DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures

Tae Jun Ham
Princeton University
tae@princeton.edu

Juan L. Aragón
University of Murcia
jlaragon@um.es

Margaret Martonosi
Princeton University
mrm@princeton.edu

ABSTRACT

Today's computers employ significant heterogeneity to meet performance targets at manageable power. In adopting increased compute specialization, however, the relative amount of time spent on memory or communication latency has increased. System and software optimizations for memory and communication often come at the costs of increased complexity and reduced portability. We propose Decoupled Supply-Compute (DeSC) as a way to attack memory bottlenecks automatically, while maintaining good portability and low complexity. Drawing from Decoupled Access Execute (DAE) approaches, our work updates and expands on these techniques with increased specialization and automatic compiler support. Across the evaluated workloads, DeSC offers an average of 2.04x speedup over baseline (on homogeneous CMPs) and 1.56x speedup when a DeSC data supplier feeds data to a hardware accelerator. Achieving performance very close to what a perfect cache hierarchy would offer, DeSC offers the performance gains of specialized communication acceleration while maintaining useful generality across platforms.

Categories and Subject Descriptors

C.0 [**General**]: Hardware/software interfaces;
C.1.3 [**Processor Architecture**]: Other Architecture Styles – Heterogeneous (hybrid) systems

Keywords

Accelerators, Communication Management, Decoupled Architecture, DeSC

1. INTRODUCTION

Challenges in Moore's Law performance scaling have spurred wide-spread adoption of on-chip parallelism [29, 43] and increasing amounts of specialization and heterogeneity [12, 31]. Future systems from mobile to exascale will employ ecosystems mixing general purpose cores, specialized cores, and accelerators [40, 51].

While heterogeneous and specialized parallelism shows great leverage improving computation performance at manageable power, its effective use raises additional challenges. First, the long-troubling "memory wall" becomes even more challenging in many accelerator centric designs. 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 [27, 50].

A second challenge in accelerator-oriented design is that the software-managed communication tailoring used to reduce communication cost often increases software complexity and reduces performance predictability and portability. For example, for a loosely-coupled accelerator [12, 13] 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 accelerator design to adjust such storage's capacity or port count can require code to be rewritten or reoptimized.

While using cache memories instead of scratchpads can mitigate some concerns about programmer effort and software portability, many issues remain. For example, caches still require programmer effort to balance computation and communication. Furthermore, since caches expose variable communication latency to the accelerator, this can force 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.

Attacking these 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 [46], DeSC employs compiler techniques to automatically separate data access and address calculations from value computations. Once separated, the code is targeted at different hardwares which can either be general-purpose cores or hardware 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

MICRO-48, December 05-09, 2015, Waikiki, HI, USA

DOI: http://dx.doi.org/10.1145/2830772.2830800

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 SuppD, performs value computations, and where needed, pushes output back to the SuppD to be stored in memory.

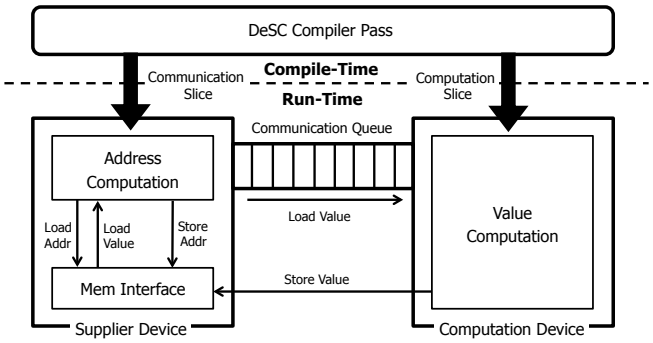

Figure 1: DeSC overview.

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

- We propose a *Decoupled Supply-Compute* (DeSC) framework which 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 (Sections 3, 4).
- We introduce novel architectural support and compiler optimizations to avoid Loss of Decoupling events (Section 5).
- We evaluate the optimized DeSC approach with modern acceleratable workloads and explore specialization opportunities (Section 6).
- We use DeSC to significantly improve on-chip accelerator performance compared to a baseline accelerator with its own cache (Section 7).

2. MOTIVATION

2.1 Decoupled Access/Execution (DAE)

In an early attempt to overcome memory wall issues while retaining implementation simplicity, Smith proposed the Decoupled Access/Execute Architecture (DAE) [46]. 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 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.

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)`
		`}`

Table 1: Decoupled code example.

Table 1 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 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, 3, 14, 25, 46], 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 [22]), and that the SuppD is specialized for data supply.

2.2 Challenges in Out-of-Order DAE

Figure 2 motivates our work by showing execution timelines for the same operations 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 in order to run programs faster and use hardware more efficiently. In-order DAE architectures (Fig. 2b) were originally envisioned as a lower-complexity latency tolerance alternative to out-of-order processors (Fig. 2a). However, they are not exclusive approaches. In fact, Fig. 2c shows them as complementary techniques.

One limitation with both Figs. 2b and c is that all communications need to happen in order. The original DAE idea [46] —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 (Fig. 2c) 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

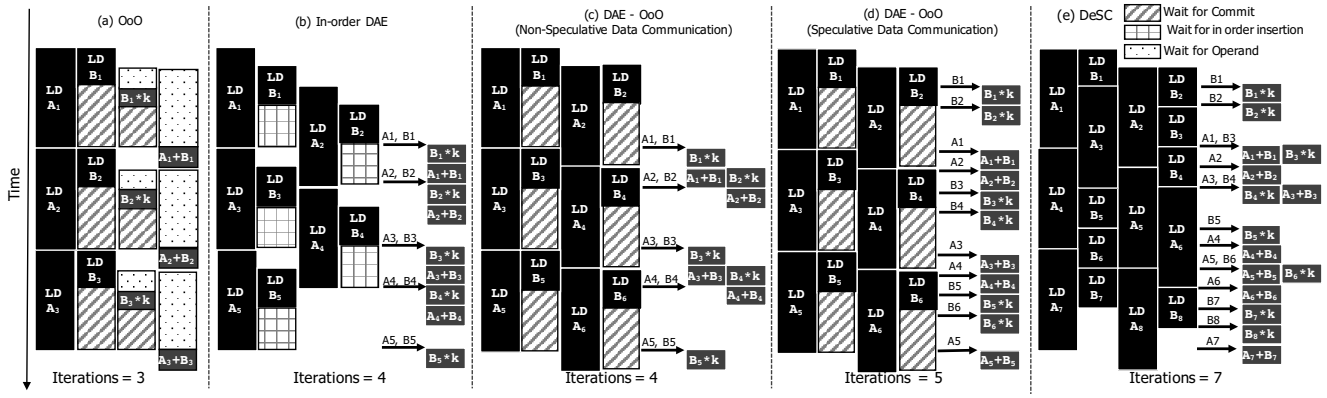

Figure 2: Motivational example for DeSC. For the code in Table 1, the dynamic instruction schedules are shown for several approaches. Some instructions (address computation, store) were omitted for clarity. Arrows represent a data communication between both sides on decoupled cases. ROB size of 4 and 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).

earlier loads are done.

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 Fig. 2d. This speculative approach can communicate data earlier at the expense of mis-speculated data being propagated, making it necessary to flush communication queues on all mispredicted branches. DeSC overcomes this limitation by supporting out-of-order communication without complex speculation recovery mechanisms.

Even with out-of-order communication, Fig. 2d's design still fails to achieve significant performance improvement. This is because its instruction commit happens in-order. For example, in Fig. 2d, 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 original program order. This requirement of non-speculative, in-order communication or commit and resulting resource constraints form a bottleneck. DeSC (Fig. 2e) overcomes this second problem by allowing out-of-order commit for certain loads with architectural/-compiler support (Section 3.2).

Summarizing, DeSC (i) communicates data when ready (out-of-order) without the need for mis-speculation recovery; and (ii) utilizes the SuppD's OoO pipeline more efficiently by allowing OoO commits. With these features, DeSC offers significantly more latency-hiding benefits than prior DAEs, and reduces key resource bottlenecks.

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 [3, 17, 52]) that limit DAE performance. LoD events occur when the data or control of the AP depends on the EP, which limits AP runahead 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 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]`

Table 2: Code examples incurring LoD events.

(a) Data Aliasing occurs when data is computed, stored to memory and then later re-loaded 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, the AP should stall until the value is computed by the EP and passed to the AP. DeSC instead uses a novel hardware optimization technique that enables store-to-load forwarding within a decoupled scenario (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, making the AP stall. 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 stalls until it receives the address from the EP. Section 5.3 describes our software approach to mitigate this.

3. DeSC HARDWARE ORGANIZATION

Decoupled Supply-Compute (DeSC) communication management consists of two specialized hardware units: a *supplier device* (SuppD) and a *computation device* (CompD), along with a hardware-software interface for their interactions, and compiler techniques for targeting them. Where DAE primarily envisioned two instruction-

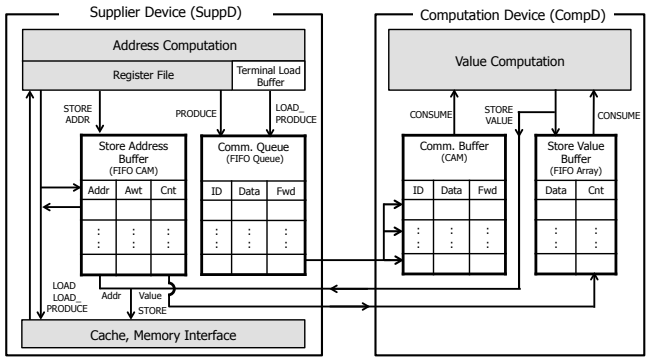

Figure 3: Hardware implementation of DeSC.

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.

Fig. 3 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, CompD is tailored to its role by removing memory hierarchy access; the SuppD supplies it with data as needed.

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 Buffer for updating the memory hierarchy when a computed value is returned. Finally, Table 3 lists the added instructions on either side to support DeSC.

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

Table 3: ISA extensions for 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, 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), 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). 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 (Fig. 2e).

Terminal loads are defined as loads in which the fetched value is going to be used only on the CompD side for computational purposes. Thus, they have no dependent instructions on the SuppD core. In traditional processors, a load instruction will 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 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 (with the same aim as many past approaches [37, 48, 15, 8]), 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.

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)`
		`}`

Table 4: Code example for terminal loads.

Table 4 shows a code example for terminal loads. In this example, `LOAD(&a[i])` is not a terminal load 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 CompD). Thus, the compiler transforms the latter load into a `LOAD_PRODUCE` instruction on the SuppD side.

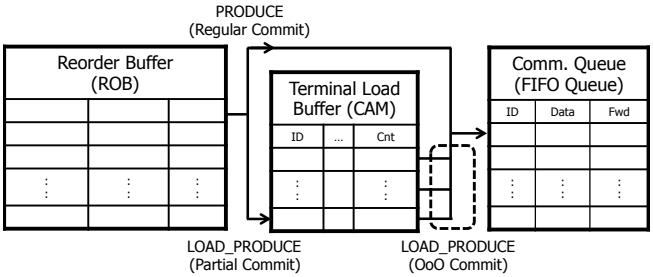

Figure 4: Out-of-order commit for terminal loads.

Figure 4 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.

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 shown in Sec. 6.

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 (Fig. 3). 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 Fig. 3) 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) 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 which 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. In order to provide precise exceptions, if any instruction in the SuppD ROB causes a fault, all ongoing loads in the Terminal Load Buffer must first complete and retire (since they are older) before the fault can be serviced. On the other hand, in case of a faulting *terminal load* (we assume the only possible faults at this point are due to late rechecks of virtual memory translations, since most other cases would have arisen before reaching the Terminal Load Buffer), it can be serviced in appearance order with no additional implications.

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, it might allow N or more younger terminal loads (or produce) to pass an older one. If those younger N fill up the CommBuf, 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 Fig. 4). 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, 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` 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, the ISA additions could include additional ordering enforcement constructs.

4. DeSC COMPILER SUPPORT

The DeSC compiler (based on LLVM [32]) splits the given source code into a communication slice and a computation slice, which target the SuppD and CompD respectively. By working on the LLVM intermediate representation (IR), the DeSC compiler can handle source codes written in any language with an LLVM front-end.

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

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

Table 5: Input for the slice construction algorithm.

1. Slicing: Using a fairly standard compiler slicing approach [53], 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 5. If it encounters a disallowed instruction during the process, 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 5) 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 (Fig. 2a), this section presents a novel hardware optimization that enables a decoupled store-to-load forwarding mechanism. As briefly stated in Section 2.3, the AP of the traditional DAE 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, for "terminal loads" (i.e., `LOAD_PRODUCE`) whose only purpose is to insert data into the CommQ, there is no reason to stall the SuppD. Furthermore, the consumer of the `LOAD_PRODUCE` (on 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 `LOAD_PRODUCE` on SuppD, we let this dependent `LOAD_PRODUCE` 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 (Fig. 3) 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, 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, 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 Fig. 3) 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

Original (SuppD)	Original (CompD)	Transformed (SuppD)	Transformed (CompD)

STORE_ADDR(Addr)
val = LOAD(Addr)
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_CHK(&k)
		STORE_ADDR(&k)
			val = a[i]-b[i]
			k = **CONSUME**()
			if (val > 0.1) {
			  k++
			  STORE_VAL(k)
			}
			else
			  STORE_INV()

Table 6: Conditional branch optimization example.

As stated in Section 2.3, a conditional branch depending on computation causes the SuppD to stall until the computed value returns (Fig. 2b). 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 6 shows the example case where this technique is beneficial. In the original 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_CHK` 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 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_CHK`, 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`).

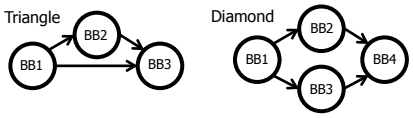

Figure 5: Conditional branch optimization target.

Currently, our compile framework performs this optimization for triangle and diamond patterns, as shown in Fig. 5 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, 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 reduces the temporal locality of a computed address can reduce the effect of this LoD. The most representative example is *Loop Distribution* [52]. If a data address is computed and used in the same loop, 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.

6. DeSC EVALUATION: CMP

6.1 Evaluation Methodology

For cycle-level simulations of DeSC, we use a modified version of Sniper [6]. Specifically, we extended Sniper's cycle-level out-of-order processor model (instruction window-centric [7]) to support DeSC's extended ISA and proposed hardware components. Table 7 summarizes the baseline simulation parameters used.

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
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

Table 7: Architectural simulation parameters.

Our experiments are run on 16 workloads from the Parboil [49] and Rodinia [9] 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.

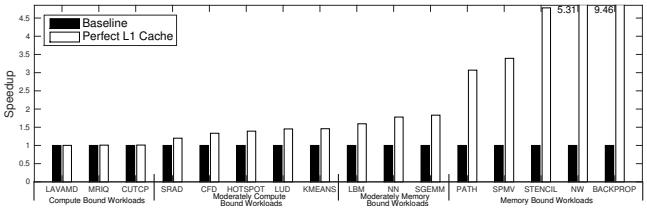

Figure 6: Workload categorization.

For the 16 studied workloads, Figure 6 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. Workloads which 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 which get more than 50% but less than 100% speedup are categorized as moderately memory-bound. Lastly, workloads with more than 100% speedup are categorized as memory-bound.

6.2 DeSC on homogeneous CMP

Figure 7 shows DeSC's speedup with varying degrees of optimizations. For each workload, the 1^{st} 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 (2^{nd} 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 (Section 5.1, 5.2, 5.3) for each workload with LoD event, those workloads get significant performance improvement (3^{rd} bar) compared to the case before LoD optimizations were applied (2^{nd} bar). The key to better performance is out-of-order commit for Terminal Loads (Section 3.2). As shown by the 4^{th} 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 7 shows the performance comparison between DeSC and perfect L1

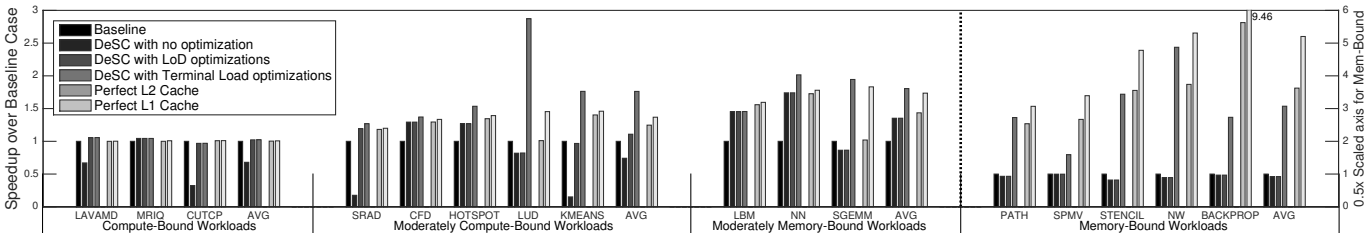

Figure 7: Performance of DeSC system across different degrees of optimization compared against perfect L1/L2 cache case. Memory-bound workloads use right Y-axis.

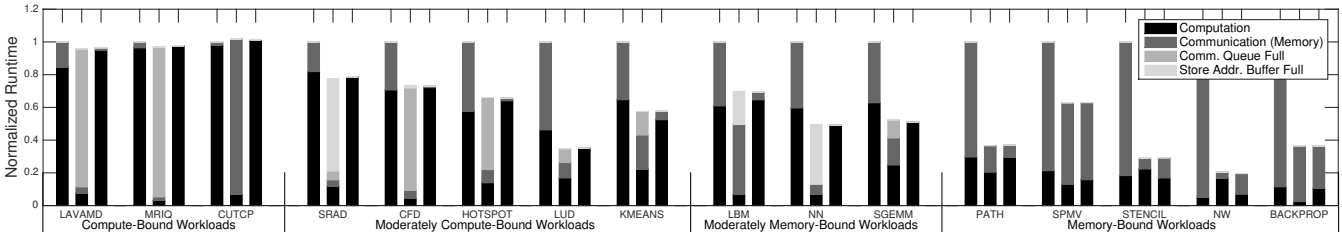

Figure 8: DeSC runtime distribution graph (from left to right: base, SuppD, CompD).

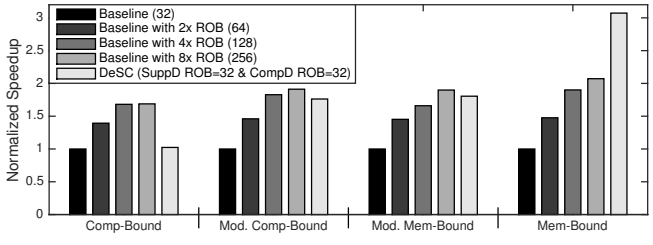

Figure 9: Average DeSC performance (per category) compared against baseline with larger ROB.

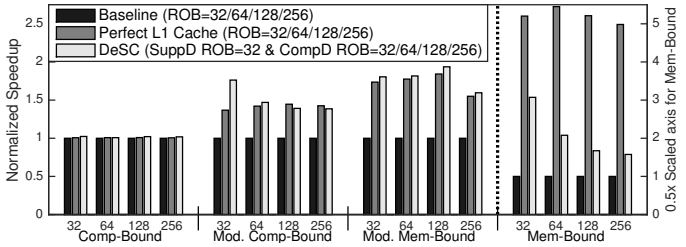

Figure 10: Average DeSC performance (per category) across varying ROB sizes.Memory-bound workloads use right Y-axis.

or L2 cache cases. Perfect L2 cache (5^{th} 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 (6^{th} 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 communication buffer access delay compared to the L1 cache.

Figure 8 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 9 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 4x or 8x 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 10 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 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;

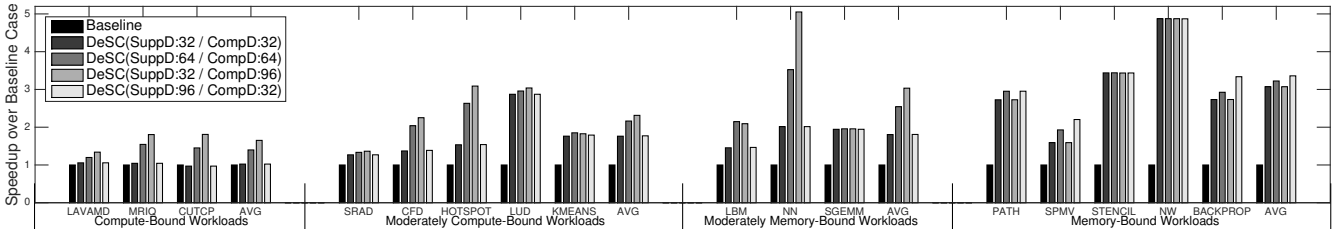

Figure 11: Effect of ROB resource allocation for each device side in DeSC.

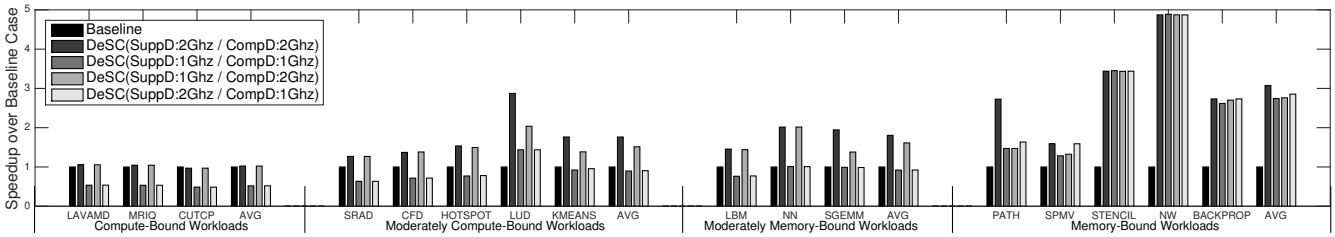

Figure 12: Potential for frequency/voltage scaling in DeSC.

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 1.6x-3x speedup over a single OoO core with varying ROB sizes for memory-bound workloads.

6.3 DeSC on heterogeneous CMP

One feature of DeSC is that it supports considerable diversity between SuppD and CompD design. This section explores moderate design specialization, while the following explores DeSC's usage in accelerator scenarios.

Figure 11 explores the effect of ROB size in DeSC. Increased ROB size improves both latency tolerance and throughput of the core while increasing design complexity of the OoO core. Second bar represents the baseline configuration of DeSC which consists of two cores whose ROB size is 32. And the third bar represents the configuration where both core's ROB size is increased to 64. As expected, increased ROB case performs strictly better than baseline case. However, simply increasing the ROB size for both cores is not the best solution on DeSC. As DeSC decouples communication and computation, it is often much more beneficial to focus on enhancing one of SuppD or CompD depending on application's characteristic. For example, in (moderately) compute Bound workloads, focusing on increasing ROB size of CompD is better than simply increasing both core's ROB. Also, increasing the ROB size of SuppD does not help at all in those cases. On the other hand, in memory bound workloads, focusing on increasing ROB size often results in the best performance. Note that increasing SuppD ROB did not result in increase of performance in some memory bound workloads because they were bound by external factors (e.g., memory bandwidth, issue width).

Figure 12 explores the potential for voltage/frequency scaling in DeSC. For compute-bound applications, scaling down SuppD frequency often retains full performance while enabling power/energy saving. Likewise, for memory-bound applications, reducing CompD frequency can save power with little or no performance impact. Exceptionally, in pathfinder, decrease in either SuppD or CompD incurs large performance penalty because SuppD's throughput or CompD's throughput becomes the new bottleneck when frequency is halved.

7. DeSC EVALUATION: ACCELERATORS

While DeSC on CPUs already offers significant performance benefits, this section also sketches out DeSC's performance potential for CompDs that are implemented as a specialized hardware accelerators.

7.1 Methodology

Detailed modeling of hardware accelerator behavior is very challenging, and most existing accelerator synthesizers or modelers (e.g. [44]) 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. Fig. 13

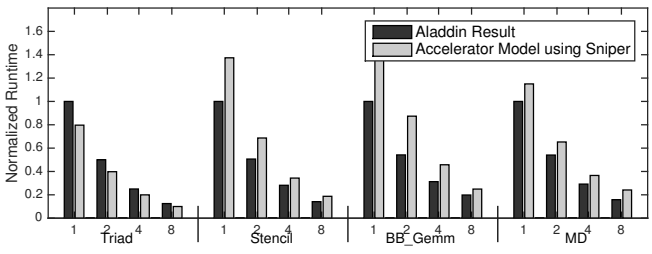

Figure 13: Accelerator model validation.

shows these validation results for four SHOC kernels [16] 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 system consisting of a CompD accelerator and a SuppD) where both sides use the same model.

7.2 DeSC with Hardware Accelerator CompD

Figure 14 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 7. 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.

8. RELATED WORK

Decoupled Access Execute Architecture: *Decoupled Access Execute* (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 [36, 46, 47]. Later DAE work extended this by exploring implementation details [21], analyzing LoD events [3], analyzing communication/computation balance [26], providing compiler framework [52], and extending for vector processors [18]. In addition, more recent work utilizes DAE for various purposes such as optimizing indirect loads [14], efficient DVFS [25, 30], and energy-efficient graphics pipelines [1]. In all these papers, DAE aimed to hide memory latency and was often viewed as a potentially simpler alternative to superscalar processors. Our work views DAE with an updated 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.

Helper Thread & Runahead for Prefetching: In addition to the split streams used in DAE, other work has envisioned helper threads either constructed by hand [11], compiler [28, 35], dynamic compilation [34, 54], or hardware [10], 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, in order to reduce memory latencies seen by the main thread. Similarly, Runahead execution [37], Performance-correctness explicitly decoupled architecture [20] and Dual core execution [55] 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 [41, 42]. 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 [48], Kilo-instruction processor [15], a Flexible heterogeneous multicore architecture [39] and simultaneous speculative threading [8] 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.

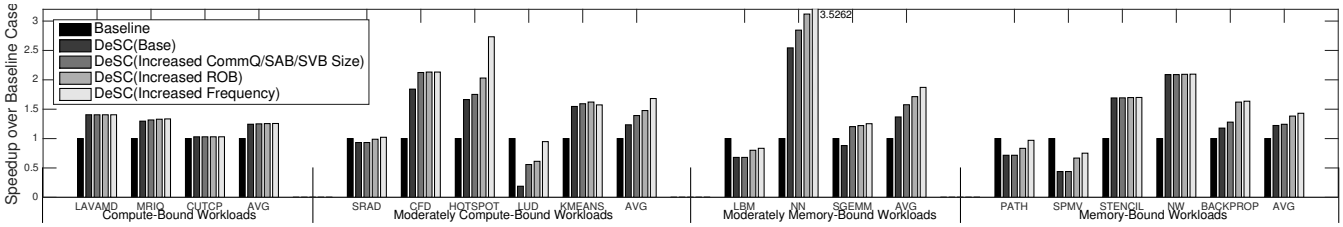

Figure 14: Speedup of a hardware accelerator CompD with varying SuppD designs over an accelerator with cache.

Automated Accelerator Design: In part thanks to high-level synthesis tools [4, 5, 19, 33, 38], 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 [23, 24] or in distributed memory systems [2, 45]. While similar in motivation to DeSC, they study distinct scenarios, such as larger memories or looser compute-memory couplings.

9. CONCLUSIONS

Overall, this paper envisions and evaluates DeSC, a framework for decoupling memory and compute that has been inspired by DAE, but updated and expanded for modern, heterogeneous processors. With modest hardware and compiler support, DeSC can offer significant speedups both for general-purpose and homogeneous scenarios as well as for more specialized or accelerator-centric cases. DeSC gains particular leverage from its optimizations for terminal loads on the memory supply device, and for streamlining store-to-load dependences from compute to supply side. With average speedups of 2.04x on CMP and 1.56x on accelerators across the evaluated workloads, DeSC strikes an important balance in terms of "specialized generality": the DeSC approach has enough specialization to achieve significant performance advantages, while still being general enough to port well across different design implementations.

Acknowledgements

The authors would like to thank the anonymous reviewers for very helpful feedback. Tae Jun Ham was supported in part by a Samsung Fellowship. Prof. Aragón was supported by a fellowship from the Spanish MEC under grant "Subprograma Estatal de Movilidad del Profesorado 2015". This work was supported in part by the DARPA PERFECT program (contract no. HR0011-13-C-0003). 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 MINECO under grant TIN2012-38341-C04-03.

10. REFERENCES

[1] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Boosting mobile GPU performance with a decoupled access/execute fragment processor," in *Proc. 39th Annual International Symposium on Computer Architecture*, 2012.

[2] A. Basumallik and R. Eigenmann, "Optimizing irregular shared-memory applications for distributed-memory systems," in *Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[3] P. Bird, A. Rawsthorne, and N. Topham, "The effectiveness of decoupling," in *Proc. 7th International Conference on Supercomputing*, 1993.

[4] "C-to-Silicon Compiler High-Level Synthesis," Cadence, Tech. Rep., 2011. [Online]. Available: http://www.cadence.com/rl/Resources/datasheets/C2Silicon_ds.pdf

[5] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: High-level synthesis for FPGA-based processor/accelerator systems," in *Proc. 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2011.

[6] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[7] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization*, 2014.

[8] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Simultaneous speculative threading: A novel pipeline architecture implemented in sun's rock processor," in *Proc. 36th Annual International Symposium on Computer Architecture*, 2009.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE International Symposium on Workload Characterization*, 2009.

[10] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *Proc. 34th Annual ACM/IEEE International Symposium on Microarchitecture*, 2001.

[11] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," in *Proc. 28th Annual International Symposium on Computer Architecture*, 2001.

[12] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich cmps," in *Proc. 49th Annual Design Automation Conference*, 2012.

[13] E. Cota, G. D. Guglielmo, P. Mantovani, and L. Carloni, "An analysis of accelerator coupling in heterogeneous architectures," in *Proc. 52nd Design Automation Conference*, 2015.

[14] N. C. Crago and S. J. Patel, "Outrider: Efficient memory latency tolerance with decoupled strands," in *Proc. 38th Annual International Symposium on Computer Architecture*, 2011.

[15] A. Cristal, O. J. Santana, M. Valero, and J. F. Martínez, "Toward kilo-instruction processors," *ACM Transactions on Architecture and Code Optimization*, vol. 1, no. 4, Dec. 2004.

[16] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

[17] A. Djabelkhir and A. Seznec, "Characterization of embedded

applications for decoupled processor architecture," in *IEEE International Workshop on Workload Characterization*, 2003.

[18] R. Espasa and M. Valero, "Decoupled vector architectures," in *Proc. 2nd IEEE Symposium on High-Performance Computer Architecture*, 1996.

[19] T. Feist, "Vivado Design Suite,," Xilinx, Tech. Rep., 2012. [Online]. Available: http://www.xilinx.com/support/documentation/white_papers/wp416-Vivado-Design-Suite.pdf

[20] A. Garg and M. C. Huang, "A performance-correctness explicitly-decoupled architecture," in *Proc. 41st Annual IEEE/ACM International Symposium on Microarchitecture*, 2008.

[21] J. R. Goodman, J.-t. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, "PIPE: A VLSI decoupled architecture," in *Proc. 12th Annual International Symposium on Computer Architecture*, 1985.

[22] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *Proc. IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[23] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically managed data for CPU-GPU architectures," in *Proc. 10th International Symposium on Code Generation and Optimization*, 2012.

[24] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU communication management and optimization," in *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[25] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. Kaxiras, "Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling," in *Proc. of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.

[26] L. K. John, V. Reddy, P. T. Hulina, and L. D. Coraor, "Program balance and its impact on high performance risc architectures," in *Proc. 1st IEEE Symposium on High-Performance Computer Architecture*, 1995.

[27] M. Kambadur, K. Tang, and M. A. Kim, "Harmony: Collection and analysis of parallel block vectors," in *Proc. 39th Annual International Symposium on Computer Architecture*, 2012.

[28] D. Kim and D. Yeung, "Design and evaluation of compiler algorithms for pre-execution," *SIGPLAN Not.*, vol. 37, no. 10, Oct. 2002.

[29] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded sparc processor," *IEEE Micro*, vol. 25, no. 2, Mar. 2005.

[30] K. Koukos, D. Black-Schaffer, V. Spiliopoulos, and S. Kaxiras, "Towards more efficient execution: A decoupled access-execute approach," in *Proc. 27th International ACM Conference on International Conference on Supercomputing*, 2013.

[31] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, vol. 38, no. 11, Nov. 2005.

[32] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Proc. International Symposium on Code Generation and Optimization.*

[33] F. Liu, S. Ghosh, N. P. Johnson, and D. I. August, "CGPA: Coarse-grained pipelined accelerators," in *Proc. 51st Annual Design Automation Conference*, 2014.

[34] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham, "Dynamic helper threaded prefetching on the sun ultrasparc cmp processor," in *Proc. 38th Annual IEEE/ACM International Symposium on Microarchitecture*, 2005.

[35] C.-K. Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *Proc. 28th Annual International Symposium on Computer Architecture*, 2001.

[36] W. Mangione-Smith, S. Abraham, and E. Davidson, "The effects of memory latency and fine-grain parallelism on astronautics ZS-1 performance," in *Proc. 23rd Annual Hawaii International Conference on System Sciences, 1990*, vol. i, 1990.

[37] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proc. 9th International Symposium on High-Performance Computer Architecture*, 2003.

[38] A. Nanda, "Accelerate Performance and Design Productivity with OpenCL on Altera FPGAs," Altera, Tech. Rep., 2012. [Online]. Available: http://wl.altera.com/education/webcasts/all/source-files/wc-2012-opencl/player.html

[39] M. Pericas, A. Cristal, F. J. Cazorla, R. Gonzalez, D. A. Jimenez, and M. Valero, "A flexible heterogeneous multi-core architecture," in *Proc. 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[40] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[41] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August, "Parallel-stage decoupled software pipelining," in *Proc. 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2008.

[42] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, "Decoupled software pipelining with the synchronization array," in *Proc. 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[43] J. Rattner, "Making the right hand turn to power efficient computing," 2002. [Online]. Available: http://www.microarch.org/micro35/keynote/JRattner.pdf

[44] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures," in *ACM/IEEE 41st International Symposium on Computer Architecture*.

[45] S. D. Sharma, R. Ponnusamy, B. Moon, Y. S. Hwang, R. Das, and J. Saltz, "Run-time and compile-time support for adaptive irregular problems," in *Proc. ACM/IEEE Conference on Supercomputing*, 1994.

[46] J. Smith, "Decoupled access/execute computer architectures," *ACM Transactions on Computer Systems*, vol. 2, no. 4, Nov. 1984.

[47] J. Smith, S. Weiss, and N. Pang, "A simulation study of decoupled architecture computers," *IEEE Transactions on Computers*, vol. C-35, no. 8, Aug 1986.

[48] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," in *Proc. 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[49] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, 2012.

[50] X.-H. Sun and Y. Chen, "Reevaluating amdahl's law in the multicore era," *J. Parallel Distrib. Comput.*, vol. 70, no. 2, Feb. 2010.

[51] "OMAP4 mobile applications platform," Texas Instruments, Tech. Rep., 2011. [Online]. Available: http://www.ti.com/lit/ml/swpt034b/swpt034b.pdf

[52] N. Topham, A. Rawsthorne, C. McLean, M. Mewissen, and P. Bird, "Compiling and optimizing for decoupled architectures," in *Proc. ACM/IEEE Conference on Supercomputing*, 1995.

[53] M. Weiser, "Program slicing," in *Proc. 5th International Conference on Software Engineering*, 1981.

[54] W. Zhang, D. M. Tullsen, and B. Calder, "Accelerating and adapting precomputation threads for effcient prefetching," in *Proc. IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[55] H. Zhou, "Dual-core execution: Building a highly scalable single-thread instruction window," in *Proc. 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.