Appears in the Proceedings of the 55th ACM/IEEE Int'l Symp. on Microarchitecture (MICRO-55), October 2022

big.VLITTLE: On-Demand Data-Parallel Acceleration for Mobile Systems on Chip

Tuan Ta, Khalid Al-Hawaj, Nick Cebry, Yanghui Ou, Eric Hall, Courtney Golden, and Christopher Batten
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{qtt2,ka429,nfc35,yo96,ewh73,ckg35,cbatten}@cornell.edu

***Abstract*—Single-ISA heterogeneous multi-core architectures offer a compelling high-performance and high-efficiency solution to executing task-parallel workloads in mobile systems on chip (SoCs). In addition to task-parallel workloads, many data-parallel applications, such as machine learning, computer vision, and data analytics, increasingly run on mobile SoCs to provide real-time user interactions. Next-generation scalable vector architectures, such as the RISC-V Vector Extension and Arm SVE, have recently emerged as unified vector abstractions for both large- and small-scale systems. In this paper, we propose novel area-efficient high-performance architectures called big.VLITTLE that support next-generation vector architectures to efficiently accelerate data-parallel workloads in conventional big.LITTLE systems. big.VLITTLE architectures reconfigure multiple little cores on demand to work as a decoupled vector engine when executing data-parallel workloads. Our results show that a big.VLITTLE system can achieve 1.6× performance speedup over an area-comparable big.LITTLE system equipped with an integrated vector unit across multiple data-parallel applications and 1.7× speedup compared to an aggressive decoupled vector engine for task-parallel workloads.**

I. Introduction

Modern mobile systems on chip (SoCs) adopt single-ISA heterogeneous multi-core architectures (e.g., Arm big.LITTLE) to offer a compelling high-performance and high-efficiency solution for task-parallel workloads [36, 38] in many commercial devices [3, 14, 21–23]. These architectures consist of several high-performance power-hungry out-of-order big cores and multiple high-efficiency low-power in-order little cores. This ISA homogeneity and micro-architecture heterogeneity enable high performance and efficiency by seamlessly distributing high- and low-intensity compute tasks to high-performance and high-efficiency cores respectively [51, 72].

In addition to task-parallel workloads, data-parallel applications are emerging in mobile SoCs to fully utilize their increasing compute power and sensing capabilities. Workloads such as augmented and virtual reality (AR/VR) [12, 24], natural language processing [7, 13], facial and voice recognition [47], and image processing [65] increasingly rely on in-device computing power instead of cloud servers to deliver real-time interactions with humans [42, 49, 68, 69, 71]. These applications often use compute-intensive data-parallel computer vision, machine learning, and data analytic algorithms to process a large amount of data in real time. Since mobile SoCs operate under a tight power and area budget, such increasing computational demand poses a significant challenge to design both **high-performance** and **high-efficiency** mobile architectures to accelerate data-parallel workloads.

The need to efficiently accelerate data-parallel workloads has led to an emergence of next-generation scalable vector architectures exemplified by the RISC-V Vector Extension (RVV) [53] and the Arm Scalable Vector Extension (Arm SVE) [61]. Traditional vector architectures are typically implemented as either large high-performance variable-length decoupled vector engines [15, 32, 56, 63] in super-computing systems or modest area-efficient fixed-length packed-SIMD integrated vector units (e.g., Intel AVX) in mobile and desktop systems. Next-generation vector architectures strive to provide unified scalable vector abstractions for both large decoupled vector engines that yield superior performance with significant area overheads and small integrated vector units that require modest extra silicon area with modest performance improvement compared to an out-of-order scalar core.

In this paper, we propose novel area-efficient high-performance architectures called big.VLITTLE that adopt next-generation vector architectures to accelerate data-parallel workloads in widely used big.LITTLE systems. big.VLITTLE architectures achieve both high performance and area efficiency by reconfiguring a cluster of little cores as a decoupled vector engine on demand when executing data-parallel workloads. When a big.VLITTLE system executes in vector mode, its big core fetches, decodes, and sends vector instructions to its associated cluster of little cores, which allows decoupling memory accesses and vector computation. Little cores reconfigure their scalar pipelines into vector execution lanes, leverage their physical register files to store vector register elements, transform their level-one cache subsystem to provide high memory bandwidth, and work together as a decoupled vector engine.

Due to its reconfigurability, big.VLITTLE architectures do not need to add area-expensive components such as wide execution pipelines and vector register files typically required in large decoupled vector engines. Compared to integrated vector units, big.VLITTLE systems can provide longer vector length and higher memory bandwidth, which results in better performance. When not executing in vector mode, big.VLITTLE systems incur no performance overhead for multi-threaded task-parallel workloads since they operate in the same way as equivalent big.LITTLE systems. Our cycle-level performance evaluation shows that a big.VLITTLE system with one big and four little cores can achieve 1.6× speedup over an area-comparable big.LITTLE system equipped with an integrated vector unit for data-parallel workloads from the Rodinia suite [10], RiVec suite [50], and a genomics benchmark suite. For task-parallel applications, the big.VLITTLE system is 1.7× faster than an aggressive decoupled vector engine for applications from the Ligra benchmark suite [58]. Our post-synthesis area evaluation shows the big.VLITTLE system incurs less than 5% overhead compared to a cluster of four little cores and their L1 private caches. Our design space exploration shows the potential of using voltage/frequency scaling to boost the little cores while slowing down the big core to further increase both performance and power efficiency of the big.VLITTLE system.

Our key contributions include: (1) a new reconfigurable little core cluster that leverages its existing scalar execution pipelines and reconfigures its scalar register files to operate as a high-performance decoupled vector engine; (2) a novel reconfigurable L1 cache subsystem that can turn private L1 data caches of little cores into a logically shared multi-bank L1 data cache for vector execution and re-purpose SRAM arrays in L1 instruction caches as data buffers to enable high vector memory bandwidth; (3) a detailed cycle-level performance evaluation of a big.VLITTLE system compared to an area-comparable conventional big.LITTLE system with an integrated vector unit and an aggressive decoupled vector engine, and a VLSI-level area analysis demonstrating the big.VLITTLE system's area efficiency; and (4) a design space exploration showing the potential of voltage/frequency scaling in increasing performance and power efficiency of a big.VLITTLE system.

II. The Resurgence of Vector architectures

Traditional vector architectures can be classified into two classes: long-vector and packed-SIMD architectures. A recent resurgence of interest in adopting vector abstractions for emerging data-parallel workloads has led to next-generation vector architectures that provide unified abstractions for both high-performance and commodity systems. In this section, we discuss a taxonomy of both traditional and next-generation vector architectures as well as their key tradeoffs (see in Table I).

A. Long-Vector Architectures

Long-vector architectures target large-scale systems, such as supercomputers, to execute highly data-parallel and regular workloads. Most long-vector architectures support a variable vector length that scales with a specific implementation of the architecture [15, 56]. The width of each vector element is typically fixed. Supporting cross-element instructions, such as reducing and shuffling vector elements, requires expensive hardware due to long vector length, so such instructions are usually not fully supported in long-vector architectures. Instead, memory gather and scatter instructions are available to support complex data movements through memory.

Vector units in supercomputing vector machines [1, 16, 32, 56, 63] are typically decoupled from their control cores. Vector units have separate large vector register files and wide execution lanes. To sustain a high compute throughput and fully utilize all execution lanes, long-vector machines require large memory bandwidth, so they are typically connected to highly banked memory systems (e.g., 1024 memory banks [59]).

B. Packed-SIMD Architectures

Since packed-SIMD ISAs often target multimedia workloads in commodity hardware, their vector lengths are typically limited and fixed. Early packed-SIMD architectures, such as Intel MMX [48] and SEE [25], are designed to handle sub-word computations on general-purpose registers. More recent packed-SIMD architectures, such as Intel AVX-128, extend their vector lengths beyond the width of a single word (e.g., 128 bits). To support multiple SIMD operations in a fixed hardware vector length, packed-SIMD ISAs support variable element widths to dynamically change the effective number of elements depending on applications. For example, a 128-bit wide packed SIMD ISA can support two 64-bit and four 32-bit operations. Since packed-SIMD ISAs are designed for commodity hardware, their support for complex vector memory instructions, such as gather-load and scatter-store instructions, is limited.

Packed-SIMD units are often tightly integrated with their control processors. Most of them share the same register files (i.e., typically floating-point register files) with the control processors although some recent short-vector units, such as ones in Intel Knights Landing [60], may have dedicated SIMD register files. Floating-point and SIMD instructions typically share the same execution pipelines. Since the number of vector elements is small, SIMD units typically share the same memory interface with their control processors to private data caches. Therefore, compared to long-vector machines, short-vector units have relatively modest memory bandwidth.

TABLE I. A Taxonomy of Vector Architectures

	Features	Long vector	Packed SIMD	Next generation
ISA	Vector length	scalable, long	fixed, short	scalable
	Element width	fixed	variable	variable
	Predication	full	limited	full
	Cross-element ops	limited	full	full
	Memory gather/scatter	full	limited	full
uArch	Vector register file	decoupled	integrated	either
	Speculative execution	yes	no	either
	Compute pipeline	decoupled	integrated	either
	Memory bandwidth	large	modest	either
	Memory latency	high	low	either

C. Next-generation Vector Architectures

Conventional vector architectures target two drastically different domains: high-performance computing in large-scale systems and multimedia workloads in commodity hardware. However, recent interest in data-parallel workloads have driven a trend to converge both conventional design approaches into next-generation vector architectures that are flexible enough to cover a wider spectrum of workloads and hardware implementations [61].

Modern vector ISAs, such as the Arm Scalable Vector Extension (SVE) [61] and the RISC-V Vector Extension (RVV) [53], adopt a vector-length-agnostic (VLA) design similar to conventional long-vector architectures. This VLA design enables such ISAs to target a wide range of implementations with different hardware resource constraints. It also allows executing the same vector code on multiple vector machines with different hardware vector lengths without recompiling the code and/or rewriting compiler intrinsics. Cross-element, load-gather, and store-scatter instructions are also supported in these ISAs to increase the overall scope of applications that can be vectorized.

Due to their flexibility, these next-generation vector ISAs can target both large-scale (i.e., decoupled from a control processor) micro-architectural implementations and small-scale (i.e., tightly integrated with a control processor). Example machines include Xuantie-910 [11] and Ara [9] implementing RVV, and Fugaku A64FX [57] implementing Arm SVE.

III. BIG.VLITTLE ARCHITECTURES

The reconfigurability of big.VLITTLE architectures helps achieve the performance level of decoupled long-vector engines while minimizing area overheads as in integrated vector units. In this section, we first provide an overview of big.VLITTLE architecture and then provide details on how multiple aspects of a next-generation vector architecture are implemented in big.VLITTLE.

A. Architectural Overview

big.VLITTLE architectures support both scalar and vector execution modes. In the scalar mode, big and little cores execute instructions independently as they do in conventional big.LITTLE systems, and components added to support the vector execution mode are disabled. In the vector mode, the big core becomes a control core, and the little cores work together as a single decoupled vector engine called VLITTLE. The big core executes scalar instructions while vector code is executed in the VLITTLE engine. A vector instruction waits at a vector dispatching unit in the big core until it is at the head of the ROB, and then is dispatched to the VLITTLE engine. If the instruction does not write back to a scalar register, the big core can commit and remove it from the ROB. Otherwise, the big core waits for the VLITTLE engine to respond with a scalar value, writes the value back, wakes up any dependent instruction(s), and finally commits the vector instruction. Although the big.VLITTLE concept is applicable to both Arm SVE and RISC-V RVV, we use RISC-V RVV version 1.0 [53] to explore the big.VLITTLE idea in the context of this paper.

A big.VLITTLE system includes additional components to facilitate its vector execution on top of an equivalent big.LITTLE system. Figure 1 shows a big.VLITTLE instance with one big, four little cores, and additional vector-specific components. First, a *vector control unit* (VCU) controls the global architectural states of a VLITTLE engine (e.g., effective vector length), communications with the big core (e.g., receiving vector instructions), and the execution of all little cores. Second, a *vector cross-element unit* (VXU) handles inter-core data communications to support cross-element vector instructions including vector permutation and reduction. Lastly, a *vector memory unit* (VMU) manages vector memory instructions by issuing requests to the memory subsystem and delivering data to little cores. Those additional components are pipelined so that they do not affect common critical paths and increase the cycle time of the little core cluster. Therefore, in the scalar mode, big.VLITTLE performs exactly the same as an equivalent big.LITTLE system. Several muxes are added to select the right input signals based on the current execution mode of the little cores. For example, an L1 data cache takes input requests from its little core's back-end in the scalar mode while receiving requests from the VMU in the vector mode.

B. Vector Control Support

We envision using a simple application interface managing an OS-privilege control status register (CSR) to switch between scalar and vector modes on demand. Applications running on the big core can switch into or out of the vector mode by requesting the OS to change the CSR. When switching into the vector mode, the OS allocates a group of little cores to form a VLITTLE engine, and those cores become unavailable to other processes. If one or multiple little cores are not readily available (e.g., busy with other processes), the OS can decide to either wait, pre-empt processes running on those little cores, or simply allocate a light-weight integrated vector unit in the big core for vector execution. Such OS-level resource scheduling decisions are left for future work. Once a little core is allocated to a VLITTLE cluster, its current thread context is saved to memory, and its pipeline is flushed. The overhead of saving a thread context into memory and flushing an in-order short pipeline is relatively small (e.g., 500+ cycles), especially when the target vectorized region is large. A control register is then updated to indicate the core is now working in the vector mode. When switching out of the vector mode, the OS returns those cores to its scheduling pool, and they become available independent scalar cores. The switching typically happens at a coarse-grained level (e.g., application and kernel levels) to amortize its overhead.

big.VLITTLE architectures implement a weak memory consistency model which is a work in progress in RISC-V RVV. We introduce a vector memory fence `vmfence` to handle vector/scalar memory dependencies (e.g., unit-stride vector store followed by a scalar load to the same address) in software to avoid complex hardware checking for such dependencies between scalar and vector pipelines. The big core executes `vmfence`, waits for all outstanding scalar loads and stores to retire before sending a memory fence command to the VCU. The VCU blocks subsequent vector memory instructions from being issued to the VMU until all outstanding vector memory instructions to retire. Effectively all scalar and vector memory instructions before `vmfence` in their program order happen before all scalar and vector memory instructions after the `vmfence`. It is important to note that this software-managed vector/scalar memory fence solution is common in most decoupled vector machines [1, 4, 17] due to their large vector lengths and decoupled vector execution pipelines. Future auto-vectorization and compiler techniques, which are being actively researched for next-generation vector architectures [2, 61], can help insert vector/scalar memory fences to guarantee the correctness of applications. In addition, any efficient hardware-managed solution for other decoupled vector machines would also be applicable to the big.VLITTLE architecture. Vector/vector memory dependencies (e.g., unit-stride vector store followed by an indexed vec-

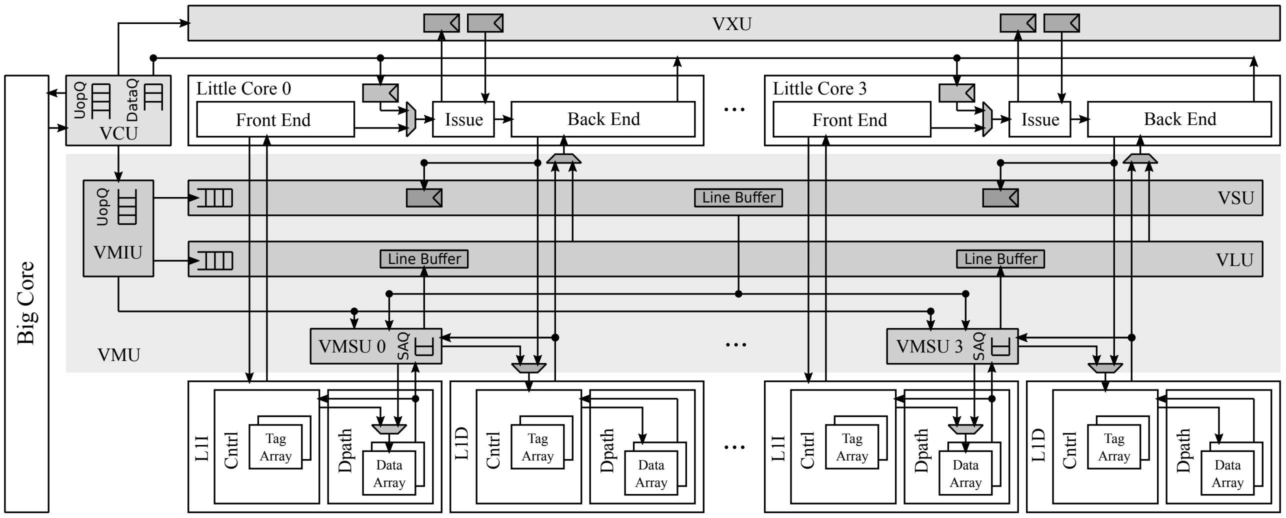

Figure 1. A big.VLITTLE system with One Big and Four LITTLE Cores – VCU = vector control unit; VXU = vector cross-element unit; VMU = vector memory unit; VMIU = vector memory issue unit; VMSU = vector memory slice unit; VLU = vector load unit; VSU = vector store unit; UopQ = micro-operation queue; DataQ = scalar data queue; SAQ = store-address queue. Components added to support the vector mode are shaded.

tor load to the same address) are handled in hardware by the VLITTLE engine's VMU.

In a big.VLITTLE system, the VCU executes *vsetvl* that is a control instruction setting the effective vector length and vector element type. For each non-control vector instruction, the VCU issues multiple micro-operations to little cores and the VMU (only for memory instructions). The VCU buffers those micro-operations and their corresponding scalar data (only for vector instructions reading scalar register values) in command and data FIFO queues in order to enable decoupling of vector memory accesses and vector executions by issuing memory micro-operations to the VMU ahead of time. Not all vector instructions need to carry scalar values, so the scalar data queue needs not to be as deep as the command queue to minimize area overheads. In each cycle, the VCU processes the oldest micro-operation from the command queue and broadcasts it and its associated scalar data (if any) to all little cores via a shared bus as shown in Figure 1. This command bus is pipelined to account for physical distance between little cores in a cluster so that it does not affect existing critical paths in the little cores.

C. Reconfigurable Little Cores

In big.VLITTLE architectures, scalar physical registers existing in little cores are re-purposed to implement all general vector registers except *v0*, which makes big.VLITTLE architectures area-efficient by not adding area-expensive vector register files as in conventional long-vector engines. Since *v0* register is used to store mask values according to the RISC-V RVV specification, predicated instructions can read up to three source operands. To avoid adding a read port to existing register files, *v0* is implemented using an extra register(s) added to each little core, which allows predicated instructions to read mask values in parallel with reading other source operands.

	Core 0		Core 1		Core 2		Core 3	
x0*	v0[0]	v0[1]	v0[2]	v0[3]	v0[4]	v0[5]	v0[6]	v0[7]
x1	v1[0]	v1[1]	v1[2]	v1[3]	v1[4]	v1[5]	v1[6]	v1[7]
	...		...		...		...	
x31	v31[0]	v31[1]	v31[2]	v31[3]	v31[4]	v31[5]	v31[6]	v31[7]
f0*	v0[8]	v0[9]	v0[10]	v0[11]	v0[12]	v0[13]	v0[14]	v0[15]
f1	v1[8]	v1[9]	v1[10]	v1[11]	v1[12]	v1[13]	v1[14]	v1[15]
	...		...		...		...	
f31	v31[8]	v31[9]	v31[10]	v31[11]	v31[12]	v31[13]	v31[14]	v31[15]

Figure 2. Mapping of 32-bit Vector Elements to Scalar 64-bit Registers in Four Little Cores – *xN* = scalar integer registers. *fN* = scalar floating point registers. *vN[m]* = m-th element in a vector register. Elements of the vector register 0 are mapped to newly added physical registers *x0** and *f0** in little cores.

To maximize the hardware vector length in big.VLITTLE architectures, both integer and floating-point physical registers in little cores can be effectively used to support multiple vector element groups (chimes). Vector elements of the same group are always executed together in time. The actual number of element groups depends on the number of available physical registers in little cores. Figure 2 shows an example of a VLITTLE engine with four little cores, each of which has 32 integer and 32 floating-point physical registers (i.e., *x0-x31* and *f0-f31* respectively). The VLITTLE engine supports two vector element groups. Vector elements in the first and second groups can be stored in the integer and floating-point registers respectively across all little cores. In addition, multiple consecutive vector elements can be packed into the same physical scalar register if their element width is less than the physical register's width. Figure 2 shows a case in which two 32-bit adjacent vector elements are packed into the same 64-bit physical register. With multiple element groups and packed vector elements, the example VLITTLE engine in Figure 2 can support a 512-bit hardware vector length by effectively using all physical registers in four little cores. Both optimizations increase the hardware vector length, reduce front-end instruction overheads, and hide long execution latency induced by complex instructions (e.g., multiplication, division, and memory instructions) in big.VLITTLE architectures.

For each vector instruction, the VCU issues multiple per-element-group micro-operations to little cores in order. Little cores receive micro-operations from the VCU at their issue stages. Their fetch and decode stages are not used in vector mode and hence disabled. In a little core, micro-operations are issued to its back-end execution pipelines in order as if they were normal scalar instructions. Except reading mask values from *v0*, no other change is added to a little core's issue stage to handle issuing micro-operations and reading operand values from the core's register file.

Back-end execution pipelines in little cores require minimal changes to support packed vector elements. For simple integer arithmetic micro-operations (e.g., addition), multiple vector elements packed into the same physical register can be processed in parallel with small area overheads to the existing little cores [39]. For more complex integer micro-operations (e.g., division) and floating-point micro-operations, we serialize the execution on different packed vector elements in multiple cycles to avoid adding non-trivial hardware overheads to the existing little cores.

D. *Cross-Element Instruction Support*

The RISC-V RVV supports two types of cross-element instructions: vector permutation and vector reduction. Vector permutation instructions (e.g., `vrgather`) read per-element values from a source vector register and write them to different elements of a destination vector register. Vector reduction instructions read per-element values from a source register, perform a reduction operation to a single value, and write it to either the first element of a destination vector register (e.g., `vredsum`) or a scalar register (e.g., `vpopc`).

For each vector permutation instruction, the VCU generates two micro-operations: `vxread` and `vxwrite` per vector element group to little cores. `vxread` micro-operations read values of their source vector elements and send them to the VXU. `vxwrite` micro-operations wait for the source values at the issue stage of each little core. Once receiving source values from the VXU, `vxwrite` micro-operations write the values to register files in little cores.

For each vector reduction instruction, the VCU first issues per-element-group `vxread` micro-operations to little cores to read values of source vector elements. The VCU then issues `vxreduce` micro-operation only to the first little core to perform a reduction. Once receiving a `vxreduce` micro-operation, the first little core's issue stage receives one value for each source vector element each cycle from the VXU, issues it to an execution pipeline, and waits for all source element values to arrive before completing issuing the micro-operation.

In order to move values across the little cores, we implement a light-weight uni-directional ring network connecting all little cores in the VXU. The ring network is pipelined to avoid affecting the cycle time of existing little core cluster. Other lower-latency network topologies (e.g., crossbar) are also viable although they may potentially incur higher area overhead compared to the uni-directional ring topology. The VXU receives per-element source values from little cores executing `vxread` micro-operations. The VXU receives requests for specific source elements from little cores executing `vxwrite` and `vxreduce` micro-operations. Once receiving all source values, the VXU iteratively shifts all per-element values by one hop each cycle. If a value's source element index matches with a request's source element index, the value is returned to the requesting core. The VXU completes shifting all elements after N cycles where N is the number of source vector elements. To avoid inter-instruction deadlocks and further complexities, the VXU processes at most one cross-element instruction at a time. Subsequent cross-element instructions must wait in the VCU for an outstanding instruction in the VXU to complete.

E. *Reconfigurable Cache Subsystem*

In a big.VLITTLE system, the VMU is the interface between its VLITTLE engine and memory subsystem. The VMU consists of a vector memory issue unit (VMIU), multiple vector memory slice units (VMSU), a vector load unit (VLU), and a vector store unit (VSU). Each VMSU corresponds to a private L1 data cache of a little core. In vector mode, private L1 data caches of all little cores work together as a logically shared cache with multiple address-interleaved slices or banks for the entire VLITTLE engine. We adopt an addressing scheme similar to a previous work [27] to distribute memory accesses across multiple L1 data caches. Given an effective address, its bank bits are located between the block offset and index bits to minimize bank conflicts in the case of consecutive requests to adjacent cache lines. The VMU uses the bank bits to select an L1 cache for a given address. The remaining bits and the bank bits are used as a tag in L1 caches to disambiguate cache lines properly regardless of which mode the caches operate under and to avoid expensive cache flushes when the system switches between modes. After switching to vector mode, a cache line that is not in the right bank will eventually either be evicted (i.e., if not used) or migrated to the right bank (i.e., if used) by the cache coherence protocol.

For unit- and constant-stride memory instructions, their base virtual addresses are translated in the big core before they are dispatched to the VLITTLE cluster. The big core also checks their access ranges (i.e., spanning across multiple pages) using both base addresses and strides for potential page faults or invalid memory accesses. This early address translation mechanism allows the VMU to decouple unit- and constant-stride memory accesses from vector executions happening in the little cores. However, for indexed vector memory instructions, since their index values are stored in the VLITTLE cluster, per-element address translations happen in the little cores using their existing address translation hardware.

To enable decoupling of memory accesses and vector execution, the VCU sends load and store micro-operations to the VMIU as soon as it receives and processes memory instructions from its associated big core so that load requests can be issued to memory ahead of vector executions using their loaded values. In addition, the VCU also sends per-element-group micro-operations to little cores to write back values from the VLU to register files (`wb_ld`), read and send data to the VSU for store instructions (`rd_data`), and read and send memory indices to the VMIU for indexed memory instructions (`rd_idx`).

Vector memory issue unit (VMIU) – The VMIU processes load and store micro-operations from the VCU in the order they arrive. It breaks down a micro-operation into one or multiple cache-line-sized requests depending on its base address, stride, and indices. For unit-stride and constant-stride micro-operations, the VMIU uses their base addresses and strides to generate one memory request for a cache line per cycle. For indexed memory micro-operations, the VMIU waits for index values sent by `rd_index` micro-operations from little cores before generating memory requests. The VMIU tries to coalesce a small number of consecutive indices (e.g., four) into a single cache-line request in each cycle. Multiple memory requests can be generated for a memory micro-operation if it accesses across different cache lines. Each generated memory request is tagged with a bit mask indicating active bytes in its cache-line-sized data. Once generated by the VMIU, a memory request is issued to a corresponding VMSU, based on its cache line address via a shared pipelined command bus. A small sequence number per request is sent to the VLU (for load requests) or VSU (for store requests) so that load and store data is processed in the order of their corresponding instructions in those units.

Vector memory slice unit (VMSU) – The VMSUs receive requests from the VMIU and operate at cache-line granularity to communicate with their corresponding L1 data caches. They also check memory dependencies between load and store requests. Each VMSU has a small content-addressable memory (CAM) holding addresses of outstanding store requests not yet issued to memory memory. Every load request arriving at a VMSU is checked against all previous outstanding store requests for potential address overlapping using their cache-line addresses. If a dependency is detected between a load and an outstanding store request, the load request is stalled until the store request is sent to the memory subsystem since the load request may read data written by the store request. Otherwise, the load request can be issued ahead of the store request to the memory.

The VMSUs need to buffer cache-line-sized data of all outstanding load requests (i.e., waiting for their cache responses) and store requests (i.e., waiting for their data from the VSU). To maximize the memory-level parallelism and enable memory accesses to run far ahead of vector execution, the amount of data buffering can be significant for each VMSU in memory-intensive workloads. Therefore, to minimize area overheads, we reconfigure SRAM data arrays in L1 instruction caches, which are unused by little cores in the vector mode, as circular FIFO queues for outstanding load and store requests. We simply use the SRAMs as FIFO queues and do not modify the cache control logic to avoid any timing overhead in the caches. Each VMSU controls head and tail pointers to its data queues and arbitrates between enqueueing and dequeueing operations since there is one read/write port in each SRAM. A VMSU writes response data from its L1 data cache into the load data queue and store data from the VSU into the store data queue. Once data for the oldest load request in a VMSU is ready, the VMSU sends the data to the VLU. Similarly, once data for the oldest store request in a VMSU is received from the VSU, the request is sent to the corresponding L1 data cache.

Vector load unit (VLU) – The VLU receives data responses from multiple VMSUs, breaks them down into per-core responses, and sends them to little cores. There are multiple line buffers, each corresponding to a VMSU, to hold ready cache-line data responses from the VMSUs. The VLU processes data responses in the order their corresponding requests are generated by the VMIU since little cores expect their data to arrive in order.

For each unit- and constant-stride load response, a small hardware logic uses its first vector element index and stride information to slice its cache-line data into multiple vector-element-width responses, which are then pushed to existing load-store queues inside corresponding little cores. Since the VLU actively pushes data to little cores through `wb_ld` micro-operations, the cores can directly read the data from their internal load queues without sending extra requests to the VLU for the data. This reduces the latency of `wb_ld` micro-operations.

For indexed load micro-operations, actively pushing data to little cores from the VLU would require extra storage for index values and more complex logic to slice, shuffle, and align data elements of a cache-line response for multiple little cores. Therefore, little cores pull data from the VLU by sending per-vector-element requests to the VLU when executing `wb_ld` micro-operations. Each little core handles indexed memory `wb_ld` micro-operations as if they were scalar load instructions by leveraging its existing address calculation logic. Once receiving an indexed load request from a little core, the VLU extracts data from one of its line buffers and returns it to the core.

Vector store unit (VSU) – The VSU receives data elements from little cores and assembles them into cache-line-sized data blocks for store requests. The VSU processes store requests in the order they are issued by the VMIU. In each cycle, the VSU waits for little cores to send data elements for the oldest store request. Each little core executes `rd_data` micro-operations as if they were scalar store instructions. Per-vector-element data requests including both address and data are sent by little cores to the VSU. The VSU takes data elements from the cores and assemble them into a cache-line-sized data block by updating its line buffer. Once data in the line buffer is ready, the VSU sends the data to a VMSU, based on its cache-line address.

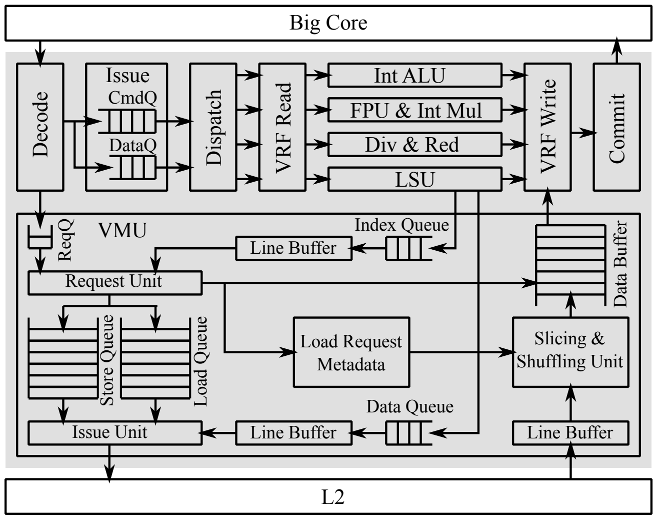

Figure 3. A Decoupled Vector Engine – CmdQ = command queue; DataQ = scalar data queue; VRF = vector register file; Int ALU = integer arithmetic & logic unit; FPU = floating-point unit; Int Mul = integer multiplication unit; Div = division unit; Red = reduction unit; LSU = load store unit; ReqQ = request queue.

TABLE II. SIMULATOR CONFIGURATION

Little Core (L)	• RISC-V ISA (RV64GC), single-issue, in-order • L1I cache: 1-cycle hit latency, 2-way, 32KB • L1D cache: 2-cycle hit latency, 2-way, 32KB
Big Core (b)	• RISC-V ISA (RV64GC), 8-way-issue out-of-order, 16-entry LSQ, 90 physical integer and 90 physical floating-point registers, 60-entry ROB • L1I cache: 1-cycle hit latency, 4-way, 64KB • L1D cache: 2-cycle hit latency, 4-way, 64KB
L2 Cache	• For the big.LITTLE and big.VLITTLE systems: 4-way, 4-bank, 8-cycle hit latency, 256KB for each big and little core cluster • For the decoupled vector system: 8-way, 8-bank, 8-cycle hit latency, 512KB shared by both the big core and the vector engine
LLC	Shared, 16-way, 12-cycle hit latency, 2MB
Main Memory	DDR4-2400

IV. EVALUATION METHODOLOGY

In this section, we describe a set of simulated systems, our cycle-level modeling methodology, and application benchmarks used to evaluate the performance of our big.VLITTLE architectures.

A. Simulated Systems

We use gem5 [8, 44, 62], a cycle-approximate simulator, to evaluate the performance of different hardware systems studied in this work. We use gem5's out-of-order processor model for our simulated big core and our in-house model to simulate in-order single-issue little cores. Our simulated cache subsystem is based on an Arm AMBA 5 CHI cache-coherent model provided in gem5 [18], and we use its simple network model for our simulated on-chip network. We model one-cycle address translation overhead per memory access (i.e., assuming memory accesses always hit in level-one TLBs) for all evaluated designs. For performance evaluations in Section V, we keep the big core, little cores, and caches running at the same frequency (i.e., 1GHz) to isolate micro-architecture-level behaviors of all designs from potential performance impacts of voltage/frequency scaling. In Section VII, we then explore a performance/power design space when considering voltage/frequency scaling of big and little core clusters. Table II shows the details of our simulated processors and memory subsystem.

Table III shows a list of evaluated systems and their configurations. *1bIV* supports next-generation vector architectures by integrating a small vector unit into its big core. The integrated vector unit supports 128-bit hardware vector

TABLE III. EVALUATED SYSTEMS

1bIV	**One big core with an integrated vector unit** • 128-bit-long vector unit capable of issuing instructions out of order • Two vector arithmetic execution pipelines capable of executing integer and floating point vector instructions • The big core's load/store unit capable of handling 128-bit wide unit-stride memory requests
1b-4L	**Conventional big.LITTLE system** • One big core and a cluster of four little cores • Private L1I and L1D cache per core • Private L2 cache for each core cluster
1bIV-4L	**big.LITTLE system with an integrated vector unit** • One big core and a cluster of four little cores with L1 and L2 caches similar to *1b-4L* • The big core including the same integrated vector unit as in *1bIV*
1bDV	**Long-vector system with a decoupled vector engine** • One big core with an aggressive decoupled vector engine • 2048-bit-long vector engine with four vector element groups, a 8KB vector register file with eight read and four write ports, 64-entry command queue, 16-entry load queue, 16-entry store queue, and 64-entry data buffers • The vector engine connected directly to L2 cache
1b-4VL	**big.VLITTLE system** • One big core and a VLITTLE engine of four little cores with L1 and L2 caches similar to *1b-4L* • 64-entry micro-operation queue and 16-entry scalar data queue in the VLITTLE's VCU • 512-bit hardware vector length with two element groups, 64 entries in load data queues, and 32 entries in store data queues in VMU

TABLE IV. TASK-PARALLEL APPLICATIONS

Name	Input	1L		1b-4L		1b-4VL vs
		DIns	Cycles	DIns	DTsk	1bDV
bc	rMat_1M	332M	1.7B	800M	0.5M	1.3x
bf	rMat_1M	782M	4.7B	1.4B	0.8M	2.3x
bfs	rMat_1M	56M	0.3B	203M	0.2M	1.0x
bfsbv	rMat_1M	113M	0.3B	241M	0.2M	1.7x
cc	rMat_1M	480M	1.9B	1.0B	0.4M	1.7x
mis	rMat_1M	337M	1.8B	864M	0.2M	1.1x
tc	rMat_1M	748M	1.7B	1.0B	0.1M	2.4x
prd	rMat_1M	3.9B	23.9B	5.9B	2.8M	3.2x
geomean						1.7x

All task-parallel applications are taken from Ligra suite [58]. DIns = dynamic instruction count in billions; DTsk = dynamic task count in millions; Cycles = cycle count in billions.

TABLE V. DATA-PARALLEL APPLICATIONS

			1L		1bDV																1bIV-4L			1b-4VL vs	
Name	**Suite**	**Input**	**DIns**	**Cyc**	**DIns**	**VIns**	**ctrl**	**ialu**	**imul**	**fpu**	**xe**	**us**	**st**	**idx**	**prd**	**DOp**	**VOp**	**VPar**	**WInf**	**ArInt**	**DIns**	**VIns**	**DTsk**	**1bIV**	**1bIV-4L**
vvadd	k	8.4M	75.5M	210M	1.6M	42%	20	20	0	0	0	60	0	0	0	35.1M	97%	22.3	0.47	0.33	52M	10%	4.3K	4.0x	2.6x
saxpy	k	8.4M	75.5M	291M	1.6M	50%	17	0	0	33	0	50	0	0	0	42.9M	98%	27.2	0.57	0.67	50M	16%	4.3K	2.9x	2.2x
mmult	k	512x512	1.08B	3.18B	18.9M	44%	25	25	25	0	0	25	0	0	0	416M	97%	22.0	0.39	2.00	567M	18%	23.4K	2.7x	2.4x
k-means	ro	10Kx34	4.65B	13.6B	67.9M	46%	1	13	∼0	57	∼0	∼0	21	7	1	2.01B	98%	29.6	0.43	2.41	2.8B	25%	11.7K	1.4x	1.2x
pathfinder	ro	5Mx10	1.08B	2.92B	22.5M	50%	31	37	0	0	0	31	0	0	25	510M	98%	22.7	0.47	1.20	764M	16%	36.9K	4.0x	3.0x
jacobi-2d	rv	2Kx10	1.59B	9.64B	35.4M	44%	8	17	0	42	17	17	∼0	0	0	942M	98%	26.6	0.59	4.50	1.1B	16%	640	2.8x	2.4x
blackscholes	rv	2.5M	726M	2.75B	10.6M	90%	4	13	0	72	0	3	8	0	5	567M	100%	53.5	0.78	7.48	467M	26%	2.0K	1.8x	1.1x
lavamd	rv	4x4x4	1.05B	3.58B	25.0M	79%	7	11	0	57	0	5	20	0	5	966M	99%	38.7	0.92	2.72	680M	29%	64	2.0x	1.2x
backprop	ro	524K	1.17B	16.4B	21.5M	39%	13	∼0	0	44	0	12	31	0	0	484M	97%	22.5	0.41	1.00	1.2B	11%	130	1.5x	1.5x
particlefilter	rv	6K	1.24B	3.52B	43.1M	51%	1	72	1	26	0	∼0	∼0	∼0	∼0	1.41B	99%	32.8	1.14	200.9	906M	16%	3.9K	2.8x	1.1x
sw	ge	2048	1.34B	2.09B	132.7M	4%	10	58	2	0	10	10	12	0	9	404M	69%	3.0	0.30	3.20	932M	6%	1.0K	1.7x	1.8x
geomean																								2.1x	1.6x

1L = one little core; 1bIV, 1bIV-4L, 1bDV and 1b-4VL = see Table III; ro = Rodinia; rv = RiVEC; ge = Genomics; DIns = dynamic instruction count in regions of interest; Cyc = cycle count; VIns = percent of dynamic instructions that are of vector type; ctrl = vector control instructions; ialu = vector integer ALU instructions; imul = vector integer multiplication and division instructions; fpu = vector floating-point instructions; xe = vector cross-element instructions; us = unit-stride memory instructions; st = constant-stride instructions; idx = indexed memory instructions; prd = predicated instructions; DOp = total number of performed operations (i.e., scalar instructions + vector instructions $\times$ active vector length); VOp = percent of operations performed by a vector unit; VPar = logical parallelism (i.e., the total number of performed operations divided by the total dynamic instructions in vectorized code); WInf = work inflation (i.e., the total number of performed operations divided by the total dynamic instructions in scalar code); ArInt = arithmetic intensity (i.e., arithmetic operations / memory operations); DTsk = dynamic number of parallel tasks; geomean = calculated for Rodinia, RiVec, and genomics applications.

length that is similar to a typical SIMD width in common mobile SoCs (e.g., Samsung M3 [55]) implementing traditional packed-SIMD architectures such as Arm NEON. This unit also leverages two of its existing execution pipelines in its associated big core for vector execution and shares the same data cache port with the big core to minimize area overheads. This unit exemplifies future modest integrated vector units implementing next-generation vector architectures [61]. *1b-4L* is a conventional big.LITTLE system including one big and four little cores without any vector execution support. *1bIV-4L* includes a big core with an integrated vector unit and a cluster of four little cores.

1bDV is a long-vector system with a decoupled vector engine connected to a big core, which is similar to aggressive vector machines such as Tarantula [17]. Figure 3 shows its vector engine's micro-architectural details. *1bDV* includes a large vector register file (i.e., 2048-bit vector length), wide multi-lane execution pipelines (e.g., 16 arithmetic operations can be processed in parallel on 32-bit vector elements), a high-bandwidth connection to an L2 cache that can support more requests in parallel than L1 caches, and deep command and data buffers to aggressively decouple memory accesses from vector computation. Those significant resources enable best-in-class performance for data-parallel workloads at the cost of extra silicon area.

1b-4VL is a big.VLITTLE system that has an equivalent area compared to *1bIV-4L*. To ensure no cycle time penalty to the existing little cores, we conservatively model fully pipelined communication paths between multiple vector-specific components and the cores. For example, it takes a full cycle to broadcast a command from the VCU to all little cores, to send a request from the VMIU to a VMSU, to send data from a core to the VSU and the VXU, and to forward a data response from the VLU to a core. We added a fixed penalty of 500 cycles to the beginning of each vector region to account for switching overheads (e.g., saving thread contexts and flusing little core pipelines).

B. Application Benchmarks

We evaluate the systems using eight task-parallel applications from Ligra benchmark suite [58] and eight data-parallel applications from Rodinia suite [10], RiVec suite [50], and a genomics benchmark suite. We also study three data-parallel kernels to further understand the performance of the simulated systems. *vvadd* and *mmult* do vector addition and matrix multiplication respectively. *saxpy* performs a single-precision $A \times X + Y$ on two vectors. Table V and Table IV summarize these applications and kernels. The set of studied applications and kernels represent real-world workloads running in mobile SoCs such as smartphones, drones, and AR/VR systems. For example, *backprop* performs a forward classification on fully connected layers, and *kmeans* clusters items into similar groups. Both algorithms are used in machine learning mobile applications. *particlefilter* is an image processing algorithm tracking an object in each frame of an input video, which can be used to do image processing in smartphones and AR/VR headsets. *blackscholes* and *jacobi2d* are data analytics applications that represent big-data processing workloads such as natural language processing. *sw* (i.e., Smith-Waterman) implements a local genome sequence alignment algorithm that finds regions of similarity between reference and query DNA sequences. Graph analytics are important to perform fast on-device analysis of large datasets in mobile devices without relying on the cloud.

For data-parallel applications and kernels, we manually vectorize them using RISC-V RVV vector intrinsics supported in LLVM 13. We parallelize task-parallel applications using a task runtime system (i.e., similar to Intel TBB [52] and Cilk Plus [26]) implementing a random work-stealing algorithm that helps distribute tasks dynamically and evenly across heterogeneous cores. Since the *1bIV-4L* system can support both vector execution on its big core and scalar tasks on its little cores, we implement both scalar and vectorized versions of each data-parallel application. The task-parallel runtime system dynamically chooses which version of a task to run depending on which core executes the task.

V. Performance Evaluation

In this section, we describe our cycle-level performance results comparing the *1b-4VL* system to the *1bIV-4L* and *1b-DV* baseline systems for both task-parallel and data-parallel workloads. We then analyze performance impacts of multiple vector element groups, packed vector element support, and reconfigurable cache subsystem on the *1b-4VL* system's performance.

A. Overall Performance

Figure 4 shows the overall performance of all simulated systems normalized to *1L* for both sets of task-parallel and data-parallel applications.

Task-parallel applications – *1bIV-4L* and *1b-4VL* perform the same since they both execute the same scalar code without using their integrated vector unit and VLITTLE vector engine respectively. In scalar mode, *1b-4VL* simply bypasses all additional vector-specific components, which incurs no performance overheads. *1bIV-4L* and *1b-4VL* are able to achieve 1.7× speedup over the *1bDV* system since the *1bDV* system is able to use only its big core to execute scalar code. Since not all workloads can be efficiently vectorized (e.g., irregular graph applications) and task-parallel applications remain an important set of workloads in mobile SoCs, it is hard to justify a large decoupled vector engine in small mobile SoCs to accelerate only data-parallel applications. Both *1bIV-4L* and *1b-4VL* are more efficient than *1bDV* in using their computing resources with the help of the work-stealing runtime system that dynamically distributes tasks to available cores.

Data-parallel applications – The *1b-4VL* system performs 1.6× faster than *1bIV-4L* while being able to achieve roughly half of *1bDV*'s performance. The *1bDV* system supports 2048-bit hardware vector length that is significantly larger than the 128-bit vector length of the integrated vector unit in *1bIV-4L* and the 512-bit vector length of *1b-4VL*. The larger a hardware vector length is, the better a system can amortize its front-end instruction overheads by performing more computation per vector instruction. Figure 5 shows that across all vectorized kernels and applications, *1bDV* and *1b-4VL* perform significantly fewer instruction fetch requests than the *1bIV-4L* system does. In addition, the four little cores in *1bIV-4L* independently execute tasks, which results in duplicated instruction fetches among the four little cores and runtime overheads to dynamically distribute tasks across the system.

The *1bDV* supports higher compute throughput using its wide execution pipelines that are capable of performing up to 16 arithmetic operations on 32-bit data elements in parallel. In contrast, *1bIV-4L*'s integrated vector unit is able to perform four operations on 32-bit data elements per cycle, and its four little cores can issue four scalar instructions in total per cycle. Meanwhile, *1b-4VL* can perform eight simple integer arithmetic and multiplication operations, and four complex integer and floating-point operations per cycle on 32-bit vector elements. Moreover, *1b-4VL* and *1bDV* support respectively two and four element groups that can effectively hide the latency of complex instructions (e.g., multiplication and division) in compute-intensive workloads such as *mmult*, *blackscholes*, *jacobi-2d*, and *lavamd* (see Table V).

1b-4VL and *1bDV* systems are also able to fetch data more efficiently from memory than *1bIV-4L* does. Figure 6 shows the normalized number of data memory requests in the three systems. For workloads with regular memory access patterns (i.e., using unit-stride and constant-stride memory instructions) such as *vvadd*, *saxpy*, *pathfinder*, and *lavamd*, *1b-4VL* and *1bDV* can efficiently fetch multiple per-element pieces of data using a single wide memory request. In contrast, the integrated vector unit's limited hardware vector length, the scalar execution of four little cores, and runtime overheads in the *1bIV-4L* system require significantly more memory requests compared to both *1b-4VL* and *1bDV*.

B. Reconfigurable Compute Pipeline

To evaluate performance impacts of packed-vector-element support and multiple vector element groups on the performance of *1b-4VL*, we study three configurations: (1) *1c* - one element group and no packed element support, (2) *1c+sw* - one element group with packed element support, and (3) *2c+sw* - two element groups with packed element support. Figure 7 shows their execution time breakdown.

Since all studied data-parallel applications use 32-bit data types, enabling packed-vector-element support effectively doubles the *1b-4VL*'s hardware vector length and increases its compute throughput. This reduces the number of executed instructions, which results in less dependency stalls (e.g., *raw_mem* and *raw_llfu* cycles in *saxpy*, *pathfinder*, and *lavamd*). In addition, the utilization of execution pipelines in little cores is increased since more per-element arithmetic operations can be performed in the same cycle (e.g., integer addition) and back to back in consecutive cycles (e.g., floating-point multiplication).

The *2c+sw* configuration introduces a second element group to *1b-4VL* compared to the *1c+sw* configuration. The second element group helps hide the long latency of complex instructions such as floating-point multiplication by overlapping the execution of the first and second element group, which reduces further read-after-write dependency stalled cycles in compute-intensive applications such as *blackscholes*, *particlefilter*, and *lavamd*. Some of memory latency can be hidden as well in memory-intensive workloads such as *saxpy*, *jacobi-2d*, and *pathfinder* since more memory requests from multiple element groups can be in flight at the same time. In some cases such as *vvadd* and *backprop*, adding the second element group slightly increases *simd* stalled cycles. The little cores run out of sync due to more memory requests conflicting for resources (e.g., accessing the same L1D bank) in the cache subsystem, which eventually stalls the VCU from issuing micro-operations in lock step to all little cores.

C. Performance Impacts of Data Buffering

We evaluate performance impacts of data buffering in the *1b-4VL* system by varying the size of its VMU's load

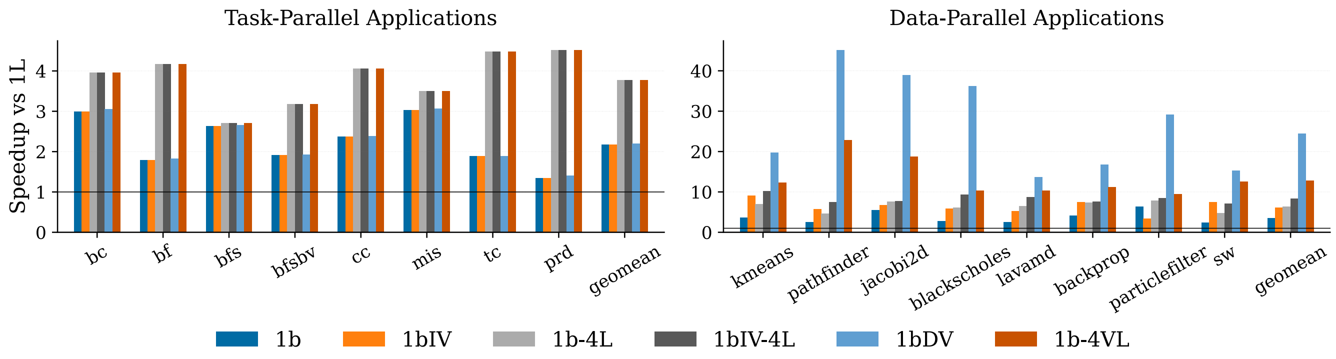

Figure 4. Speedup over 1L - *1L* = one little core; *1b* = one big core; *1bIV* = one big core with an integrated vector unit; *1b-4L* = one big & four little cores; *1bIV-4L* = one big core with an integrated vector unit & four little cores; *1bDV* = one big core with a decoupled vector engine; *1b-4VL* = big.VLITTLE system with one big and a VLITTLE engine of four little cores.

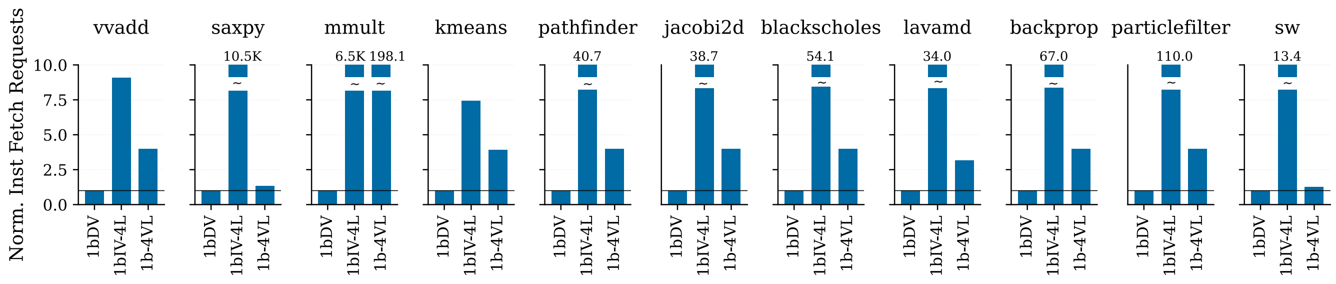

Figure 5. Number of Instruction Fetch Requests to Memory - All numbers are normalized to *1bDV*.

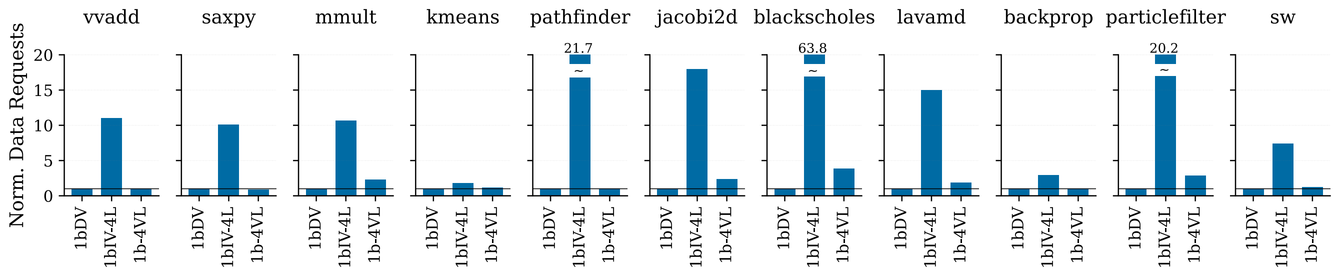

Figure 6. Normalized Number of Data Requests to Memory - All numbers are normalized to *1bDV*.

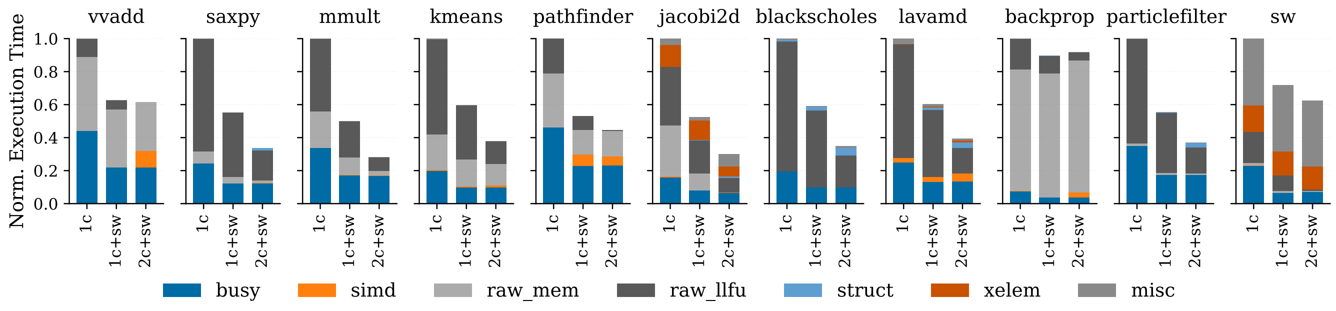

Figure 7. Average Execution Time Breakdown of Four Little Cores in *1b-4VL* – *1c* = *1b-4VL* with one chime (vector element group); *1c+sw* = *1b-4VL* with one chime & packed vector elements; *2c+sw* = *1b-4VL* with two chimes & packed vector elements; *busy* = cycles in which little cores are not stalled; *simd* = stalled cycles due to lock-step issuing of micro-ops in the VCU; *raw_mem* = stalled cycles due to waiting for memory; *raw_llfu* = stalled cycles due to little cores waiting for long-latency micro-ops to complete; *struct* = stalled cycles due to structural hazards; *xelem* = stalled cycles due to cross-element micro-ops; *misc* = other stalled cycles (e.g., no micro-op from the VCU).

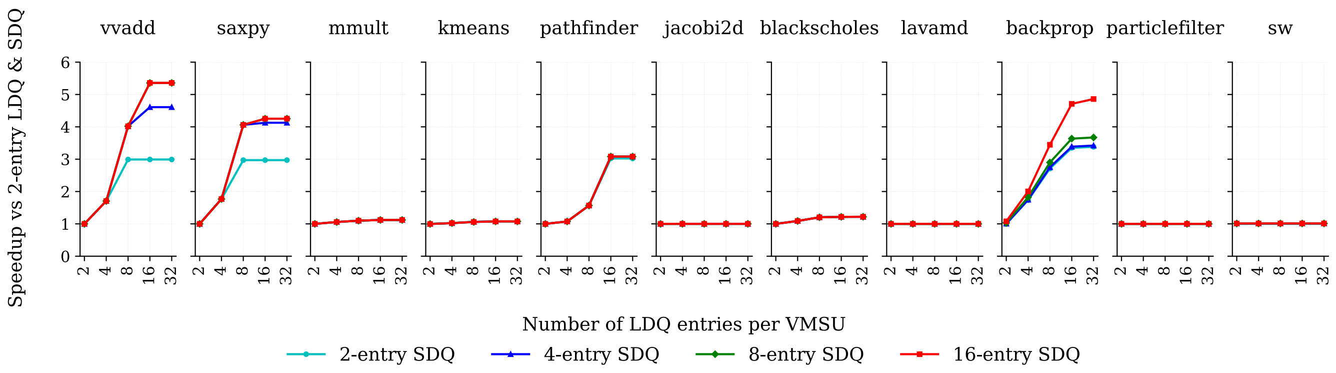

Figure 8. Performance Impacts of VMU's Load Data Queue (LDQ) and Store Data Queue (SDQ)

and store data queues. Figure 8 shows that by increasing the amount of data buffering in the VMU, the performance of memory-intensive workloads such as *vvadd*, *saxpy*, *pathfinder*, and *backprop* can be improved significantly. Supporting larger load and store data queues allows more in-flight memory requests to fully take advantage of the available bandwidth provided by the logically shared multi-bank L1 data cache in the VLITTLE engine. This enables more decoupling of memory accesses and arithmetic computation, which can effectively hide long memory latency in memory-sensitive workloads. However, deep data buffering comes at significant area cost. Our technique to re-purpose SRAM data arrays in L1 instruction caches, which are otherwise unused in the vector mode, as data buffers for load and store requests provides an area-efficient way to unlock more memory-level parallelism and memory-computation decoupling without adding non-trivial area overheads.

VI. Area Evaluation

In this section, we first evaluate area overheads of additional vector-specific components in the *1b-4VL* system using a post-systhesis area model. We then estimate the area of the *1bDV* system by referencing an open-source RISC-V decoupled vector machine.

Methodology – We use a post-synthesis component-level area modeling methodology to evaluate area overheads of extra hardware added to support a VLITTLE engine composed of four little cores. We implement key components of the VLITTLE engine in RTL. We use two different RTL models for the little core: simple and Ariane [70]. The simple core is our in-house single-issue in-order processor implementing RISC-V RV64IMAF. The Ariane model is an open-source Linux-capable RISC-V RV64G in-order core. For L1 instruction and data caches, we use a 32KB two-way set-associative cache model that is configured to support either 64-bit or 512-bit data path. For the VCU and VMU, we model multiple micro-operation, scalar data, command queues, and store address CAM according to the configuration of the *1b-4VL* system shown in Table III. For the VXU, we implement a unidirectional 64-bit-wide ring network. We then use a commercial standard-cell-based toolflow in a 12-nm technology node to generate post-synthesis area results.

Table VI. Post-Synthesis Area Results

Component	Area	Simple		Ariane	
	(k μm^2)	4L	4VL	4L	4VL
Simple core	26.1	×4	×4		
Ariane core	41.8			×4	×4
32KB L1I with 64b data path	40.3	×4	×4	×4	×4
32KB L1D with 64b data path	40.3	×4		×4	
32KB L1D with 512b data path	41.6		×4		×4
VXU: Ring network	0.3		×1		×1
VMU:	2.9		×1		×1
• Micro-op & command queues	1.7				
• Store-address CAM	0.8				
• Line buffers	0.4				
VCU:	2.0		×1		×1
• Micro-op queue	1.0				
• Data queue	1.0				
Total		427.0	437.4	489.8	500.1
4VL vs. 4L overhead			**2.4%**		**2.1%**

4L = a cluster of four little cores with L1I and L1D caches; 4VL = a VLITTLE engine with four little cores, L1I, and L1D caches.

Area overheads of big.VLITTLE – Table VI shows the detailed area comparison between a cluster of four little cores and an equivalent VLITTLE engine. The 4VL engine only adds around 2% area overhead (i.e., 2.4% if using the simple little cores and 2.1% if using Ariane little cores) compared to the 4L cluster including their private L1 data and instruction caches. The main area overheads come from the VCU and VMU that includes multiple FIFO queues for micro-operations, scalar data, and VMU commands. For a complete big.LITTLE system including a big core, its private L1 and L2 caches, and interconnect network, we expect the area overhead of big.VLITTLE architectures to be less than 1% of the entire system.

First-order area estimate of 1bDV – We reference Ara [9], an open-source decoupled vector machine, to estimate the area of our simulated decoupled vector engine. We use an Ara configuration that includes eight 64-bit compute lanes that are equivalent to the 16x 32-bit lanes in our simulated decoupled vector engine in the *1bDV*, which makes the areas of the two vector engines comparable. The work reported that the Ara configuration has an area of around 6,000 kilo-gates (kGE) (i.e., 738 kGE per lane) and that an Ariane core without its L1 caches has an area of 524 kGE. Table VI shows that one L1 32KB cache's area is roughly the same as one Ariane core's area without caches. Therefore, a cluster of four Ariane cores with their L1 instruction and data caches is as large as an eight-64-bit-lane Ara vector engine (i.e., roughly 6,000 kGE). Since our VLITTLE cluster incurs less than 3% of area overhead compared to a cluster of four Ariane cores with their L1 caches, a four-core VLITTLE cluster's area is comparable to the simulated decoupled vector engine used in *1bDV*. More detailed area analysis of the *1bDV* system is left for future work.

VII. Power & Energy Evaluation

In this section, we first qualitatively evaluate power and energy efficiency of big.VLITTLE architectures. We then explore the potential of voltage/frequency scaling to further improve the performance and power efficiency of big.VLITTLE architectures for data-parallel workloads.

A. Qualitative Power and Energy Efficiency Analysis

In terms of power, a big.VLITTLE system leverages existing little core pipelines (i.e., functional units and register files) for vector execution and the big core for scalar control flow. Extra vector-specific components mainly consist of small FIFO command/data buffers and control logics, and they can be power-gated in the scalar mode to avoid leakage power. In the vector mode, front-end components (e.g., fetch, decode stages, and branch predictor) in little cores and control logic in instruction caches are not used, so they do not contribute to the overall dynamic power consumption. Therefore, we do not anticipate a big.VLITTLE system to draw significantly more power than an equivalent big.LITTLE system.

Regarding energy efficiency, by reconfiguring little cores as a medium-sized decoupled vector engine, big.VLITTLE architectures can reduce significantly the number of instruction and data memory accesses due to less dynamic instructions (Figure 5) and wider data memory requests (Figure 6). This reduction translates directly to less energy consumed in the memory subsystem for data-parallel workloads compared to an equivalent big.LITTLE system with an integrated vector unit. In addition, higher performance at a similar power consumption yields higher energy efficiency. Previous work [40,41] has also studied and reported the energy efficiency of vector architectures. Future work can explore a more detailed power/energy evaluation of big.VLITTLE.

B. Voltage/Frequency Scaling Design Space Exploration

Methodology – We assume the voltage/frequency of the big and little core clusters can be scaled independently, which is similar to typical commercial big.LITTLE systems (e.g., Samsung Exynos [31] and Qualcomm Snapdragon [22]). We use previously reported average power consumption of a big and little core at different voltage/frequency levels [67] to estimate the average power consumption of the big and little core clusters in our simulated big.LITTLE and big.VLITTLE systems. Table VII shows the selected voltage/frequency levels for big and little core clusters and their corresponding average power consumption as reported in the previous work.

Table VII. Average Power Consumption of a Big and Little Core at Multiple Voltage/Frequency Levels

Big core			Little core		
	Frequency (GHz)	Avg Power (W)		Frequency (GHz)	Avg Power (W)
b0	0.8	0.432	l0	0.6	0.043
b1	1.0	0.591	l1	0.8	0.059
b2	1.2	0.841	l2	1.0	0.095
b3	1.4	1.205	l3	1.2	1.450

The average power consumption of a big and little core at different voltage/frequency levels was reported in previous work [67]. The work used an Odroid XU+E board that includes a Samsung Exynos 5410 SoC and per-cluster voltage/current sensors for the measurement. This SoC consists of four big Arm Cortex-A7 cores and four little Arm Cortex-A15 cores. The power consumption was measured by running 26 benchmarks on all cores at different frequencies (i.e., 500-1200MHz for the little cores and 800-1500MHz for the big cores at corresponding appropriate voltage levels).

In this design space exploration study, we assume that both *1bIV-4L* and *1b-4VL* have similar average power consumption compared to *1b-4L*. To estimate the power consumption of *1bDV*, we reference the decoupled vector engine in Tarantula [17]. The work reported its vector engine consumed roughly 40% more power than its out-of-order super-scalar core. Both the vector engine and the out-of-order core were clocked at the same frequency, which is similar to our simulated *1bDV* system. We assume the same power consumption ratio between the big core and its decoupled vector engine at different voltage/frequency levels. More accurate power models for all designs are left for future work.

Performance impacts of voltage/frequency scaling – Figure 9 show the performance of *1bIV-4L* and *1b-4VL* at different combinations of voltage/frequency levels for big and little core clusters. For *1bIV-4L*, whether to boost the big core or the little core cluster for higher performance depends on specific workloads and existing voltage/frequency levels. For example, in *blackscholes*, boosting the big core cluster (e.g., from $(b1,l1)$ to $(b2,l1)$) always yields better performance than boosting the little core cluster (e.g., from $(b1,l1)$ to $(b1,l2)$). In contrast, in *sw*, boosting the little core cluster is more beneficial than boosting the big core cluster.

For *1b-4VL*, boosting the big core cluster while keeping the voltage/frequency level of the little core cluster yields insignificant performance benefits across all applications except *sw*. Different from *1bIV-4L* in which both the big and little cores work together on the main computation, in *1b-4VL*, the big core mainly works as a control core for the VLITTLE engine that handles all heavy vector computation. Slowing down the big core to a certain limit does not cause the VLITTLE engine to stall since the engine has a deep command buffer and long vector length. For *sw*, since only 69% of the work is vectorized (see *VOp* in Table V) and the rest is executed by the big core, boosting the big core while keeping the little cores running at the same voltage/frequency level helps increase the overall performance.

Performance and power consumption trade-offs – Figure 10 shows the performance and estimated average power consumption of all studied voltage/frequency combinations

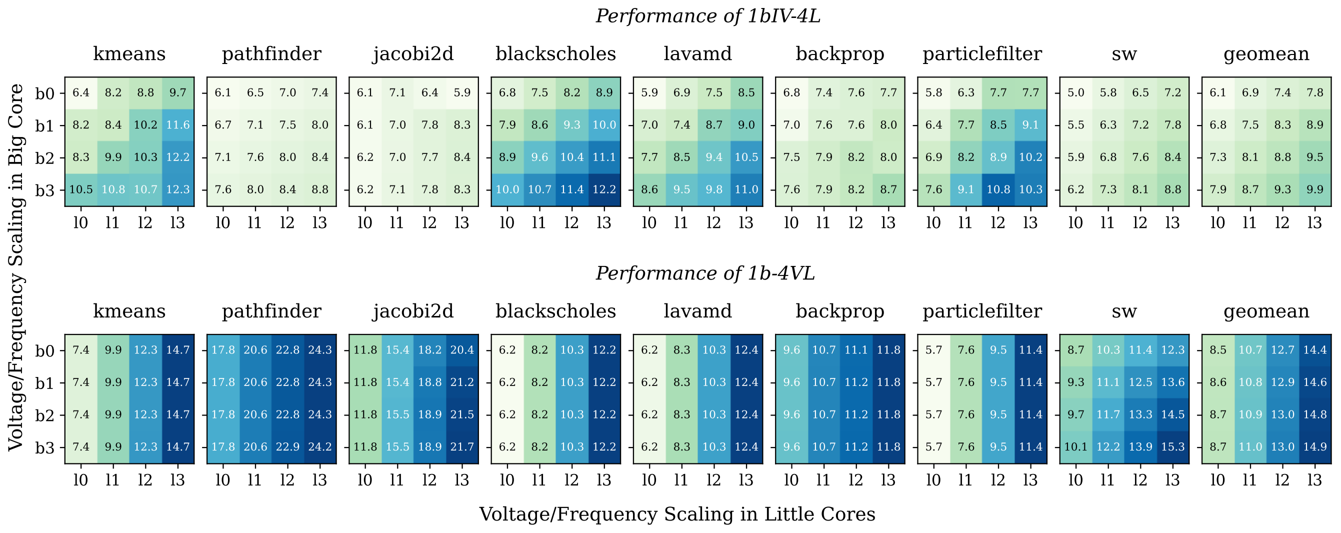

Figure 9. Performance of 1bIV-4L and 1b-4VL at Different Voltage/Frequency Scaling Levels for Big and Little Cores - All studied voltage/frequency levels (i.e., $\{b0, b1, b2, b3\}$ and $\{l0, l1, l2, l3\}$) are shown in Table VII. The performance numbers show speedup over *1L* system running at 1GHz. The color scaling for each application is the same for both 1bIV-4L and 1b-4VL.

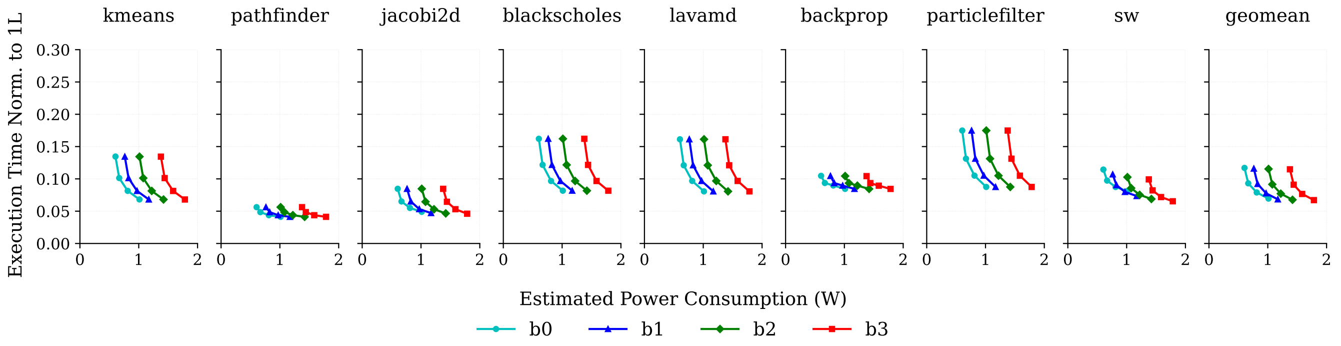

Figure 10. Execution Time and Estimated Power Consumption of *1b-4VL* at Different Voltage/Frequency Levels - Each performance-power data point corresponds to a combination of big and little core's voltage/frequency levels shown in Table VII.

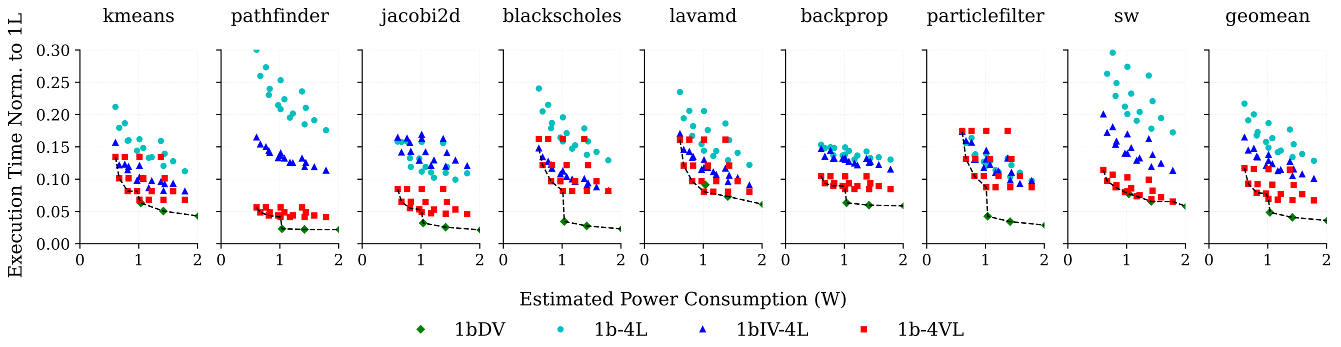

Figure 11. Execution Time and Estimated Power Consumption of Multiple Designs at Different Frequencies - The dotted lines show Pareto frontier curves. Each performance-power data point corresponds to a combination of big and little core's voltage/frequency levels shown in Table VII.

for *1b-4VL*. Boosting the little core cluster and slowing down the big core help *1b-4VL* achieve the Pareto optimal performance/power curve. Given a certain power budget, the power saved by lowering down the big core's speed can be used to boost the little cores that execute most of the vector computation in data-parallel workloads. This power trading translates to higher performance and efficiency for *1b-4VL*.

Figure 11 shows performance/power data points for all designs including *1bDV*, *1b-4L*, *1bIV-4L*, and *1b-4VL*. In the low-power region (i.e., less than 1W), *1b-4VL* stays on the Pareto optimal performance/power curve across all data-parallel applications. For *1bDV*, despite its ability to deliver high performance for data-parallel applications, its power-hungry decoupled vector engine makes it not feasible in the low-power region. In the high-power region (i.e., more than 1W), *1b-4VL* is able to get close to the performance/power efficiency of *1bDV*. It is important to note that unlike *1bDV*, *1b-4VL* does not sacrifice the performance of important task-parallel applications to achieve this power/performance efficiency for data-parallel workloads (see Section V).

VIII. Related Work

Rockcress [6] extends many-core architectures with vector-like execution support by dynamically grouping multiple small cores together into vector groups executing the same stream of instructions. Different from big.VLITTLE, Rockcress targets scale-out many-core systems with scratchpads and mesh-based tiled network by loosely coupling multiple cores in a vector group together, which requires frequent intra-vector-group synchronizations, a nontrivial amount of data buffering, and a dedicated instruction-forwarding network in each core's scratchpad as the group size grows. In contrast, big.VLITTLE architectures aim to provide vector execution in small mobile systems, which allows a small number of OS-capable little cores in a VLITTLE engine to execute strictly in lock step, which greatly simplifies its design and implementation. Rockcress adopts a non-standard vector-thread abstraction [34] and requires extensive compiler-level support to insert implicit instruction barriers so that its scalar cores do not run out of resources in vector mode. In contrast, big.VLITTLE architectures support next-generation vector architectures and compilers out of the box.

Vector-thread architectures [5, 33–35, 40] enable a SIMD-like micro-architecture to execute MIMD code. They propose a non-standard hybrid vector/thread ISA abstraction that would require non-trivial programming model and compiler support. Vector-thread architectures strive to achieve a single abstraction for both task- and data-parallel workloads with certain trade-offs in performance, programmability, and energy efficiency. In contrast, big.VLITTLE provides both multi-thread and vector solutions in a single micro-architecture to provide the best multi-thread support when running task-parallel workloads and the most efficient vector support when running data-parallel applications.

Cray X1 [15] provides options to group multiple vector engines into a single long-vector machine, which is more applicable to large-scale super-computing systems already equipped with vector engines than to small mobile SoCs. Taking an opposite approach compared to big.VLITTLE architectures, vector lane threading [54] reconfigures multiple lanes in a vector engine as individual scalar cores that can execute independently from each other. Similarly, Libra [46] attempts to overcome the inflexibility of SIMD accelerators by allowing different lanes to work in either SIMD or VLIW execution styles.

Some prior work has proposed to gang multiple scalar threads dynamically to amortize their front-end instruction overheads [19, 30, 33–35, 37, 43, 45]. While preserving the simplicity of multi-thread programming abstractions, those approaches spend extra energy at run time to dynamically align multiple streams of scalar execution. Some other reconfigurable architectures aim to exploit both thread-level and instruction-level parallelism such as CoreFusion [27], MorphCore [28], and others [20,29,64,66]. Unlike big.VLITTLE architectures, they do not explore data-level parallelism.

IX. Conclusion

This paper has demonstrated that big.VLITTLE architectures offer a compelling high-performance and area-efficient solution to accelerating data-parallel workloads in heterogeneous multi-core mobile systems. The reconfigurability of big.VLITTLE architectures resolves the fundamental tension between performance and area in implementing next-generation vector architectures, which opens up opportunities to provide the performance level of decoupled vector engines for data-parallel workloads in small mobile systems without sacrificing either valuable silicon area on chips or performance of task-parallel workloads. This work provides a small but important step toward a future era of efficient next-generation vector architecture support in mobile SoCs. Future research can explore the scalability of big.VLITTLE architectures beyond the scope of mobile SoCs.

Acknowledgments

This work was supported in part by NSF PPoSS Award #2118709, NSF SHF Award #2008471, and the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, and equipment, tool, and/or physical IP donations from Intel, Synopsys, and Cadence. The authors acknowledge and thank Alejandro Rico and Jose Joao for their valuable feedback. The authors also thank Tiago Muck for his help in understanding the implementation of Arm CHI protocol in gem5 and Moyang Wang for his baseline work-stealing runtime system. We also thank the PyMTL3 team for their help in implementing RTL models in PyMTL3 and the RISC-V community for RISC-V vector extension support in GNU and LLVM toolchains. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation theron. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

X. Artifact Appendix

A. Abstract

This guide describes how to set up and run experiments to reproduce the cycle-level timing results shown in Section V and Section VII. More specifically, this appendix provides:

- How to access a Docker image and a `README` file used to run the experiments
- Import a Docker image containing necessary tools (e.g., python, GNU toolchain, and LLVM)
- Build our custom gem5 simulator
- Build the applications and kernels reported in Table IV and V
- Run the experiment to reproduce performance results reported in Figure 4, 5, 6, 7, 8, 9, 10, and 11.

B. Artifact check-list

- **Program:** Our custom gem5 simulator, along with all applications and kernels reported in Table IV and V, are included in the Docker image.
- **Compilation:** We include a python-based flow to compile both gem5, applications, and kernels in the Docker image. More details can be found in the `README` file.
- **Binary:** We provide pre-built RISC-V GNU toolchain and LLVM compiler in the Docker image. Once the Docker image is started, gem5 and application binaries can be generated using our compilation flow.
- **Data set:** All necessary data sets are included in the Docker image. More details are included in the `README` file.
- **Run-time environment:** We ran our simulations on a system with Centos-7, devtoolset-7, and python-3.7.4. We expect our workflows to work on different Linux distros (e.g., Ubuntu) and with newer toochains. However, we have not tested them on any system other than above one.
- **Hardware:** We have tested our simulation flow on `x86_64` machines with Intel processors.
- **Execution:** We provide scripts to run all performance experiments and to produce results as described in the paper. A more detailed description is provided in the `README` file.
- **Output:** Running the scripts as instructed in the `README` file generates multiple plots that should be similar to Figure 4, 5, 6, 7, 8, 9, 10, and 11.
- **Experiments:** All experiments using our custom gem5 simulator are captured in our simulation workflow. Details about how to use the workflow can be found in the `README` file.
- **Required disk space:** The Docker image takes about 16 GB of disk space once loaded. It is recommended to have at least 20 GB of free disk space to run all simulations using the Docker image.
- **Workflow preparation time:** The compilation and simulation workflows are already provided in the Docker image. Please refer to the `README` file for more details.
- **Simulation time:** There are roughly 1000 gem5 simulation runs to produce all performance results reported in the paper. The shortest simulation took about 15 minutes while the longest simulation took approximately 20 hours on an Intel Xeon E7-8867 v4 CPU. Each simulation run is single-thread, and we recommend running as many simulations as possible in parallel. Please refer to our `README` file regarding how to run multiple simulations in parallel using our workflow.
- **Publicly available:** A pre-built Docker image containing all source code and necessary environment setup is publicly available at `https://doi.org/10.5281/zenodo.7029093`.
- **Code licenses:** Berkeley-style open source License; MIT Licence; Creative Commons Attribution 4.0 International
- **Workflow framework:** We use `doit`, a Python-based automation tool, to automate running our simulations.
- **Archive DOI:** `10.5281/zenodo.7029093`

C. Description

- **How to access:** A Docker image containing source code and pre-built software dependencies and its corresponding `README` file are publicly available at: `https://doi.org/10.5281/zenodo.7029093`.
- **Hardware dependencies:** There is no restriction on hardware dependencies for running our simulations. However, we have only tested our simulation flow on `x86_64` machines with Intel processors.
- **Software dependencies:** We have tested our simulation flow on Centos-7 with devtoolset-7 and python-3.7.4. All dependencies (e.g., RISC-V GNU toolchain, LLVM 14.0.0, and Python-3.7.4) are pre-built and provided in the Docker image.
- **Datasets:** We include datasets used by our applications and kernels in the Docker image.
- **Models:** Our provided gem5 source code includes detailed cycle-level models of the little core, big core, decoupled vector unit, integrated vector unit, VLITTLE engine, cache subsystem, on-chip network, and DRAM as described in Table II and III.

D. Installation

Please refer to the `README` file for details about how to load and set up the provided Docker image. The image provides necessary environment software dependencies to build gem5 and cross-compile applications. Our `README` file shows how to build gem5 and applications steps by steps using our compilation workflow.

E. Experiment workflow

We use `doit`, a Python-based automation tool, to manage running our simulations. Detailed instructions on how to use our workflows are provided in the `README` file. We also provide multiple scripts to process raw simulation data and plot the generated performance results. The `README` file also provides instructions on how to use our scripts.

F. Evaluation and expected results

Our provided simulation flow and data processing scripts can reproduce results reported in Section V. All plots are generated in the following directory `/usr/local/artifacts/scripts/plots` in the Docker image:

- `plot-perf.svg` – Speedup of all studied systems over *1L* shown in Figure 4.
- `plot-inst_reqs_breakdown.svg` and `plot-data_reqs_breakdown.svg` – Normalized numbers of instruction fetch and data requests to memory shown in Figure 5 and 6.
- `plot-amc_exec_time_breakdown.svg` – Average execution time breakdown of four little cores in *1b-4VL* configuration as shown in Figure 7.
- `plot-lsq_perf.svg` – Performance impacts of VMU's load and store data queues as shown in Figure 8.
- `plot_freq_perf_heatmap.py` – Performance of *1bIV-4L* and *1b-4VL* at different voltage/frequency scaling levels as shown in Figure 9.
- `plot_freq_power.py` – Execution time and estimated power consumption of all designs at different voltage/frequency scaling levels as shown in Figure 10 and 11.

References

[1] Abts, Dennis and Bataineh, Abdulla and Scott, Steve and Faanes, Greg and Schwarzmeier, Jim and Lundberg, Eric and Johnson, Tim and Bye, Mike and Schwoerer, Gerald. The Cray BlackWidow: A Highly Scalable Vector Multiprocessor. *ACM/IEEE Conference on Supercomputing (SC)*, Nov 2007.

[2] N. Adit and A. Sampson. Performance Left on the Table: An Evaluation of Compiler Auto-Vectorization for RISC-V. *IEEE Micro*, 2022.

[3] Apple Unleashes M1. Online Webpage, 2020.

[4] K. Asanović. *Vector Microprocessors*. Ph.D. Thesis, EECS Department, University of California, Berkeley, 1998.

[5] C. Batten, H. Aoki, and K. Asanović. The Case for Malleable Stream Architectures. *Workshop on Streaming Systems (WSS)*, Nov 2008.

[6] P. Bedoukian, N. Adit, E. Peguero, and A. Sampson. Software-Defined Vector Processing on Manycore Fabrics. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2021.

[7] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A Neural Probabilistic Language Model. *The Journal of Machine Learning Research*, 13, 2003.

[8] N. Binkert, B. M. 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. *SIGARCH Computer Architecture News (CAN)*, 39(2):1–7, Aug 2011.

[9] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini. Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI. *TVLSI*, 28(2):530–543, 2020.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, , and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. *Int'l Symp. on Workload Characterization (IISWC)*, Oct 2009.

[11] C. Chen, X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen, C. Li, Y. Pu, J. Meng, X. Yan, Y. Xie, and X. Qi. Xuantie-910: A Commercial Multi-core 12-stage Pipeline Out-of-order 64-bit High Performance RISC-V Processor with Vector Extension: Industrial Product. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2020.

[12] H. Chen, Y. Dai, H. Meng, Y. Chen, and T. Li. Understanding the Characteristics of Mobile Augmented Reality Applications. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2018.

[13] R. Collobert and J. Weston. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. *Int'l Conference on Machine Learning (ICML)*, Jul 2008.

[14] M. Demler. MediaTek Steps Up to Tablets: MT8135 Brings Heterogeneous Multiprocessing to Big.Little. *Microprocessor Report*, Aug 2013.

[15] T. H. Dunigan, J. S. Vetter, J. B. White, and P. H. Worley. Performance Evaluation of the Cray X1 Distributed Shared-Memory Architecture. *IEEE Micro*, 25(1):30–40, 2005.

[16] C. Eoyang, R. H. Mendez, and O. M. Lubeck. The Birth of the Second Generation: the Hitachi S-820/80. *Conference on Supercomputing*, 1988.

[17] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: A Vector Extension to the Alpha Architecture. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2002.

[18] Arm's AMBA 5 CHI Ruby Model in gem5. Online Webpage, accessed Nov 20, 2021.

[19] J. Gonzalez, Q. Cai, P. Chaparro, G. Magklis, R. N. Rakvic, and A. Gonzalez. Thread fusion. *Int'l Symp. on Low-Power Electronics and Design (ISLPED)*, Aug 2008.

[20] S. Gupta, S. Feng, A. Ansari, and S. Mahlke. Erasing Core Boundaries for Robust and Configurable Performance. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2010.

[21] L. Gwennap. Renesas Mobile Goes Big (and Little). *Microprocessor Report*, Feb 2013.

[22] L. Gwennap. Qualcomm Tips Cortex-A57 Plans: Snapdragon 810 Combines Eight 64-Bit CPUs, LTE Baseband. *Microprocessor Report*, Apr 2014.

[23] L. Gwennap. Samsung First with 20 nm Processor. *Microprocessor Report*, Sep 2014.

[24] M. Huzaifa, R. Desai, X. Jiang, J. Ravichandran, F. Sinclair, and S. V. Adve. Exploring Extended Reality with ILLIXR: A New Playground for Architecture Research. *arXiv preprint arXiv:2004.04643*, 2020.

[25] Intel SSE4 Programming Reference. Intel Reference Manual, 2007.

[26] Intel Cilk Plus Language Extension Specification, Version 1.2. Intel Reference Manual, Sep 2013.

[27] E. Ipek, M. Kirman, N. Kirman, and J. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2007.

[28] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt. MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2012.

[29] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable Lightweight Processors. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2007.

[30] J. Kim, S. Jiang, C. Torng, M. Wang, S. Srinath, B. Ilbeyi, K. Al-Hawa, and C. Batten. Using Intra-Core Loop-Task Accelerators to Improve the Productivity and Performance of Task-Based Parallel Programs. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2017.

[31] M. Kim, H. Kim, H. Chung, and K. Lim. Samsung Exynos 5410 Processor-Experience the Ultimate Performance and Versatility. *White Paper*, 2013.

[32] K. Kitagawa, S. Tagaya, Y. Hagihara, and Y. Kanoh. A Hardware Overview of SX-6 and SX-7 Supercomputer. *NEC Research & Development Journal*, 44(1):2–7, Jan 2003.

[33] R. Krashinsky, C. Batten, and K. Asanović. Implementing the Scale Vector-Thread Processor. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 13(3), Jul 2008.

[34] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović. The Vector-Thread Architecture. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[35] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović. The Vector-Thread Architecture. *IEEE Micro*, 24(6):84–90, Nov/Dec 2004.

[36] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2003.

[37] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-Core Chip Multiprocessing. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2004.

[38] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[39] R. Lee. Multimedia Extensions for General-purpose Processors. *IEEE Workshop on Signal Processing Systems (SiPS) Design and Implementation*, 1997.

[40] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerator Cores. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2011.

[41] C. Lemuet, J. Sampson, J. Francios, and N. Jouppi. The Potential Energy Efficiency of Vector Acceleration. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*, Nov 2006.

[42] Y. Leng, C.-C. Chen, Q. Sun, J. Huang, and Y. Zhu. Energy-efficient Video Processing for Virtual Reality. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2019.

[43] G. Long, D. Franklin, S. Biswas, P. Oritz, J. Oberg, D. Fan, and F. T. Chong. Minimal Multi-threading: Finding and Removing Redundant Instructions in Multi-threaded Processors. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2010.

[44] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian. The gem5 Simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[45] M. Mckeown, J. Balkind, and D. Wentzlaff. Execution Drafting: Energy Efficiency Through Computation Deduplication. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[46] Y. Park, J. J. K. Park, H. Park, and S. Mahlke. Libra: Tailoring SIMD Execution Using Heterogeneous Hardware and Dynamic Configurability. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2012.

[47] O. M. Parkhi, A. Vedaldi, and A. Zisserman. Deep Face Recognition, 2015.

[48] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, 1996.

[49] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling Interactive Perception Applications on Mobile Devices. *Int'l Conference on Mobile Systems, Applications, and Services*, Jun 2011.

[50] C. Ramírez, C. A. Hernández, O. Palomar, O. Unsal, M. A. Ramírez, and A. Cristal. A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures. *ACM Trans. on Architecture and Code Optimization (TACO)*, 17(4):1–30, 2020.

[51] R. Randhawa. Software Techniques for ARM big. LITTLE Systems. Arm Whitepaper, 2013.

[52] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O'Reilly, 2007.

[53] RISC-V Vector Extension (Version 0.10). Online Webpage, 2021.

[54] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis. Vector Lane Threading. *Int'l Conference on Parallel Processing (ICPP)*, Aug 2006.

[55] J. Rupley, B. Burgess, B. Grayson, and G. D. Zuraski. Samsung M3 Processor. *IEEE Micro*, 39(2):37–44, 2019.

[56] R. M. Russel. The Cray-1 Computer System. *Communications of the ACM*, 21(1):63–72, Jan 1978.

[57] M. Sato. The Supercomputer "Fugaku" and Arm-SVE Enabled A64FX Processor for Energy Efficiency and Sustained Application Performance. *Int'l Symp. on Parallel and Distributed Computing (ISPDC)*, 2020.

[58] J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Feb 2013.

[59] M. L. Simmons, H. J. Wasserman, O. M. Lubeck, C. Eoyang, R. Mendez, H. Harada, and M. Ishiguro. A Performance Comparison of Four Supercomputers. *Communications of the ACM*, 1992.

[60] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46, 2016.

[61] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. The ARM Scalable Vector Extension. *IEEE Micro*, 37(2), 2017.

[62] T. Ta, L. Cheng, and C. Batten. Simulating Multi-Core RISC-V Systems in gem5. *Workshop on Computer Architecture Research with RISC-V*, 2018.

[63] S. Tagaya, M. Nishida, T. Hagiwara, T. Yanagawa, Y. Yokoya, H. Takahara, J. Stadler, M. Galle, and W. Bez. The NEC SX-8 Vector Supercomputer System. *High Performance Computing on Vector Systems*, May 2006.

[64] D. Tarjan, M. Boyer, and K. Skadron. Federation: Repurposing Scalar Cores for Out-Of-Order Instruction Issue. *Design Automation Conf. (DAC)*, Jun 2008.

[65] R. Thabet, R. Mahmoudi, and M. H. Bedoui. Image Processing on Mobile Devices: An Overview. *Int'l Image Processing, Applications and Systems Conference (IPAS)*, Nov 2014.

[66] A. Tino, C. Collange, and A. Seznec. SIMT-X: Extending Single-Instruction Multi-Threading to Out-of-Order Cores. *ACM Trans. on Architecture and Code Optimization (TACO)*, 17(2), 2020.

[67] E. Vasilakis, I. Sourdis, V. Papaefstathiou, A. Psathakis, and M. G. Katevenis. Modeling Energy-Performance Tradeoffs in ARM big. LITTLE Architectures. *Int'l Symp. on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Sep 2017.

[68] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra. High-throughput CNN Inference on Embedded ARM Big.LITTLE Multicore Processors. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 39(10), 2019.

[69] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang. Machine Learning at Facebook: Understanding Inference at the Edge. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2019.

[70] F. Zaruba and L. Benini. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, 27(11):2629–2640, 2019.

[71] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang. Edge Intelligence: Paving the Last Mile of Artificial Intelligence with Edge Computing. *Proceedings of the IEEE*, 107(8):1738–1762, 2019.

[72] Y. Zhu and V. J. Reddi. High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2013.