Decoupling Data Supply from Computation for Latency-Tolerant Communication in Heterogeneous Architectures

TAE JUN HAM, Princeton University
JUAN L. ARAGÓN, University of Murcia
MARGARET MARTONOSI, Princeton University

In today's computers, heterogeneous processing is used to meet performance targets at manageable power. In adopting increased compute specialization, however, the relative amount of time spent on communication increases. System and software optimizations for communication often come at the costs of increased complexity and reduced portability. The Decoupled Supply-Compute (DeSC) approach offers a way to attack communication latency bottlenecks automatically, while maintaining good portability and low complexity. Our work expands prior Decoupled Access Execute techniques with hardware/software specialization. For a range of workloads, DeSC offers roughly 2× speedup, and additional specialized compression optimizations reduce traffic between decoupled units by 40%.

CCS Concepts: • **Computer systems organization → Architectures**; **Heterogeneous (hybrid) systems;**

Additional Key Words and Phrases: Accelerators, communication management, decoupled architecture

ACM Reference Format:
Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2017. Decoupling data supply from computation for latency-tolerant communication in heterogeneous architectures. ACM Trans. Archit. Code Optim. 14, 2, Article 16 (June 2017), 27 pages.
DOI: http://dx.doi.org/10.1145/3075620

1. INTRODUCTION

The deceleration in Moore's Law and Dennard scaling have over the years led to the rise first of on-chip parallelism [33, 46] and subsequently of specialization and heterogeneity [14, 35]. From data centers to embedded systems, computing devices now all employ mixes of general purpose cores, specialized cores, and accelerators [43, 55]. Effective use of heterogeneity and specialization can, however, be complex. First, from an Amdahl's Law point of view, as specialized accelerators speed up computations, the communication or memory operations that feed them represent even more of the remaining performance slowdown [31, 54]. Thus, the long-troubling "memory wall" becomes even more challenging in accelerator-oriented designs.

A second challenge in accelerator-oriented design lies in tailoring software-managed communication to reduce communication cost; software complexity often increases and

---

This is an extension of the conference paper "DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures" presented at the *48th Annual IEEE/ACM International Symposium on Microarchitecture* [25].

Authors' addresses: T. J. Ham and M. Martonosi, Department of Computer Science, Princeton University, Princeton, NJ 08540 USA; emails: {tae, mrm}@princeton.edu; J. L. Aragón, Computer Engineering Department, University of Murcia, 30100 Murcia, Spain; email: jlaragon@ditec.um.es.

DOI: http://dx.doi.org/10.1145/3075620

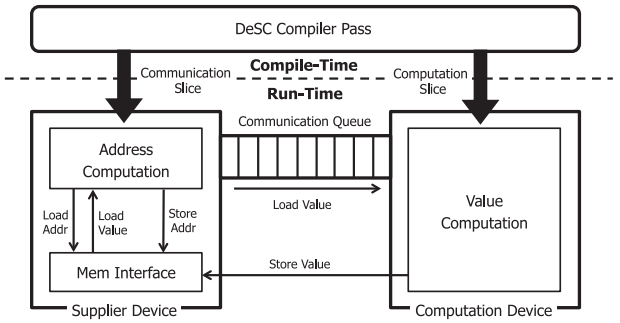

Fig. 1. DeSC overview.

performance predictability and portability usually decreases. For example, for a loosely coupled accelerator [14, 15] with scratchpad memory, transfers in and out of it are typically tightly tailored to the scratchpad size. In addition to blocking computations to fit the scratchpad, programmers must also work to maximize the overlap of computation and communication. Even worse, small variations in such storage's capacity or port count can require designs and codes to be rewritten or reoptimized.

Relative to scratchpads, cache memories can seem preferable in terms of programmer effort and software portability, but many issues remain. For example, caches still require programmer effort to balance computation and communication. In addition, by exposing variable communication latency to the accelerator, caches can end up requiring a more conservative hardware design, either regarding computation speed or regarding the use of at-accelerator data buffering. Finally, a cache's demand-fetched and line-at-a-time nature can incur performance overhead when compared to a carefully managed scratchpad memory system.

In order to attack the aforementioned challenges, our work seeks to improve the performance, programmer effort, and software portability of heterogeneous systems. *Decoupled Supply-Compute* (DeSC) is a communication management approach that aims to provide the performance and energy efficiency of scratchpad memory, while offering programmability similar to a cache-based approach. Inspired by the Decoupled Access/Execute (DAE) architecture model initially proposed by Smith [50], DeSC employs compiler techniques to automatically separate data access and address calculations from value computations. Once separated, each slice is targeted at different hardware—either general-purpose cores or specific accelerators tailored to their role.

Figure 1 shows an overview of the proposed framework. DeSC decouples host data memory access, performed by a *Supplier Device* (SuppD), from value computation performed by a *Computation Device* (CompD) using an LLVM-based compile-time framework. Program source code is input to the DeSC compiler, which then divides the original program stream into the communication slice on the SuppD device and the computation slice on the CompD device.

In running the communication slice, the SuppD fetches and provides necessary memory data to the CompD running the computation slice. On the other side, the CompD receives input data from the SuppD, performs value computations, and where needed, pushes output back to the SuppD to be stored in memory.

Decoupling communication from computation has several advantages. First, the SuppD can be tailored to the needs of address computation and memory access. Units can be appropriately sized for a memory-heavy workload, and the SuppD need not include floating point units, for example. Second, the SuppD can run ahead of the CompD

Table I. Decoupled Code Example

Original	AP slice	EP slice

`for (i=0; i<100; i++) {`
`  v1 = LOAD(&a[i]);`
`  v2 = LOAD(&b[i]);`
`  val = v1 + v2 * k;`
`  STORE(&c[i], val);`
`}`
	`for (i=0; i<100; i++) {`
	`  v1 = LOAD(&a[i]);`
	`  PRODUCE(v1);`
	`  v2 = LOAD(&b[i]);`
	`  PRODUCE(v2);`
	`  STORE_ADDR(& c[i]);`
	`}`
		`for (i=0; i<100; i++) {`
		`  v1 = CONSUME();`
		`  v2 = CONSUME();`
		`  val = v1 + v2 * k;`
		`  STORE_VAL(val);`
		`}`

to hide memory latency. Third, a chip can be provisioned with arbitrary mixes of SuppDs and CompDs to strike a good balance for an expected workload. Having evaluated DeSC using LLVM and Sniper, our contributions are:

- We propose a DeSC framework that automates and optimizes communication for heterogeneous systems.
- We improve the original DAE by introducing hardware/compiler support to benefit from out-of-order communication and out-of-order commit of terminal loads (Section 3).
- We introduce novel architectural support and compiler optimizations to avoid Loss of Decoupling events (Section 5).
- We present a compression scheme to reduce traffic between the supplier device and the compute device, which allows DeSC to be deployed in more bandwidth-constrained scenarios (Section 6).
- We evaluate the optimized DeSC approach with modern accelerator workloads and explore specialization opportunities (Section 7).
- We use DeSC to significantly improve on-chip accelerator performance compared to a baseline accelerator with its own cache (Section 8).

2. MOTIVATION

2.1. Decoupled Access/Execution (DAE)

DAE architecture model was proposed by Smith [50] in an early attempt to overcome memory wall issues while retaining implementation simplicity. DAE divides a program into two independent streams, one containing all memory-related instructions (including address calculation and memory accesses) and another containing all compute-related instructions. A pair of *Access Processor* (AP) and *Execute Processor* (EP), connected by several First-In, First-Out (FIFO) queues, are responsible of executing both streams. DAE improves memory latency tolerance, since the AP can run ahead of the EP while overlapping data accesses with computation.

Table I shows an example of program code split into access and execute portions. Data items $a[i]$ and $b[i]$ must be accessed from memory and then passed over to the execute side for computation. While different implementations vary, in some DAE-style systems, specific instructions such as `PRODUCE` and `CONSUME` would support this, and DeSC adopts this approach. The code example also illustrates that in DeSC, a `STORE` instruction in the original program is split into `STORE_ADDR` and `STORE_VAL` to decouple address computation and value computation.

Prior work has explored many different aspects of the DAE approach [1, 4, 16, 29, 50], but they do not specifically focus on data supply challenges for heterogeneous systems. Our work more fully embraces today's specialization trends by assuming that the CompD has no direct access to memory (much like current loosely-coupled accelerators or DySER [24]) and that the SuppD is specialized for data supply.

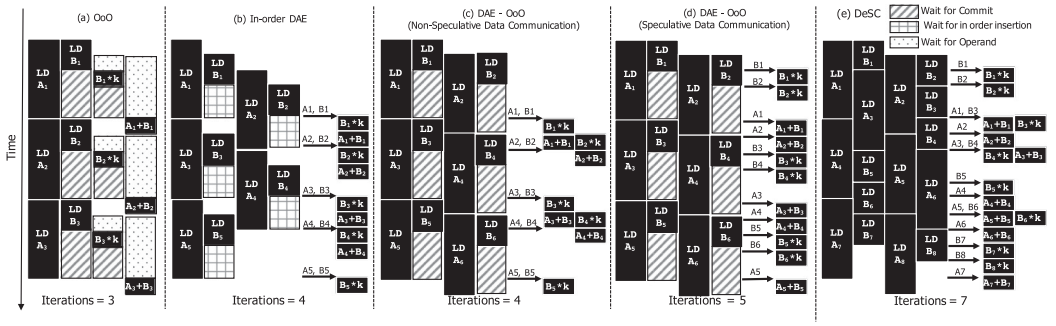

Fig. 2. Motivational example for DeSC. For the code in Table I, the dynamic instruction schedules are shown for several architecture models. Some instructions (address computation/stores) were omitted for clarity. Arrows represent a data communication between both sides on decoupled cases. ROB size of 4- and 2-issue width are assumed for OoO cores. Outstanding loads are limited to 4. Each column can be understood as the occupancy of either the ROB (OoO cases) or the MSHRs (in-order case).

2.2. Challenges in Out-of-Order DAE

Figure 2 motivates our work by showing execution timelines for the same operations (following the example code in Table I) on different architectures. Using small ROB and MSHR sizes, the figure illustrates key bottlenecks that different approaches experience or overcome. DeSC's goal is to reduce or tolerate different types of memory or execution latencies to run programs faster and use hardware more efficiently.

In-order DAE architectures (Figure 2(b)) were originally envisioned as a lower-complexity latency tolerance alternative to out-of-order processors (Figure 2(a)). However, they are not exclusive approaches. In fact, Figure 2(c) shows them as complementary latency-tolerance techniques. One limitation with both approaches in Figures 2(b) and 2(c) is that all communications need to happen in order. The original DAE idea [50]—striving for simplicity compared to then-nascent OoO ideas—had to perform communication in-order, because the computation device was an in-order core with only access to the head of a queue. On the other hand, an out-of-order DAE architecture (as in Figure 2(c)) still requires in-order (commit time) communication to avoid propagating mis-speculated data. In such cases, even if a later load (in program order) has lower memory latency, its data cannot be passed to the EP until all earlier loads are completed.

One way of achieving out-of-order communication in an OoO DAE is simply inserting data into communication queues at issue time (i.e., out-of-order) as in Figure 2(d). This speculative approach can communicate data earlier at the expense of propagating mis-speculated data, making it necessary to flush communication queues on all mispredicted branches. As we will show later, DeSC (Figure 2(e)) overcomes this limitation by supporting out-of-order communication without complex speculation recovery mechanisms.

Even with out-of-order communication, Figure 2(d)'s design still fails to achieve significant performance improvement. This is because its instruction commit happens in-order. For example, in Figure 2(d), after communicating a short-latency `Load B`, another load cannot be issued, since the long-latency `Load A` blocks the commit of `Load B` and its ROB entry cannot be freed. A similar argument applies to the in-order core case. While the original DAE assumed fixed memory latency and thus all memory requests naturally returned in-order without extra structures (e.g., load queue, MSHRs), modern memory systems with variable memory latency require structures, like the load queue and MSHRs, to buffer returned data values to insert them into the communication queue in program order. This requirement of non-speculative, in-order communication (or commit) and resource constraints form a bottleneck. DeSC

Table II. Code examples incurring in LoD events.

Case (a)	Case (b)	Case (c)

`for (i=0;i<10;i++)`
`  a[i] = a[i]*x;`
`v = a[5] * y;`
	`v = a[i] * x;`
	`if (v>0.5)`
	`  d = b[i];`
		`v = a[i] * x;`
		`d = b[(int)v];`

(Figure 2(e)) overcomes this second problem by allowing out-of-order commit for certain load instructions with architectural/compiler support (Section 3.2).

2.3. Loss of Decoupling Events in DAE

DeSC also improves on general DAE performance by attacking several of the loss of decoupling (LoD) events (as termed in [4, 19, 57]) that limit DAE performance. LoD events occur when the data or control flow of the AP depends on results coming from the EP, which limits AP run-ahead distance. There are three primary categories of inter-core dependences that cause LoD events as described below. Our work on DeSC reexamines DAE approaches with additional hardware/software support to reduce LoDs.

(a) Data Aliasing occurs when data is computed, stored to memory and then later reloaded from memory. In this case, the AP may or may not stall depending on the distance between the preceding store address instruction and following load instruction. If the value was not computed by the time a dependent load happened, then the AP must stall until the value is computed by the EP and passed back to the AP. DeSC, alternatively, uses a novel hardware optimization technique that enables store-to-load forwarding within a decoupled scenario as we will explain in Section 5.1.

(b) Computation Dependent Control Flow occurs when a computed result determines the AP's control flow (e.g., a conditional exit). This happens in applications where a computed value must be communicated back to the AP to determine a branch outcome. In such applications, the AP should stall until it receives the computed value from the EP. DeSC uses a compiler transformation introduced in Section 5.2 to mitigate this.

(c) Computation Dependent Data Address occurs when a computed result is used as an address for a subsequent data load. This behavior is seen in some scientific codes where a computed result is quantized (e.g., histogram, interpolation). In such cases, the AP must stall until it receives the address from the EP. Section 5.3 describes our software approach to mitigate this.

2.4. DAE AP-EP Bandwidth Consumption Challenge

One of the notable costs in employing DAE architectures such as DeSC is that they incur significant amounts of data traffic between the AP and EP. In such approaches, the EP does not have a cache-like structure to allow it to reuse data that the SuppD has supplied. As a result, the AP must re-supply a data item every time the EP needs something not present in its extremely limited local storage (e.g., register file).

AP-EP traffic can be substantial and therefore can significantly affect a system's performance and power. When both the AP and the EP are placed on the same die, the AP-EP traffic exerts considerable pressure on a network-on-chip. On the other hand, if the AP and the EP are placed on different dies (e.g., off-chip accelerator working as the EP), this traffic places stress on often-limited off-chip bandwidth. Both of these scenarios can easily lead to performance degradation of the overall system. To make matters worse, as DeSC improves the performance of the data supplier with respect to the conventional DAE, it leads to higher AP-EP bandwidth consumption compared to DAE. Figure 3 shows that DeSC AP-EP bandwidth consumption exceeds 5GB/s in certain applications (refer to Section 7 for evaluation details).

Fortunately, the characteristics of DeSC AP-EP traffic offer hints at ways to reduce it. First, since the EP has no cache, data items supplied from the AP to the EP often have

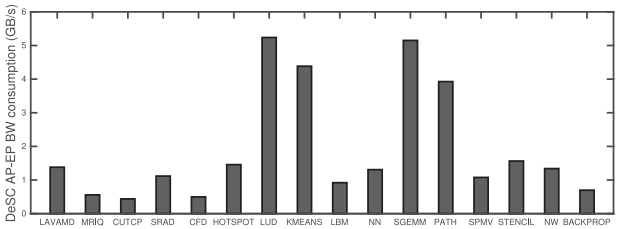

Fig. 3. DeSC AP-EP BW consumption.

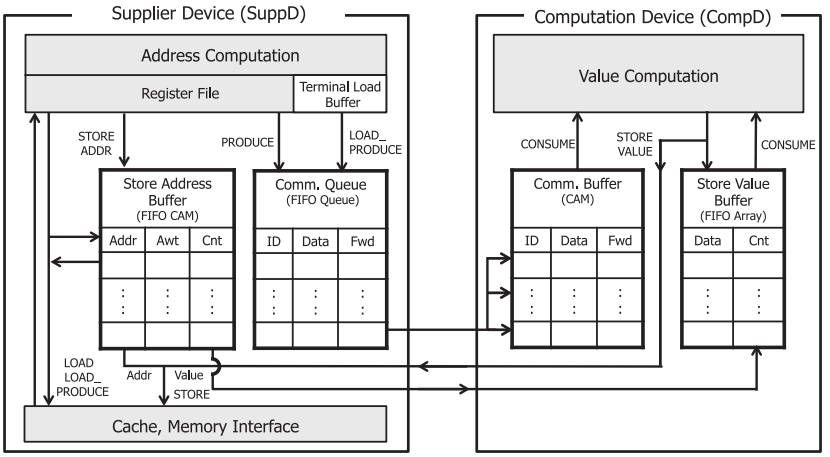

Fig. 4. Hardware implementation of DeSC.

relatively high *value temporal locality*. That is, the exact same value is communicated multiple times. Second, on some of the accelerator-friendly workloads, the data supplied from the AP to the EP can have high *value spatial locality*. That is, the differences within some values sent from the AP to the EP are small. To reduce the traffic between the AP and the EP by exploiting these two characteristics, Section 6 explores a simple and effective compression technique targeting these patterns.

3. DeSC ORGANIZATION

DeSC communication management consists of two specialized hardware units: an SuppD and a CompD, along with a hardware-software interface for their interactions and compiler techniques for targeting them. Where DAE primarily envisioned two instruction-programmable processor cores with different roles, our work is open to more specialization. That is, both SuppD and CompD could be either processors or accelerators, or (as we discuss here) processors with tailoring to each of their roles.

Figure 4 shows DeSC's hardware implementation. Grey boxes represent an abstracted view of the hardware modules that either calculate the memory addresses or compute the output values. Here, SuppD is a nearly general-purpose core—an out-of-order pipeline with ROB, RegFile, and a number of integer functional units for calculating memory addresses—but sizing choices are tailored to its role and no floating point functional units are needed. Likewise, CompD can be another out-of-order core or a specialized hardware accelerator for a particular application. Either way, the CompD is tailored to its role by removing memory hierarchy access while the SuppD supplies it with data as needed.

Table III. ISA Extensions for DeSC

Supplier Device	Computation Device
PRODUCE(Reg)	Reg=CONSUME()
LOAD_PRODUCE(Addr)	
STORE_ADDR(Addr)	STORE_VAL(Reg)

For data supply, a *Communication Queue* (CommQ) interconnects SuppD to CompD, and feeds into a *Communication Buffer* (CommBuf) from which value lookup can be performed. The SuppD also includes a *Store Address Buffer* for updating the memory hierarchy when a computed value is returned. Finally, Table III lists the added instructions on both sides to support DeSC.

3.1. Communication Mechanism

The CommQ is logically a FIFO hardware queue for interconnecting SuppD and CompD. Data items are placed into the CommQ by a `PRODUCE` instruction executed on SuppD. In addition to a data value, each entry in CommQ also holds a program-order *id* assigned at a `PRODUCE` instruction's dispatch. In an out-of-order SuppD, data is inserted at commit, thus guaranteeing that no mis-speculated data can pollute the queue. Therefore, there is no need for a recovery/flush mechanism. From there, data are transmitted to CommBuf as space is available. The CommQ size dictates the maximum run-ahead distance allowed between SuppD and CompD. Our evaluations use a 512-item queue. Physically, this queue can be implemented as RAM, with storage on both SuppD and CompD sides. If CompD is a hardware accelerator, then the queue can also be logically mapped into the CompD's scratchpad memory.

The CommBuf is a CAM-based array on the CompD side. In addition to a data value, each entry in CommBuf holds a program-order *id* originated on SuppD and propagated from the CommQ. This *id* allows a `CONSUME` instruction (which also gets program-order *id* at dispatch) to find the data produced by its counterpart. Because we do not allow speculative data in the queue, for every producing instruction on the SuppD side there will be a consuming counterpart on the CompD side. If the `CONSUME` is dispatched before its data arrives to the CommBuf (rare), then it remains in the CompD's instruction window, waiting for its data to arrive. Whenever a new data is moved into the CommBuf, the *id* and value are reported to the instruction window, which eventually wakes-up the `CONSUME`. This CAM buffer enables data to be consumed out-of-order for better performance, but the entry is not released until commit (enforcing that no mis-speculated `CONSUME` instructions could evict any item). The CommBuf size limits the degree of out-of-order data consumption and load reordering (we used a 64-entry buffer for our evaluation). Compared to a single large searchable buffer, the combination of CommQ and CommBuf lets DeSC benefit from OoO data consumption with lower area and energy consumption.

3.2. Exploiting Terminal Loads

Decoupled execution has highest leverage when the SuppD side remains well ahead of the CompD. To support that, the goal is to insert data into the queue as soon as possible. Inserting speculative data to the CommQ could achieve that but would incur large overhead when speculation turns out to be wrong. On the other hand, forcing queue insertion to wait until traditional commit time often kills the benefit of out-of-order processing. To overcome those limitations, DeSC allows an out-of-order commit for limited cases called *terminal loads* while preserving the benefit of out-of-order processing.

Table IV. Code Example for Terminal Loads

Original	DeSC SuppD	DeSC CompD

`for (i=0; i<100; i++) {`
`  idx = LOAD(&a[i]);`
`  tmp = LOAD(&v[idx]);`
`  val = tmp * c;`
`  STORE(&b[i], val);`
`}`
	`for (i=0; i<100; i++) {`
	`  idx = LOAD(&a[i]);`
	`  LOAD_PRODUCE(&v[idx]);`
	`  STORE_ADDR(&b[i];`
	`}`
		`for (i=0; i<100; i++) {`
		`  tmp = CONSUME();`
		`  val = tmp * c;`
		`  STORE_VAL(val);`
		`}`

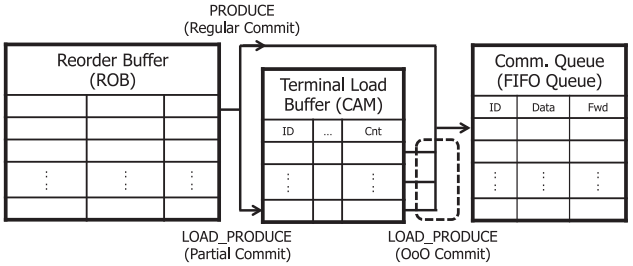

Fig. 5. Out-of-order commit for terminal loads.

Terminal loads are defined as loads in which the fetched value is going to be used *only* on the CompD side for computational purposes. One unique feature we will exploit is that they have no dependent instructions on the SuppD core, which enables an early commit as explained next. Note that in traditional processors, a load instruction will typically have subsequent users of the data, and thus terminal loads would be rare or non-existent. However, they frequently appear in a decoupled architecture where a load's consumer is commonly part of the CompD instruction stream. To the best of our knowledge, DeSC is the first approach that exploits the specific characteristics of terminal loads to avoid unnecessary stalls in the SuppD core due to long latency loads and thus preventing stalls due to lack of space in the SuppD ROB. In DeSC, terminal loads are determined at compile-time (Section 4) and thus need relatively modest hardware support. The compiler marks them using a special `LOAD_PRODUCE` instruction (on the SuppD side) that combines the original load request together with a `PRODUCE` into one single instruction.

Table IV shows a code example for a terminal load. In this example, `LOAD(&a[i])` is not *terminal*, because its value will be reused for the next instruction. On the other hand, `LOAD(&v[idx])` is a *terminal* one, because its value will be only used for computation purposes (on the CompD). Thus, the compiler transforms the latter load into a `LOAD_PRODUCE` instruction on the SuppD slice.

Figure 5 illustrates the hardware aspects of the terminal load optimization. A `LOAD_PRODUCE` instruction that reaches the head of the ROB is allowed to "partially" commit if it is already issued. Note that partial commit happens even if it has not completed yet (e.g., upon a cache miss still awaiting to be serviced). It is then retired from the ROB and moved to a separate Terminal Load Buffer, a CAM-based structure, where it will remain until the data value is received and then inserted into the CommQ. At that point, the terminal load can fully commit, out-of-order. (Section 3.5 discusses exception handling and consistency.) On the CompD side, its `CONSUME` counterpart will eventually receive the value on a successful value lookup in the CommBuf.

It is important to note that any load in the Terminal Load Buffer is non-speculative, since it reached the head of the ROB, and so we still enforce that no mis-speculated data can pollute CommQ/CommBuf (similarly as for `PRODUCE` instructions). Second, there are no dependent instructions on the SuppD waiting for the loaded value, so

no special actions need be taken when the commit occurs. Because this optimization directly enqueues the value to be passed to the CompD, it does not use either registers or an extra instruction unnecessarily on the SuppD side. Even more, it also allows later loads to proceed, so the SuppD ROB does not fill waiting on this load to finish.

Summarizing, even with a relatively small Terminal Load Buffer (32 entries), this technique efficiently reduces stalls, providing significant speedup as we will show in Section 7.

3.3. Memory Update Mechanism

After computing a value, CompD might need to communicate it back to the SuppD side, which is the interface to the memory hierarchy. To manage that, a *Store Address Buffer* (SAB) is needed on the SuppD side (Figure 4). This structure keeps information about all in-flight store instructions in program order. Similar to the LSQ's *store queue* in a conventional processor, the SAB is a FIFO structure that supports associative searches of a memory address to (i) detect memory dependences and (ii) allow decoupled store-to-load forwarding (Section 5.1).The size of SAB limits the number of stores a SuppD device can perform without waiting for CompD to generate value of the store. Our evaluation uses a 128-entry SAB.

In DeSC, a store from the original instruction stream is split into two: one for the SuppD in charge of providing the address and another for the CompD device providing the data to be stored. In the SuppD, a `STORE_ADDR` reserves an empty entry in the SAB at dispatch time, which holds the destination address when it becomes ready. When this instruction reaches the head of the ROB, it can safely retire from the SuppD regardless of whether the value on the CompD side has been computed or not. At that point, the "awaiting" bit (see Figure 4) is set to indicate there is an outstanding store waiting for the data to arrive from the CompD.

Note that a `STORE_ADDR` instruction is always paired (in program order) with a `STORE_VAL` instruction on the CompD side. The `STORE_VAL` instruction communicates the value back to the oldest entry (head entry) in the SAB at commit time, which checks the "awaiting" bit. If set (which is the common case), then the entry can be freed and the value is submitted to the memory hierarchy (and the *store* completes). In the rare case of finding the "awaiting" bit not set (CompD has temporary surpassed the SuppD core), the CompD core is stalled until the `STORE_ADDR` pair commits (setting the "awaiting" bit), and we can submit the value to memory and release the SAB entry.

3.4. Control Flow Management

Different from previous DAE-based approaches that communicated branch outcomes between access/execute processors through separate queues, the proposed DeSC framework lets each side manage its own control flow independently. This is a desired property, since DeSC is aimed at decoupled *heterogeneous* systems where the CompD side could be implemented as a specialized computing accelerator with its own internal control flow management (or as a general-purpose core with a traditional branch predictor). The SuppD side will typically implement a conventional branch predictor and its corresponding recovery mechanism.

Two key points allow DeSC to act asynchronously in terms of control flow. First, CommQs never contain mis-speculated data, a condition enforced by the SuppD commit-time insertion policy used by `PRODUCE`s and `LOAD_PRODUCE`s. Second, the CommBuf's commit-time deletion policy avoids wrong evictions by mis-speculated `CONSUME` instructions (recall this does not prevent CompD from reading data from the buffer out-of-order). As a result, as long as commited `PRODUCE`s and `CONSUME`s follow matching sequences, DeSC allows for control flow divergences with the guarantee that a mispredicted path on either SuppD or CompD side will be eventually flushed, transparently

to the other side, affecting neither the correctness of the communication nor the computation.

3.5. Potential Issues and Solutions

Precise Exceptions. DeSC supports precise exceptions for most of the instructions but there are few exceptions. If any instruction (including a terminal load) in the SuppD ROB causes a fault, then precise exceptions are supported by letting all ongoing loads in the Terminal Load Buffer first complete and retire (note that all instructions in the Terminal Load Buffer are older than any instruction in the ROB) before the fault can be serviced. On the other hand, in the case of a faulting *terminal load* already in the Terminal Load Buffer, the evaluated implementation of DeSC does not support a precise exception. Alternatively, in a processor where a fault can be guaranteed to be detected in certain fixed cycles from the issue time, precise exceptions on terminal loads can be supported by letting a terminal load move to the Terminal Load Buffer only after that amount of cycles has passed since its issue time. For example, precise exceptions can be supported for *page faults* by letting a terminal load move to the Terminal Load Buffer only upon a TLB hit (recall that partial commits happen when a terminal load reaches the head of the ROB and by that cycle the TLB has been commonly proved), otherwise forcing the terminal load to continue in the ROB.

Deadlock Prevention. To save on area and power, the CommBuf has limited size (assume N), and thus CompD can only consume from the N oldest (in program order) instructions that were inserted into the CommQ. If the out-of-order commit aggressively reorders many terminal loads, then it might allow N or more younger terminal loads (or produce) to pass an older one. If those younger N fill up the CommBuf, then it is possible (though unlikely) that none of the CompD's in-flight `CONSUME` instructions would be able to find their data (if they all were waiting for older terminal loads). In that case, both the CompD and SuppD would stall, resulting in a deadlock.

To prevent this, our deadlock avoidance mechanism limits the degree of reordering (i.e., how many younger terminal loads or produce can "pass" a given terminal load). The mechanism works as follows: each new terminal load buffer entry resets its "counter" field to zero (see Figure 5). When a `LOAD_PRODUCE` fully commits, all older terminal load's counters are incremented. Similarly, when a `PRODUCE` commits, all terminal load's counters are incremented. If the oldest entry's counter ever reaches N-1, then it must commit before other entries. This mechanism guarantees at least one item in the CommBuf to be the oldest data that has not been consumed, thus, avoiding deadlocks. Note, however, that the CompD still has N-1 out-of-order items in the CommBuf to feed from. Our experiments (CommBuf with $N = 64$) show this mechanism has negligible performance impact for almost all workloads.

Memory Consistency. As explained in Section 3.2, DeSC benefits from out-of-order commit for terminal loads. By doing so, we give up the capability of supporting load-to-load ordering and store-to-load in hardware for better performance. On the other hand, DeSC guarantees store-to-store ordering (with in-order memory update) and load-to-store ordering (store can only be visible when both `STORE_ADDR` and `STORE_VAL` commits). This results in a unique memory consistency model that is stronger than some weak memory models (e.g., ARM, POWER) but weaker than stronger memory models (e.g., x86-TSO). If a stronger consistency model is desired for some instructions, then the Instruction Set Architecture (ISA) additions could include additional ordering enforcement constructs.

4. DESC COMPILER SUPPORT

The communication slice is responsible for all loads; it supplies required data to the computation slice using `PRODUCE` or `LOAD_PRODUCE` instructions. The communication

Table V. Input for the Slice Construction Algorithm

Slice	Starting Set	Operands Not Tracked	Disallowed Instruction
Communication Slice	Load and Store	Value operand of Store	Fadd, Fmul, Fdiv, and so on
Computation Slice	Store	Addr operand of Store	Load

slice is also responsible for all address calculations for both loads and stores. Store instructions in the original code are split into a `STORE_ADDR` instruction for the SuppD and a `STORE_VAL` pair for the CompD.

The computation slice has the following characteristics. First and foremost, since it has no direct memory system access, it cannot have load or store instructions. Instead of loads, it uses a `CONSUME` instruction to receive the data from a communication slice. In addition, the computation slice performs all of the program's value computations. Where those are to be stored to memory, the CompD calculates the data values and asks the SuppD to handle their storage using the `STORE_VAL` instruction. To generate the code slices, the compiler goes through three primary steps as discussed here.

1. Slicing: Using a fairly standard compiler slicing approach [58], a backward slice is constructed for SuppD and another for CompD. In each case, the algorithm extracts a subset of the program based on propagations backwards from the starting set given in Table V. If it encounters a disallowed instruction during the process, then the propagation stops. Instead, values needed at these points will be received from the other slice through special instructions that are inserted in later compiler stages.

2. Communication: The second compiler stage inserts decoupling instructions to appropriately link SuppD and CompD value communications. Within this stage, all load instructions from the original program must be replaced by a `PRODUCE` instruction inserted into the communication slice and a corresponding `CONSUME` instruction in the computation slice. Compiler dependence analysis identifies terminal loads by checking for dead values on the SuppD after the point of the load. In such cases, the `LOAD_PRODUCE` instruction is used instead of a `PRODUCE`, to indicate terminality and allow for commit optimizations.

Similarly, store instructions must be handled as well. All stores in the communication slice are replaced with the `STORE_ADDR`, while the store counterparts in the computation slice are replaced with the `STORE_VAL` instruction. For rare cases when a communication slice needs to receive the value from a computation slice, a special identifier (or magic address) is used to indicate that this store is for CompD to SuppD communication. Thus, `STORE_ADDR(MagicAddr)` and `Load(MagicAddr)` are inserted into the SuppD slice while a `STORE_VAL` instruction is added to the CompD slice.

3. Integrating Control Flow: Finally, a third phase of compilation handles control flow issues, particularly between the SuppD and the CompD. By default, both slices include all instructions that terminate basic blocks (branch, jump, etc.) from the original source code. On each side, the compiler then removes redundant instructions, because some control flows are only useful in on one side or the other. From this simplified set of control instructions, two backward control slices are constructed for SuppD and CompD. In some cases, this may cause additional disallowed instructions (Table V) to return to the code; if so, the compiler's second step (Communication transformations) is re-run to adjust for these.

5. LoD OPTIMIZATIONS FOR DeSC

5.1. Decoupled Store-to-Load Forwarding

To address the data alias LoD event (Figure 2(a)), this section presents a novel hardware optimization that enables a *decoupled* store-to-load forwarding mechanism. In

traditional DAE, the AP must stop whenever it sees a Load instruction dependent on a previous Store whose value the EP has yet to calculate. This conservative approach stalls the AP until the data arrives back from the EP. However, in case of being a "terminal load" (i.e., LOAD_PRODUCE) waiting for a store value on the CompD, we can exploit another feature: as the only purpose of the terminal load is to insert data into the CommQ for the CompD, and the value will be originated (computed) in the latter, there is no reason to stall the SuppD. Furthermore, the consumer of the LOAD_PRODUCE (on the CompD) may need the data much later if both devices are decoupled enough (or it might not even been dispatched yet). Instead of blocking the execution of the dependant LOAD_PRODUCE on the SuppD, we let it proceed into the CommQ, eventually reaching the CompD side, as any other PRODUCE or LOAD_PRODUCE, but carrying an *index* to its producing store rather than the value itself. Once the value is computed, the *index* allows the CONSUME pair to find its data on CompD.

To support this technique, a *Store Value Buffer* (SVB) is used to hold computed values on the CompD side (Figure 4) in case they need to be forwarded to upcoming dependent loads. The SVB is implemented as a FIFO array, and it is the counterpart to the SAB (that holds addresses on the SuppD side). A STORE_VAL executed by CompD reserves an entry in the SVB at dispatch time (to preserve the program order). It updates the data field after the value has been calculated. In addition, a pair of global counters are needed (one on each side) to track the number of removed entries for each SAB and SVB buffer. By adding the number of older entries in SAB/SVB to these counters, we can define an unique *st_id* per entry. Each of these global counters must be large enough to guarantee that *st_id*s are always unique for in-flight instructions.

The matching mechanism works as follows. Every LOAD_PRODUCE checks the SAB for a matching preceding STORE_ADDR. If found, then the *st_id* is inserted into the CommQ (instead of a data value) and the "Fwd" bit is set. On the CompD side, the CONSUME pair will check the "Fwd" bit for a forwarded item. If so, then it subtracts the SVB's global counter value from the *st_id* (in the data field) to obtain the entry in the SVB from which the CONSUME will get the computed data.

Finally, a value in the SVB must be kept long enough to handle any upcoming forwarded LOAD_PRODUCE. To precisely know when an SVB entry can be released, a per-entry counter ("Cnt" in Figure 4) is needed on both SAB and SVB buffers to track the number of forwarded items as follows. In the SuppD, each time a LOAD_PRODUCE finds a match in the SAB, the entry's "Cnt" is incremented. When a STORE_ADDR commits and leaves the SAB, its counter is sent to the STORE_VAL pair in CompD. There, every time a forwarded item uses the value in SVB, the counter is decremented. When the oldest SVB entry's counter becomes zero, it can be safely released. Sending the counter value for every Store does not incur much overhead, since the counter can be very small (e.g., 4bit).

5.2. Conditional Branch Optimization

As stated in Section 2.3, a conditional branch depending on computation causes the SuppD to stall until the computed value returns (Figure 2(b)). However, if executing both paths of a branch is not as expensive as waiting until the data value returns, it can be more beneficial to simply execute both branch paths.

To take advantage of such a case, we propose a compiler transformation with architectural support. Table VI shows the example case where this technique is beneficial. In the base decoupled code, as the val variable depends on computation, the SuppD has to wait until val is provided by the STORE_VAL instruction. Then, the branch will be evaluated and both LOAD_PRODUCE and STORE_ADDR instructions will be executed if it is taken. However, in this case, unconditionally executing the branch can be much more beneficial than waiting for the CompD to provide the value. Thus, from the communication slice, the branch and those instructions required to calculate the branch

Table VI. Conditional Branch Optimization Example

Original Code	Base Decoupled (SuppD)	Base Decoupled (CompD)	Transformed (SuppD)	Transformed (CompD)

`val = a[i]-b[i];`
`if(val > 0.1)`
`{`
`  tmp = LOAD(&k);`
`  tmp = tmp + 1;`
`  STORE(&k, tmp);`
`}`
	`STORE_ADDR(*);`
	`val = LOAD(*);`
	`if (val > 0.1) {`
	`  LOAD_PRODUCE(&k);`
	`  STORE_ADDR(&k);`
	`}`
		`val = a[i]-b[i];`
		`STORE_VAL(val);`
		`if (val > 0.1) {`
		`  k = CONSUME();`
		`  k++;`
		`  STORE_VAL(k);`
		`}`
			`LOAD_PRODUCE(&k);`
			`STORE_ADDR(&k);`
				`val = a[i]-b[i];`
				`k = CONSUME();`
				`if (val > 0.1) {`
				`  k++;`
				`  STORE_VAL(k);`
				`}`
				`else`
				`  STORE_INV();`

Fig. 6. Conditional branch optimization target.

outcome are removed. Then, instructions from the branch target address are executed unconditionally.

To adjust to these changes in the SuppD slice, the CompD slice is also transformed. All CONSUME instructions inside the branch are moved to the point just before the branch. In addition to these, for every STORE_VAL instruction inside the branch, a STORE_INV instruction is added to the other path of the branch. This way, every extra STORE_ADDR instruction executed in the communication slice will be invalidated accordingly. Note that this transformation can potentially lead to a load exception. When an exception is found in LOAD_PRODUCE, SuppD stalls and delay its processing until matching CompD instruction is executed. Depending on the matching CompD instruction, the exception can be processed (if matching CompD inst is STORE_VAL) or ignored (if matching CompD inst is STORE_INV).

Currently, our compile framework performs this optimization for triangle and diamond patterns, as shown in Figure 6 but it is also possible to apply this transformation for more complex patterns. To avoid the potential performance degradation from converting a large conditional basic block to an unconditional one, our framework only performs this optimization when a conditional basic block contains few instructions. More advanced heuristics based on cost-benefit analysis are also possible.

5.3. Optimizations for Computed Address

A data address that depends on computation can cause a stall, as mentioned earlier (Section 2.3c). If the time is sufficiently large, however, between when the CompD generates a data item and when the SuppD consumes it, then the SuppD may not need to stall, because data may have already been updated to memory by the time SuppD reads it. Therefore, the effect of this LoD can be reduced by using compiler techniques that try to compute the address as early as possible, with sufficient spacing before its use. In addition, for loops, some transformations that reduce the temporal locality of a computed address can reduce the effect of this LoD. The most representative example is *Loop Distribution* [57]. If a data address is computed and used in the same loop, then the loop can be distributed at a point between data address computation and a load using computed data address. When a loop is sufficiently large, this is often enough to avoid the LoD.

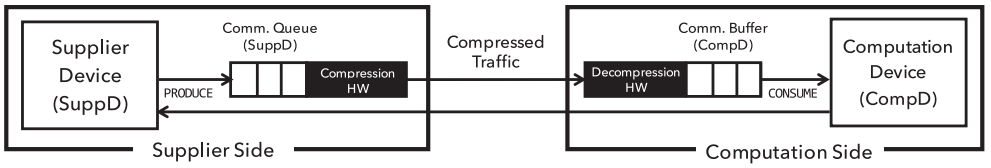

Fig. 7. DeSC SuppD-CompD Traffic Compression Overview.

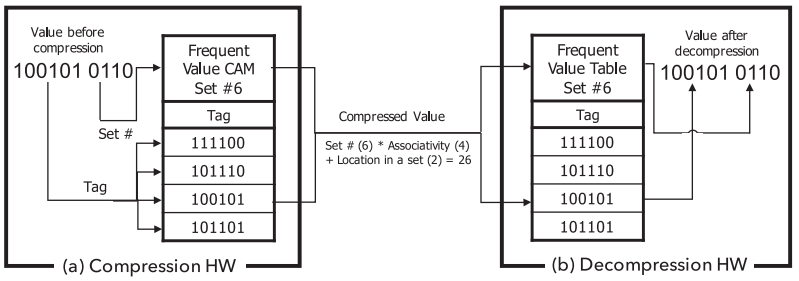

Fig. 8. Block diagram for Compression and Decompression Units.

6. DeSC INTERNAL TRAFFIC COMPRESSION

6.1. Base Scheme Design and Operation

Figure 7 shows an overview of the DeSC SuppD-CompD traffic compression scheme. A compression unit and a decompression unit (both in hardware) are placed on the SuppD and the CompD, respectively. The compression unit compresses data in the communication queue and sends compressed data over the link, which connects the SuppD and the CompD. The decompression unit decompresses data received from the link and passes this data to the communication buffer on the CompD. For simplicity, this diagram omits minor components, such as a small extra buffer or SerDes if serial link is used.

Base Compression Scheme (Figure 8(a)). In order to exploit the value locality present in DeSC SuppD-CompD traffic, a new hardware component named *Frequent Value CAM* (FVC) is introduced on the SuppD side. It is a SRAM CAM-based structure similar to a read-only associative cache except for the fact that it uses a value for addressing and does not have a separate data field. When the compression unit on the SuppD receives a new value to be sent to the CompD, it checks if this value is already in the SuppD FVC by comparing its tag to the tags of existing values in a set. If so, then this value is encoded to a smaller-width number representing its location in the FVC (i.e., set_id * associativity + location in a set) and the encoded value is sent to the CompD. If the value is not present in the FVC, then it simply replaces a Least-Recently Used (LRU) (or pseudo LRU) entry in a set and the unencoded value is sent. For both cases, a single bit indicating whether it is compressed or not is sent through a separate link. Assuming a 2^n-entry FVC ($n = 4$ has been used for our experiments), the encoded value will use $(n + 1)$-bits (including a single-bit compression indicator).

Base Decompression Scheme (Figure 8(b)). As a counterpart to the FVC on the SuppD side, a new structure named *Frequent Value Table* (FVT) is introduced on the CompD side. It has the same size and set structure as the FVC on the SuppD side, but the FVT does not perform any tag search. Instead, when data is received from the link, it determines whether this data is compressed or not by checking the single bit indicator. If the data is compressed, then it simply reads the FVT using the compressed value, which is an index to the FVT. Otherwise, if the data is not compressed, it updates the FVT by replacing a LRU (or pseudo LRU) entry in a set with the newly received value.

6.2. Extended Scheme Design and Operation

While the base design is sufficient to capture the *temporal value locality*, this alone often results in limited FVC hit rate. In order to further increase the effectiveness of the approach, we present an extended scheme that also targets *spatial value locality*.

Capturing Integer Value Spatial Locality. The key goal of this design is to utilize limited FVC spaces to cover a wide range of values. For integer values, simply discarding last k bits ($k = 6$ has been used for our experiments) of a value and caching the remaining bits in the FVC achieves this goal effectively. On a FVC hit, the discarded last k bits are concatenated to the encoded bits and sent over the link for the CompD. Then, on the CompD side, the encoded bits are used to read the FVT and later appended to the last k bits to recover a full value. While this extension changes the encoded value's bit length by k bits, it increases the hit rate of the FVC by increasing the coverage of each entry in FVC.

Capturing Floating Point Value Spatial Locality. For floating point (FP) values, the range of values are not represented by the last k bits of a value, which correspond to the mantissa portion of FP values. Instead, a range of values are more aptly represented by the sign and exponent portion, which correspond to the first 9 bits in a single-precision FP value. However, caching these 9 bits for a n-entry FVC often brings limited benefit, since an encoded value will still have n ($= \log_2 2^n$) bits. Therefore, we add an even smaller four-entry FVC dedicated to compressing the 9 bits representing the FP sign and exponent. When the compression unit receives an FP value, it extracts the first 9 bits and checks whether it hits in the four-entry FVC. In parallel, a full 32-bit FP value is sent to the original FVC to check its presence. Upon a hit in the original FVC, the compression scheme works as in the base case. When it misses in the original FVC but its sign and exponent portion hits in the four-entry FVC, its sign and exponent portion is encoded to 2 bits, while the mantissa remains uncompressed. On the CompD side, upon receiving a compressed FP value, the decompression unit uses the indicator bits to decide whether the whole FP number is compressed or only the sign and exponent part, and then it decompresses them accordingly. The extended scheme also requires a 2-bit indicator noting the type of compression. For example, 00 can indicate no compression; 01 integer compression; 10 whole FP compression; and 11 sign and exponent only compression.

7. DeSC EVALUATION: CMP

7.1. Evaluation Methodology

For cycle-level simulations of DeSC, we use a modified version of Sniper [7]. Specifically, we extended Sniper's cycle-level out-of-order processor model (instruction window-centric [8]) to support DeSC's extended ISA and proposed hardware components. Table VII summarizes the baseline simulation parameters used. Our experiments are run on 16 workloads from the Parboil [53] and Rodinia [10] suites. In each case, the compiler pass operates on the regions-of-interest as marked by the suite developers (with start-timer calls). Some benchmarks from these suites (e.g., bfs, b+tree, tpacf, MummerGPU) are so communication-bound—with insufficient value computation to overlap—that we do not address them. Without value computation to balance against, DeSC is no better than a single SuppD. The compiler passes can identify imbalanced benchmarks and only employ DeSC when promising. In addition, we excluded few computation-intensive benchmarks to avoid redundancy while keeping three as representative cases for them. In addition, three workloads were not included to the experiment due to incompatibilities with the our simulation framework.

For the 16 studied workloads, Figure 9 compares the baseline performance on a standard OoO processor to that same processor with perfect L1 cache. From this, we classify them into four categories based on memory-boundedness.

Table VII. Architectural Simulation Parameters

CPU	2.0Ghz 4-Way OoO Cores 32-entry Instr. Window / 32-entry ROB
L1 Cache	32KB / 4-way / 2ns Latency
L2 Cache	1024KB / 8-way / 10ns Latency
MSHR	16 MSHRs
DRAM	12.8GB/s Bandwidth / 60ns access latency (excl. contention)
DeSC Interface	512-item Comm. Queue / 1 cycle Push Lat. 64-entry Comm. Buffer / 1 cycle Read Lat. CommQ to CommBuf / 1 cycle Latency 128-entry SAB / SVB
DeSC Internal Traffic Compression	16-entry (32 bits each) / 4-way FVC and FVT 4-entry (9 bits each) / 4-way FVC and FVT (for FP sign and exponents)

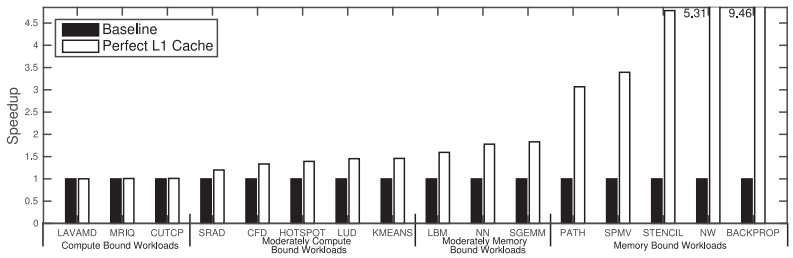

Fig. 9. Workload categorization.

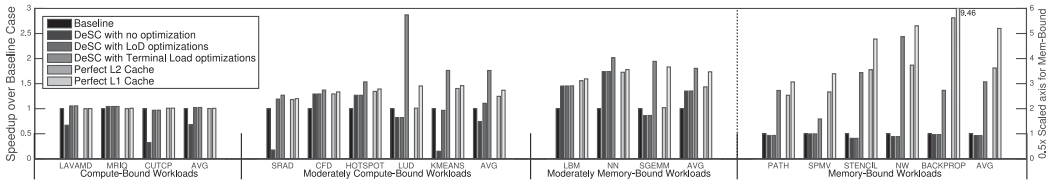

Fig. 10. Performance of DeSC system across different degrees of optimization compared against perfect L1/L2 cache case. Memory-bound workloads use right Y-axis.

Workloads that get less than 5% speedup from the perfect cache configuration are categorized as computation-bound. Workloads with 5% to 50% speedup are categorized as moderately computation-bound. Workloads that get more than 50% but less than 100% speedup are categorized as moderately memory-bound. Last, workloads with more than 100% speedup are categorized as memory-bound.

7.2. DeSC Performance Results

Figure 10 shows DeSC's speedup with varying degrees of optimizations. For each workload, the first bar represent the baseline case where single core is running a workload. The next three bars show DeSC (a baseline OoO core for SuppD/a baseline OoO core for CompD) speedups for the different optimizations. The baseline DeSC (second bar) gets little to no speedup over a single core. In particular, for workloads with LoD events, performance was much worse than baseline case. After applying specific LoD optimizations (Sections 5.1, 5.2, 5.3) for each workload with LoD event, those workloads get significant performance improvement (third bar) compared to the case before LoD

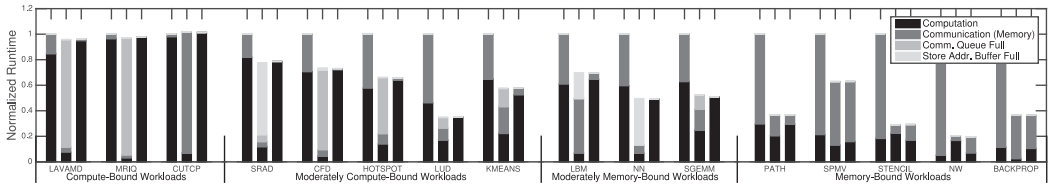

Fig. 11. DeSC runtime distribution graph (from left to right: base, SuppD, CompD).

optimizations were applied (second bar). The key to better performance is out-of-order commit for Terminal Loads (Section 3.2). As shown by the fourth bar, with the Terminal Load optimization, around 70% speedup is achieved for moderately compute-bound or memory-bound workloads and 200% speedup is achieved for memory-bound workloads. Since compute-bound workloads are not limited by memory performance, they see less speedup from DeSC.

It is natural to consider comparing DeSC against other memory latency tolerance optimizations, and with that in mind, the rightmost two bars in Figure 10 shows the performance comparison between DeSC and perfect L1 or L2 cache cases. Perfect L2 cache (fifth bar) can be interpreted as a realistic upper bound for existing work on prefetching techniques, which typically fetch to the L2. Similarly, perfect L1 cache (sixth bar) can be interpreted as an extreme upper bound for prefetching techniques. DeSC performs better than perfect L2 cache in most cases. Where perfect L2 cache performed better (e.g., stencil), DeSC is, in fact, limited by memory bandwidth, which does not affect the perfect cache. In cases such as spmv or backprop, histogramming behavior or other value dependencies mean that DAE-based prefetching would not approach these perfect L2 cache speedups due to latency on their critical path. We also note that DeSC sometimes outperforms perfect L1. This occurs where DeSC benefits from (i) offloading of address computation-parallelization; and (ii) lower CommBuf access delay compared to the L1 cache.

Figure 11 shows the distribution of runtime for either baseline or SuppD and CompD. As expected, compute-bound workloads spend most of the runtime on computation while memory bound workloads spend most of the runtime on memory accesses. For many workloads, however, DeSC removes almost all the communication time. In memory-bound workloads, where it cannot, this is either, because the system is bandwidth-bound (nw, stencil), or because memory latency is exposed, because the application limits run-ahead distance (path, spmv, backprop). Despite these limits, memory-bound workloads still show huge speedup, because DeSC eliminates large portion of the time spent on communication. Finally, in some computation-bound workloads (mriq, lavamd, cfd, hotspot), SuppD stalls frequently due to a full CommQ. In cases like this where CompD's data consumption speed is low, the SuppD could be power-gated based on CommQ occupancy.

Figure 12 compares DeSC performance against a baseline OoO core with varying degree of reorder buffer sizes. On compute-bound workloads, DeSC performs similarly to the baseline with a 32-entry ROB, because its computing ability is limited by the small CompD's ROB size (just 32 entries). For moderately compute/memory-bound workloads, DeSC performance is similar to the baseline with 4× or 8× ROB sizes. Finally, for memory-bound workloads, DeSC performs much better than even a baseline with a 256-entry ROB, because it can benefit more from its superior latency hiding capability.

Figure 13 shows DeSC speedup against a baseline OoO core with varying ROB sizes. However, for this experiment, DeSC SuppD's ROB size was fixed to 32 while CompD's ROB sizes were matched to the baseline OoO core. Speedup is pretty insensitive to ROB sizes for the first three workload categories. In these cases, a SuppD core with a

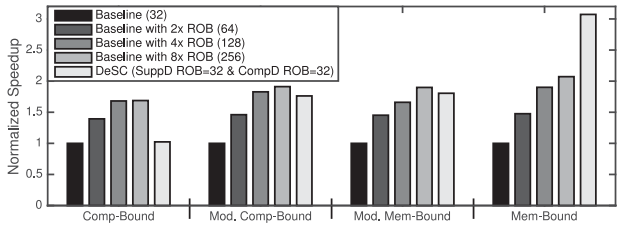

Fig. 12. Average DeSC performance (per category) compared against baseline with larger ROB.

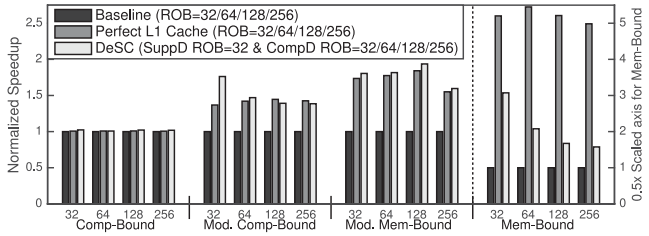

Fig. 13. Average DeSC performance (per category) across varying ROB sizes. Memory-bound workloads use right Y-axis.

32-entry ROB is enough to maintain the benefit of DeSC, achieving a performance close to a baseline with perfect L1. On the other hand, for memory-bound workloads, the average speedup decreases with increasing ROB size. This is because (i) SuppD with ROB = 32 fails to supply enough data for increased CompD capability; and (ii) external factors such as memory bandwidth (or issue width) that limits the increase in performance (resulting decrease in relative speedup). However, note that DeSC still shows 1.6×–3× speedup over a single OoO core with varying ROB sizes for memory-bound workloads.

7.3. DeSC Runahead Distance Analysis

Figure 14 shows the average occupancy of both the CommQ and the CommBuf (i.e., average number of items residing in either communication structure) at run time for each workload. This is often a good indicator for the SuppD runahead distance. As shown in the figure, most of non-memory bound workloads have occupancy close to the theoretical maximum (512 in our setup). Two exceptions are cutcp and srad, where their runahead distance is limited by LoD events and the Store Address Buffer (SAB) size. These high queue occupancies indicate that the SuppD's data supply rate exceeds the CompD's data consumption rate. On the other hand, most memory bound workloads exhibit low CommQ and CommBuf occupancy. This indicates that their data supply rate (even with DeSC Optimizations—Section 3.2, 5) is acting as the bottleneck for DeSC performance.

On the other hand, Figure 15 shows the average lifetime of supplied data. This shows the average time a data value spends inside the CommQ and the CommBuf. Usually, this metric is highly correlated with the CommQ and CommBuf occupancy shown in Figure 14. The key difference is that this metric shows the runahead distance in a time scale, rather than in the number of `PRODUCE` instructions executed in the runahead region. For example, while workloads like lud, kmeans, and sgemm exhibit high queue occupancy, they show relatively lower supplied data lifetime, because the data consumption rate on the CompD is high (i.e., less computation between `CONSUME` instructions).

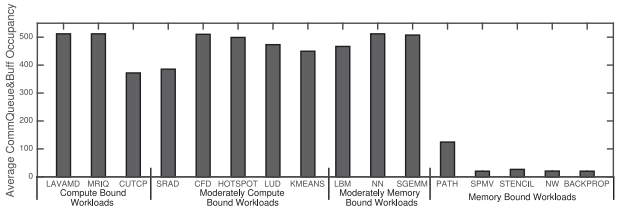

Fig. 14. Average CommQ+CommBuf Occupancy.

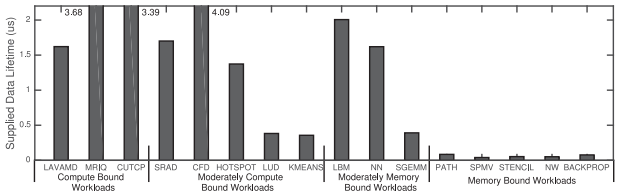

Fig. 15. Average Supplied Data Lifetime.

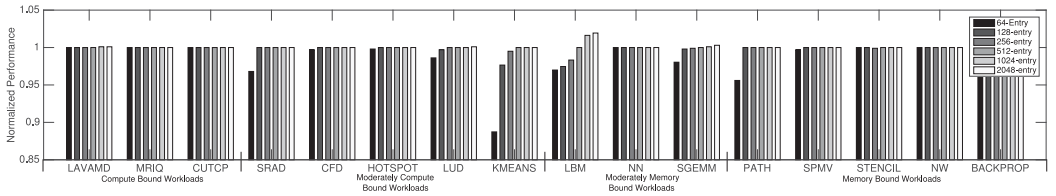

Fig. 16. DeSC Performance Sensitivity to the CommQ+CommBuf Size. Performance is normalized to the base case where CommQ+CommBuf size is 512. Note that Y-axis starts from 0.85.

Another interesting analysis is to determine how DeSC performance is dependent on CommQ size. To that end, Figure 16 shows DeSC performance with varying CommQ+CommBuf sizes. For this experiment, the CommBuf size has been fixed to 64-entry (as in Table VII) and CommQ size is varied accordingly. As shown in the figure, in 10 of 16 workloads, the performance difference across varying CommQ+CommBuf sizes were less than 1%. Even in the worst case (kmeans), the performance for 64-entry case is within 12% of the performance for 512-entry case.

Basically, the size of the CommQ+CommBuf determines the maximum runahead distance. By scaling them down, the maximum runahead distance becomes smaller than the previous baseline case. First, this does not really affect memory bound workloads, because their average runahead distance is low (as shown in Figure 14). Since they do not achieve maximum runahead distance anyway, reducing or increasing communication queues size does not really affect DeSC performance. On the other hand, most of the compute-bound workloads operate at near-maximum runahead distance, because their data production rate exceeds their data consumption rate. In such cases, their average runahead distance in time (as shown in Figure 15) is limited by CommQ+CommBuf size (i.e., maximum runahead distance). For example, when CommQ+CommBuf size becomes one-eighth (64) of the base case (512), lavamd's average runahead distance in time (Figure 15) changes from 1.6us to 200ns. Since this runahead distance is still large enough to avoid load latency being exposed to the CompD, lavamd's performance is not affected. For the same reason, the performance of most of the compute-bound applications is not significantly affected by a reduced CommQ+CommBuf size. Still, there are some exceptional cases, because the runahead distance changes dynamically during execution time. For example, if the data

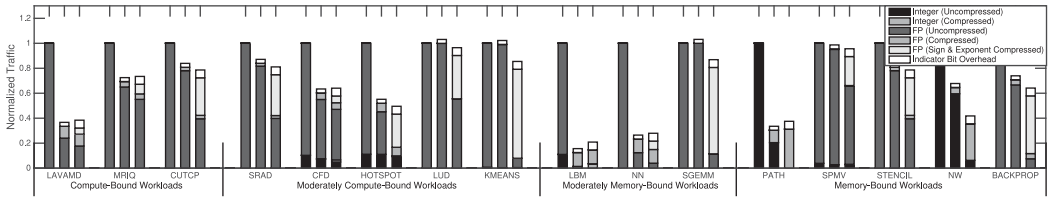

Fig. 17. DeSC Traffic w/ Compression (left to right: no compression, base compression, extended compress.).

consumption rate temporarily exceeds the production rate on a compute-bound workload (e.g., a long-latency load is found), having a large maximum runahead (i.e., having more data ready in the CommQ/CommBuf) can avoid data supply disruptions being exposed to the CompD. For this reason, a workload (e.g., lbm) performance can be sensitive to CommQ/CommBuf size even though its *average* runahead distance is high.

7.4. DeSC Internal Traffic Compression Analysis

Figure 17 shows how DeSC SuppD-CompD traffic changes with the compression scheme presented in Section 6. For each benchmark, three stacked bars represent: (i) traffic w/o compression; (ii) traffic with the base compression scheme; (iii) traffic with the extended compression scheme. Overall, both base compression and extended compression schemes are effective in reducing the traffic between the SuppD and the CompD. On average, the base scheme reduces 35% of the original traffic and the extended scheme reduces 38% of the traffic. However, the schemes' effectiveness varies greatly with the workload. For example, compression schemes are particularly effective for lbm, nn, and lavamd, reducing traffic by nearly a factor of 3× compared to the original traffic. The extended compression scheme works better than base compression scheme in 10 of 16 workloads. The extended compression scheme shows it potential for exploiting *value spatial locality* but some workloads lack such locality (e.g., lavamd, mriq); or simply the base scheme works sufficiently well just with *value temporal locality* (e.g., path). In such cases, either the extra bit overhead of the extended scheme or the limited maximum integer compression rate make it perform slightly worse than the base compression scheme.

The next question is whether the use of a compression scheme, which incurs in an extra latency, can hurt DeSC performance. Our experiments show that compression has no impact on DeSC performance across all workloads, even with arbitrarily large compression/decompression latencies (e.g., 512 cycles—due to space limits, we do not include graphs showing the negligible performance loss). The reason is the latency tolerance inherent in DeSC: neither compression or decompression (in addition to CommQ-to-CommBuf latency) is on the critical path for the SuppD or the CompD. Thus, it does not affect the data supply/consumption rate of the SuppD/CompD. The increased communication time between the SuppD and the CompD is *pipelined* and, therefore, it is equivalent to the CompD starting a few cycles later than the SuppD. Thanks to the inherent *latency tolerance* in DeSC SuppD-CompD communication, traffic compression schemes can be applied at no performance overhead. In contrast, in prior works compression and decompression happen on a likely-critical path (e.g., between processor and memory [3, 47, 56, 59]).

Finally, Figure 18 shows the effect of FVC/FVT size on DeSC internal traffic. Changing the size of the FVC/FVT has two contrasting effects on internal traffic compression rate. First, increasing the FVC/FVT size naturally increases hit rate for FVC/FVT. As a result, more data ends up being compressed and thus it reduces the total amount of traffic. On the other hand, increasing the FVC/FVT size makes encoded/compressed

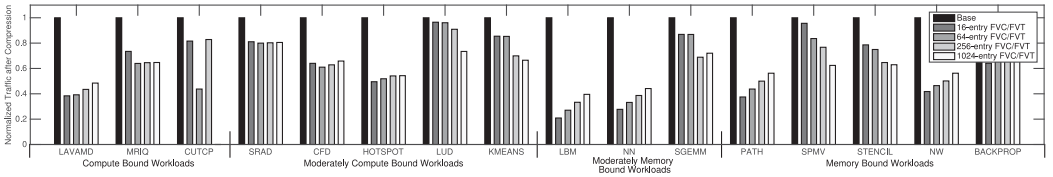

Fig. 18. DeSC Internal Traffic Compression Scheme Sensitivity to the FVC/FVT size.

results wider. For example, when a 256-entry FVC/FVT is used, a compressed outcome has 8-bit width. Analogously, for a 16-entry FVC/FVT the compressed outcome has 4-bit width, achieving 2× better compression rate. Because of these two contrasting effects, each workload's traffic shows different trends with varying FVC/FVT size. While larger FVC/FVT sizes have their own merits and drawbacks, we concluded that a 16-entry FVC/FVT perform reasonably well across most workloads at a significantly lower cost than larger alternative, especially with the extended compression scheme (Section 6.2), which exploits *spatial value locality*.

8. DeSC EVALUATION: ACCELERATORS

DeSC on CPUs clearly offers significant performance benefits. This section explores DeSC's performance potential for CompDs implemented as specialized hardware accelerators.

8.1. Methodology

Detailed modeling of hardware accelerator behavior is very challenging, and most existing accelerator synthesizers or modelers (e.g., see Reference [48]) do not directly connect to the CPU simulator we require for the SuppD side. To explore the design space, we *approximate* the behavior of a hardware accelerator by deeply modifying Sniper.

The core idea is to mimic the behavior of nearly-perfect operation scheduling that would occur in a non-instruction accelerator, with performance primarily limited by true data dependencies. In essence, this is approximated by using an OoO simulator with as few resource constraints as possible. Thus for the performance of our kernels to be run as if on an accelerator, we make the issue/dispatch/commit width of the processor very large (e.g., 256), and we likewise make the ROB unrealistically large as well (e.g., 16K). We assume perfect ICache and Branch Predictor behavior, and we change the instruction latency to match the assumed computation latency for an accelerator (e.g., 1ns @ 1Ghz as in Aladdin). We unroll important loops by a certain factor to allow considerable parallelism within the kernel. Finally, to enclose the "accelerated instructions" as if in specialized hardware, we simulate a `Sync` instruction just before and after the accelerated kernel, which forces them to complete without any overlap with preceding or following instructions.

While this model for accelerated kernels is approximate, it has sufficient fidelity to support our goal of broad exploration of SuppD and CompD tradeoffs for accelerator-style usage. In order to validate our approach, we compared our results against the state-of-the art pre-RTL hardware accelerator simulator Aladdin.

Figure 19 shows these validation results for four SHOC kernels [18] that ship with Aladdin. Across different loop unrolling factors, the validation shows that while our Sniper approach is not highly accurate (30% on average), the accelerator execution times are within sufficient accuracy for our design goals here.

Note that our experiments are not entirely dependent on an absolute accuracy of the model, because we compare two cases (an accelerator with a cache hierarchy *vs* a DeSC

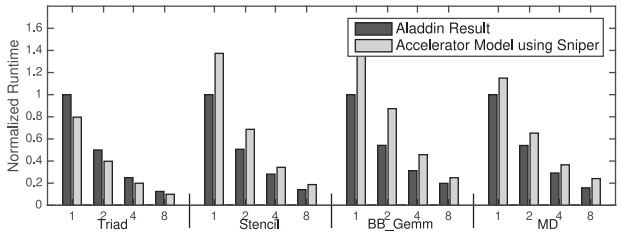

Fig. 19. Accelerator model validation.

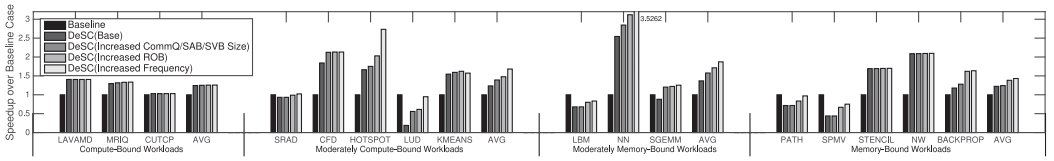

Fig. 20. Speedup of a accelerator CompD with varying SuppD designs over an accelerator with cache.

system consisting of a CompD accelerator and a SuppD) where both sides use the same model.

8.2. DeSC with Hardware Accelerator CompD

Figure 20 explores the case where the CompD is a hardware accelerator. Here, the baseline design assumes an accelerator having its own cache hierarchy (identical to the one assumed in the OoO core). DeSC bars assume a computing accelerator having no memory hierarchy or access (as previously described), but rather a CommQ and CommBuf with a SuppD supplying data for it. The second bar assumes a baseline DeSC configuration where the SuppD is a small OoO core whose parameters are as in Table VII. Remaining bars give the SuppD more resources. On average, a baseline DeSC performs better than cache-based accelerators in more than 60% of the evaluated workloads. In the end, DeSC provides more than 50% average speedup for three workload categories and provides around 30% for computation-bound workloads. One interesting thing to note is that even originally computation-bound workloads can noticeably benefit from communication optimization with hardware accelerators. This is because a computation hardware accelerator, as expected, accelerates computation greatly while leaving communication mostly intact.

There are few exceptional cases where DeSC performs worse than cache-based accelerators such as spmv. Hardware accelerators integrated with the cache can issue all independent loads between synchronization points while OoO based SuppD can only detect independent loads within the instruction window. As SPMV has non-terminal loads, SuppD often had limited effective instruction window, because non-terminal load frequently blocked the head of ROB. On the other hand, hardware accelerators were able to utilize higher levels of parallelism for loads in that case. However, as mentioned, this is not a general case. Usually, DeSC provides more performance benefit, mainly because it utilizes the SuppD communication queue to dynamically manage the communication in a fine granularity rather than relying on static coarse grained communication synchronization planned by hardware designers.

9. RELATED WORK

Decoupled Access Execute Architecture: DAE architectures attack memory latency by decoupling a program's *access* and *execute* streams and letting them run

largely independently while communicating data through architectural queues [39, 50, 51]. Subsequent DAE work explored implementation details [23], analyzed LoD events [4], analyzed communication/computation balance [30], provided compiler frameworks [57], and extended to vector processors [20]. More recent work utilizes DAE for various purposes such as optimizing indirect loads [16], efficient DVFS [29, 34], and energy-efficient graphics pipelines [1]. In all these articles, DAE aimed to hide memory latency and was often viewed as a potentially simpler alternative to superscalar processors. Our work offers an updated DAE perspective aimed not just at latency tolerance, but also as a solution to the data-supply problem for heterogeneous or accelerator-based processors. In addition, DeSC unifies and exploits both out-of-order and DAE techniques.

Decoupled Architecture for Accelerators: Recent works have also explored the possibility of applying decoupled architecture for accelerator-based systems [11, 26]. These works suggests an idea of utilizing specialized hardware accelerator for energy-efficient data supplier. On the other hand, our work presents a lightweight extension to an existing OoO core that enables OoO core to work as a high-performance data supplier.

Helper Thread & Runahead for Prefetching: In addition to the split streams used in DAE, other work has envisioned helper threads either constructed by hand [13], compiler [32, 38], dynamic compilation [37, 60], or hardware [12], which run in parallel with a main thread. Helper threads speculatively prefetch some of the data that the main thread may (or may not) use, to reduce memory latencies seen by the main thread. Similarly, Runahead execution [40], Performance-correctness explicitly decoupled architecture [22] and Dual core execution [61] utilizes idle/extra hardware resources to prefetch useful data before main thread needs it. While DeSC and helper threads/runahead share latency reduction and tolerance as a goal, they operate speculatively and heuristically as prefetchers. In contrast, DeSC offers a true data-supply solution that obviates the need for memory connections from the CompD.

Automatic Parallelization Techniques: DeSC relates to some automatic parallelization research, most notably DSWP [44, 45]. DSWP parallelizes the program to increase its memory latency tolerance utilizing a hardware-aided inter-thread communication mechanism called a synchronization array. A core difference between DSWP and DeSC is that DSWP still targets a system where all cores have access to the memory system while we assume only some cores able to access it, which allows us to encompass loosely-coupled accelerators.

Out-of-order Commit for latency tolerance: Continual flow pipeline [52], Kilo-instruction processor [17], a Flexible heterogeneous multicore architecture [42] and simultaneous speculative threading [9] tries to avoid ROB-blocking on long-latency loads by allowing out-of-order commits or offloading for loads and its dependent instructions. While the exact implementation varies, most utilize relatively high-cost mis-speculation recovery mechanisms such as checkpointing. In DeSC, we get the benefits of OoO commit of terminal loads, but our hardware is much simpler, because terminal loads have no SuppD dependents and because no speculation recovery is needed.

Automated Accelerator Design: In part thanks to high-level synthesis tools [5, 6, 21, 36, 41], accelerator-centric design is easier and more widely-used than ever before. However, the burden of communication management for accelerators still primarily lies on programmers or library writers. DeSC enables portable, low-effort, high-performance data supply approaches.

Communication Management: Other related research has studied automating and optimizing data communication between CPU and GPU [27, 28] or in distributed memory systems [2, 49]. While similar in motivation to DeSC, they study distinct scenarios, such as larger memories or looser compute-memory couplings.

Link Compression: Link compression has been one of the popular, traditional topics in computer architecture. Most of the previous works [3, 47, 56, 59] explore how data traffic can be compressed to reduce processor (or GPU)—memory link bandwidth. On the other hand, our work studies how a scheme derived from such works can be applied for a different type of traffic (i.e., DeSC SuppD-CompD) communication without any performance degradation.

10. CONCLUSIONS

This article envisions and evaluates the DeSC framework. In its decoupling of memory and compute, DeSC is inspired by DAE, but has been updated and expanded for modern, heterogeneous processors. With modest hardware and compiler support, DeSC offers significant speedups both for general-purpose and homogeneous scenarios as well as for more specialized or accelerator-centric cases. DeSC improves traditional DAE approaches by (i) optimizing terminal loads on the supply device, (ii) offering hardware and software extensions to avoid loss-of-decoupling events, and (iii) using traffic compression schemes to reduce bandwidth consumption between the SuppD and the CompD. These optimizations allow DeSC's SuppD to run ahead of the CompD and avoid exposing load latencies to the computation device. By doing so, DeSC offers average speedups of 2.04× on CMP and 1.56× on accelerators across the evaluated workloads. Furthermore, the use of a compression scheme reduces DeSC SuppD-CompD traffic by around 40%, thereby allowing DeSC to be deployed for more bandwidth-constrained scenarios. DeSC strikes an important balance in terms of "specialized generality"; DeSC has enough specialization to offer significant performance advantages, while still being general enough to port well across different implementations.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. This work was supported in part by the NSF under the grant CCF-1117147. This work was also supported in part by the Spanish State Research Agency under grants TIN2015-66972-C5-3-R and TIN2016-75344-R (AEI/FEDER, EU).

REFERENCES

[1] José-María Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. 2012. Boosting mobile GPU performance with a decoupled access/execute fragment processor. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*.

[2] Ayon Basumallik and Rudolf Eigenmann. 2006. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*.

[3] L. Benini, D. Bruni, B. Ricco, A. Macii, and E. Macii. 2002. An adaptive data compression scheme for memory traffic minimization in processor-based systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, Vol. 4.

[4] Peter Bird, Alasdair Rawsthorne, and Nigel Topham. 1993. The effectiveness of decoupling. In *Proceedings of the 7th International Conference on Supercomputing (ICS'93)*.

[5] Cadence 2011. *C-to-Silicon Compiler High-Level Synthesis*. Technical Report. Cadence. Retrieved from http://www.cadence.com/rl/Resources/datasheets/C2Silicon_ds.pdf.

[6] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'11)*.

[7] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of the of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. Article 52.

[8] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An evaluation of high-level mechanistic core models. *ACM Trans. Arch. Code Optim.*, Article 5 (2014).

[9] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. 2009. Simultaneous speculative threading: A novel pipeline architecture implemented in sun's rock processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*.

[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*.

[11] T. Chen and G. E. Suh. 2016. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *Proceedings of the 49th International Symposium on Microarchitecture (MICRO'16)*.

[12] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. 2001. Dynamic speculative precomputation. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'01)*.

[13] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. 2001. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*.

[14] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. 2012. Architecture support for accelerator-rich CMPs. In *Proceedings of the 49th Annual Design Automation Conference*.

[15] Emilio Cota, Giuseppe Di Guglielmo, Paolo Mantovani, and Luca Carloni. 2015. An analysis of accelerator coupling in heterogeneous architectures. In *Proceedings of the 52nd Design Automation Conference (DAC'15)*.

[16] Neal Clayton Crago and Sanjay Jeram Patel. 2011. OUTRIDER: Efficient memory latency tolerance with decoupled strands. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*.

[17] Adrián Cristal, Oliverio J. Santana, Mateo Valero, and José F. Martínez. 2004. Toward kilo-instruction processors. *ACM Trans. Arch. Code Optim.* 1, 4 (Dec. 2004).

[18] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU'10)*.

[19] Assia Djabelkhir and Andre Seznec. 2003. Characterization of embedded applications for decoupled processor architecture. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC'03)*.

[20] R. Espasa and M. Valero. 1996. Decoupled vector architectures. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA'96)*.

[21] Tom Feist. 2012. *Vivado Design Suite,*. Technical Report. Xilinx. Retrieved from http://www.xilinx.com/support/documentation/white_papers/wp416-Vivado-Design-Suite.pdf.

[22] Alok Garg and Michael C. Huang. 2008. A performance-correctness explicitly-decoupled architecture. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*.

[23] J. R. Goodman, Jian-tu Hsieh, Koujuch Liou, Andrew R. Pleszkun, P. B. Schechter, and Honesty C. Young. 1985. PIPE: A VLSI decoupled architecture. In *Proceedings of the 12th Annual International Symposium on Computer Architecture (ISCA'85)*.

[24] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*.

[25] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15)*.

[26] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. 2015. Efficient execution of memory access phases using dataflow specialization. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA'15)*.

[27] Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu, and David I. August. 2012. Dynamically managed data for CPU-GPU architectures. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12)*.

[28] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. 2011. Automatic CPU-GPU communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*.

[29] Alexandra Jimborean, Konstantinos Koukos, Vasileios Spiliopoulos, David Black-Schaffer, and Stefanos Kaxiras. 2014. Fix the code. Don'T tweak the hardware: A new compiler approach to voltage-frequency scaling. In *Proceedings of the of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'14)*. Article 262.

[30] L. K. John, V. Reddy, P. T. Hulina, and L. D. Coraor. 1995. Program balance and its impact on high performance RISC architectures. In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture (HPCA'95)*.

[31] Melanie Kambadur, Kui Tang, and Martha A. Kim. 2012. Harmony: Collection and analysis of parallel block vectors. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*.

[32] Dongkeun Kim and Donald Yeung. 2002. Design and evaluation of compiler algorithms for pre-execution. *SIGPLAN Not.* 37, 10 (Oct. 2002).

[33] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. 2005. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro.* 25, 2 (March 2005).

[34] Konstantinos Koukos, David Black-Schaffer, Vasileios Spiliopoulos, and Stefanos Kaxiras. 2013. Towards more efficient execution: A decoupled access-execute approach. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS'13)*.

[35] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. 2005. Heterogeneous chip multiprocessors. *Computer* 38, 11 (Nov. 2005).

[36] Feng Liu, Soumyadeep Ghosh, Nick P. Johnson, and David I. August. 2014. CGPA: Coarse-grained pipelined accelerators. In *Proceedings of the 51st Annual Design Automation Conference (DAC'14)*. Article 78.

[37] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G. Abraham. 2005. Dynamic helper threaded prefetching on the sun ultraSPARC CMP processor. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*.

[38] Chi-Keung Luk. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*.

[39] W. Mangione-Smith, S. G. Abraham, and E. S. Davidson. 1990. The effects of memory latency and fine-grain parallelism on Astronautics ZS-1 performance. In *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences, 1990*, Vol. i.

[40] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*.

[41] Anisha Nanda. 2012. *Accelerate Performance and Design Productivity with OpenCL on Altera FPGAs*. Technical Report. Altera. Retrieved from http://wl.altera.com/education/webcasts/all/sourcefiles/wc-2012-opencl/player.html.

[42] Miquel Pericas, Adrian Cristal, Francisco J. Cazorla, Ruben Gonzalez, Daniel A. Jimenez, and Mateo Valero. 2007. A flexible heterogeneous multi-core architecture. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*.

[43] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*.

[44] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. 2008. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'08)*.

[45] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. 2004. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*.

[46] Justin Rattner. 2002. Making the Right Hand Turn to Power Efficient Computing. (2002). Retrieved from http://www.microarch.org/micro35/keynote/JRattner.pdf.

[47] Vijay Sathish, Michael J. Schulte, and Nam Sung Kim. 2012. Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*.

[48] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA'14)*.

[49] Shamik D. Sharma, Ravi Ponnusamy, Bongki Moon, Yuan Shin Hwang, Raja Das, and Joel Saltz. 1994. Run-time and compile-time support for adaptive irregular problems. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'94)*.

[50] James Smith. 1984. Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst.* 2, 4 (Nov. 1984).

[51] J. E. Smith, Shlomo Weiss, and N. Y. Pang. 1986. A simulation study of decoupled architecture computers. *IEEE Trans. Comput.* C-35, 8 (Aug 1986).

[52] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. 2004. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*.

[53] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and W-m Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report IMPACT-12-01. University of Illinois at Urbana-Champaign.

[54] Xian-He Sun and Yong Chen. 2010. Reevaluating Amdahl's law in the multicore era. *J. Parallel Distrib. Comput.* 70, 2 (Feb. 2010).

[55] Texas Instruments 2011. *OMAP4 Mobile Applications Platform*. Technical Report. Texas Instruments. Retrieved from http://www.ti.com/lit/ml/swpt034b/swpt034b.pdf.

[56] Martin Thuresson, Lawrence Spracklen, and Per Stenstrom. 2008. Memory-link compression schemes: A value locality perspective. *IEEE Trans. Comput.* 57, 7 (July 2008).

[57] Nigel Topham, Alasdair Rawsthorne, Callum McLean, Muriel Mewissen, and Peter Bird. 1995. Compiling and optimizing for decoupled architectures. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'95)*. Article 40.

[58] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE'81)*.

[59] Jun Yang, Rajiv Gupta, and Chuanjun Zhang. 2004. Frequent value encoding for low power data buses. *ACM Trans. Des. Autom. Electron. Syst.* 9, 3 (July 2004).

[60] Weifeng Zhang, Dean M. Tullsen, and Brad Calder. 2007. Accelerating and adapting precomputation threads for effcient prefetching. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*.

[61] Huiyang Zhou. 2005. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*.

Received February 2017; accepted March 2017