Sim-D: a SIMD accelerator for hard real-time systems

Roy Spliet and Robert D. Mullins

Abstract—Emerging safety-critical systems require high-performance data-parallel architectures and, problematically, ones that can guarantee tight and safe worst-case execution times. Given the complexity of existing architectures like GPUs, it is unlikely that sufficiently accurate models and algorithms for timing analysis will emerge in the foreseeable future. This motivates our work on Sim-D, a clean-slate approach to designing a real-time data-parallel architecture. Sim-D enforces a predictable execution model by isolating compute- and access resources in hardware. The DRAM controller uninterruptedly transfers tiles of data, requested by entire work-groups. This permits work-groups to be executed as a sequence of deterministic access- and compute phases, scheduling phases from up to two work-groups in parallel. Evaluation using a cycle-accurate timing model shows that Sim-D can achieve performance on par with an embedded-grade NVIDIA TK1 GPU under two conditions: applications refrain from using indirect DRAM transfers into large buffers, and Sim-D's scratchpads provide sufficient bandwidth. Sim-D's design facilitates derivation of safe WCET bounds that are tight within 12.7% on average, at an additional average performance penalty of $\sim$9.2% caused by scheduling restrictions on phases.

Index Terms—Real-time and embedded systems, Parallel architectures

1 Introduction

Many emerging safety-critical systems require high-performance data-parallel architectures and, problematically, ones that can facilitate the provision of *hard real-time* (HRT) guarantees. Autonomous transport systems require time-critical image processing, AI classification and decision making algorithms [1], while interventional radiology uses image processing algorithms to reconstruct high resolution visualisations in real-time [2]. To pursue ambitious goals in these and other fields, researchers and equipment manufacturers are increasingly looking at applying massively parallel accelerators in their safety-critical devices. Semiconductor companies are keen to fill this gap in the market, exemplified by NVIDIA's Drive platform [3].

For such HRT systems, the holy grail of timing analysis is the derivation of a *safe* and *tight* worst-case execution time (WCET). WCET derivation is the problem of finding the worst of all possible run-time variations for a given workload. In this context, a safe WCET is a bound that is guaranteed never to be exceeded, while tightness refers to minimising the pessimism associated with the distance between the derived WCET and the true execution time.

As a prerequisite to constructing the search space of run-time variations we require an accurate model of the targeted microarchitecture. Efforts that advance such models for existing GPUs include the reverse-engineering of microarchitectural details, e.g. outstanding DRAM request handling resources [4], instruction latency [5], kernel-instance and work-group scheduling policies [6], [7] and process context switching latency [8]. These insights have contributed to WCET computation techniques that can be applied in soft real-time systems [9], [10], but sadly have not resulted in a closing timing model suitable for deriving safe WCETs.

A fundamental challenge in deriving a safe WCET lies in the influence of variation in one subcomponent on the latency of others. In an ideal world, the search space of run-time variations is limited by e.g. a simple, monotonic relationship between the local latency in each subcomponent and the overall latency. Unfortunately, many throughput optimisations in existing architectures frustrate such a relationship. For example, with a commonplace request-reordering DRAM controller, a DRAM request on the critical execution path can be *delayed* by another request when issued one cycle *earlier*. A shorter latency in one subcomponent can thus lead to a longer overall latency. Without a clear understanding of the relation between local and overall latency, all possible run-time variations must be deemed a WCET-candidate rather than just those that e.g. maximise latency in each subcomponent, quickly growing the search space to a size for which a full exploration is infeasible.

To tackle this problem, we believe that tailoring parallel architectures towards HRT systems is not just desirable, but *required* to achieve both good performance and safe, tight bounds on the execution times of workloads. Designing architectures specifically for HRT systems allows us to eliminate the complexities and unknowns of off-the-shelf hardware that impair accurate timing analysis. Given the many recent success-stories of domain-specific architectures (for e.g. machine learning and cryptocurrency mining), we believe the time is right to fill this surprising gap in the architecture design space.

To demonstrate the feasibility of a clean-slate approach, we present Sim-D: the first (GPU-like) wide-SIMD architecture to be specifically designed for HRT systems. Evaluation using our cycle-accurate model shows that Sim-D's performance often resembles that of the embedded-grade NVIDIA Tegra K1 GPU [11], while allowing algorithmic computation of safe and tight WCETs for kernel-instances.

Similar to GPUs, Sim-D performs *hardware strip-mining* to schedule the work for a compute kernel in entities called *work-groups*. Inspired by the *PRedictable Execution Model*

(PREM) [12], Sim-D schedules the work for each work-group as a sequence of uninterruptible access- and compute *phases*, interleaving the phases of two work-groups at a time. Owing to strict performance isolation between Sim-D's compute- and memory resources, the execution time of each phase can be tightly bound through static analysis. Static WCET derivation of a *kernel-instance* is done by analysing the possible interleavings of these phases. Various phase scheduling policies are enforced in hardware to reduce the number of possible interleavings, at a performance penalty of ∼9.2%. The result is a WCET analysis algorithm tailored to the Sim-D architecture that derives a safe bound which is measured to be tight within 12.7% on average.

We make the following contributions:

- An overview of the Sim-D processor pipeline and memory system (Section 3),
- A program model that permits reasoning about execution on Sim-D (Section 4),
- Two work-group scheduling policies, each permitting the derivation of a WCET by computing that of a single worst-case schedule of phases (Section 4),
- A novel WCET computation algorithm, combining control flow analysis with a simulation-based processor behaviour analysis, that takes into account scheduling of work-groups as sequences of phases (Section 5).

2 Background

Many aspects of Sim-D's design are inspired by existing GPUs. We first introduce the terminology used in this work, based on that of OpenCL [13] and CUDA [14], followed by a brief introduction of WCET calculation practices and timing-predictable DRAM transfers.

2.1 Execution model

To off-load data-parallel work to an accelerator, a program must first upload a *kernel*, a binary non-interactive function. Then, for each invocation of this kernel, the program uploads all required data and launches a *kernel-instance*.

A *kernel-instance* is parametrised with the required vector size. This size, called the *NDRange*, is specified in up to 3 dimensions, and dictates how many *work-items* must be launched. Each work-item is identified by a *global ID*, a unique identifier in the NDRange space. Work-items are grouped into *work-groups*. In GPUs, hardware is responsible for *strip-mining* [15] and scheduling execution of the work-groups required to span the entire NDRange.

GPUs divide work-groups into *warps*, groups of 32 (NVIDIA) or 64 (AMD) work-items. Each *compute unit* (CU) in a GPU can issue multiple instructions per cycle from the warps in a work-group. This plurality of active warps helps to mask data movement latencies between the memory hierarchy and the individual warps, maximising occupancy of the available compute- and memory resources. All warps in a work-group execute on the same CU, allowing data sharing within each work-group through *local memory*.

2.2 Worst-case timing analysis

WCET analysis generally consists of three analysis passes: control flow analysis (CFA), processor behaviour analysis (PBA) and bound calculation [16]. CFA is used to model the possible execution paths of a program. PBA characterises the execution cost of all parts of a program through modelling pipelines [17] and memory systems (e.g. [18]), system simulation [19] or a hybrid of the two. Bound calculation finds the cost of the longest path through a program.

Table 1
Configuration and relevant timing properties of Micron [26] (1) MT40A512M16JY-062E and (2) MT40A1G8SA-062E DDR4-3200AA.

Symbol	(1) 2BG	(2) 4BG	Description
	64-bit		Effective bus width
nB	8	16	Banks / Rank
nBG	2	4	Bank groups / Rank
nR	65536		Rows / Bank
nC	1024		Columns / Row
nBW	16		Words/burst (8 beats, 64B)
Latency in cycles, $tCK = 0.625ns$			
$nRCD$	22		Row-activate to CAS delay
$nCAS$	22		RD $\rightarrow$ first burst distance
$nCWD$	16		WR $\rightarrow$ first burst distance
nRP	22		Row Precharge delay
$nBURST$	4		Cycles per burst (for DDR: beats / 2)
$nRAS$	52		Row Activate Strobe
$nRTP$	12		Read-to-Precharge
nWR	24		Write Recovery
$nRFC$	560		Refresh cycle (act, pre) time
$nREFI$	12480		Refresh interval
$nCCD_S$	4		Column R/W to CAS delay, diff. bg
$nCCD_L$	8		Column R/W to CAS delay, same bg
$nRRD_S$	9	4	Row-act. to Row-act. delay, diff. bg
$nRRD_L$	11	8	Row-act. to Row-act. delay, same bg

Sim-D's bound calculation differs from prior work as a consequence of its execution model, as we will explain in detail in Section 5. CFA and simulation-based PBA is used to determine the cost of the execution paths for a single work-group. Sim-D's bound calculation then extracts access- and compute phases from the critical path, and calculates the WCET of a kernel-instance by taking into account the scheduling of these phases for each work-group.

2.3 Real-time DRAM transfers

The biggest challenge for HRT DRAM transfers is to achieve predictable latencies with a minimal sacrifice of data bus utilisation. Predictable latency is generally achieved using closed-page DRAM policies [20], [21], [22]. These policies eliminate timing interference effects between requests by precharging all banks between any two DRAM requests.

DDR4 DRAM storage is organised in a hierarchy of bank groups, banks, rows and columns. Data is transferred in *bursts* spanning 8 consecutive columns [23] over 4 cycles. The number of bits in each column is determined by the *data bus* (DQ) width.

Akesson et al. [24] observe that for a closed-page DDR3 DRAM controller to achieve a net efficiency above 80%, transfers must span 16 bursts. Krishnapillai et al. [25] note a trend of increasing data bus widths and associated burst sizes. Extrapolating these trends to a contemporary DDR4-3200AA DRAM [26] system with a 64-bit data bus, a request must exceed 4KiB to achieve 80% utilisation. Sim-D issues requests of such size by issuing work-group-wide reads and writes that convey a linear relationship between the elements in a tile of data and the work-group's work-items.

The worst-case DRAM latency used in our WCET calculation algorithm is the *longest issue delay* (LID) [22]: the interval between issuing the first command for this request and the first possible instant a command for the next request can be issued. We express the LID in DRAM clock cycles as a function of the DRAM timing parameters listed in Table 1. For a comprehensive overview of their meaning, we refer the reader to Jacob et al.'s "Memory Systems: Cache, DRAM, Disk". [23]

3 Architecture

Sim-D aims to provide both high-performance data-parallel processing, and the means to derive safe and tight bounds on execution times. Three key features lie at the heart of this: phase-wise work scheduling, performance isolation between the compute, DRAM and scratchpad resources, and large, work-group wide DRAM requests.

Inspired by PREM [12], Sim-D schedules each work-group as a succession of compute- and access *phases*. Each phase runs uninterruptibly on a single resource. Because Sim-D's resources do not interfere with each other, the execution time of each phase can be determined statically. Treating each access- and compute phase in a program as an independent critical section reduces the problem of finding a kernel's WCET to finding the worst-case schedule of phases.

To maximise resource occupancy, Sim-D's CU has two active work-group *slots* to schedule phases from. This is permitted because, like OpenCL and Cuda, we forbid dependencies between work-groups that necessitate synchronisation of the entire NDRange. The *work-group scheduler* is responsible for strip-mining an NDRange into work-groups of 1024 work-items each. This work-group size is chosen such that in the common case a work-group will request blocks of 4KiB of data from DRAM, equal to 16 bursts over a 64-bit DQ. As explained in Section 2.3, this permits a DQ occupation approaching 80%.

Work-groups are consumed by the CU's instruction scheduler, which proceeds to schedule its operations. Performance isolation is achieved by separating instruction- from data memory (Harvard architecture [27]), and by replicating all storage resources for each slot. This isolation permits interference-free overlapping of a DRAM/scratchpad transfer from one work-group with a compute phase of another. To schedule work in phases, Sim-D issues instructions from the two active work-groups following a policy similar to greedy-then-round-robin (GTRR) [10]: a work-group continues to occupy the compute resources until a DRAM or scratchpad transfer is issued or an exit is encountered. The resource is then yielded to the work-group in the other slot.

Finally, large work-group wide DRAM transfers help to achieve high DQ utilisation at a predictable (worst-case) request execution time. Whereas commodity GPUs perform best-effort coalescing of DRAM requests at run-time, Sim-D programs contain *scalar* DRAM and scratchpad request instructions that transfer either a 1D/2D tile of data, or data corresponding with a stream of offsets (indexes) for a whole work-group. By explicitly conveying the relation between memory locations and work-items in the instruction and performing this memory request uninterruptibly for the whole work-group, the memory controller can deterministically achieve high bank locality in a way that can be statically analysed with little pessimism.

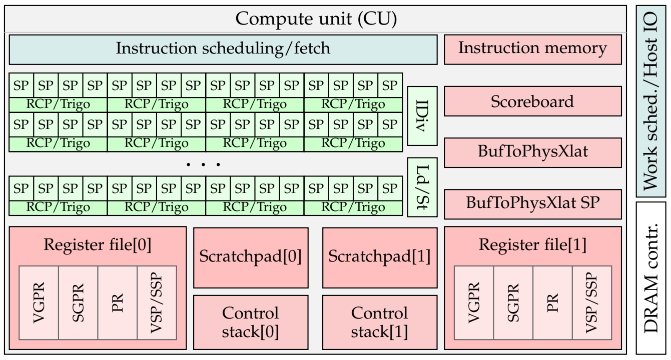


Fig. 1. High level overview of the Sim-D architecture.

To issue work-group wide DRAM request, Sim-D requires to keep all 1024 work-items in a work-group in lock-step. A drawback of this requirement is that for workloads with many divergent branches, there may be an efficiency loss compared to GPUs that schedule at the granularity of (sub-)warps (e.g. [28]). For the workloads evaluated in Section 6.1, we did not encounter significant negative performance effects from this coarser scheduling granularity.

We currently omit modelling host$\leftrightarrow$device communication, including buffer up- and downloads, and leave studying its performance implications as future work.

A high-level overview of the Sim-D architecture is given in Figure 1. In the remainder of this section we explain the architecture in greater detail.

3.1 Compute unit

Sim-D's CU implements an in-order, single issue four-step pipeline: fetch, decode, execute, write-back. The instruction set supports a mix of scalar- and vector instructions. In this paper, Sim-D is evaluated with a three-stage decode and operand fetch step and a five-stage execute pipeline step, thus totalling 10 pipeline stages.

Sim-D is configured with 128 single-precision arithmetic units (SP-units), performing boolean, integer and floating-point operations on 32-bit values. Justified by various NVIDIA publications [29], [30], [31], pipelined floating-point reciprocal, reciprocal square root and trigonometric operations are modelled using four multipliers and various look-up tables. We assume the SP-units' multipliers can be re-used, and thus model one *RCP-unit* for every four SP-units. Scalar operations are performed on the first SP-unit. One non-pipelined 8-cycle scalar Radix-16 integer divider is included, modelled after Intel Core 2's divider design [32].

Each register file consists of four types of registers: vector-, scalar-, predicate-, and special purpose registers. Special-purpose registers are further subdivided into vector special purpose registers (e.g. global- and local ID, control masks) and scalar special purpose registers (e.g. work-group dimensions and offset within the NDRange).

The vector register file (VRF) can sustain three operand fetches and one operand write-back per cycle. This matches the bandwidth requirement of a vector multiply-addition (MAD) operation. Guided by GPU design practices documented in the literature [33], [34], the VRF is modelled as eight banks of two-port, 1R1W SRAM cells, one bank for every group of 128 work-items (henceforth *warp*). Each of the three instruction decode stages fetches one operand for an instruction. By scheduling the warps for each vector instruction in round-robin, no VRF bank conflicts occur.

Sim-D supports two forms of control flow: per-work-item vector control flow, and work-group-wide scalar (un)conditional branches. The former allows work-items within a work-group to follow diverging code paths. Vector control flow is implemented using implicit predicated execution [35] : control masks (CMASKs) in the special purpose vector register file determine which work-items are enabled. A dedicated control stack (CSTACK) contains (PC,mask,type) 3-tuples that allow restoring control flow at the end of an if- or else-block, a loop, or a function call.

A *control stack pop (cpop) injection* mechanism prevents starvation that results from disabling all work-items. When a vector control flow instruction causes the implicit control mask to become all-0, the instruction decoder injects a cpop operation into the pipeline to restore a control mask and PC from the top of the CSTACK. This process is repeated until at least one work-group is re-enabled.

3.2 DRAM controller

Sim-D's DRAM controller implements a pipeline that dynamically translates a large request, issued by a work-group, into a sequence of DRAM commands. To eliminate variations in latency caused by interference between successive requests, it implements a closed-page policy on the boundary of each large DRAM requests. Within a request, DRAM bank locality is exploited in a statically analysable manner through deterministic command scheduling policies.

The DRAM controller pipeline consists of four stages, as shown in Figure 2. The *front-end* accepts large DRAM requests, and translates each to a set of burst requests. The *command generator* performs address translation on these burst requests and determines which commands must be sent to DRAM. Scheduling these commands under the constraints of DRAM timing is done by the *command arbiter*. For read- and write commands, it creates a DQ reservation. In the final stage, the *DQ scheduler* generates control signals for the CU to synchronise DRAM data movement with the register file or scratchpad.

Currently the DRAM controller supports one 64-bit channel, consisting of one DRAM rank. Scaling Sim-D to the performance of a higher-end GPU is left for future work.

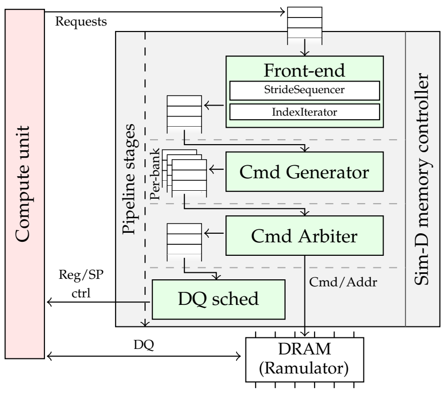


Fig. 2. High-level architecture for Sim-D memory controller

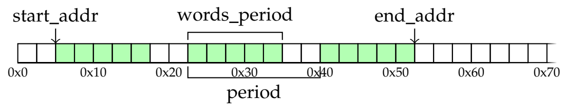


Fig. 3. Example stride request

3.2.1 Front-end

The DRAM controller front-end accepts two types of DRAM requests: 1D/2D tile requests and iterative indexed requests. Such requests are processed by the *stride sequencer* and *index iterator* subcomponents respectively.

The *stride sequencer* translates a 1D/2D tile request into a sequence of burst requests, one per cycle. Figure 3 visualises an example stride pattern, reading a 5×3 tile from a 7×7 buffer. Reading is performed with start address 0x8, a period of 7, 5 words per period and a period count of 3. The resulting end address is $8 + (2 * 7 + 5) * 4 = 84 = 0x54$. Since a DRAM burst must be aligned to a multiple of its size (Sim-D: 64 bytes), this stride pattern translates to two burst requests: one burst starting at address 0x0, and a second request at address 0x40.

The stride sequencer generates these burst requests, one per cycle. For each burst request, it generates a wordmask that indicates which words from each burst (marked in green) must be routed to registers. Additionally, the stride sequencer computes the destination vector register lane or scratchpad address for each word in each burst.

The *index iterator* supports requests for data that cannot be expressed as such a linear relationship, for example for data-dependent (indirect) loads and stores. It accepts a stream of indexes as its input, and generates one burst request for each word. No attempt is made to coalesce burst requests as such logic has the potential of becoming quite complex yet offers no obvious benefits to worst-case performance for buffers exceeding 1024 bursts (64KiB).

3.2.2 Command scheduling

The remaining pipeline stages deterministically schedule the commands to service these burst requests.

The command generator translates burst requests into a stream of activate, read/write and precharge commands. It performs address translation following a scheme optimised for DDR4 DRAM similar to Paired Bank-Group Interleaving [36], where two adjacent bursts always access alternating bank groups except on the boundary of a bank-group pair. These commands are enqueued into per-bank FIFOs.

The command arbiter monitors the top entries of each of these FIFOs and deterministically schedules their commands compliant with DRAM timing restrictions. To pick which eligible command to schedule in a given cycle, it adheres to six prioritisation rules, in order:

1) Read/write commands are issued as early as possible.
2) Read/write commands from the *currently active bank-pair* are preferred. When this bank pair is precharged, the next active bank pair is selected.
3) Read/write commands have priority over activate.
4) Activate commands have priority over precharge.
5) Row activate commands are prioritised according to the number of read/write operations present in the

respective bank FIFOs targeting said row, tie-broken by distance from the currently active bank pair.

6) Refresh operations are scheduled between two requests.

When issuing a read or write command, the command scheduler generates a reservation to schedule the data movement between DRAM and the scratchpad or register file. The *DQ scheduler* processes these reservations and timely generates the control signals for the CU.

3.2.3 Snoopy indexed transfers

For HRT systems, indexed transfers are problematic. Regardless of how indexed transfers are implemented in hardware, there exists a worst case for which the relevant data is stored so sparsely that for *every word* a burst read/write request must be issued. With DDR4 DRAM, all words could reside in the same bank group, spacing consecutive bursts $nCCD_L$ cycles apart. Hence the worst-case data bus utilisation for such large buffers will never exceed:

$$\frac{1}{\frac{nBW}{nBURST} * nCCD_L} \quad \rightarrow \quad \frac{nBURST}{nBW * nCCD_L} \tag{1}$$

For a DDR4-3200AA configuration with two bank-groups [26] this bounds worst-case bus utilisation to $\frac{4}{16*8} = 3.125\%$. Despite this low data bus utilisation, use-cases might still necessitate indexed transfers.

When the size of the buffer or tile in which indexes reside can be bound a priori, we can improve on this bound with a novel technique called *snoopy indexed transfers*. A snoopy indexed transfer works by streaming an entire buffer or tile on the DRAM bus. For read operations, dedicated per-work-item content-addressable memories (CAMs) detect on each cycle whether the data currently present on the bus is for the index requested by this work-item. If so, it snoops the word from the bus into its corresponding data register. For write-operations, when a CAM indicates that its index matches one of the words written in a cycle, it signals a match to set the corresponding DQ mask bits and writes the word from its data register to the correct DQ lines.

From the DRAM controller's perspective, a snoopy indexed transfer is functionally and timing-wise equivalent to a regular 1D or 2D tile request. To route the data to the correct vector register lanes, each register file is extended with an array of CAMs. By replicating these CAMs in both register files, they can be re-used for scratchpad transfers. Isolated experiments (not included for brevity) show that snoopy indexed transfers can outperform iterative indexed transfers for DRAM buffers or tiles of up to 1.1MiB for read operations, and 1.37MiB for write-operations.

3.2.4 Mapped buffers

Sim-D uses mapped buffers to provide a memory protection mechanism that isolates data between different kernel-instances, and to provide parameters to load/store instructions. We chose mapped buffers over paged memory as the former can be queried in a single cycle, whereas the latency of a page-walk is difficult to bound. To facilitate mapped buffers, each SimdCluster contains two dedicated fixed-size BufToPhysXlat units: one for DRAM buffers and one for scratchpad buffers. Each BufToPhysXlat unit contains a mapping from buffer IDs to their (physical address, x-dimension, y-dimension) 3-tuples. The buffer dimensions are used both to determine e.g. the *period* of a 2D stride request, as well as to detect out-of-bounds accesses.

Upon launching a kernel-instance, the work scheduler uploads all mappings to the two BufToPhysXlat units. These mappings persist throughout kernel-instance execution.

3.3 Scratchpads

The CU is equipped with two SRAM-backed on-chip scratchpads, one per active work-group. This permits a DRAM↔scratchpad transfer to run in parallel with a scratchpad↔register file transfer. In our simulation model, the width of a scratchpad line and thus the width of the scratchpad bus can be configured in powers of two between 4- and 32 data words (128-1024 bits).

Compared to DRAM, the operation of SRAM cells is relatively simple: there are no row-buffers mandating lengthy activate or precharge operations, and cells do not need a periodical refresh. Consequentially, the control logic for the scratchpads is simpler than that of the DRAM controller. We have given each scratchpad a StrideSequencer front-end similar to that of the DRAM controller. However, rather than generating burst requests, it issues read/write requests directly to a DQ scheduler. This DQ scheduler synchronises SRAM control signals with those for the target register file.

Given the limited size of most scratchpad buffers, we omit support for iterative indexed transfers between the scratchpad and the register file. Indexed transfers must use the snoopy method instead.

4 Real-time work-group scheduling

In the previous section we explained how Sim-D schedules work from work-groups in access- and compute phases. Owing to performance isolation, the WCET of each phase can be bound statically. Deriving the WCET of a kernel instance is now a problem of finding the worst-case schedule of phases onto the CU's resources.

This section analyses this scheduling problem in greater detail. After introducing a formal program model, we first demonstrate that exhaustively testing all possible schedules is intractable. We introduce two work-group scheduling policies that reduce this search space to a single *serialisation*, followed by a formal proof that for each serialisation we only need to consider a single *worst case* schedule. We conclude this section with an upper- and lower bound on the possible efficacy of a work-group scheduling policy, assuming that the bounds found for each phase are tight.

4.1 Program model

The formal foundation of Sim-D's worst-case timing analysis relies on four abstractions: a *system*, a *kernel-instance*, a *serialisation* and a *schedule*.

A *system* is described by a set of *resources* R, each resource R_i represented by a (τ_i, I_i)-pair containing the resource type $\tau \in$ {COMPUTE, DRAM, SP}, and the instance number I. The Sim-D architecture is represented by the set of resources $R = \{(\text{COMPUTE}, 1), (\text{DRAM}, 1), (\text{SP}, 1), (\text{SP}, 2)\}$.

Inspired by PREM [12], we model a work-group's execution as a succession of *phases*. On Sim-D, each phase exclusively reserves one resource. As every DRAM or scratchpad request is issued by an instruction, a work-group must alternate between *compute* and DRAM/scratchpad *access* phases. The final phase is a DRAM store to write back results.

A *kernel-instance* $K = (w, (\Phi_1, \ldots, \Phi_n))$ is modelled as a sequence of n phases Φ and a number of work-groups w. Each phase Φ_i is described as a (ρ_i, c_i)-pair describing the resource type $\rho_i \in \{\text{COMPUTE, DRAM, SP}\}$ and the phase's WCET (cost) c_i. Each work-group executes the sequence of phases Φ in order.

To reason about the problem of scheduling phases on resources, two further abstractions are introduced: *schedules* and *serialisations*.

A *serialisation* is a list that describes an order in which the phases of every work-group in a kernel-instance K start executing on their resource. In other words, it describes a way in which the phases of work-groups in K interleave at run-time. Formally, a serialisation $S(K) = (\nu_1, ..., \nu_m)$ is a totally ordered set of phase instances $\nu_j = (\phi_j, w_j, r_j, z_j)$, $\phi_j \in \Phi$ the corresponding phase from K, w_j the work-group number, $r_j \in R$ the resource occupied by this phase instance and $z_j \in \{0, 1\}$ the slot this phase instance occupies. In an execution following $S(K)$, each phase instance ν_j must start before or at the same time as ν_{j+1}. If two phase instances start at the same time, a consistent tie-breaking rule (e.g. "lowest work-group number first") determines the order of phase instances in the serialisation. To honour the dependencies between instructions, each work-group must launch their phases in order $(\Phi_1, \Phi_2, \ldots, \Phi_n)$.

A *schedule* can be informally thought of as a (run-time) instance of a serialisation with associated time information. Formally, a *schedule* $s(K) = (\sigma_1, ..\sigma_n)$ is a totally ordered set of resource reservations, each entry representing a 5-tuple $\sigma_i = (\phi_i, w_i, r_i, z_i, [t_i^{start}, t_i^{end}])$ with $\phi_i \in \Phi$ and w_i the phase and work-group for this reservation, $r_i \in R$ the resource occupied by this reservation, z_i its work-group slot, and $[t_i^{start}, t_i^{end}]$ the interval during which this reservation is active. In this work, all time is measured in discrete clock cycles at the rate of the compute resource. A schedule is said to run for the interval $[1, t^{end}]$, with t^{end} defined as follows:

Definition 1. *For a given schedule, t^{end} is the last time-instant that the corresponding kernel-instance runs.*

$$t^{end} = max(\{t_m^{end} \mid \sigma_m \in s(K)\})$$

There is a one-to-many relationship between serialisations and schedules. From a schedule, its corresponding serialisation is obtained by ordering the elements by start time, and extracting the relevant program phase instances.

4.2 Scheduling policies

On every cycle or event, an on-line work-group scheduler takes two binary decisions: whether or not to fetch the next work-group from the queue, and whether or not to enqueue the next program phase for execution. The resulting serialisations are therefore the permutation of a multiset containing two elements, "work-group slot 0" and "work-group slot 1", each with a multiplicity equal to the number of work-groups assigned to each slot multiplied by the number of program phases in the kernel-instance. Assuming optimistically that work-groups are evenly distributed over the two slots, the number of possible permutations p for a kernel-instance K is determined by:

$$p = \frac{(w * |\Phi|)!}{(\lceil \frac{w}{2} \rceil * |\Phi|)! * (\lfloor \frac{w}{2} \rfloor * |\Phi|)!} \tag{2}$$

Given both w and $|\Phi|$ can run into hundreds, considering all possible serialisations is intractable. Such exhaustive evaluation seems unnecessary, as even the simplest greedy policy that schedules work as early as possible will preclude the vast majority of these serialisations. However, establishing criteria for filtering out impossible schedules and schedules that are strictly worse than others is problematic given that the run time of phases varies between executions.

To overcome this tractability problem, we propose two on-line scheduling policies: *scratchpad as compute* and *scratchpad as access*. Both policies build on the observation that when a system is modelled as exactly two resources, a greedy scheduler only schedules program phases according to a single serialisation. This serialisation is one where the two active work-groups swap resources every time they both finish their current program phase.

To permit work-groups to conditionally exit before executing all of its phases, both policies schedule work-groups in pairs. When a work-group exits, its slot remains empty until the other slot starts its final phase. Consequently, early exit does not alter the order of the remaining program phase instances in the serialisation. Retaining this order causes a safe bound derived from a serialisation without early exit to be safe under early exit, for which we sketch a proof in Section 4.4.

The resource utilisation diagrams in Figure 4 demonstrates the effects of pair-wise work-group scheduling. The black line shows how allowing immediate release of a new work-group in slot 0 once its previous work-group has finished causes a delay on the final DRAM request for

Table 2
Common symbols and definitions.

Symbol	Description
R_i	Resource
τ_i	Type of resource R_i
I_i	Resource instance of resource R_i
K	Kernel-instance
w	# Work-groups launched for kernel-instance K
Φ_j	Program phase j for kernel-instance K
ρ_j	Resource type required by program phase Φ_j
c_j	Worst-case execution time (cost) of program phase Φ_j
$S(K)$	Serialisation of kernel-instance K
ν_m	Program phase instance of serialisation $S(K)$
ϕ_m	Program phase for m'th entry in $S(K)$
w_m	Work-group of m'th entry in $S(K)$
r_m	Resource occupied by m'th entry in $S(K)$, $r_m \in R$
z_m	Work-group slot for the m'th entry in $S(K)$
$s(K)$	Schedule of kernel-instance K, sequence of reservations
σ_i	Reservation i of schedule $s(K)$
ϕ_i	Program phase for i'th entry in $s(K)$, $\phi_i \in \Phi$
w_i	Work-group of i'th entry in $s(K)$
r_i	Resource occupied by i'th entry in $s(K)$, $r_i \in R$
z_i	Work-group slot for the i'th entry in $s(K)$
t_i^{start}	Start time of i'th entry in schedule $s(K)$
t_i^{end}	End time of i'th entry in schedule $s(K)$
t^{end}	End time of the last-finishing entry in schedule $s(K)$

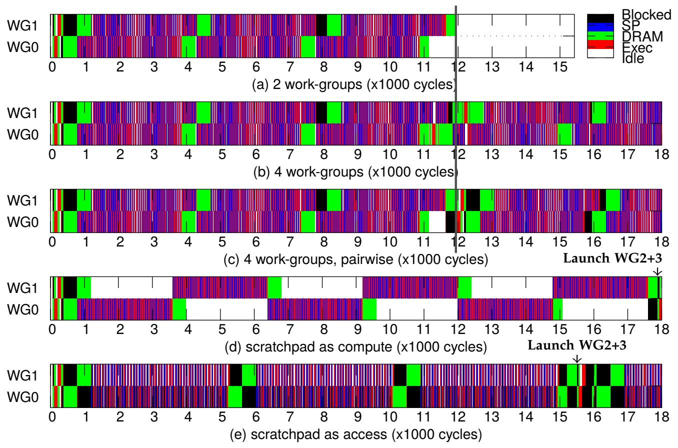


Fig. 4. Occupation graphs for CNN convolution.

work-group 1. Under unconstrained execution (b), at the line the work-groups in the two slots have drifted apart by several phases with no sign of reconvergence later on. Under *pairwise work-group* scheduling (c), the start of the second work-group for slot 0 is delayed. As a result, the final write request of the first work-group in slot 1 is no longer blocked by a work-group that is launched later, restoring the execution schedule of the first two work-groups to that of diagram (a). The cost of this resynchronisation is the idle time between the final DRAM write request of work-group 0 and that of work-group 1, indicated by the white region between cycle 11000 and 12000.

Diagrams (d) and (e) show the cost of enforcing scratchpad access as compute or DRAM access respectively. Evidently there is a cost attached to policies that limit the scope for parallel occupation of resources, which we will quantify for a set of 15 benchmarks in Section 6.

4.3 Worst-case minimal valid schedule

In this section we prove that for each serialisation, WCET analysis only needs to consider one schedule: the *worst-case minimal valid schedule*. We first give a definition of *valid* and *minimal* in the context of a schedule derived from a specific serialisation, after which we prove that the *worst-case* valid minimal schedule is the one where the execution time of each program phase is maximised.

Definition 2. *A* ***valid schedule*** $s(K)$ *for a kernel-instance* K *is a schedule for which a valid serialisation* $S(K)$ *exists, and additionally no resource is allocated to more than one reservation at any point in time.*

A *minimal valid schedule* derived from a serialisation is a valid schedule that schedules each program phase as early as possible, provided the requested resource is available, the serialisation's order is honoured and for each work-group its program phases do not overlap. Before presenting a formal definition of a minimal valid schedule, we first provide three auxiliary functions:

Definition 3. $RRes(s, \rho)$ *is a function that for a given schedule* s *returns the end time of the last resource reservation* $\sigma_i \in s$ *for which* $r_i = \rho$*, or 1 if no such reservation exists.*

$$RRes(s, \rho) = max(\{t_i^{end} \mid \forall \sigma_i \in s : r_i = \rho\} \cup \{1\}) \tag{3}$$

Definition 4. $WEnd(s, \zeta)$ *is a function that for a given schedule* s *returns the end time of the last resource reservation* $\sigma_i \in s$ *for which* $z_i = \zeta$*, or 1 if no such reservation exists.*

$$WEnd(s, \zeta) = max(\{t_i^{end} \mid \forall \sigma_i \in s : z_i = \zeta\} \cup \{1\}) \tag{4}$$

Definition 5. $Pre(s, m)$ *is a function that returns the sub-schedule (prefix) of schedule* s *containing all elements up to but excluding element* m*:*

$$Pre(s, m) = (\sigma_i \in s \ : \ i < m) \tag{5}$$

Using these three functions, the minimal valid schedule is defined as follows:

Definition 6. *A* ***minimal valid schedule*** $s(K)$ *with respect to a serialisation* $S(K)$ *is a valid schedule for which* $\forall \sigma_m \in s(K)$ *the following holds:*

$$t_1^{start} = 1 \tag{6}$$

$$t_{m>1}^{start} = max \begin{pmatrix} t_{m-1}^{start}, \\ RRes(Pre(s(K), m), r_m) + 1, \\ WEnd(Pre(s(K), m), z_m) + 1 \end{pmatrix} \tag{7}$$

$$t_m^{start} < t_m^{end} \leq t_m^{start} + c_m \tag{8}$$

Phases in a minimal valid schedule are required to finish within c_i cycles from start, but no other bounds are placed on the execution time of a phase. Hence a serialisation can result in many different minimal valid schedules. The next theorem introduces the worst-case minimal valid schedule derived from a serialisation $S(K)$, being the one that maximises the execution time for every element.

Theorem 1. *The* ***worst-case*** *minimal valid schedule* $s(K)$ *for a given serialisation* $S(K)$ *is one where each phase executes for its worst case execution time.*

$$\forall \sigma_i \in s(K) : t_i^{end} = t_i^{start} + c_i \tag{9}$$

Proof: By induction. Let $s(K)$ and $s'(K)$ be minimal valid schedules derived from $S(K)$, $s(K)$ being a *worst-case minimal valid schedule* and $s'(K)$ being a minimal valid schedule where at least one program phase executes for a shorter amount of time. We prove $t'^{end} \leq t^{end}$ by proving two invariants: $\forall i \in [1 : |s(K)|] : t_i'^{start} \leq t_i^{start}$ and $\forall i \in [1 : |s(K)|] : t_i'^{end} \leq t_i^{end}$.

Let d be the index of the first resource reservation in both $s(K)$ and $s'(K)$ such that $t_d^{end} \neq t_d'^{end}$ and $Pre(s(K), d) = Pre(s'(K), d)$. If $d = 1$, the first element in $s(K)$ and $s'(K)$ already differ in execution time. Trivially, $\forall i \in [1 : d) : t_d'^{start} \leq t_d^{start}$ and $t_d'^{end} \leq t_d^{end}$ by definition of d.

Consider d the base case for the induction argument. Consider two cases. If $d = 1$, $t_d'^{start} = t_d^{start} = 1$ per Equation 6. For $d > 1$, Equation 7 consists of three components. The first component is equal for both scheduler as per the start-time invariant. The second and third component are equal as, by definition, $Pre(s(K), d) = Pre(s'(K), d)$. Because all three components are equal for both schedules we conclude that $t_d^{start} = t_d'^{start}$. The first invariant holds.

Since $t_d^{start} = t_d'^{start}$, the definitions given by Equations 8 and 9 guarantee that $t_d'^{end} \leq t_d^{end}$. Hence the second invariant holds.

For the induction case $d + 1$, the start-time invariant guarantees that $t_d'^{start} \leq t_d^{start}$. For the second and third component of the maximum in Equation 7, the end-time invariant guarantees that:

$$\begin{aligned} RRes(Pre(s'(K),d+1), r_{d+1}) \leq \\ RRes(Pre(s(K), d+1), r_{d+1}) \text{ , and} \\ WEnd(Pre(s'(K),d+1), z_{d+1}) \leq \\ WEnd(Pre(s(K), d+1), z_{d+1}) \end{aligned}$$

Since all three components of the maximum in Equation 7 must be smaller or equal for $s'(K)$, the maximum must be smaller or equal too.

For the second invariant, recall from Equation 9 that $t^{end}_{d+1} = t^{start}_{d+1} + c_{d+1}$. Substituting this into Equation 8 gives $t'^{end}_{d+1} \leq t'^{start}_{d+1} + c_{d+1} \leq t^{start}_{d+1} + c_{d+1} = t^{end}_{d+1}$, proving the second invariant holds too.

This concludes the induction argument demonstrating that both invariants hold for all elements in the schedules. From Definition 1 it now follows that:

$$\begin{aligned} t'^{end} &= max(\{t'^{end}_i : \forall \sigma_i \in s'(K)\}) \\ &\leq max(\{t^{end}_i : \forall \sigma_i \in s(K)\}) = t^{end}_i \end{aligned} \qquad \square$$

This theorem reduces the problem of finding the worst-case schedule to one of finding the worst-case serialisation, as each serialisation only has a single worst-case minimal valid schedule to consider.

4.4 Work-group early exit

Some kernels may require whole work-groups to exit early, thus skipping their remaining program phases. Within a limited scope, this can be facilitated under pair-wise work-group scheduling. We provide the following theorem and a basic sketch of its proof, but omit the full proof for brevity.

Theorem 2. *Removing a program phase from a serialisation does not increase the run-time of its worst-case minimal valid schedule.*

The proof follows the same inductive nature as that of Theorem 1. Given two serialisations, the second a copy of the first with one element removed, we can construct the two worst-case minimum valid schedules one program phase instance at a time according to Equations 6, 7 and 9. For equivalent reservations in both schedules, the reservation from the second schedule must have both a start- and end-time earlier or at the same time as the reservation in the first schedule, proving that the latter schedule must have an end time earlier or at the same time as the former.

4.5 Bounds

Assuming the WCET bounds on phases are tight, there are limits to how well (or poorly) a work-group scheduler can perform. We next introduce upper and lower bounds that help put the performance of the scheduling policies in perspective. An upper bound is given by serial execution of the program, formally:

Definition 7. *For any work-group scheduler, the WCET of a given kernel-instance K running on Sim-D is upper bounded by:*

$$WCET^{U}(K) = w * \sum_{i=1}^{|\Phi|} c_i \tag{10}$$

A lower bound on the WCET is obtained by taking the maximum of the sum of costs on each resource. Formally:

Definition 8. *For any work-group scheduler, the WCET of a given kernel-instance K running on Sim-D is lower bounded by:*

$$WCET^{L1}(K) = w * max \begin{pmatrix} C(K, COMPUTE), \\ C(K, DRAM), \\ C(K, SP) \end{pmatrix} \tag{11}$$

with

$$C(K,r) = \sum_{i=1}^{|\Phi|} \begin{cases} c_i & \text{if } r = \rho_i \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

Note that this bound is not tight as it does not take into account any constraints on schedules that result from data dependencies between phases. Serialisation is expected to occur as compute phases cannot start until their data is loaded or stored in full. As a result, for a kernel-instance K no minimal valid schedule might exist with an execution time not exceeding $WCET^{LB1}(K)$.

A second non-tight lower bound is one where two work-groups execute in parallel with zero blocking, formally:

Definition 9. *For any work-group scheduler, the WCET of a given kernel-instance K running on Sim-D is lower bounded by:*

$$WCET^{L2}(K) = \left\lceil \frac{w}{2} \right\rceil * \sum_{i=1}^{|\Phi|} c_i \tag{13}$$

In the remainder of this work, the lower bound is reported as the maximum of these two:

Definition 10. *For any work-group scheduler, the WCET of a given kernel-instance K running on Sim-D is lower bounded by:*

$$WCET^{L}(K) = max(WCET^{L1}(K), WCET^{L2}(K)) \tag{14}$$

5 WCET COMPUTATION

This section presents an algorithm to determine a safe WCET of a kernel-instance running on Sim-D. This algorithm combines path-based CFA with simulation-based PBA. The main contribution of this algorithm is its bound calculation: from the critical path through kernel code, the algorithm extracts the list of access- and compute *phases* that each work-group will execute. A safe WCET is derived from this sequence by computing the cost of the *worst-case minimal valid schedule*. As explained in Section 4.2, this schedule is composed trivially for both the *scratchpad as access* and *scratchpad as compute* scheduling policies.

Sim-D's WCET analysis tool performs the following 8 steps, explained in detail in the remainder of this section:

1) Parse the kernel source,
2) Perform CFA to construct a control flow graph (CFG) [37] whose nodes consist of basic blocks (BBs) - linear sequences of instructions - and the edges represent the possible transitions between these blocks.
3) Perform PBA to compute/simulate bounds for DRAM and scratchpad requests, for the execution time of each BB and for the cost of BB→BB transitions,
4) Transform the CFG into a weighted directed acyclic graph (DAG), unrolling loops using programmer-annotated iteration bounds,

5) Find the critical path through the DAG,
6) Transform the critical path into a list of access- and compute phases in accordance with the chosen work-group scheduling policy,
7) Compute the WCET of the phase list,
8) Inflate the WCET with the DRAM refresh cost.

5.1 Parsing the program

The first step is to parse the assembly program into a list of BBs. BBs terminate with an instruction that performs control flow, issues a DRAM or scratchpad request or that could cause the instruction decoder to inject a cpop operation into the pipeline. The latter two BB terminators aid both with accounting for the cost of pipeline warm-up after each request, as well as with the construction of the DAG in step four and the program phase listing in step six.

5.2 Control flow analysis

Control flow analysis (CFA) transforms the list of BBs into a CFG. Sim-D's CFA differs from common practice as it must analyse branch targets reached by injected cpop operations.

To implement branch target analysis of injected cpops, each edge is annotated with the state of the control stack at the end of executing its source's BB. Each entry of the annotated stack state is a *(branch target, entry type)*-pair. To ensure that the number of edges is bound and known a priori, the CFA pass currently enforces the constraint that all incoming edges for a BB must have an equal stack state.

We assume that any CMASK write may lead to unrolling the entire CSTACK. When such an instruction terminates a BB, an edge is created for each entry in the stack state found on the BB's incoming edges. If the BB contains CSTACK push operations, these are included as potential branch targets. Each generated edge is annotated with the number of CSTACK entries that must be popped to reach this branch.

The CSTACK constraint on incoming edges of a BB limits the legal control flow constructions. Specifically it limits code sharing through function calls, and forbids recursive calling and loops where the stack grows on each iteration. The latter two are common limitations in real-time systems as such loops and recursive calls prohibit analysis of the worst-case control flow [38]. Code sharing, although desirable for code maintainability, is not functionally required. Since a program's binary size is generally not a constraint, developers can in-line shared subroutines instead. There may be scope for relaxing this restriction in the future.

5.3 Worst-case performance simulation

Step three annotates the CFG with worst-case execution times of both compute and DRAM/scratchpad accesses.

The LID of a strided DRAM request is determined using exhaustive simulation of a request for all possible alignments. The region covered by a transfer is known a priori, and the number of words taken from this region is statically upper bound. For most DRAM requests, these parameters can be derived statically from the instruction and the buffer mapping. For requests that compute values from the *custom stride* special purpose registers, developers may need to annotate these registers with upper bounds.

From analysing such simulations for 1D tiled transfer, we derived the following equations for the LID (in DRAM clock cycles) of read- and write requests of b bursts:

$$\begin{aligned} tID_R(b) =& max\begin{pmatrix} AC(b) + nRTP + nRP, \\ min(b-1,3) * nRRD_S + nRAS + nRP \end{pmatrix} \\ tID_W(b) =& AC(b) + nCWD + nBURST + nWR + nRP \end{aligned} \tag{15}$$

With the activation+read/write delay defined as:

$$AC(b) = \begin{cases} \text{(b-1)*nRRD}_\text{S}\text{+nRCD} & \text{iff } b \leq 4 \\ \text{2*nRRD}_\text{S}\text{+nRCD+(b-4)*nCCD}_\text{L}\text{+nCCD}_\text{S} & \text{iff } b \in [5,6] \\ \text{3*nRRD}_\text{S}\text{+nRCD+nCCD}_\text{L}\text{+nCCD}_\text{S} & \text{iff } b \in [7,8] \\ \text{nRRD}_\text{S}\text{+nRCD+2*nCCD}_\text{L}\text{+(b-4)*nCCD}_\text{S} & \text{iff b odd} \\ \text{2*nRRD}_\text{S}\text{+nRCD+nCCD}_\text{L}\text{+(b-5)*nCCD}_\text{S} & \text{otherwise} \end{cases}$$

For indexed-iterative transfers, the following equations define the LID for b bursts in a buffer of s words covering $rows(s)$ rows:

$$\begin{aligned} tIIID_R(b,s) =& \begin{cases} \text{IIAC(b,s) + nRTP + nRP} & \text{iff } rows(s) \leq nB \\ \text{b * (nRAS+nRP)} & \text{otherwise} \end{cases} \\ tIIID_W(b,s) =& \begin{cases} \text{IIAC(b,s) + nCWD + nBURST + nWR + nRP} & \text{iff } rows(s) \leq nB \\ \text{b}*\text{(nRCD+nCWD+nBURST+nWR+nRP)} & \text{otherwise} \end{cases} \end{aligned} \tag{16}$$

with

$$IIAC(b,s) = \begin{cases} \text{nRCD + (b-1)*nCCD}_\text{L} & \text{iff } rows(s) = 1 \\ \text{nRCD + nRRD}_\text{S}\text{ + (b-2) * nCCD}_\text{L} & \text{iff } 1 < rows(s) \leq nBG \\ \text{nRCD + nRRD}_\text{L}\text{ + nRRD}_\text{S}\text{ - 1 + (b-3) * nCCD}_\text{L} & nBG < rows(s) \leq nB \end{cases}$$

The LID of a scratchpad request is determined by counting one cycle for every line read/written in the worst-case alignment, plus one cycle for its front-end overhead.

The WCET of a BB's instruction execution is determined by simulating the timing of the Sim-D pipeline twice: once with a cold pipeline and once with a warm pipeline. For the warm-pipeline case, the entire kernel is simulated in linear order as if all branches are not taken. The absence of a branch predictor in Sim-D ensures that any other entry into a BB is with a cold pipeline.

The compute time simulation accounts for all pipeline behaviour, specifically:

- The expansion of vector instructions into many sub-vector instructions: 8 for single-precision operations, and 32 for operations occupying the RCP-units,
- Pipeline stalls due to read-after-write (RAW) hazards,
- Issue delays caused by the non-pipelined scalar integer divider, both to enforce instruction commit ordering and to keep a minimal distance between two issued integer divide/modulo operations.

For both the cold- and the warm pipeline simulation, the cost of each BB is computed by counting the number of cycles between it's first instructions' write-back and the write-back of the following BB's first instruction. The cost of a cold run is the base cost of a BB's execution. The difference between the cost for a warm and a cold run determines the penalties incurred by pipeline effects between a BB and the next. These penalties are stored as weights on the BB's outgoing "fall-through" edge. Branch-taken edges have their weight set to the cost of a pipeline flush. The edges for injected cpops have their weight set to the cost of cpop

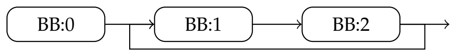


Fig. 5. Example control flow graph fraction

execution, multiplied by the number of CSTACK entries between the top of the CSTACK and the entry represented by the edge. For the latter two types of edges, if the destination BB starts with a scalar integer division or modulo instruction, the pipeline penalty includes the (three) extra cycles required to reach the write-back stage.

To justify why the two simulation paths produce safe WCET bounds even in the light of pipeline effects that propagate beyond two BBs, consider the CFG in Figure 5.

The cost for transitioning from BB:1 to BB:2 is given by the warm-pipeline overhead as simulated along the straight-line path BB:0→BB:1→BB:2. However, entry of BB:2 could also occur through the alternative [. . .]BB:2→BB:1→BB:2 path. Theoretically the pipeline state on the transition from BB:1 to BB:2 could differ between these two paths. However, pipeline effects that cross BB boundaries are the result of temporarily reserved resources like registers or functional units. The alternative path must enter BB:1 with no resource reservations. From this it follows that the cost of executing BB:1 plus the transition to BB:2 must be smaller than the bounds derived from the straight-line warm-pipeline path. From this observation we conclude that the two simulations (warm-pipeline branch never taken, and cold-pipeline) are sufficient to produce safe bounds on execution time.

5.4 Construct a DAG

The fourth step transforms the CFG into a weighted DAG by eliminating cycles from the CFG. Cycles in the CFG are the result of for- and while-loops in the program. To eliminate these cycles from the CFG, each loop is transformed into a DAG containing one node for each *execution* of a BB, similar to loop unrolling. A simple annotation-based scheme is used to provide mandatory bounds on loop iterations.

Loops are processed in depth-first order according to their nesting depth. After transforming a loop, the resulting DAG is cached in a map indexed by its entry BB, potentially overwriting an inner loop's cache entry. If, during the transformation of a loop or the main program, an edge is added to the DAG that points to a BB present in the loop-DAG cache, the cached DAG is appended to the parent's DAG.

Caching and in-lining partial DAGs for loops imposes the following limitations:

- Each loop has a single entry point and exit target,
- Loops must be properly nested,
- An inner loop's iteration bound annotation may not depend on outer scopes.

Multiple jump instructions with the same loop (re-)entry target are analysed as if they are multiple nested loops. Early breaking out of a loop is correctly analysed as long as each exit branch is annotated with its "taken" bound.

The resulting DAG has the cost for a DRAM or scratchpad request stored as weights associated with each node. All compute costs are represented as the weights on the edges of the DAG. Weights for these edges are computed by adding the CFG node's (cold-pipeline) compute cost to the weight of each of the CFG node's outgoing edges.

5.5 Critical path analysis

Given a DAG, its critical path is found using a dynamic programming algorithm that determines the longest path for successively larger sub-graphs. By adding nodes from the DAG to this sub-graph in topological order, no node needs to be processed more than once. For a DAG $D = (v, e)$ with no unconnected nodes, the resulting algorithm is of $O(|e|)$ time complexity.

This critical path algorithm requires that all DRAM and SP requests are present on the critical path. Violation of this requirement could cause the work-group scheduler to schedule program phases following different serialisations than the one assumed by the WCET computation in Section 5.7 with potential worse run-times. As an implication, conditional DRAM or scratchpad requests are only permitted in two cases: either when the condition can be described with a branch annotation, or when a work-group exits early.

5.6 Access/compute phase lists

In this step, this list of phases $\Phi = \{(\rho_1, c_1)..(\rho_i, c_i)\}$ is extracted from the critical path. To this end, the critical path is traversed from source to sink, aggregating the weight of each edge in an accumulator until a node with an associated DRAM or scratchpad transfer is encountered. Depending on the scheduling strategy (*scratchpad as access* or *scratchpad as compute*) this access cost must now be accounted for.

For both strategies, if the node has an associated DRAM request, two phases are added to the list: a compute phase whose cost is the compute cost gathered in the accumulator, followed by an access phase with its cost equal to the DRAM request WCET. Subsequently, the accumulator is reset to 0 and the algorithm continues path traversal.

If the encountered node is a scratchpad access, depending on the scheduling strategy chosen there are two ways to proceed. For the *scratchpad as access* policy, two phases are created similarly to the DRAM request case. For the *scratchpad as compute* scheduling policy, the scratchpad access cost is added to the compute-time accumulator and traversal of the critical path continues without creating two new entries.

5.7 WCET computation

From a phase list, the WCET is extracted by constructing the worst-case serialisation. Under the constraints of both scheduling policies, this is the serialisation under which two work-groups alternate between resources.

Since the phases of a work-group always alternate between access and execute, the cost for executing all n phases for a pair of work-groups is determined by $c_{2wg} = max(c_n, c_1) + \sum_{i=1}^{n-1} max(c_i, c_{i+1})$. If a work-group contains an odd number of work-groups, one work-group will execute serially without interleaving. The cost of such execution is defined as $c_{1wg} = \sum_{i=1}^{n} c_i$.

The WCET is found by multiplying c_{2wg} with the number of work-group pairs in a program. If an odd number of work-groups was launched, c_{1wg} is added to the total. For an even number of work-groups, the calculated total is adjusted for the tails of the schedule by adding $min(c_1, c_n)$. Finally, the program upload time must be added, which is calculated from the size of the program using Equations 15.

Formally, for a program binary spanning b bursts of data in DRAM the WCET (cost) c of a program is given by:

$$c_{edge} = \begin{cases} min(c_1, c_n) & \text{iff } w \text{ is even} \\ c_{1wg} & \text{otherwise} \end{cases} \tag{17}$$

$$c = \left\lfloor \frac{w}{2} \right\rfloor * c_{2wg} + c_{edge} + tID_R(b) \tag{18}$$

Additionally, the phase list constructed for the *scratchpad as access* scheduling policy is used to calculate $WCET^U(K)$ (Equation 10) and $WCET^L(K)$ (Equation 14).

5.8 DRAM refresh inflation

Finally, following an approach proposed by Park et al. [39], the derived WCET or bound is inflated by the cost of DRAM refresh. Assuming the ratio between the DRAM clock and Sim-D's compute clock is rCK, inflation is performed using the following equation:

$$c^{inflated} = c + \left\lceil \frac{c * rCK}{nREFI - nRFC} \right\rceil * \frac{nRFC}{rCK} \tag{19}$$

Inflation of WCET equates to a case where refresh occurs in a "stop the world" fashion as soon as required, halting both compute and DRAM. This is a pessimistic model of accounting for refresh cost as Sim-D's actual refresh behaviour differs in two ways:

1) Compute/scratchpads continue to run during refresh,
2) Refresh is deferred until after the active DRAM request.

Point 2 is covered safely by this inflation method despite assuming a refresh penalty of $nRFC$, thus without introducing a precharge-activate cycle required to *preemptively* execute the refresh. In most cases the DDR4 standard permits safe deferral of refresh until a request finishes and all banks are precharged; preemptive refresh is only required when a request takes longer than $8 * (nREFI - nRFC)$ cycles, e.g. a snoopy indexed request into a very large buffer. In these cases, the indexed iterative method has a better worst case and should thus be used instead.

Point 1 implies that inflation introduces a pessimism, specifically for compute- or scratchpad I/O bound kernels. The total cost of refresh inflation assuming DDR4-3200AA DRAM [26] is ∼4.5%, which is also an upper bound on this pessimism.

Table 3
Summary of hardware configurations.

Parameter	Sim-D	NVIDIA GeForce GT710	NVIDIA GeForce GTX780
		Compute	
Clock	1GHz	1GHz	992MHz
Compute units	1	1	12
Work-items/WG	1024	≤ 1024	≤ 1024
SP/RCP-units	128/32	192/32[1]	2304/384[1]
		DRAM	
Configuration	DDR4-1866M, DDR4-3200AA [26]	DDR3, 1866MHz	GDDR5, 6.4GHz
Bus width	64 bits (DDR)	64-bits (DDR)	384-bits (DDR)
Throughput	14.4, 23.8 GiB/s	14.4 GiB/s	288.4 GiB/s
		Scratchpads	
Clock	DRAM clock	unknown	unknown
Bus width	128, 256, 512, 1024 b; 4, 8, 16, 32 words	2048 bits/CU [40]; 64 words/CU	2048 bits/CU [40]; 64 words/CU
Capacity	64KiB/WG	16KiB/CU (+L1)	16KiB/CU (+L1)

[1] "SFUs", special function units, described by Oberman et al [31].

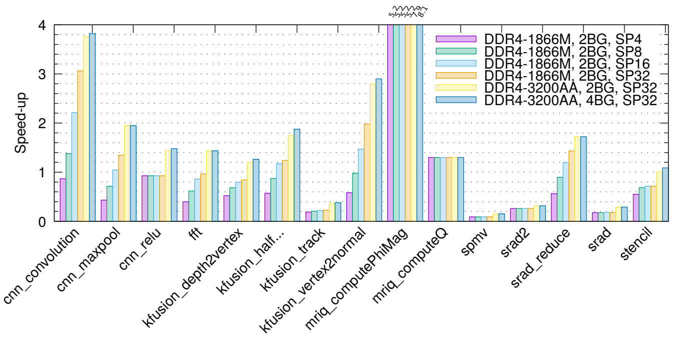


Fig. 6. Average performance of Sim-D, normalised to GeForce GT710.

6 Evaluation

In the previous sections we introduced both the Sim-D architecture and a WCET computation algorithm. Together they form the first full-stack solution for data-parallel compute in a way that permits calculating *safe* and *tight* WCETs for kernel-instances. We next evaluate the efficacy of Sim-D.

To this end, we gathered average- and worst-case performance numbers for 15 kernels. We ported 12 kernels from OpenCL to Sim-D's ISA, which were selected from Rodinia [41], Parboil [42] and KinectFusion [43] for their relevance to safety-critical systems. We additionally developed 3 convolutional neural network kernels, based on work by colleague Daniel Bates (personal communication, 8 September 2016), to OpenCL and Sim-D[1].

Execution times were measured using Sim-D's single-threaded cycle accurate performance model[2]. This simulation model is written in SystemC [44], and interacts with Ramulator [45] to enforce correct DDR4 timing. All benchmarks were run on a mid-range desktop computer from 2013 (Intel Core i5-4670, 3.4GHz). The cycle accurate simulator ran at a rate exceeding 45K cycles per second, thus requiring several hours to finish the simulation of all benchmarks for a single Sim-D configuration on one core. WCET computation, including exhaustive DRAM simulation for each access phase, took ∼1 minute per kernel.

6.1 Average-case performance

We first compare Sim-D under two DRAM configurations with the NVIDIA GeForce GT710 and NVIDIA GeForce GTX780 graphics cards. Parameters of these devices are given in Table 3. As shown, Sim-D's configuration roughly resembles NVIDIA's GeForce GT710, which in turn is representative of the embedded NVIDIA Tegra K1 GPU [11].

Figure 6 shows the measured performance of Sim-D, normalised to the performance of GeForce GT710. This figure shows mixed results. Provided sufficient scratchpad bandwidth, Sim-D is able to match or slightly outperform NVIDIA's low-end GPU for 8 benchmarks: CNN, FFT, KFusion's halfsample and vertex2normal kernels, SRAD reduce and the compute-bound MRI-Q computeQ. This is also true for MRI-Q computePhiMag, but with a run time of less than 2000 cycles, we have little trust in the significance of this observation. Such short kernels might exacerbate NVIDIA's

[1]https://github.com/RSpliet/CLaxon
[2]https://github.com/RSpliet/Sim-D

Table 4
Average case performance vs. NVIDIA GeForce GTX780.

Benchmark	Sim-D	GeForce GTX780	
	cycles	cycles	speed-up
CNN Convolution	18342388	5901185	×3.11
CNN Maxpool	368726	78122	×4.72
CNN RELU	3413172	903232	×3.78
FFT	216062	20106	×10.75
KFusion depth2vertex	350894	40212	×8.73
KFusion halfSample[...]	102770	19008	×5.41
KFusion track	7015098	256715	×27.33
KFusion vertex2normal	522664	137421	×3.80
MRI-Q computePhiMag	1291	5091	×0.25
MRI-Q computeQ	71988180	7669264	×9.39
SPMV	1438668	38622	×37.25
SRAD2	1653467	34357	×48.13
SRAD reduce	286514	58928	×4.73
SRAD	1745937	47182	×37.00
Stencil	510227	46040	×11.08

fixed-cost overheads caused by the OpenCL run-time (e.g. command submission through the PCIe bus). These overheads are not included in Sim-D's reported run times.

In the remaining six cases, Sim-D achieves at most 82% of the GT710's performance, with less than 10% for the SPMV benchmark. The four worst performing kernels (SRAD, SRAD2, SPMV and KFusion track) are DRAM I/O-bound as a consequence of Sim-D's absence of transparent caches. For example, SRAD and SRAD2 rely on indexed transfers that load arbitrary data points from a DRAM buffer of ∼898KiB. Because there is no scope to bound the region a work-group's indices can refer to, neither indexed transfer method performs well. NVIDIA's cache hierarchy drastically speeds up such DRAM accesses in the average case. Unfortunately, existing solutions using transparent caches are out of scope for Sim-D as they are not able to improve the worst case.

Finally, the results show that the architecture still benefits from higher DRAM throughput. The performance of DRAM I/O-bound benchmarks (e.g. CNN RELU, KFusion track, stencil) scales nearly linearly with the theoretical bandwidth provided by the DRAM technology. Furthermore, four bank-group DRAM chips consistently outperform two bank-group configurations.

Table 4 shows that, compared to Sim-D, the high-end NVIDIA GTX780 achieves 14.36× better performance . With over 10× more DRAM bandwidth and 18× as many SP-units, this is an unsurprising result. Scaling Sim-D to such high-end specifications is an open problem that poses two main challenges for future research: increasing the DRAM bus width without sacrificing data bus utilisation, and adapting the WCET derivation algorithm to account for parallel SimdClusters. We expect that a solution to the latter challenge can provide a foundation for research towards temporal- and spatial multitasking methods for HRT data-parallel accelerators.

6.2 Scheduling policies

To understand the run-time overheads of the *scratchpad as access* and *scratchpad as compute* scheduling policies, and the fraction of this overhead attributable to the *pairwise work-groups* scheduling constraint, Figure 7 demonstrates the run time of benchmarks under all three constraints. Execution times are normalised to the unconstrained scheduling case.

In line with expectations, this figure shows that kernels without scratchpad buffers (i.e. SRAD2 and MRI-Q computePhiMag) are unaffected by the proposed constraints. These benchmarks use only two resources and never exit early, hence even the unconstrained scheduler will always schedule them following a single serialisation.

Performance of the other benchmarks drops by 5.9% on average and 17.5% in the worst case under pairwise work-group scheduling. This penalty reflects the cost of synchronising work-groups that drift apart as a result of executing alternating compute and scratchpad phases in parallel with a single DRAM request, as shown in Figure 4.

The average penalty for treating scratchpad requests of these kernels as either access or compute is 14.1% and 12.8% respectively, with a worst-case observed penalty of 36.5%. This cost reflects the reduced scope for parallel resource occupation plus the cost of pair-wise work-group scheduling.

Interestingly the KFusion depth2vertex kernel benefits from the *scratchpad as compute* policy, performing 6.1% better than under unconstrained scheduling. This improvement is entirely caused by more favourable scheduling decisions.

We observe that neither work-group scheduling policy is a universally superior choice. For the KFusion, SPMV and stencil benchmarks the *scratchpad as compute* policy delivers better performance, whereas the CNN convolution, CNN maxpool and SRAD reduce benchmarks perform better under the *scratchpad as access* policy. When comparing for each benchmark their performance under unconstrained scheduling to that of the best-performing constrained scheduler, the measured performance degradation is 10.7% on average, or 9.2% when including the two unaffected kernels.

6.3 Worst-case execution time

Concluding, we evaluate the WCET of the benchmarks under both scheduling policies along two metrics: tightness of the produced bounds with respect to simulated execution, and performance under worst case conditions with respect to the lower- and upper bounds on WCET from Section 4.5.

Figure 8 visualises the simulated execution time of 11 benchmarks under both scheduling policies, alongside the calculated WCETs and bounds. The ranges depicted in red represent the (non-tight) lower- and upper bounds on WCET. The remaining four benchmarks are outliers that we summarised in Table 5. The data for columns labelled "avg" is obtained using the cycle-accurate simulation model, while the columns labelled "wcet" contain the WCET of a kernel as determined by the algorithm described in Section 5.

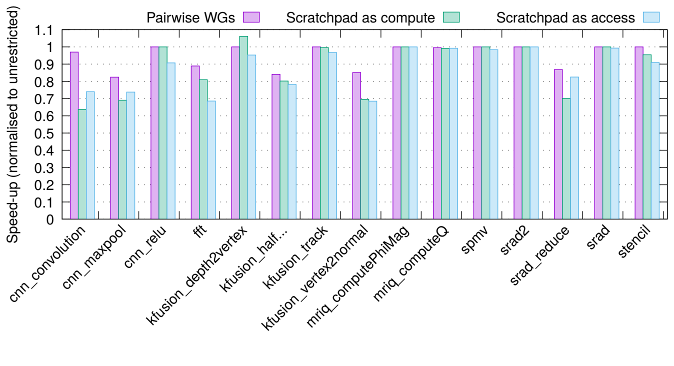


Fig. 7. Scheduling policy impact on avg. case performance.

Table 5
Run-time vs. WCET of outliers under various scheduling constraints, in compute cycles.

Benchmark	Unconstr.	$WCET^L$	$WCET^U$	SP as access			SP as compute		
	avg			avg	wcet	% diff	avg	wcet	% diff
KFusion track	7015098	91111265	92146915	7249308	91744883	1165.6	7042173	91488533	1202.8
MRI-Q computePhiMag	1291	1669	1777	1291	1687	30.6	1291	1687	30.6
SRAD2	1653467	11814286	11890837	1653467	11814605	618.8	1653467	11814605	618.8
SRAD	1745937	23199814	23413990	1737462	23252862	1238.3	1725240	23238710	1247.0

Three of the outliers (KFusion track, SRAD and SRAD2) show a large discrepancy between the measured execution time and calculated WCET. This is caused by their reliance on indexed read requests from large buffers. Section 3.2.1 explained that such transfers have an upper bound on their LID of 3.125%, which may not reflect average case performance. Unfortunately we see little room for improvement for these cases unless more details about the indexes into these buffers is known a priori. The last outlier (MRI-Q computePhiMag) is an extremely short-running kernel that is required in preparation for the computeQ kernel. Its data is displayed for completeness, but bears little significance.

Ignoring the outliers, we observe that the WCET derivation algorithm produces a bound which is on average tight within 12.7% and 11.8% under the *scratchpad as access* and *scratchpad as compute* policies respectively. This discrepancy represents the combined pessimism of assuming the longest execution path for every work-group, assuming the worst-case memory transfer properties (alignment, size, indices), and assuming maximum blocking from DRAM refresh. Bar two exceptions, MRI-Q ComputeQ and SPMV, the distance between the lower- and upper bounds indicate that there is much potential for improving kernel run times by occupying compute- and access resources in parallel. As the figure shows, the most effective scheduling policy for a benchmark generally achieves half of that potential.

In line with the findings from Section 6.2, this graph shows that neither scheduling policy is universally better than the other. There appears to be an (unproven) correlation between which scheduling policy produces the best WCET and which policy performs better during run-time.

7 Conclusion

To substantiate our claim that custom architectures are essential for applying data-parallel architectures in safety-critical systems, we studied Sim-D: a wide-SIMD architecture designed for HRT systems. Sim-D schedules the work for each work-group as a sequence of uninterruptible *access*- and *compute* phases, interleaving the phases of two work-groups. By providing performance isolation between the memory- and compute resources, the execution time of each phase can be tightly bounded through static analysis. Sim-D's DRAM controller and scratchpad front-ends process requests that transfer either 1D- or 2D blocks of data or the data corresponding with a set of buffer off-sets. By explicitly conveying the relation between work-group and data element, worst-case DRAM request efficiency can be analysed statically with relatively little pessimism.

Sim-D can often achieve performance on-par with an embedded-grade NVIDIA GPU under two conditions: Sim-D must provision sufficient scratchpad bandwidth, and applications must avoid indexed transfers from large buffers.

We presented a tailored WCET calculation algorithm capable of deriving safe bounds. Its key novelty is the consideration of work-group scheduling effects in the final bound calculation. This algorithm is paired with two hardware work-group scheduling policies that deterministically reduce run-time variance. We show that these policies degrade performance on average by 9.2%, but permit the calculation of WCETs that are tight within 12.7% on average for benchmarks that avoid inefficient indexed transfers.

With the presented study on Sim-D, we hope to provide a stepping stone for research in data-parallel architectures for HRT systems. Indeed, we see many opportunities for applying more advanced architectural- and algorithmic concepts from throughput-oriented architectures and identifying their merit in a HRT environment. Specifically, we expect that Sim-D can be further improved through advancements in its DRAM controller, work-group (phase) scheduling policies and general pipeline optimisations proposed in existing GPU literature. Moreover, we see value in applying and refining our architectural blueprint to domain-specific architectures, where practicable and bounded performance is an often undervalued aspect of a system's safety and security properties.

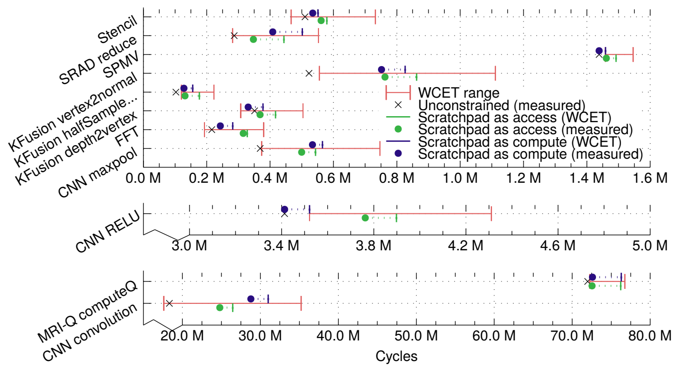


Fig. 8. Performance of scheduling constraints.

References

[1] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An Open Approach to Autonomous Vehicles," *Micro, IEEE*, vol. 35, no. 6, pp. 60–68, Nov 2015.

[2] A. Eklund, P. Dufort, D. Forsberg, and S. LaConte, "Medical image processing on the GPU – Past, present and future," *Medical Image Analysis*, vol. 17, no. 8, pp. 1073 – 1094, 2013.

[3] "NVIDIA Tegra X1 - NVIDIA's New Mobile Superchip," 2015, retr. Mach 2020, http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf.

[4] A. Lashgar, E. Salehi, and A. Baniasadi, "A case study in reverse engineering gpgpus: Outstanding memory handling resources," *ACM SIGARCH Comp. Arch. News*, vol. 43, no. 4, pp. 15–21, 2016.

[5] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *IEEE Int Symp on Perf. Analysis of Systems & Software*, 2010, pp. 235–246.

[6] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed," in *Proc. 38th Real-Time Systems Symp.*, Dec 2017, pp. 104–115.

[7] J. Bakita, N. Otterness, J. Anderson, and F. Smith, "Scaling Up: The Validation of Empirically Derived Scheduling Rules on NVIDIA GPUs," *OSPERT 2018*, p. 49, 2018.

[8] R. Spliet and R. Mullins, "The case for limited-preemptive scheduling in GPUs for real-time systems," in *ECRTS, Operating Systems Platforms for Embedded Real-Time applications*, Jul 2018.

[9] A. Betts and A. Donaldson, "Estimating the WCET of GPU-Accelerated Applications Using Hybrid Analysis," in *25th Euromicro Conf. on Real-Time Systems*, July 2013, pp. 193–202.

[10] Y. Huangfu and W. Zhang, "Static WCET Analysis of GPUs with Predictable Warp Scheduling," in *20th IEEE Int. Symp. on Real-Time Distributed Computing*, May 2017, pp. 101–108.

[11] NVIDIA Corp., *Tegra K1 Technical Reference Manual*, Sep. 2016, retr. Feb. 2017, https://developer.nvidia.com/embedded/downloads.

[12] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A Predictable Execution Model for COTS-Based Embedded Systems," in *IEEE Real-Time and Embedded Tech. and Appl. Symp.*, April 2011, pp. 269–279.

[13] Khronos, *The OpenCL specification 2.2*, jul. 2019.

[14] NVIDIA corp., *nVidia Cuda C Programming guide*, apr. 2012.

[15] M. Weiss, "Strip Mining on SIMD Architectures," in *Proc. 5th Int. Conf. on Supercomputing*, 1991, p. 234–243.

[16] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[17] S.-S. Lim, Y. Bae, G. Jang, B.-D. Rhee, S. Min, C. Park, H. Shin, K. Park, M. S.-M., and C. Kim, "An accurate worst case timing analysis for RISC processors," *IEEE Trans. Softw. Eng.*, vol. 21, no. 7, pp. 593–604, July 1995.

[18] R. Bourgade, C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "Accurate analysis of memory latencies for WCET estimation," in *16th Int. Conf. on Real-Time and Network Syst.*, Oct 2008.

[19] J. Engblom and A. Ermedahl, "Pipeline timing analysis using a trace-driven simulator," in *Proc. 6th Int. Conf. on Real-Time Computing Systems and Applications*, Dec 1999, pp. 88–95.

[20] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A Predictable SDRAM Memory Controller," in *Proc. 5th IEEE/ACM Int. Conf. on Hardware/Software Codesign and System Synthesis*, 2007, pp. 251–256.

[21] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero, "An Analyzable Memory Controller for Hard Real-Time CMPs," *IEEE Embedded Systems Letters*, vol. 1, no. 4, pp. 86–90, Dec 2009.

[22] M. Paolieri, E. Quiñones, and F. Cazorla, "Timing Effects of DDR Memory Systems in Hard Real-time Multicore Architectures: Issues and Solutions," *ACM Trans. Embedded Computing Systems*, vol. 12, no. 1s, pp. 64:1–64:26, Mar. 2013.

[23] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. Elsevier Science, 2010.

[24] B. Akesson, W. H. Jr., and K. Goossens, "Classification and Analysis of Predictable Memory Patterns," in *IEEE Int. Conf. on Embedded and Real-Time Comp. Syst. and Appl.*, Aug 2010, pp. 367–376.

[25] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni, "A Rank-Switching, Open-Row DRAM Controller for Time-Predictable Systems," in *26th Euromicro Conf. on Real-Time Systems*, July 2014, pp. 27–38.

[26] Micron Tech. Inc., "8Gb: x4, x8, x16 DDR4 SDRAM datasheet," 2017, retr. June 2018, https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf.

[27] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann Publishers Inc., 2019.

[28] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-level Warp Scheduling," in *44th IEEE/ACM Int. Symp. on Microarchit.*, 2011, pp. 308–317.

[29] N. Foskett, R. J. P. Jr., and S. Treichler, "Method and system for performing pipelined reciprocal and reciprocal square root operations," Oct 2006, US Patent 7,117,238.

[30] J. Piñeiro, S. F. Oberman, J. Muller, and J. D. Bruguera, "High-speed function approximation using a minimax quadratic interpolator," *IEEE Trans. on Comput.*, vol. 54, no. 3, pp. 304–318, March 2005.

[31] S. F. Oberman and M. Y. Siu, "A high-performance area-efficient multifunction interpolator," in *17th IEEE Symp. on Comput. Arithmetic*, June 2005, pp. 272–279.

[32] J. Coke, N. Cooray, E. Gamsaragan, P. Smith, K. Yoon, J. Abel, and A. Valles, "Improvements in the Intel Core2 Penryn Processor Family Architecture and Microarchitecture," Intel Corporation, Tech. Rep., 2008.

[33] M. Gebhart, D. Johnson, D. Tarjan, S. Keckler, W. Dally, E. Lindholm, and K. Skadron, "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 235–246, Jun. 2011.

[34] J. Lindholm, M. Siu, S. S. Moy, S. Liu, and J. Nickolls, "Simulating multiported memories using lower port count memories," Mar 2008, US Patent 7,339,592.

[35] B. Coon, J. Lindholm, and S. Tzvetkov, "Structured programming control flow using a disable mask in a SIMD architecture," Nov 2009, US Patent 7,617,384.

[36] S. Goossens, K. Chandrasekar, B. Akesson, and K. Goossens, "Power/Performance Trade-Offs in Real-Time SDRAM Command Scheduling," *IEEE Trans. on Comput.*, vol. 65, no. 6, pp. 1882–1895, Jun, 2016.

[37] C. Healy and D. Whaley, "Tighter timing predictions by automatic detection and exploitation of value-dependent constraints," in *IEEE Real-Time Tech. and Appl. Symp.*, June 1999, pp. 79–88.

[38] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Real-Time Systems*, vol. 1, no. 2, pp. 159–176, 1989.

[39] C. Park and A. C. Shaw, "Experiments with a program timing tool based on source-level timing schema," in *Proc. 11th Real-Time Systems Symp.*, Dec 1990, pp. 72–81.

[40] NVIDIA corp., "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/GK210," 2014.

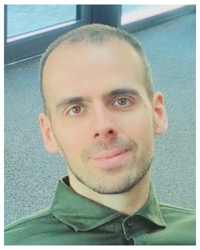
[41] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE Int. Symp. on Workload Characterization*, Oct. 2009, pp. 44 –54.

[42] "Parboil benchmark suite," 2010, http://impact.crhc.illinois.edu/Parboil/parboil.aspx.

[43] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon, "KinectFusion: Real-time dense surface mapping and tracking," in *IEEE Int. Symp. on Mixed and Augmented Reality*, Oct 2011, pp. 127–136.

[44] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011*, pp. 1–638, Jan 2012.

[45] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 45–49, Jan 2016.



Roy Spliet received his BSc degree in Computer Science, and his MSc degree in Computer Engineering from the Technical University of Delft, the Netherlands. He has been approved for a PhD degree at the University of Cambridge, United Kingdom, and is a senior research engineer at Imagination Technologies. His research interests span (massively-parallel) architectures and hard real-time systems.

Robert D. Mullins received the B.Eng degree in Computer Science and Electronics and M.Sc. and Ph.D. degrees in Computer Science from the University of Edinburgh. He is a Reader in the Department of Computer Science and Technology at the University of Cambridge. His current research is focused in the area of computer architecture, hardware accelerators and open-source chip design. Robert is a co-founder of the Raspberry Pi Foundation and lowRISC CIC.