2018 IEEE International Symposium on High Performance Computer Architecture

Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level

Anthony Gutierrez, Bradford M. Beckmann, Alexandru Dutu, Joseph Gross, John Kalamatianos, Onur Kayiran, Michael LeBeane, Matthew Poremba, Brandon Potter, Sooraj Puthoor, Matthew D. Sinclair, Mark Wyse, Jieming Yin, Xianwei Zhang, Akshay Jain[†], Timothy G. Rogers[†]

AMD Research, Advanced Micro Devices, Inc.
[†]School of Electrical and Computer Engineering, Purdue University

{anthony.gutierrez, brad.beckmann, alexandru.dutu, joe.gross, john.kalamatianos, onur.kayiran, michael.lebeane, matthew.poremba, brandon.potter, sooraj.puthoor, matthew.sinclair, mark.wyse, jieming.yin, xianwei.zhang}@amd.com
{jain156, timrogers}@purdue.edu

***Abstract*—Modern GPU frameworks use a two-phase compilation approach. Kernels written in a high-level language are initially compiled to an implementation-agnostic intermediate language (IL), then finalized to the machine ISA only when the target GPU hardware is known. Most GPU microarchitecture simulators available to academics execute IL instructions because there is substantially less functional state associated with the instructions, and in some situations, the machine ISA's intellectual property may not be publicly disclosed. In this paper, we demonstrate the pitfalls of evaluating GPUs using this higher-level abstraction, and make the case that several important microarchitecture interactions are only visible when executing lower-level instructions.**

Our analysis shows that given identical application source code and GPU microarchitecture models, execution behavior will differ significantly depending on the instruction set abstraction. For example, our analysis shows the dynamic instruction count of the machine ISA is nearly 2× that of the IL on average, but contention for vector registers is reduced by 3× due to the optimized resource utilization. In addition, our analysis highlights the deficiencies of using IL to model instruction fetching, control divergence, and value similarity. Finally, we show that simulating IL instructions adds 33% error as compared to the machine ISA when comparing absolute runtimes to real hardware.

***Keywords*-ABI; GPU; Intermediate Language; Intermediate Representation; ISA; Simulation;**

I. Introduction

Research in GPU microarchitecture has increased dramatically with the advent of GPGPUs. Heterogeneous programming features are now appearing in the most popular programming languages, such as C++ [24] and Python [18]. To evaluate research ideas for these massively parallel architectures, academic researchers typically rely on cycle-level simulators. Due to the long development time required to create and maintain these cycle-level simulators, much of the academic research community relies on a handful of open-source simulators, such as GPGPU-Sim [11], gem5 [14], and Multi2Sim [35].

Simulating a GPU is especially challenging because of the two-phase compilation flow typically used to generate GPU kernel binaries. To maintain portability between different generations of GPU hardware, GPU kernels are first compiled to an intermediate language (IL). Prior to launching a kernel, low-level software, such as the GPU driver or runtime, finalizes the IL into the machine instruction set architecture (ISA) representation that targets the GPU hardware architecture.

Unlike CPU ISAs, GPU ISAs are often proprietary, change frequently, and require the simulation model to incorporate more functional state, thus academic researchers often use IL simulation. In particular, NVIDIA's Parallel Thread Execution (PTX) ISA [33] and the HSA foundation's Heterogeneous Systems Architecture Intermediate Language (HSAIL) Virtual ISA [26] are the most popular ILs. However, AMD has recently disclosed several GPU ISA specifications, including the Graphics Core Next (GCN) 3 ISA [3] explored in this work. AMD has also released their complete GPU software stack under an open-source license [8], thus it is now feasible for academic researchers to simulate at the lower machine ISA level.

From an industrial perspective, this paper makes the case that using IL execution is not sufficient when evaluating many aspects of GPU behavior and implores academics to use machine ISA execution, especially when evaluating microarchitecture features. The primary goals of ILs (e.g., abstracting away hardware details and making kernels portable across different HW architectures) directly conflict with the goals of cycle-level simulation, such as accounting for hardware resource contention and taking advantage of unique hardware features. Furthermore, GPUs are co-designed hardware-software systems where vendors frequently change the machine ISA to simplify the microarchitecture and push more complexity to software.

While several prior works have investigated modeling challenges for the CPU and memory [12] [15] [20] [23], this paper is the first to deeply investigate the unique issues that occur when modeling GPU execution using the IL. To facilitate the investigation, this paper also introduces a new simulation infrastructure capable of simulating both

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00058

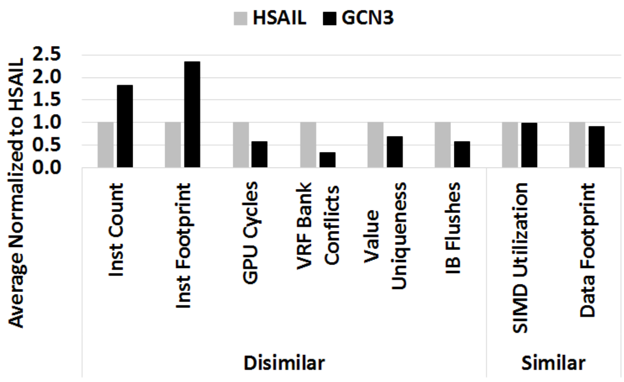


Figure 1: Average of dissimilar and similar statistics.

HSAIL and AMD's GCN3 machine ISA using the same microarchitecture model.

This paper explores the space of statistics for which IL simulation provides a faithful representation of the program under test, and those for which it does not. Figure 1 summarizes several key statistics that differ significantly between HSAIL and GCN3 and a few that do not. Specifically, due to significant code expansion, substantial differences in the application binary interface (ABI), and the lack of scalar instructions, HSAIL substantially underestimates the dynamic instruction count and the code footprint. Despite the significant underestimation in the instruction issue rate, HSAIL substantially overestimates GPU cycles, bank conflicts, value uniqueness within the vector register file (VRF), and instruction buffer (IB) flushes. Meanwhile, the lack of a scalar pipeline does not impact HSAIL's accuracy in estimating single instruction, multiple data (SIMD) unit utilization and the program's data footprint.

In summary, we make the following contributions:

- We add support for the Radeon Open Compute platform (ROCm) [8] and the state-of-the-art GCN3 ISA to gem5's GPU compute model, and we demonstrate how using the actual runtimes and ABIs impact accuracy.
- We perform the first study quantifying the effects of simulating GPUs using an IL, and we identify when using an IL is acceptable and when it is not.
- Furthermore, we demonstrate that anticipating the impact IL simulation has on estimating runtime is particularly hard to predict and application dependent, thus architects cannot simply rely on "fudge-factors" to make up the difference.

II. Background

A. GPU Programming

GPUs are programmed using a data parallel, streaming computation model. In this model a kernel is executed by a collection of threads (named work-items in the terminology applied in this paper). A programmer, compiler, or automation tool is responsible for identifying the computation forming the kernel. Modern GPUs support control flow operations allowing programmers to construct complex kernels. High-level GPU programming languages define kernels using the single instruction, multiple thread (SIMT) execution model. Threads in a kernel are subdivided into workgroups, which are further subdivided into wavefronts (WF). All work-items in a WF are executed in lock step on the SIMD units of a compute unit (CU). Additionally, AMD's GCN3 architecture includes a scalar unit with corresponding scalar instructions. The use of these scalar instructions is transparent to the high-level application programmer, as they are generated by the compilation toolchain and are inserted into the WF's instruction stream, which also includes vector instructions.

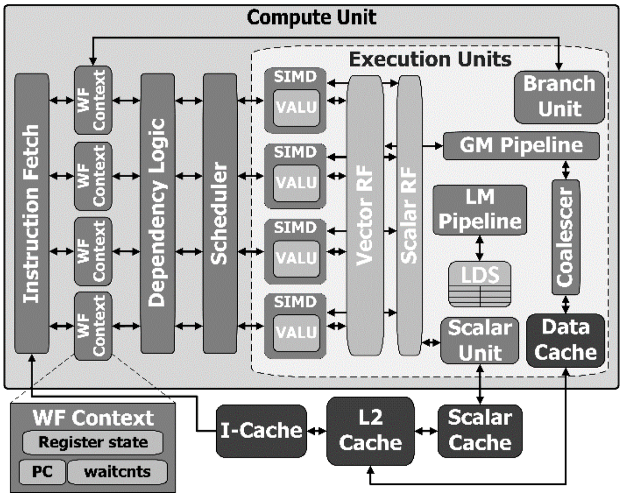


Figure 2: Block diagram of the compute unit pipeline.

B. GPU Hardware

CUs are the scalable execution units that are instantiated many times within a GPU. GPUs also typically include a multi-level cache hierarchy to improve memory bandwidth and command processors (CP), which are also called packet processors using HSA terminology [25], to manage work assignment.

Figure 2 provides a high-level block diagram of the CU model in gem5, which is based on AMD's GCN3 architecture [2]. Each CU contains four SIMD engines, a scalar unit, WF slots, local and global memory pipelines, a branch unit, a scalar register file (SRF), a VRF, a private L1 data cache, memory coalescing logic, and an on-chip local data share (LDS). Each CU is connected to shared L1 scalar data and instruction caches, which connect to memory through a shared L2 cache.

The GCN3 design splits the CU into four separate sets of SIMDs, each of which executes a single instruction (same PC) in lockstep on sixteen lanes. Thus, a single 64-work-item WF instruction is executed across 4 cycles. The GCN3 scalar unit's primary purpose is to handle control flow and to aid address generation.

HSAIL	GCN3
# Return Absolute WI ID workitemabsid $v0, 0;	# Read AQL Pkt s_load_dword s10, s[4:5], 0x04 # Wait for number of s_loads = 0 s_waitcnt lgkmcnt(0) # Extract WG.x Size from Pkt s_bfe s4, s10, 0x100000 # Calculate WG.x ID s_mul s4, s4, s8 # Calculate Global TID v_add v117, vcc, s4, v0

Table 1: Instructions for obtaining work-item ID.

III. Abstraction and Modeling Differences

A. *ABI Implementation*

A key component of GPU execution that is not captured when running HSAIL kernels is the ABI. The ABI defines the interface between the application binary (e.g., ELF), ISA, and operating system (OS). Examples of things that an ABI specifies include: function calling conventions, where data segments are placed in memory, the location of values such as the program counter (PC), which registers are used for argument passing, and the meaning of arguments passed to ISA system call instructions.

The ABI for GCN3 kernels dictates which registers must be initialized, where special values (e.g., condition codes) are stored, how code is loaded, and dispatch packet format. In contrast, HSAIL lacks an ABI definition, and allows individual vendors to customize the implementation to the details of their hardware. The result of using HSAIL is that simulators must simplify the functionality of certain instructions and features; however, the remaining subsections point out that without supporting the full ABI, many important microarchitecture interactions are missed.

1) Kernel Launch and State Initialization

One of the key aspects of GPU simulation that is particularly affected by the lack of an ABI is the kernel launch flow. In preparation for kernel launch, the runtime, kernel driver, and the CP perform several tasks, most notably initialization of register state and kernel arguments.

The kernel launch flow for HSAIL interprets the real kernel launch packet (via the HSA packet processor) and extracts relevant information, such as CU resource requirements and workgroup sizes. The simulator gathers and stores this information in order to service HSAIL instructions, which amounts to using a simulator-defined ABI. In some cases (e.g., when gathering kernel arguments) the simulator must maintain state that is not visible to the IL.

For GCN3 kernels, the real ABI information allows the simulator to model ABI initialization similar to real hardware. The CP traverses runtime data structures to extract values and initialize the necessary register state. The loader also inspects data structures in the ELF binary to obtain other pertinent information. Before a kernel launches, the required values (kernel argument addresses, workgroup sizes, etc.) are loaded into the register files, and the GCN3 instructions are aware of the semantics of each initialized register.

HSAIL	GCN3
#load kernarg at addr %arg1 ld_kernarg $v[0:1], [%arg1]	# mv kernarg base to v[1:2] v_mov v1, s6 v_mov v2, s7 #load kernarg into v3 flat_load_dword v3, v[1:2]

Table 2: Instructions for kernarg address calculation.

As an example, it is often beneficial for a work-item to know its thread ID (e.g., to index into an array). HSAIL can obtain an ID using one instruction, and Table 1 shows that this functionality requires several GCN3 instructions. Specifically, the GCN3 code must first obtain the workgroup size from its launch packet, whose address is stored in s[4:5]. Then each lane multiplies the size by its workgroup ID, which is stored in s8. Finally, each work-item adds its base ID within the WF (stored in v0), resulting in the global ID. As a result, the single *absworkitemid* instruction is expanded into five GCN3 instructions as required by the ABI.

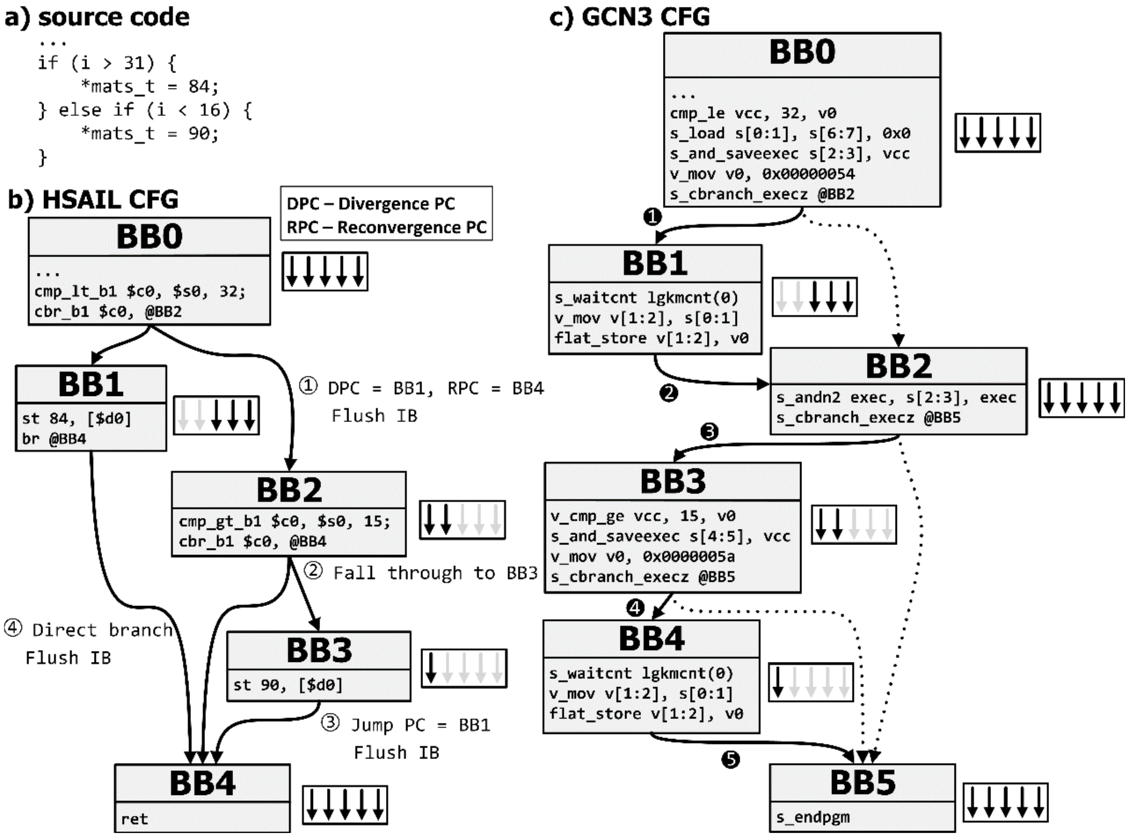
The expansion of HSAIL's absworkitemid instruction is one example that demonstrates how the lack of ABI can overly simplify the instruction stream and omit important register and memory accesses. The remainder of this section highlights several more examples.

2) Accessing Special Memory Segments

An expansion similar to that of the absworkitemid instruction, occurs for memory instructions as well. Comparable to typical CPU memory addressing, GPU memory address generation often applies an offset to a known base address stored in one or more registers. This is particularly the case for accesses to special memory regions, known as *segments*, defined by the HSA standard [25][26].

HSA specifically defines the following memory segments in the virtual address space, which have special properties and addressing requirements: *global*, *readonly*, *kernarg*, *group*, *arg*, *private*, and *spill* [25][26]. For example, the private memory segment has a base address that is shared per process, and individual work-items must use several offsets and stride sizes to get the base address for their portion of the private segment. The information required to obtain an individual work-item's private segment address is stored in a private segment descriptor, which occupies four scalar registers. In HSAIL, segment-specific memory instructions imply a base address that is usually maintained by the simulator, whereas GCN3 only includes segment-agnostic memory instructions. Specifically, GCN3 kernels store their segment base addresses in registers that are set by previously executed ISA instructions, or by the ABI initialization phase of dispatch.

Table 2 shows how the ABI impacts kernel argument accesses in GCN3 and HSAIL, respectively. Kernel argument memory is defined as a special HSA segment, similar to private memory. For GCN3, the ABI specifies that the



```
...
if (i > 31) {
    *mats_t = 84;
} else if (i < 16) {
    *mats_t = 90;
}
```

```
BB0
...
cmp_lt_b1 $c0, $s0, 32;
cbr_b1 $c0, @BB2

BB1
st 84, [$d0]
br @BB4

BB2
cmp_gt_b1 $c0, $s0, 15;
cbr_b1 $c0, @BB4

BB3
st 90, [$d0]

BB4
ret
```

```
BB0
...
cmp_le vcc, 32, v0
s_load s[0:1], s[6:7], 0x0
s_and_saveexec s[2:3], vcc
v_mov v0, 0x00000054
s_cbranch_execz @BB2

BB1
s_waitcnt lgkmcnt(0)
v_mov v[1:2], s[0:1]
flat_store v[1:2], v0

BB2
s_andn2 exec, s[2:3], exec
s_cbranch_execz @BB5

BB3
v_cmp_ge vcc, 15, v0
s_and_saveexec s[4:5], vcc
v_mov v0, 0x0000005a
s_cbranch_execz @BB5

BB4
s_waitcnt lgkmcnt(0)
v_mov v[1:2], s[0:1]
flat_store v[1:2], v0

BB5
s_endpgm
```

Figure 3: Managing control flow (HSAIL vs. GCN3). Note numbers indicate order of operations.

kernel argument base address must be placed in two scalar registers (s[6:7] in this example) and instructions are generated to move this address into the appropriate vector registers needed for the load's address operand. HSAIL, however, has no such ABI, therefore it uses abstract labels such as %arg1 to indicate the address for the first kernel argument. Because of this, HSAIL is unaware of the value redundancy of moving a single scalar value into each vector lane, and simply loads address operands, perhaps even from functional simulator state rather than from main memory.

B. Awareness of Microarchitecture Features

1) GCN Scalar Pipeline

HSAIL is a SIMT ISA, with each work-item representing one thread, whereas GCN3 is a vector ISA (i.e., the execution mask is exposed to the ISA) with some instructions targeting a scalar pipeline. Fundamentally, HSAIL instructions define the execution semantics of individual work-items, whereas GCN vector instructions semantically represent the execution of an entire WF. Each ISA has a unique view of the underlying hardware resources, and direct execution of HSAIL requires much more complex hardware. For example, the finalizing compiler (a.k.a. finalizer) inserts *waitcnt* instructions into the GCN3 instruction stream (see §III.B.2)) to greatly simplify dependency management. Similarly, the finalizer inserts scalar instructions within the GCN3 instruction stream, alongside vector instructions, in order to manage the WF's control flow and perform other scalar tasks such as address generation. The scalar unit is shared among all SIMD units within the compute unit and is also responsible for handling synchronization and pipeline scheduling via workgroup barrier instructions, NOPs, and waitcnts.

2) Dependency Management

GCN3's dependency management is another example of hardware-software co-design that enables GPUs to be fast, efficient throughput engines. Specifically, AMD GPUs do not use a hardware scoreboard, although minimal dependency logic (e.g., bypass paths) exists. Instead of relying on dedicated hardware, it is the responsibility of the finalizer to ensure data dependencies are managed correctly. For deterministic latencies, the finalizer will insert independent or NOP instructions between dependent instructions. For memory instructions, whose timing is not deterministic, waitcnt instructions are inserted to ensure stale data is not read. Waitcnt instructions will stall execution of a WF until the number of memory instructions in flight equals the specified value. For instance, if 0 is specified, the waitcnt instruction will stall the WF until all prior memory instructions complete. Table 1 demonstrates how a waitcnt may be used. In this example, the waitcnt

HSAIL

```
# Perform Divide
div $v[17:18], $v[11:12], $v[1:2]
```

GCN3

```
# Scale Denominator (D)
v_div_scale v[3:4], vcc, v[1:2], v[1:2], s[4:5]
v_mov v[5:6], s[4:5]

# Scale Numerator (N)
v_div_scale v[5:6], vcc, v[5:6], v[1:2], v[5:6]

# Calculate 1/D
v_rcp v[7:8], v[3:4]

# Calculate Quotient and Error
v_fma v[9:10],-v[3:4],v[7:8],1.0
v_fma v[7:8],v[7:8],v[9:10],v[7:8]
v_fma v[9:10],-v[3:4],v[7:8], 1.0
v_fma v[7:8],v[7:8],v[9:10],v[7:8]
v_mul v[9:10],v[5:6],v[7:8]
v_fma v[3:4],-v[3:4],v[9:10], v[5:6]

# Calculate Final Q
v_div_fmas v[3:4],v[3:4],v[7:8],v[9:10]

# Fixup Q
v_div_fixup v[1:2],v[3:4],v[1:2],s[4:5]
```

Table 3: Instructions for 64-bit floating point division.

ensures that the *s_bfe* instruction's dependence on s10 is satisfied before allowing execution to continue beyond the waitcnt.

HSAIL instructions are created by the compiler without regard for dependent instructions, therefore the simulator must include scoreboard logic to manage dependent instructions even though the logic does not exist in the actual GPU. Furthermore, the lack of intelligent instruction scheduling increases stalls due to RAW or WAR dependencies.

C. Instruction Set Functionality and Encoding

The previous subsection identified several situations related to the ABI and microarchitecture features where a few HSAIL instructions expanded into many more instructions when executing GCN3. This subsection identifies a few more situations that specifically relate to instruction set functionality and its impact on instruction fetch. Specifically, this section looks at three examples: 1) control flow management, 2) floating point division, and 3) instruction fetch.

1) Managing Control Flow

A unique issue that arises in GPUs is control flow divergence. Control flow divergence occurs when not all work-items in a WF follow the same control flow path. When handling divergence, GPUs execute both paths of a branch serially, possibly diverging multiple times down each path until the paths reconverge. A key difference between GCN3 and HSAIL is the visibility of the execution mask to the ISA. Because GCN3 instructions can view and manipulate the execution mask, the compiler is able to layout basic blocks (BB) in the control flow graph (CFG) serially, thereby mitigating the need for a reconvergence stack (RS) when control flow is reducible, which is the common case. In the case of irreducible control flow, GCN3 kernels will manage a software reconvergence stack, however this was not encountered in our benchmarks.

8 Compute Units, each configured as described below:	
GPU Clock	800 MHz, 4 SIMD units
Wavefronts	40 (each 64 lanes)/oldest-job first
D$ per CU	16kB, 64B line, fully associative
VRF/SRF	2,048 vector/800 scalar
Memory Hierarchy	
L2$ per 4CUs	512kB, 64B line, 16-way write-through (write-back for R data)
I$ per 4 CUs	32kB, 64B line, 8-way
DRAM	DDR3, 32 Channels, 500 MHz

Table 4. Simulation configuration.

When executing the SIMT instructions defined in the IL, simulators typically manage control flow divergence using an RS. If the IL does not identify reconvergence points, the simulator will parse the kernel code and identify the immediate post-dominator instructions. When executing the kernel, the simulator's RS stores the divergent and reconvergent PCs for each branch, as well as the execution mask. The result is that the simulator can estimate the performance of SIMT code running on vector hardware where each WF has only a single PC and both paths of the branch are taken in sequence.

Handling control flow divergence using a RS does not represent how AMD hardware handles control flow, and thus is fundamentally problematic for simulation when estimating front-end performance. Specifically, when reaching a reconvergence point, the simulator will often need to initiate a jump to the divergent PC causing the IB to be flushed and instructions at the divergent PC to be fetched. These extra IB flushes force the WF to stall, but they do not occur in real hardware because the ISA is able to simply mask off divergent lanes.

Figure 3 explains in detail how HSAIL uses an RS to manage control flow and how it compares to GCN3. The example illustrates a simple if-else-if statement (Figure 3a) where each work-item writes 84 or 90 to memory depending on the outcome of the condition statements. The illustration assumes five work-items per WF.

For HSAIL, the high-level compiler generates the CFG using SIMT instructions (Figure 3b) and the single-threaded nature of the encoding causes IB flushes not encountered when executing GCN3. Specifically, when reaching the branch at the end of BB0 the simulator first executes the taken path and pushes the PC and execution mask for the divergent path on the RS ①. Jumping to BB2 requires the IB to be flushed because BB2 is not sequentially after BB0, however the subsequent execution of BB3 does not require an IB flush because the RS detects that the branch in BB2 goes to the RPC (the fall through path) ②. After executing BB3 the top entry from the RS is popped and the PC jumps to BB1 ③. This jump also requires an IB flush; then after executing BB1, a final IB flush is required before executing BB4 ④. In the end, the simulator encountered three IB flushes in order to approximate the execution of the example SIMT instruction stream running on vector hardware.

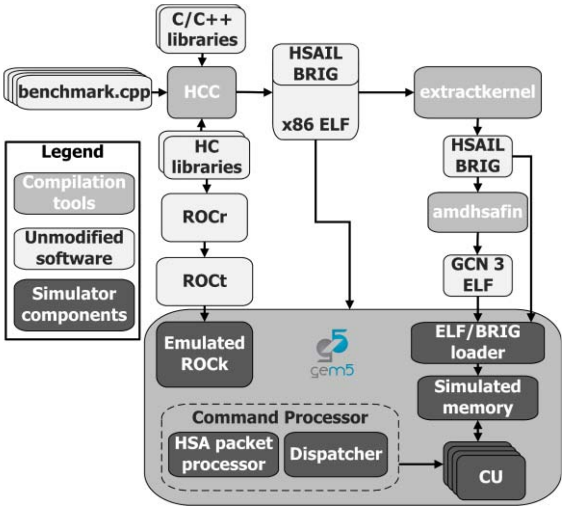


Figure 4: ROCm gem5 compilation flow.

For GCN3, the finalizer generates the CFG using both scalar and vector instructions and uses branch instructions only to bypass completely inactive paths of execution (Figure 3c). These optimized bypass paths are highlighted with dashed arrows and are not executed in this example because collectively the work-items execute all basic blocks. By ensuring the CFG is reducible, and through the use of predication, the main control flow of the program is handled without taking branches or simulator-initiated jumps ❶❷❸❹❺. No IB flushes are needed, and the front-end of the GPU executes without stalling. The result is the GCN3 code executes the divergent control flow far more efficiently than HSAIL.

2) *Floating Point Division*

Floating point division is another area for which HSAIL greatly simplifies its ISA. In HSAIL floating point division is performed by a single *div* instruction; GCN3 on the other hand, relies on several floating point instructions to implement the Newton-Raphson method [3], as shown in Table 3. While the instruction expansion can be approximated using a comparable latency for the HSAIL instruction, the effects of increased register pressure can only be simulated using the GCN3 code.

3) *Instruction Fetching*

Instruction fetch is particularly challenging to model under HSAIL simulation due to fundamental differences in encoding between HSAIL and GCN3. Specifically, HSAIL kernels are encoded in a BRIG binary, and HSAIL instructions are not meant to be fetched from memory or directly executed and decoded by hardware. Instead, HSAIL instructions are encoded as verbose data structures and may require several kilobytes of storage. This is because the BRIG format was designed to be easily decoded by finalizer software. When simulating HSAIL, all instructions in a BRIG binary are extracted and decoded into simulator specific instruction objects as the GPU kernel is loaded. In gem5's HSAIL simulator, each instruction is then represented as a fixed length 64b unsigned integer value that is stored in simulated memory and provides a way for the simulator to find the corresponding instruction object. In contrast, GCN3 uses variable length instructions: 32b, 64b, or 32b with a 32b inline constant.

Workload	Description
Array BW	Memory streaming
Bitonic Sort	Parallel merge sort
CoMD	DOE Molecular-dynamics algorithms
FFT	Digital signal processing
HPGMG	Ranks HPC systems
LULESH	Hydrodynamic simulation
MD	Generic Molecular-dynamics algorithms
SNAP	Discrete ordinates neutral particle transport app.
SpMV	Sparse matrix-vector multiplication
XSBench	Monte Carlo particle transport simulation

Table 5: Description of evaluated workloads.

IV. METHODOLOGY

Our evaluation methodology relies on gem5's GPU model, which AMD recently released [10], and the open-source ROCm software stack. Prior to this work gem5 only supported OpenCL™ [27], executed HSAIL instructions, and emulated the runtime API. A similar approach has been used by other GPU simulators [11][35]. In contrast, our GCN3 implementation faithfully supports the ROCm stack [8] and the GCN3 ISA and only emulates kernel driver functionality. Table 4 summarizes the key system parameters we use to compare HSAIL and GCN3.

Figure 4 shows several of the components that were added to the gem5 GPU compute model to execute GCN3 kernels. It also provides an overview of the compilation and code loader flow for the model. All programs are compiled using the heterogeneous compute compiler (HCC) [4]. HCC is an open-source C++ compiler for heterogeneous compute that compiles both CPU and accelerator code from single source via its compiler frontend. The simulator is able to run HCC applications on an unmodified version of the user-level ROCm stack [8]. This includes the HCC libraries, the ROC runtime, and the user-space thunk driver that interfaces with the kernel-space driver. The kernel driver is emulated in gem5's OS emulation layer.

HCC compiles and links C++ source, and produces a single multi-ISA ELF binary. The resultant ELF binary includes both the x86 instructions, and an embedded BRIG binary that stores the HSAIL instructions. The HCC toolchain includes a script called *extractkernel* that allows a user to extract the BRIG image directly from the multi-ISA ELF binary. To generate GCN3 code, we use AMD's offline finalizer tool: *amdhsafin* [6]. This allows us to generate and execute separate GPU kernel binaries containing HSAIL and GCN3 instructions from the same application. Table 5 describes the applications used in this study [1][5].

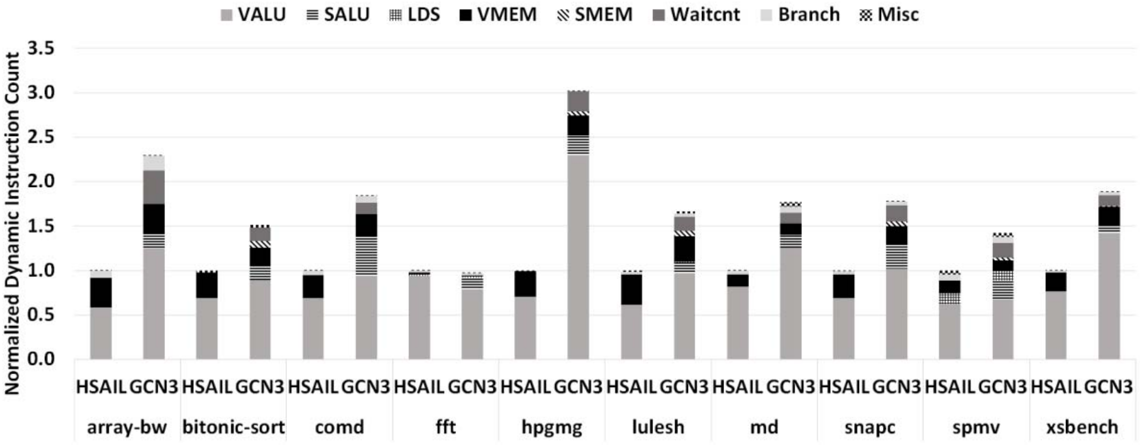


Figure 5: Dynamic instruction count and breakdown normalized to HSAIL.

V. Behavior Differences

A. Dynamic Instructions

One of the starkest behavior differences between HSAIL and GCN is their dynamic instruction executions. Figure 5 breaks down the types of HSAIL instructions executed for each workload and compares that to GCN3 (note that all HSAIL ALU instructions are vector instructions). The *Misc* classification includes NOP, barrier, and end-of-program instructions. With FFT being the notable exception, Figure 5 shows that the GCN3 kernels execute 1.5×-3× more dynamic instructions than HSAIL. As explained in §III.C and validated by comparing the breakdowns of instruction types, the significant expansion of GCN3 instructions cannot be attributed to a single factor. Instead, most workloads see a diverse expansion of instruction types. For instance, CoMD has one of the highest percentages of HSAIL branch instructions, which are then expanded to many GCN3 scalar ALU and branch instructions. Meanwhile, Array BW, LULESH, and HPGMG have a high number of HSAIL vector memory operations that get expanded into many scalar ALU, scalar memory, and waitcnt instructions.

FFT is the one benchmark whose GCN3 execution has the most similar behavior to HSAIL. This is because FFT is the most compute-bound application in our suite with around 95% of instructions being ALU instructions (for GCN3 this includes scalar ALU instructions) and very few branches. FFT uses many conditional move instructions, which mitigate the need for extra control flow instructions. It should also be noted that FFT executes no divide instructions, thus there is minimal GCN3 code expansion.

B. VRF Bank Conflicts

Contention for the VRF can have a significant impact on performance. It is crucial that simulation accurately reflects the VRF usage of kernels under test. Architecturally, HSAIL (which is register-allocated) allows up to 2,048 32-bit architectural vector registers per WF, whereas GCN3 only allows 256. In addition, GCN3 allows up to 102 scalar registers per WF. HSAIL has no concept of a scalar register file, so all HSAIL register values must fit in the VRF.

Figure 6 shows that GCN3 encounters approximately one third the port conflicts of HSAIL. The reasons for this are two-fold. First, many GCN vector instructions use at least one scalar register operand (e.g., a base address) whereas all HSAIL operands must come from the VRF. Second, the GCN3 finalizer inserts independent instructions between instructions that have register dependencies. The result is WFs simultaneously executing on the same compute unit place less demand on the VRF.

Figure 7 confirms this behavior by plotting the median reuse distance for vector registers. We define the reuse distance as the number of dynamic instructions executed by a WF between reuse of a vector register. The median reuse distance for GCN3 is nearly twice that of HSAIL confirming the significant impact the finalizer's intelligent instruction scheduling has on execution behavior. The notable exception is FFT. As previously mentioned in §V.A, FFT is compute-bound, and executes few instructions that require expansion or drastically change register file access behavior, therefore HSAIL is able to capture its reuse distance well. While FFT's relative reuse distance for GCN3 is essentially the same as it is for HSAIL, this workload has one of the highest absolute reuse distances of any of our workloads. Under GCN3, FFT takes advantage of the scalar unit for its many compare operations and conditional moves, HSAIL on the other hand may only use vector registers. Thus, while HSAIL and GCN3 may reuse individual VRF entries sparsely, HSAIL accesses many more vector registers, thereby

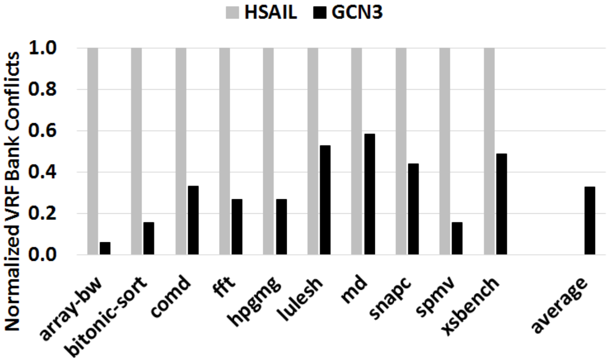


Figure 6: Number of VRF bank conflicts.

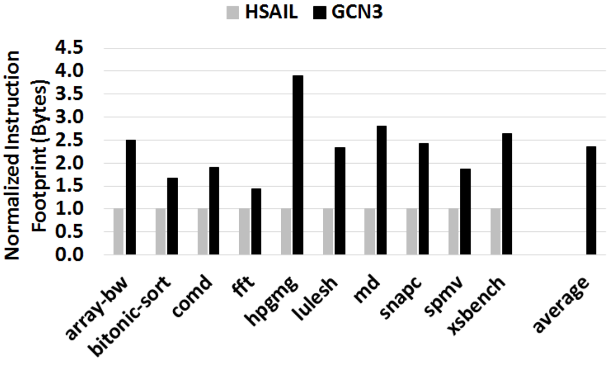


Figure 8: Instruction Footprint.

increasing the probability of its vector operands being mapped to the same bank and encountering conflicts.

C. *Instruction Fetching*

IL instructions are inaccurate for representing a kernel's instruction footprint. As described in §III.C.3) the HSAIL instructions are represented in software data structures, and the encoding was primarily designed for fast interpretation by finalization tools, rather than a hardware decoder. To approximate the storage of HSAIL instructions, the gem5 GPU model encodes each instruction using 64 bits. Figure 8 compares this approximation to the true instruction footprint of the corresponding GCN3 kernels and shows that even with the conservative approximation, HSAIL kernels underrepresent the true instruction footprint by 2.4× on average. This large relative difference is due to GCN3's code expansion, but for most applications even the GCN3 footprint fits in the 16KB I-cache.

The noteworthy exception to this trend is LULESH, which is composed of 27 unique kernels and has an absolute instruction footprint of 16KB and 40KB for HSAIL and GCN3, respectively. Because the GCN3 instruction footprint significantly exceeds the L1 instruction cache size of 16KB, LULESH sees a 10× increase in L1 instruction fetch misses and an effective L1 fetch latency increase of 8×. The result is a substantial increase in overall runtime (see Figure 12).

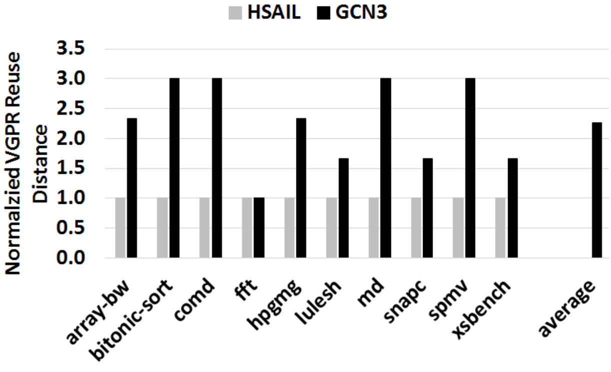


Figure 7: Median vector register reuse distance.

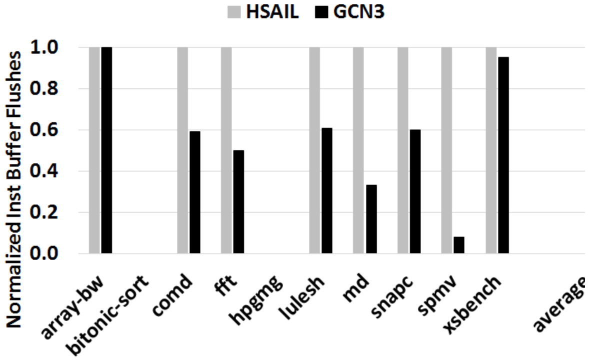


Figure 9: Instruction buffer flushes.

Beyond the instruction footprint, the higher-level IL abstraction can also cause significant differences in instruction fetch behavior due to control flow management. §III.C.1) described fetch buffer implications of using HSAIL's SIMT ISA, and Figure 9 confirms that GCN3 kernels require fewer than half as many IB flushes as their equivalent HSAIL kernels. Bitonic-Sort and HPGMG do not contain branches, and instead use predication to manage conditionals. Array BW and XSBench have simple control flow constructs (simple loops or direct branches) that are amenable to HSAIL execution. The result is that, in general, GCN3 kernels execute control flow more efficiently than their HSAIL kernel counterparts.

D. *VRF Value Redundancy*

Identifying the redundancy of individual operand values within the VRF has been the focus of several recent works [29][31][39]. We perform a simple case study to evaluate the uniqueness of operand values across all VRF accesses in our applications. Figure 10 shows the uniqueness of lane values for VRF reads and writes. A higher bar means more unique values observed. To calculate operand uniqueness, we take the cardinality of the set of all unique lane values observed for VRF accesses, divided by the total number of lanes accessed (e.g., if we have 32 active lanes for a VRF write, and we write only 8 unique values, that is a uniqueness of 25%).

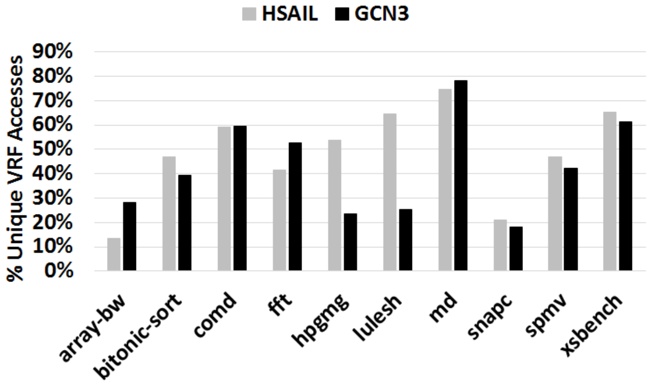


Figure 10: Uniqueness of VRF lane values.

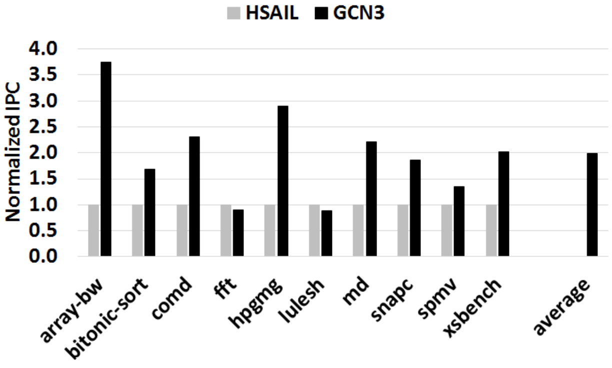


Figure 11: Normalized IPC.

As can be seen in Figure 10, very different values can be observed at the VRF simply because of the ISA, which could lead to incorrect conclusions about the effectiveness of any value compression or reuse technique. One would expect that GCN3 codes would show a universal increase in value uniqueness because of GCN3's ability to execute scalar instructions and use scalar operands (even for vector instructions), however this is not the case. In the common case, value redundancy that is inherent to the kernel remains, and in some cases redundancy is exacerbated by the GCN3 ISA.

Two reasons why GCN3 codes are not able to improve value uniqueness are worth highlighting: 1) the scalar unit in GCN3 is not generally used for computation, and 2) HSAIL's abstract ABI hides redundant base addresses when accessing special segments because the addresses are not stored registers.

Contrasting the results for Array BW and LULESH in Figure 10 demonstrates this issue. Array BW is a simple kernel that reads from a buffer stored in global memory in a very tight loop, and LULESH has many small kernels where memory accesses to special segments make up a non-trivial portion of kernel instructions. Here Array BW drastically underestimates the uniqueness of operand values, showing a uniqueness of about 12%. GCN3, however shows a significant improvement in value uniqueness at nearly 30%. Because Array BW is dominated by global memory loads, both applications experience a low uniqueness; however, under GCN3 the instructions that update the address as it traverses the array are able to utilize scalar values and explicit vector registers that hold each lane's unique ID. These actions are implicit under HSAIL.

LULESH demonstrates the opposite effect: HSAIL drastically overestimates the value uniqueness (65% for HSAIL and 25% for GCN3) because it accesses special segments (kernarg and private), whose addresses calculations are hidden from HSAIL and exposed to GCN3. In addition, floating point operations in HSAIL limit the visibility of many intermediate values. For example, the expansion of HSAIL's floating point divide instruction shown in Table 3 may cause increased value redundancy when multiple lanes generate the same values, such as NaN or 0.

E. Throughput and Runtime

Figure 11 compares the instructions per cycle (IPC) rates of HSAIL and GCN3, and, as expected, GCN3 generally achieves higher IPC because in many situations multiple GCN3 instructions are equivalent to a single HSAIL instruction. While IPC is not necessarily commensurable across ISAs, it is one of the key metrics used to evaluate performance, therefore it should track the underlying microarchitecture resource utilization accurately. HSAIL, therefore, will always do a poor job of expressing how well a workload utilizes hardware resources. In fact, some resources such as the scalar unit are never used by HSAIL, thus confirming the pitfall of using HSAIL kernels to evaluate resource usage.

The noticeable exceptions to this trend are FFT and LULESH, which encounter slight IPC degradations for GCN3. FFT's degradation is due to the fact that FFT is one application that does not experience any GCN3 instruction expansion, but some HSAIL ALU instructions are translated to GCN3 scalar ALU instructions. Since there are four vector ALU (VALU) units per CU, but only one scalar ALU unit, this results in increased contention and a slight slowdown for the GCN3 kernel. Meanwhile, LULESH's degradation is due its significant number of dynamic kernel launches, which number in the thousands. HSAIL does not have an ABI that specifies where kernarg base addresses are specified (in GCN3 the ABI specifies which registers they are stored in) therefore the simulator provides them at no cost. GCN3 kernels must retrieve kernarg base addresses from registers, and due to LULESH's larger register demand, this leads to extra cycles waiting for data in GCN3 when compared to HSAIL.

Overall, it is hard to predict and compensate for the runtime implications of using HSAIL, which underscores the importance of using GCN3. Simulating HSAIL instructions will be optimistic by assuming smaller instruction footprints and lower resource utilization.

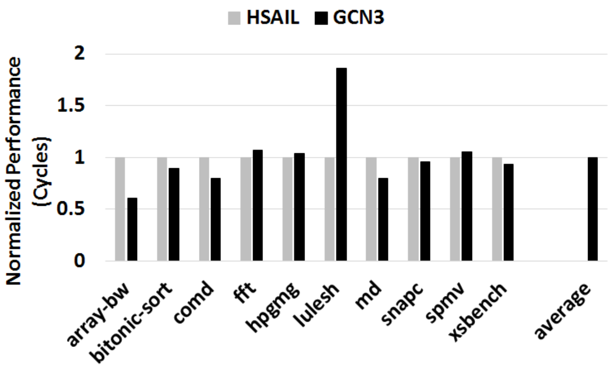


Figure 12: Normalized Performance

However, simulating HSAIL instructions will be pessimistic with regards to instruction scheduling, VRF port conflicts, and IB flushes. The result is HSAIL's runtime can vary significantly from GCN3. In particular, Figure 12 shows that HSAIL runtime is 1.6× higher for Array BW, but GCN3 is 1.85× higher for LULESH.

VI. Behavior Similarities

A. Data Footprint

Table 6 shows the data footprint for each benchmark. Fundamentally the same computation is performed on the same data regardless of which ISA is used, therefore the data footprints are precisely the same for most benchmarks. The notable exceptions, FFT and LULESH, have footprints that are 20% and 4× larger, respectively, when run in HSAIL. This is because of the simplified way the simulator handles special memory segments, such as the private or spill segments, due to HSAIL's lack of an ABI. FFT and LULESH are the only applications in our suite that use special memory segments (spill and private, respectively). Both applications use special segments to spill and fill because of their large register demands.

Because HSAIL does not have a known location from which it may retrieve its segment base addresses, the simulator maps and manages the memory for special segments independent of the real runtime. Each time a kernel is dynamically launched, the emulated HSAIL ABI must allocate new segment mappings.

GCN3 kernels require support for a robust ABI to run correctly, therefore we rely on the real runtime's support for the ABI, and model the hardware with higher fidelity. When running GCN3 kernels the simulator does not maintain segment mappings, and because the runtime allocates segment memory on a per-process basis, as opposed to a per-kernel basis, the memory is reused across kernel launches within the same process.

B. SIMD Utilization

Previously, §V highlighted behavior differences between HSAIL and GCN3 to justify simulating at the machine ISA level. There are, however, a few aspects of HSAIL's execution behavior that closely match GCN3, which can justify using HSAIL instructions in certain situations. In particular, Table 6 compares the VALU lane utilization for GCN3 and HSAIL and shows HSAIL utilizes the SIMD lanes within a few percent of GCN3. This result shows that while many aspects of the microarchitecture are unknown to HSAIL, the overall computation requirements and work-item activity of a kernel are dependent on the program behavior, not the ISA description.

	Data Footprint		SIMD Utilization	
	HSAIL	GCN3	HSAIL	GCN3
Array-BW	386kb	386kB	100%	100%
Bitonic Sort	512kB	512kB	100%	100%
CoMD	4MB	4MB	23%	21%
FFT	232MB	192MB	100%	99%
HPGMG	9MB	9MB	100%	100%
LULESH	36MB	8MB	98%	96%
MD	960kB	960kB	100%	100%
SNAPC	2MB	2MB	100%	100%
SPMV	3MB	3MB	67%	72%
XSBench	58MB	58MB	53%	52%

Table 6: Similar stats HSAIL vs. GCN3.

VII. Hardware Correlation

To determine the overall impact of evaluating GPU kernels using an IL, we compare the simulated execution time our applications, for both GCN3 and HSAIL, against GCN3-based [38] hardware. In particular, hardware data are collected on an AMD Pro A12-8800B APU. We run the same codes in both simulation and hardware, and we use the same binaries in the case of GCN3 execution. We use the Radeon Compute Profiler (RCP) [7] to collect hardware performance counters. Table 4 describes the GPU configuration we simulate, which matches the hardware CUs and cache system as closely as possible.

The mean absolute runtime error (averaged across all kernels) is shown in Table 7. While not shown in the table due to space, GCN3 error remains consistent across kernels, while HSAIL error exhibits high variance. For both simulated ISAs, the correlation is quite high. This shows that simulating under either ISA may provide performance data that correlates well with hardware (i.e., it preserves performance trends). The next set of columns in the table show the average absolute error with respect to hardware. The error for GCN3 simulations is around 45%. In contrast, the HSAIL error is significantly higher than GCN3 at 75%. It should be noted that we did not attempt to eliminate the sources of error in the open-source model itself because we are only interested in understanding the error that is *inherent* when evaluating applications under an IL. In addition to execution time, we are able to collect several microarchitecture-agnostic performance counters using the RCP. In particular, the dynamic instruction count, instruction mix, and SIMD utilization are all 100% accurate when executing GCN3.

These results demonstrate that, while some error exists in the model itself, HSAIL adds significant additional error

Correlation		Avg. Absolute Error	
HSAIL	GCN3	HSAIL	GCN3
0.972	0.973	75%	42%

Table 7: Hardware correlation and error.

that is difficult to predict. GCN3 simulation error, on the other hand, is due only due to modeling error [12][15][20][23].

VIII. Related Work

A. GPU Simulation

The most closely related work to ours is Barra [17]. The authors claim that using PTX may lead to low accuracy, and they perform simulations using a reverse-engineered ISA (decuda [36]) for the Tesla architecture [30]. However, the paper only focuses on functional simulation and micro-architiecture-agnostic metrics. In comparison, our work demonstrates why microarchitects need to consider the differences in ISA abstraction.

GPGPU-Sim [11] is currently the most popular general-purpose GPU simulator, and it models NVIDIA GPUs while executing PTX and GT200 SASS. The source code is linked to a custom runtime, which intercepts all GPGPU function calls and emulates the effects. gem5-GPU [34] integrates GPGPU-Sim with gem5.

Multi2Sim [35] is a simulation framework for CPU-GPU heterogeneous computing, and it models AMD's GCN1 ISA. Multi2Sim also uses custom runtime implementations. MacSim [28] is a trace-driven heterogeneous architecture simulator that models PTX. The PTX traces are generated using GPUOcelot [21] (a dynamic compilation framework that works at a virtual ISA level), and MacSim converts trace instructions into RISC style micro-ops.

Attila [19] models the ARB ISA for OpenGL applications, and does not have a GPU compute model. GpuTejas [32] provides a parallel GPU simulation infrastructure; however, it does so using PTX. HSAemu [22] is a full-system emulator for the HSA platform and uses HSAIL.

Other than Attila, which models the ARB ISA for OpenGL applications and does not have a GPU compute model, these simulators model various GPU microarchitectures and memory systems. However, unlike our work, most of these simulators model ILs such as PTX or HSAIL. Even in situations when simulators use the machine ISA, such as Multi2Sim executing the AMD's GCN1 ISA [9] or GPGPU-Sim executing SASS in a specialized encoding called "PTXplus," these simulators still emulate the runtime rather than supporting the full user-level software stack.

B. ISA Comparison

Blem et al. [13] provide a detailed analysis of the effects the ISA has on a microprocessor's energy, power, and performance. Their work is entirely focused on how the ARM and x86 ISAs affect the microarchitecture on which they run, and they conclude that, because most microprocessors execute RISC-like micro-ops, the high-level ISA does not matter much. Their work does not evaluate the effects of running (or simulating) different ISAs on the exact same microarchitecture.

IX. Conclusion

We have shown that, beyond the microarchitecture, the ISA used to encode applications can have significant effects on any conclusions drawn from simulation results. While accuracy may be maintained for a few statistics, architects must be aware of the pitfalls we have pointed out when drawing conclusions based on IL simulation. There are inherent differences between the instructions themselves and what knowledge the ISA has about the microarchitecture. In addition, the lack of an ABI for ILs abstracts away key hardware/software interactions. These differences can have first-order effects on the observed usage of resources and secondary effects on overall application performance, thus motivating the use of a machine ISA.

Acknowledgment

The authors would like to thank Tony Tye for his help with understanding the HSA ABI and ROCm stack. The authors also thank Steve Reinhardt, who wrote much of gem5's initial HSAIL implementation. AMD, the AMD Arrow logo, Radeon, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies. OpenCL is a trademark of Apple Inc. used by permission by Khronos.

References

[1] AMD. "Compute Applications". GitHub Repository, 2017. http://github.com/AMDComputeLibraries/ComputeApps. Accessed: February 27, 2017.

[2] AMD. "Graphics Core Next (GCN) Architecture". AMD Whitepaper, 2012. http://amd.com/Documents/GCN_Architecture_witepaper.pdf. Accessed: February 27, 2017.

[3] AMD. "Graphics Core Next Architecture, Generation 3". AMD Technical Manual, 2016. http://gpuopen.com/wp-content/uploads/2016/08/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf. Accessed: February 27, 2017.

[4] AMD. "HCC". https://www.github.com/RadeonOpenCompute/hcc/wiki. Accessed: February 27, 2017.

[5] AMD. "HCC Sample Applications", GitHub Repository, 2016. http://github.com/RadeonOpenCompute/HCC-Example-Application. Accessed: February 27, 2017.

[6] AMD. "HSAIL Offline Finalizer", GitHub Repository, 2016. https://github.com/HSAFoundation/HSA-HOF-AMD. Accessed: February 27, 2017.

[7] AMD. "Radeon Compute Profiler", GitHub Repository, 2017. https://github.com/GPUOpen-Tools/RCP. Accessed: September 20, 2017.

[8] AMD. "ROCm: Platform for Development, Discovery and Education Around GPU Computing". http://gpuopen.com/compute-product/rocm. Accessed: February 27, 2017.

[9] AMD. “Southern Islands Series Instruction Set Architecture”. AMD Technical Manual, 2012. http://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf. Accessed: February 27, 2017.

[10] AMD. “The AMD gem5 APU Simulator: Modeling Heterogeneous Systems in gem5”. 2015. http://www.gem5.org/wiki/images/f/fd/AMD_gem5_APU_simulator_micro_2015_final.pptx. Accessed: February 27, 2017.

[11] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. “Analyzing CUDA Workloads Using a Detailed GPU Simulator”. In the proceedings of the *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 164–174, 2009.

[12] B. Black and J.P. Shen. “Calibration of Microprocessor Models”. In *IEEE Computer*, 31(5), pp. 59–65, 1998.

[13] E. Blem, J. Menon, and K. Sankaralingam. “Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures”. In the proceedings of the *19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–12, 2013.

[14] N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill, and D.A. Wood. “The gem5 Simulator”. In *SIGARCH Computer Architecture News*, 39(2), pp. 1–7, 2011.

[15] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli. “Accuracy Evaluation of gem5 simulator system”. In the proceedings of the *7th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, pp. 1–7, 2012.

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.H. Lee, and K. Skadron. “Rodinia: A Benchmark Suite for Heterogeneous Computing”. In the proceedings of the *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, 2009.

[17] S. Collange, M. Daumas, D. Defour, and D. Parello. “Barra: A Parallel Functional Simulator for GPGPU”. In the proceedings of the *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Compute and Telecommunication Systems (MASCOTS)*, pp. 351–360, 2010.

[18] Continuum Analytics. “ANACONDA”. http://continuum.io. Accessed: February 27, 2017.

[19] V.M. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa. “ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures”. In the proceedings of the *2006 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 231–241, 2006.

[20] R. Desikan, D. Burger, and S.W. Keckler. “Measuring Experimental Error in Microprocessor Simulation”. In the proceedings of the *28th International Symposium on Computer Architecture (ISCA)*, pp. 266–277, 2001.

[21] G.F. Diamos, A.R. Kerr, S. Yalamanchili, and N. Clark. “Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems”. In the proceedings of the *19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 353–364, 2010.

[22] J.-H. Ding, W.-C. Hsu, B.-C. Jeng, S.-H. Hung, and Y.-C. Chung. “HSAemu – A Full System Emulator for HSA Platforms”. In the proceedings of the *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 26:1–26:10, 2014.

[23] A. Gutierrez, J. Pusdesris, R.G. Dreslinski, T. Mudge, C. Sudanthi, C.D. Emmons, M. Hayenga, and N. Paver. “Sources of Error in Full-System Simulation”. In the proceedings of the *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 13–22, 2014.

[24] J. Hoberock. "Working Draft, Technical Specification for C++ Extensions for Parallelism, Revision 1". Technical Manual, 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3960.pdf. Accessed: February 27, 2017.

[25] HSA Foundation. “HSA Platform System Architecture Specification 1.1”. 2016. http://hsafoundation.com/standards. Accessed: February 27, 2017.

[26] HSA Foundation. “HSA Programmer’s Reference Manual 1.1: HSAIL Virtual ISA and Programming Model, Compiler Writer, and Object Format (BRIG)”. 2016. http://hsafoundation.com/standards. Accessed: February 27, 2017.

[27] Khronos Group™. “The Open Standard for Parallel Programming of Heterogeneous Systems”. https://www.khronos.org/opencl. Accessed: February 27, 2017.

[28] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho. “MacSim: A CPU-GPU Heterogeneous Simulation Framework”. Georgia Institute of Technology Technical Report, 2012.

[29] S. Lee, K. Kim, G. Koo, H. Jeon, W.W. Ro, and M. Annavaram. “Warped-Compression: Enabling Power Efficient GPUs through Register Compression”. In the proceedings of the *42nd International Symposium on Compute Architecture (ISCA)*, pp. 502–514, 2015.

[30] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. “NVIDIA Tesla: A Unified Graphics and Computing Architecture”. In *IEEE Micro*, 28(2), pp. 39–55, 2008.

[31] Z. Liu, S. Gilani, M. Annavaram, and N.S. Kim. “G-Scalar: Cost-Effectiv Generalized Scalar Execution Architecture for Power-Efficient GPUs”. In the proceedings of the *2017 IEEE International Symposium on High Performance Compute Architecture (HPCA)*, pp. 601–612, 2017.

[32] G. Malhotra, S. Goel, and S.R. Sarangi. “GpuTejas: A Parallel Simulator for GPU Architectures”. In the proceedings of the *21st International Conference on High Performance Computing (HiPC)*, pp. 1–10, 2014.

[33] NVIDIA. “Parallel Thread Execution ISA Version 5.0”. NVIDIA Technical Manual 2017. http://www.docs.nvidia.com/cuda/parallel-thread-execution. Accessed: February 27, 2017.

[34] J. Power, J. Hestness, M.S. Orr, M.D. Hill, and D.A. Wood. “gem5-gpu: A Heterogeneous CPU-GPU Simulator”. In *IEEE Computer Architecture Letters*, 14(1), pp. 34–36, 2015.

[35] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. “Multi2Sim: A Simulation Framework for CPU-GPU Computing”. In the proceedings of the *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 335–344, 2012.

[36] W.J. van der Laan. Decuda and Cudasm, the CUDA Binary Utilities Package, 2008. https://www.github.com/laanwj/decuda. Accessed: February 27, 2017.

[37] V. Volkov and J. W. Demmel. “Benchmarking GPUs to Tune Dense Linear Algebra”. In the proceedings of the *2008 ACM/IEEE Conference on Supercomputing (SC)*, pp. 31:1–31:11, 2008.

[38] K. Wilcox, D. Akeson, H.R. Fair, J. Farrell, D. Johnson, G. Krishnan, H. McIntyre, E. McLellan, S. Naffziger, R. Schreiber, S. Sundaram, and J. White. “4.8A 28nm x86 APU Optimized for Power and Area Efficiency”. In the proceedings of the *2015 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 1–3, 2015.

[39] D. Wong, N.S. Kim, and M. Annavaram. “Approximating Warps with Intra-Warp Operand Value Similarity”. In the proceedings of the *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 176–187, 2016.