. The wavefront scheduler selects the wavefront which should execute based on if any of its work-items are ready for execution.

Reverse-engineering commercial GPUs. Several prior works by Otterness and Anderson have aimed to publish more information about closed-source or poorly documented implementation details of NVIDIA and AMD GPUs [4, 16, 21] in order to better understand their behaviour for real-time and safety-critical systems. In [21], the authors identify the issues with using NVIDIA GPUs for safety-critical systems. For example, the behaviour of NVIDIA GPUs is often different than what is presented in the official documentation and there are undocumented implicit synchronization constructs which impede real-time analysis of these GPUs. They also investigate AMD GPUs in [16]. They show that, since the AMD software stack is open source, it makes AMD GPUs a

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

more attractive choice for supporting real-time systems than NVIDIA GPUs. Although these prior works have advanced the state-of-the-art in being able to use GPUs for safety-critical systems, the performance impact of branch divergence continues to be a challenge for these works as well. PWS address this challenge by proposing a predictable approach for wavefront splitting.

Vector processors in safety-critical systems. Some prior works have proposed vector processor and GPU designs for safety-critical systems. Platzer and Puschner present Vicuna, a vector coprocessor implementing the vector extension to the RISC-V standard [17]. They show that their implementation is timing predictable by showing that its execution is free of timing anomalies. The architecture of Vicuna eliminates timing anomalies by enforcing a strict ordering of instructions and removing optimizations that make analysis more complex. A similar work by Spliet and Mullins presents Sim-D [20], which is a vector processor that separates workgroup execution into several uninterruptable compute and access phases. The execution times of these phases can be determined statically because they cannot interfere with each other on the available functional units.

7 RESULTS

In this section, we use an implementation of PWS to demonstrate its impact on the WCET of a kernel. We also show that PWS can still offer a benefit to the average-case performance of a workload, similar to DWS. We summarize the results in graphical form in Figures 10 and 11.

Experimental setup. To the best of our knowledge, a freely available RTL implementation of an AMD GPU is not available. Further, we find that GPU micro-architectural simulators are both appropriate and commonly used in architectural exploration. As a result, we use the AMD gem5 implementation. Recent research papers that propose architectural changes to the GPU also make use of this simulator to evaluate proposed micro-architectural extensions, such as [7]. To that end, we implement DWS and PWS in the gem5 simulator [13], using the AMD GCN3 ISA and use it for evaluation.

We evaluate PWS against existing GPU architectures, which do not split wavefronts, and GPUs that implement DWS [15], the state-of-the-art approach to improving performance under branch divergence. When existing GPUs encounter branch divergence, they serialize the execution of the two branch paths. We refer to this configuration as *no splitting*. To test this configuration, we use the existing gem5 GPU implementation without any modifications. By contrast, DWS provides a mechanism for the GPU to respond to branch divergence by dividing the work-items in a wavefront into separate SWFs. At runtime, DWS detects when the wavefront encounters a divergent branch in the kernel and splits the wavefront into two SWFs. The work-items in the wavefront are partitioned based on the branch path they take. These SWFs share the same SIMD unit execution resources. Our version of DWS implements these attributes, making it most similar to prior DWS implementations by Damani et al. [10] and Rhu and Erez [18].

Our experimental implementation of PWS functions as described in Section 4. The splitting mechanism is the same as DWS other than two key differences. First, we use the `split` and `merge` instructions to determine when a wavefront should split. Second, we provision dedicated SpSIMD units to guarantee parallel execution of SWFs. We compare PWS both to the no splitting and DWS configurations. Note that both PWS and DWS have a maximum number of SWFs that the GPU can create for each wavefront. We denote this with $\mathcal{S}$ for our analysis.

Benchmarks. We use the Rodinia benchmark suite [19] to evaluate PWS. The Rodinia suite is designed for heterogeneous computing across a variety of applications. AMD maintains a version of these benchmarks written in HIP, the AMD counterpart to CUDA, making it natively compatible with the simulator GPU architecture. This benchmark suite is a good representation of today's

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

GPU workloads. The Rodinia suite consists of a variety of programs from different domains such as machine learning, graphs analytics, and computer vision. Furthermore, it is a key benchmark suite used in recent research works, such as [23].

In addition, we supplement the Rodinia benchmarks with our set of synthetic benchmarks. These synthetic benchmarks are designed to exercise the worst-case scenario, which the Rodinia suite may not do. Furthermore, in the synthetic benchmarks, the number of divergent instructions can be controlled enabling us to perform qualitative comparisons against state-of-the-art approaches.

The basic structure of our synthetic benchmarks consists of three parts: (1) non-divergent instructions where all work-items in the wavefront execute the instructions unconditionally, (2) conditionals in the form of nested if-else statements where the conditionals are predicated on work-item identifiers, and (3) divergent instructions where a subset of work-items in the wavefront execute instructions based on the outcome of the conditionals. Both (1) and (3) constitute arithmetic and memory operations. All three parts are configurable in the synthetic benchmarks.

In Figure 10a, the horizontal axis represents the percentage of divergent instructions present in the application. The 100% divergent instructions scenario in Figure 10a corresponds to a synthetic benchmark setup where each work-item in the wavefront executes a different instruction; there are no non-divergent instructions. The 25% divergent instructions scenario in Figure 10a corresponds to a synthetic benchmark setup where 75% of instructions (other than conditionals) are non-divergent instructions (all threads in the wavefront execute the same instructions albeit on different data) and 25% of the instructions (other than conditionals) are divergent instructions.

In Figure 10b, the horizontal axis represents the percentage of memory instructions. For the data shown in Figure 10b, all the instructions in the synthetic benchmark are divergent. We change the composition of the divergent instructions by changing the number of memory instructions present in the divergent instructions. For example, the 25% memory instructions scenario in Figure 10b corresponds to a synthetic benchmark setup where all instructions are divergent, and the percentage of memory and arithmetic instructions in the divergent instructions are 25% and 75%, respectively.

Worst-case behaviour of PWS. We compute the WCET of PWS using a measurement-based approach to obtain values for each e^j_{bb} and d_i. The starting point for the measurements for each wavefront occurs when it transitions to the running state and is ready to fetch its first instruction. The end point for these measurements is when the wavefront executes an end-of-program instruction and transitions to the stopped state. Accurately computing E_{WFi} also requires upper bounds on the number of iterations on each loop. We statically control the number of loop iterations in the synthetic benchmark and use this bound to solve for E_{WF}. Finally, we measure the total execution time of each kernel for performance data and to determine the observed WCET.

We compute three different WCET values: observed, analytical for PWS, and analytical for DWS. The first is the observed WCET, which is computed as the maximum of all kernel execution times for a given benchmark of fixed input size, if applicable. If a GPU program has multiple kernels, the WCET is the sum of the individual kernel WCETs. To compute the analytical WCET, we first construct a CFG using each e^j_{bb}. Next, we use Select and Prune with this CFG to compute E_{WF}. Finally, we use Lemma 5.5 with the maximum measured d_i and the benchmark parameters G and N_{WIWG} to compute E^{PWS}_{kernel}. To model the analytical WCET of DWS, we use the same measured e^j_{bb} and d_i values as inputs. Since DWS does not guarantee parallel execution, we do not apply Select and Prune to the CFG, and use Equation 3 to determine E_{WF}, where $C_{out} = C$ and $|SB| = \mathcal{S}$. Finally, we use Lemma 5.7 to compute E^{DWS}_{kernel}.

We start by comparing the impact of PWS on the WCET compared to DWS in Table 3. The table compares the WCET of both techniques for the synthetic benchmark given a maximum number of divergent instructions. It presents the cycle counts of the observed and analytical WCETs; note

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

that the data is presented in terms of thousands of cycles for brevity. For PWS, we annotate as many split points as there are SpSIMD units. Hence, all SpSIMD units are utilized and we do not need to perform any re-provisioning. With DWS, the analytical WCET increases linearly with the value of $\mathcal{S}$. This result is a consequence of the the $\mathcal{S}+1$ factor in Lemma 5.7. Hence, splitting with DWS results in a much larger WCET than PWS. We highlight this as a reduction percentage in the rightmost column of the table in Table 3. It shows how PWS provides a much better reduction in the WCET than DWS, as much as 88% when $\mathcal{S}=7$. Finally, these results also show that the analytical WCET serves as an upper bound for the observed WCET.

Table 3. WCET cycle counts, reported in thousands of cycles, for PWS and DWS executing the synthetic benchmark with maximum branch divergence and the associated reduction under PWS compared to DWS.

$\mathcal{S}$	No Splitting		DWS		PWS		PWS Analytical WCET Reduction Compared to DWS
	Observed	Analytical	Observed	Analytical	Observed	Analytical	
1	500	680	140	600	150	300	50%
3	500	680	50	870	30	220	75%
7	500	680	45	1440	15	180	88%

In Figure 10, we show data for a range of values of $\mathcal{S}$. The reductions in WCET that we report are compared to the case when wavefronts do not split as the baseline. Again, we annotate as many split points as there are SpSIMD units such that we do not need to perform any re-provisioning.

Using Figure 10, we start by showing the impact of PWS on the WCET of the synthetic benchmark through several parameter sweeps. Figure 10a shows the impact of the number of instructions that are divergent on the overall reduction in the WCET. As expected, the data shows that workloads with greater amounts of divergence benefit more from PWS. Next, Figure 10b shows the impact of number of memory instructions. Here, each version of the synthetic benchmark has a varying number of branches that contain memory instructions. The SWFs created by PWS interfere with each other in the memory hierarchy. Hence, there is less of a WCET reduction for workloads that have larger amounts of memory instructions.

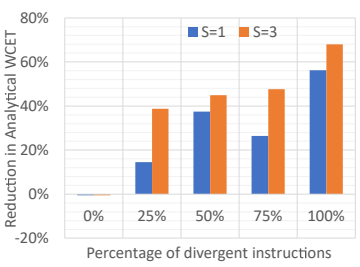
To reduce hardware costs, the design of PWS also supports keeping the total number of SIMDs constant and designating a subset of them to be SpSIMDs. For example, if there are a total of 16 SIMDs, 4 SIMDs can be designated for the original wavefronts while the remaining 12 can be provisioned as SpSIMDs to allow for $\mathcal{S}=3$. Figure 10c shows the WCET reduction for GPU processors with a varying total number of available SIMDs. The result demonstrates that PWS can also be a useful strategy for WCET reduction even with tight hardware budgets.

Figure 10e shows the analytical WCET for each of the Rodinia benchmarks. These benchmarks have relatively little branch divergence so there is limited reduction in the WCET.

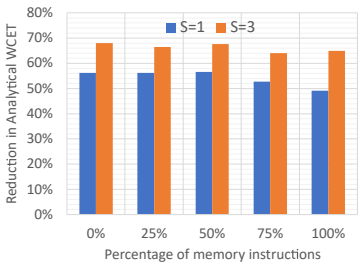
Average-case performance under PWS. The WCET is the primary requirement for being able to deploy safety-critical applications on GPUs. However, it is also important to offer as much performance as possible while still providing WCET guarantees to allow the application to deliver its required quality of service. Therefore, we also evaluate the performance improvement offered by PWS compared to existing architectures.

We use the Rodinia benchmark suite [19] to evaluate the average-case performance of PWS across a variety of applications. Figure 11 shows the average-case performance improvement demonstrated by PWS when $\mathcal{S}=1$ compared to no splitting as the baseline. The total number of SIMDs for both splitting and no splitting cases is held constant. The benchmarks are sorted in order

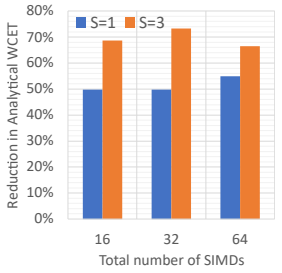
A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.



(a) Reduction in WCET given the proportion of divergent instructions.



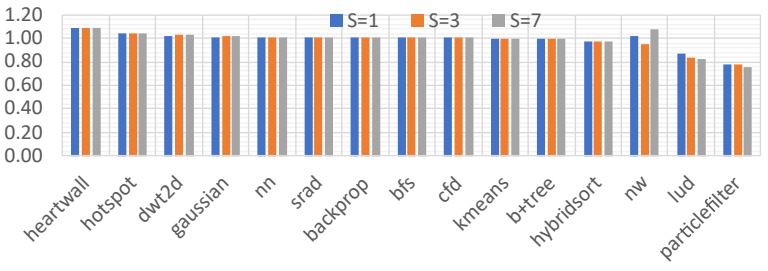
(b) Reduction in WCET given the proportion of memory instructions.



(c) Reduction in WCET given the total number of SIMDs.

S	PWS	DWS
0	0%	0%
1	56%	12%
3	68%	-28%
7	73%	-112%

(d) WCET reduction for PWS and DWS.



(e) Analytical WCET for each Rodinia benchmark, normalized to the analytical WCET for $\mathcal{S} = 0$.

Fig. 10. Plots of each result running PWS on synthetic and Rodinia benchmarks, compared to no splitting as the baseline.

of smallest to largest speedup. The first segment of benchmarks, coloured orange, demonstrate a minor performance loss. The second segment of benchmarks, coloured green, demonstrate a performance gain. Finally, the blue columns represent the geometric and arithmetic mean of all speedups, which is a performance improvement of approximately 11% and 18%, respectively. Notably, the bfs benchmark demonstrates a performance improvement of 3x. The structure of the benchmark includes a loop which contains conditional logic based on the value of the data for each work-item, a pattern which often leads to branch divergence. Since the divergent code block is found within the main loop of the kernel, wavefront splitting offers a significant increase in performance. Overall, the average-case performance improvement by 11% shows that PWS can also enhance GPU performance on top of reducing the WCET.

Reusing SpSIMD units. Recall that PWS may re-provision some SpSIMD units if the number of split points required by the programmer is less than $\mathcal{S}$. Reusing the SpSIMD units does not have an impact on the kernel WCET under PWS; however, it may offer some performance benefit. In our experiments, we assess the impact of reusing SpSIMD units on the performance of PWS. Figure 12 compares the average-case performance under PWS with and without this described SpSIMD reuse. We use a GPU configuration where each CU has four SIMD units with three SpSIMDs allocated per SIMD unit. We evaluate the configuration using our synthetic benchmark with 100% divergent instructions and 0% memory instructions with zero, one, two, and three annotated split points. Note that the fewer annotated split points there are, the better the observed performance improvement.

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

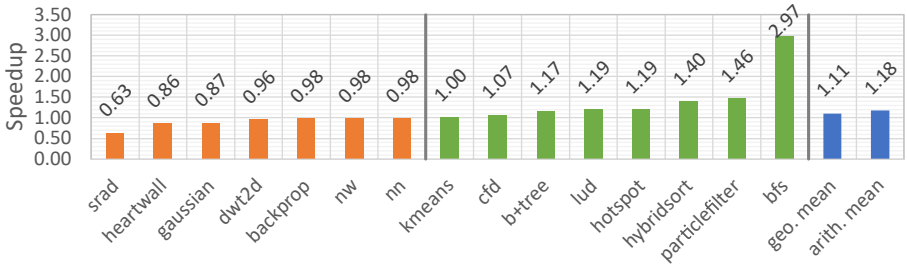


Fig. 11. Average-case PWS speedup for each Rodinia benchmark compared to no splitting as the baseline.

This is because there are a larger number of re-provisioned SpSIMD units. The data shows that reusing the idle SpSIMDs results in an up to 31% performance improvement under PWS compared to leaving them idle.

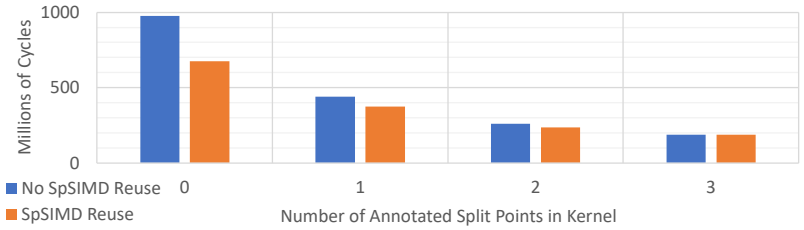


Fig. 12. Average-case execution time under PWS with and without SpSIMD reuse, where $\mathcal{S}$=3.

8 CONCLUSION

In this work, we present PWS, a technique that allows for predictable wavefront splitting to address the impact of GPU branch divergence for safety-critical systems. Timing predictability requires that the WCET can be modeled to provide an upper bound on the kernel execution time. PWS achieves timing predictability in three key ways. First, it uses explicit instructions in the form of `split` and `merge` instructions which inform the static analysis techniques where split points will occur in the kernel. Next, it guarantees that SWFs execute in parallel with each other, which contributes to obtaining a low WCET. Finally, it adds a compiler pass to eliminate execution paths in the CFG that will not be taken by SWFs, allowing it to reduce the upper bound on the kernel execution time. Our analysis gives programmers a way to compute the WCET, and we show how PWS can reduce this WCET compared to prior works. Our results show that PWS can also improve GPU performance while also providing programmers a mechanism to compute and reduce the WCET of a kernel, making PWS a suitable strategy for safety-critical systems.

REFERENCES

[1] Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, and Margaret Martonosi. 2018. *General-Purpose Graphics Processor Architecture.* Morgan & Claypool. 21–26 pages.
[2] Advanced Micro Devices. 2016. Graphics Core Next Architecture Reference Guide.
[3] Advanced Micro Devices. 2019. Introducing RDNA Architecture.
[4] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. 2017. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS).* 104–115. https://doi.org/10.1109/RTSS.2017.00017

A. Klashtorny, Z. Wu, A. M. Kaushik, and H. Patel, "Predictable GPU Wavefront Splitting for Safety-Critical Systems," ACM Trans. Embed. Comput. Syst., vol. 22, no. 5s, Sep. 2023, doi: 10.1145/3609102.

[5] Pete Bannon, Ganesh Venkataramanan, Debjit Das Sarma, and Emil Talpes. 2019. Computer and Redundancy Solution for the Full Self-Driving Computer. In *2019 IEEE Hot Chips 31 Symposium (HCS), Cupertino, CA, USA, August 18-20, 2019*. IEEE, 1–22. https://doi.org/10.1109/HOTCHIPS.2019.8875645

[6] Adam Betts and Alastair Donaldson. 2013. Estimating the WCET of GPU-Accelerated Applications Using Hybrid Analysis. In *2013 25th Euromicro Conference on Real-Time Systems*. 193–202. https://doi.org/10.1109/ECRTS.2013.29

[7] Srikant Bharadwaj, Shomit Das, Yasuko Eckert, Mark Oskin, and Tushar Krishna. 2021. DUB: Dynamic Underclocking and Bypassing in Nocs for Heterogeneous GPU Workloads. In *Proceedings of the 15th IEEE/ACM International Symposium on Networks-on-Chip* (Virtual Event) *(NOCS '21)*. Association for Computing Machinery, New York, NY, USA, 49–54. https://doi.org/10.1145/3479876.3481590

[8] Benjamin Brosgol. 2011. DO-178C: The Next Avionics Safety Standard. In *Proceedings of the 2011 ACM Annual International Conference on Special Interest Group on the Ada Programming Language* (Denver, Colorado, USA) *(SIGAda '11)*. Association for Computing Machinery, New York, NY, USA, 5–6. https://doi.org/10.1145/2070337.2070341

[9] Nicolas Brunie, Caroline Collange, and Gregory Diamos. 2012. Simultaneous branch and warp interweaving for sustained GPU performance. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 49–60. https://doi.org/10.1109/ISCA.2012.6237005

[10] Sana Damani, Mark Stephenson, Ram Rangan, Daniel Johnson, Rishkul Kulkarni, and Stephen W. Keckler. 2022. GPU Subwarp Interleaving. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.

[11] Wilson W.L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 407–420. https://doi.org/10.1109/MICRO.2007.30

[12] Yijie Huangfu and Wei Zhang. 2017. Static WCET Analysis of GPUs with Predictable Warp Scheduling. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*. 101–108. https://doi.org/10.1109/ISORC.2017.24

[13] Jason Lowe-Power et al. 2020. The gem5 Simulator: Version 20.0+. *CoRR* abs/2007.03152 (2020). arXiv:2007.03152 https://arxiv.org/abs/2007.03152

[14] Kuen-Long Lu and Yung-Yuan Chen. 2019. ISO 26262 ASIL-Oriented Hardware Design Framework for Safety-Critical Automotive Systems. In *2019 IEEE International Conference on Connected Vehicles and Expo (ICCVE)*. 1–6. https://doi.org/10.1109/ICCVE45908.2019.8965235

[15] Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France) *(ISCA '10)*. Association for Computing Machinery, New York, NY, USA, 235–246. https://doi.org/10.1145/1815961.1815992

[16] Nathan Otterness and James H. Anderson. 2021. Exploring AMD GPU Scheduling Details by Experimenting With "Worst Practices". In *29th International Conference on Real-Time Networks and Systems* (NANTES, France) *(RTNS'2021)*. Association for Computing Machinery, New York, NY, USA, 24–34. https://doi.org/10.1145/3453417.3453432

[17] Michael Platzer and Peter Puschner. 2021. Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 196)*, Björn B. Brandenburg (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1:1–1:18. https://doi.org/10.4230/LIPIcs.ECRTS.2021.1

[18] Minsoo Rhu and Mattan Erez. 2013. The dual-path execution model for efficient GPU control flow. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 591–602. https://doi.org/10.1109/HPCA.2013.6522352

[19] Corbin Robeck and Aryan Salmanpour. 2016. ROCm Developer Tools: HIP Examples. https://github.com/ROCm-Developer-Tools/HIP-Examples.

[20] Roy Spliet and Robert D. Mullins. 2022. Sim-D: A SIMD Accelerator for Hard Real-Time Systems. *IEEE Trans. Comput.* 71, 4 (2022), 851–865. https://doi.org/10.1109/TC.2021.3064290

[21] Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H. Anderson, and F. Donelson Smith. 2018. Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems. In *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain (LIPIcs, Vol. 106)*, Sebastian Altmeyer (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:21. https://doi.org/10.4230/LIPIcs.ECRTS.2018.20

[22] Sharad Malik Yau-Tsun Steven Li. 1995. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *32nd Design Automation Conference*. 456–461. https://doi.org/10.1109/DAC.1995.249991

[23] Wei Zhang, Quan Chen, Kaihua Fu, Ningxin Zheng, Zhiyi Huang, Jingwen Leng, and Minyi Guo. 2022. Astraea: Towards QoS-Aware and Resource-Efficient Multi-Stage GPU Services. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 570–582. https://doi.org/10.1145/3503222.3507721