Efficient Data Supply for Parallel Heterogeneous Architectures

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

Decoupling techniques have been proposed to reduce the amount of memory latency exposed to high-performance accelerators as they fetch data. Although decoupled access-execute (DAE) and more recent decoupled data supply approaches offer promising single-threaded performance improvements, little work has considered how to extend them into parallel scenarios. This article explores the opportunities and challenges of designing parallel, high-performance, resource-efficient decoupled data supply systems. We propose MERCURY, a parallel decoupled data supply system that utilizes thread-level parallelism for high-throughput data supply with good portability attributes. Additionally, we introduce some microarchitectural improvements for data supply units to efficiently handle long-latency indirect loads.

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

Additional Key Words and Phrases: Heterogeneous architecture, decoupled architecture, data access optimization

ACM Reference format:
Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2019. Efficient Data Supply for Parallel Heterogeneous Architectures. *ACM Trans. Archit. Code Optim.* 16, 2, Article 9 (April 2019), 23 pages.
https://doi.org/10.1145/3310332

1 INTRODUCTION

In response to both application trends fueling increasing compute capability demands and the end of Moore/Dennard technology scaling, specialized accelerators have emerged as an important alternative to conventional cores. Although specialized accelerators show great potential in improving compute performance and performance-per-watt, reaching their full potential still requires overcoming the challenge of keeping them supplied with data.

This is a new article, not an extension of a conference paper.
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; by the NSF under grant SHF-1617732; and by the Spanish State Research Agency under grants TIN2015-66972-C5-3-R and TIN2016-75344-R (AEI/FEDER, EU).
Authors' addresses: T. J. Ham, Seoul National University, 1 Gwanak-ro, Gwanak-gu, Seoul 08826, Republic of Korea; email: taejunham@snu.ac.kr; J. L. Aragón, Facultad de Informática, Campus de Espinardo, University of Murcia, 30100 - Murcia, SPAIN; email: jlaragon@um.es; M. Martonosi, Dept. of Computer Science, Princeton University, 35 Olden St. Princeton, NJ 08540; email: mrm@princeton.edu.

1544-3566/2019/04-ART9
https://doi.org/10.1145/3310332

Challenges in supplying data from memory to compute elements have been present and growing for more than three decades now—the so-called memory latency wall [53]. These challenges become even more difficult in the era of specialized accelerators. The success of specialized accelerators at speeding up particular problems (e.g., encryption, graph analytics, image analysis) in turn makes memory latency look—from a relative perspective—even larger. Accelerators widen the gap between the computation capability and data accesses, making the *memory wall* more severe. Without successful solutions to this data supply problem, accelerators will not reach their performance potential.

To achieve the goal of minimizing and tolerating memory latency, current specialized accelerator designs usually place an additional burden on programmers. For example, programmers are asked to manually partition data to a size that fits in a particular scratchpad memory while scheduling data transfer in a way that minimizes the exposed memory latency. Moreover, such optimization is often tied to a specific configuration (e.g., scratchpad memory size, port count), so each configuration change from one implementation to another requires rewriting the optimized communication code. Thus, the question of how to efficiently feed the increasing number of fine-grain accelerators without burdening programmers is a major problem that remains unsolved.

In part to address memory latency concerns, the emergence of specialized hardware accelerators has led to a resurgence of interest in *decoupled* approaches. Drawing from the early decoupled access-execute (DAE) approach [44, 45], recent works evolve and adapt such ideas for modern processors [8, 15, 16, 22, 25, 37]. Both the original DAE proposal and more recent decoupling approaches seek to mitigate the performance impact of memory latency by *decoupling* the memory access operations from the compute operations that subsequently operate on those values. Instead of relying on manual programmer effort, these approaches can use compiler support to automatically generate separate code slices for the *access* portion (i.e., data supply) of the application and for the *execute* portion (i.e., compute). Compared to the generic CPU used for access in the original DAE, recent decoupled approaches specialize and optimize the *decoupled data supplier* (DDS) unit specifically to minimize the memory latency exposed to the *compute unit* (CU).

Until now, most works on decoupled data supply systems have primarily focused on them in single-threaded contexts: a *single* DDS unit and a *single* CU operating as a pair. This one-to-one pairing offers single-threaded speedup, but with today's workloads, we seek to support larger amounts of on-chip parallelism. This work explores the opportunities and challenges of designing an efficient, high-performance decoupled data supply system for *parallel* configurations where one or more DDS units supply in parallel to multiple CUs. Such approaches allow decoupled data supply paradigms to leverage larger amounts of on-chip parallelism, to offer greater speedups. In the process, we also show that they allow for better resource sharing that can reduce hardware overheads while maintaining speedup.

The key contributions of this article are the following:

- We propose Mercury, a parallel, decoupled data supply system that extends high-throughput decoupled data supply techniques to parallel environments where thread-level parallelism (TLP) offers high performance and efficient use of resources. In many cases, this allows DDS speedups to be multiplicative on top of conventional parallel speedups. We show that the best parallel DDS designs are often not simple replications of individual DDS approaches.
- The Mercury-N approach operates as a set of N individual DDS units paired with N individual CUs. This scalable design offers an average 3.7x speedup for the evaluated workloads over a conventional CMP.
- The Mercury-Shared approach utilizes a shared DDS unit leveraging simultaneous multithreading (SMT) techniques to drive multiple CUs. This approach has significant advantages

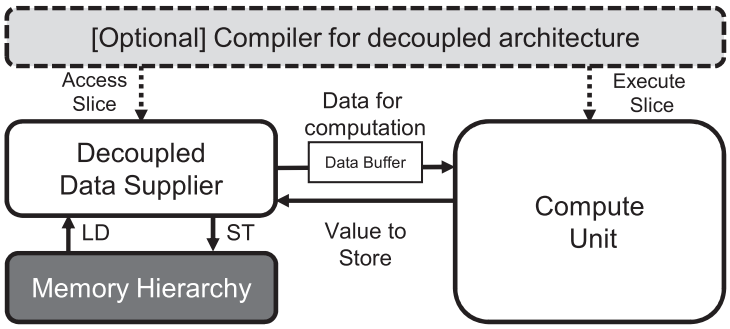


Fig. 1. Generic decoupled data supply and compute system. Details vary across implementations [8, 15, 22, 37].

in terms of resource sharing. Mercury-Shared offers a comparable average speedup (3.5x on multithreaded workloads and 2.9x for multiprogrammed ones) over a CMP yet using 2.5x less area than Mercury-N.

- In addition to gains through parallelism, we further extend the DDS microarchitecture by presenting an optimization that enables the DDS to tolerate the effect of *indirect loads*. Such loads have a high potential to limit the system performance and are very common on applications processing graphs or sparse matrices. For workloads with heavy use of indirect loads, Mercury achieves 61% to 83% *additional* speedup.

2 BACKGROUND

Decoupled access-execute. The DAE architecture was originally envisioned as a lower-complexity alternative to out-of-order processors with the goal of reducing or better tolerating memory latency [44, 45]. In DAE approaches, a program code is sliced into an access instruction stream and an execute instruction stream to improve memory latency tolerance by more efficiently overlapping data accesses with computation. DAE speedups hinge on ensuring that the access slice can run sufficiently ahead of the execute slice. DAE approaches have high potential to outperform other data prefetching approaches, particularly due to the effectiveness of their lookahead approach for hard-to-predict access patterns. Early DAE approaches, however, fell short of their full performance potential in the case of both (a) loss of decoupling events (LoD as termed in other works [2, 11, 49]), basically due to dependencies, that limit the runahead distance between the access and execute threads, and (b) lack of re-order buffer (ROB) space resulting in limited instruction-level parallelism (ILP) opportunities whenever a long-latency load was blocking the head of the ROB (in a similar way as for conventional out-of-order (OoO) cores).

Decoupled data supply. One key aspect that differentiates DAE from many other prefetch techniques is that DAE is not speculative. In other words, its access unit (data supplier) supplies all data its execution unit (CU) needs. Based on this advantage, more recent work—expanding on the key intuitions of DAE—has proposed decoupled data supply system designs feeding a diverse set of CUs including accelerators [8, 15, 22, 37]. Figure 1 shows a general decoupled data supply and compute system, although different proposals vary in their specific hardware and compiler support. In these works, a DDS unit is utilized to supply data for a CU with limited latency tolerance, such as an application-specific accelerator, a programmable accelerator, or a conventional out-of-order core. A DDS supplies data to a designated storage near the CU (e.g., scratchpad memory, hardware queue, content-addressable memory (CAM)) ahead of time, which allows the CU to retrieve data with a very low access latency. There are several different design

philosophies embodied in different DDS proposals. For example, Ho et al. [22] utilize a custom ISA and programming model to design a programmable DDS unit that performs as well as an out-of-order core data supplier for better energy efficiency. Another approach taken in Chen and Suh [8] is to design a custom DDS unit for each compute accelerator, which also results in an energy-efficient design. DeSC [15] takes a third approach, as explained next.

DeSC system. DeSC [15] utilizes a specialized out-of-order core as a DDS unit. Starting from a conventional out-of-order core, DeSC removes unneeded functionality and then specializes the core so that it can work as a very effective data supplier unit achieving much higher memory-level parallelism (MLP) and instruction-level parallelism (ILP). In particular, DeSC utilizes special instructions such as `PRODUCE` for its supplier to fill a *data buffer* from where the CU will later retrieve data. DeSC's supply side can work with minimal modification with various types of compute sides (e.g., CPUs, accelerators). For example, if the compute side is a CPU, the conventional memory instructions (i.e., `LOAD`, `STORE`) are replaced with instructions that in turn access the DeSC data buffer (i.e., `CONSUME`, `STORE_VAL`). However, if the CU is a custom accelerator, the memory access units should be modified to access the DeSC data buffer instead of main memory or scratchpad memory. DeSC facilitates this process with a LLVM-based compiler that utilizes program slicing (specifically, backward slicing) [52] and other techniques to split the original code into the access (supply) and execute (compute) streams. With this compiler, the software for both the supplier and the compute sides can be automatically generated without programmer intervention. If the CU being fed is intended to be a custom accelerator, the compiler's auto-generated execute slice can help to automate (with high-level synthesis tools) or ease the custom design process of the accelerator hardware that serves as a DeSC-compatible CU.

3 A PARALLEL DECOUPLED DATA SUPPLY SYSTEM

3.1 Challenge in Balancing the DDS and the CU

One of the key factors for a decoupled architecture to perform efficiently is to properly balance the *data supply rate* with the *data consumption rate*. Otherwise, one part of the system will end up waiting for the other to supply/consume data items. The data supply rate is defined as the number of data items a DDS can supply to the data buffer per unit time, which mainly depends on both the effectiveness of the DDS hardware and the application characteristics. For example, a powerful data supplier (e.g., implementing effective structures to exploit more ILP, higher frequency, larger L1 cache) can achieve a higher data supply rate than a weaker DDS without such advantages. In addition, application characteristics such as an easy-to-utilize ILP/MLP, high data locality, or simple (direct) address calculations let the DDS achieve a higher data supply rate. Similarly, the data consumption rate—defined as the number of data items consumed from the data buffer per unit time—is determined by the effectiveness of the CU hardware and the application characteristics. Having a more effective hardware (e.g., ability to exploit ILP, a large number of ALUs, specialized functional units) and running applications with certain characteristics (i.e., easy-to-utilize ILP, low computation-to-data-access ratio) results in a higher data consumption rate.

Summarizing, a decoupled architecture is a classical producer-consumer design. When a powerful CU (i.e., the consumer) is paired with a weak DDS unit (i.e., the producer), the former frequently stalls waiting for data to be supplied by the DDS, killing any potential benefit from decoupling (as shown in Figure 2(a)). The straightforward solution is to overprovision the DDS side to increase its data supply capability to avoid it being the bottleneck of the system. However, when a powerful and large DDS unit is paired with a CU that does not consume data so often, the DDS unit will stay idle most of the time waiting for the data buffer to have free space to supply new data. This will lead to an underutilization of the DDS supply capability (as illustrated in Figure 2(b)).

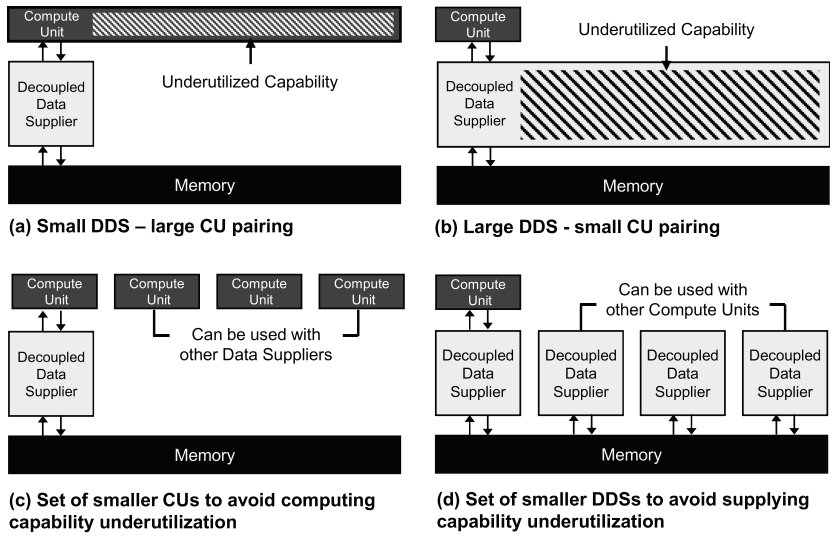


Fig. 2. How the use of multiple, finer-grain DDS unit and CU avoids the capability underutilization issue.

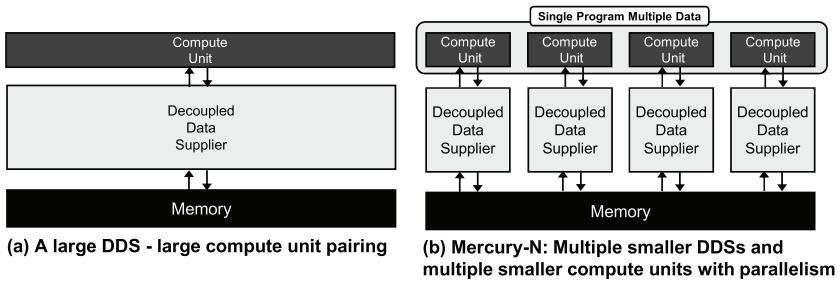
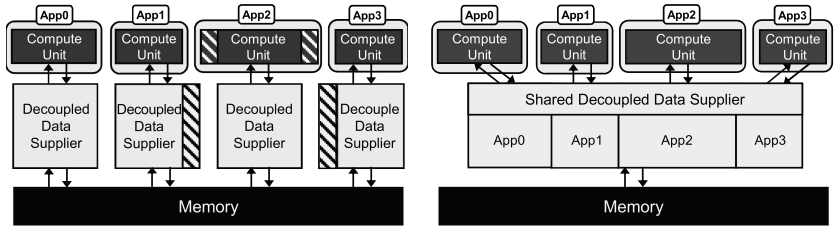


Fig. 3. How TLP enables higher performance with multiple smaller DDS units and CUs.

A natural solution to avoid the capability underutilization issue is to fragment the units following a finer-grain approach. By using multiple (but smaller) DDS units and CUs instead of single ones with a larger amount of resources, it is possible to avoid the underutilization of resources. As shown in Figure 2(c) and (d), underutilized original resources can now be used by other DDS units or CUs. Still, this finer-grain pairing has two main limitations. First, a single small DDS and a small CU pair naturally achieves lower performance compared to the pairing of large DDS unit and large CU. Second, a finer-grain DDS and CU pairing still can suffer from capability underutilization at either side (albeit to a lesser degree when compared to Figure 2(a) and (b)). The proposed Mercury systems aim to address these limitations through the use of TLP. The next section introduces two different Mercury configurations.

3.2 Overview of Mercury Systems

Mercury-N: A replicated DDS approach using TLP for better performance. To address the aforementioned limitations, this article first proposes a parallel decoupled data supply system named Mercury-N. Mercury-N employs N pairs of DDS units and CUs connected in a one-to-one fashion. Instead of utilizing a single large DDS unit paired with a single CU (as in Figure 3(a)), Mercury-N follows a fine-grain approach by using multiple but smaller DDS units and CUs (as



(a) Multiple pairs of DDS and compute units where each pair runs different application. Hatched area shows underutilized capability.

(b) Mercury-Shared: A single shared DDS supplying data for compute units each running a different application.

Fig. 4. How a shared DDS further avoids capability underutilization by sharing supplier capability.

in Figure 3(b)). Although the mismatch between the data supply rate and data consumption rate can still happen in this configuration, its degree is naturally much more limited because both DDS units and CUs are smaller compared to the case where a large, powerful DDS is paired with a weak CU (or contrarily, a large CU is paired with a small DDS). When high performance is demanded for a parallel (multithreaded) application, Mercury-N utilizes multiple pairs of DDS units and CUs, each pair running a thread of the application. However, when there are multiple programs to run (multiprogrammed workload), each pair of DDS units and CUs can run a different one.

TLP exploitation is not straightforward in all DDS implementations. For example, for a custom DDS unit design (e.g., those proposed in other works [8, 22, 37]) to fully support TLP, a substantial extension is needed to their programming frameworks, compilers, and hardware. However, designs like DeSC can naturally support TLP because they build on top of a conventional out-of-order core with full support for TLP. A DeSC-based DDS unit can utilize both existing programming frameworks (e.g., Pthreads, OpenMP, C++ Threads) and the existing hardware support for synchronization required for TLP. For this reason, Mercury builds on top of a DeSC-based supply unit to implement a *parallel* decoupled data supply system. Additionally, note that the CUs in Mercury do not need any explicit support for parallelism either since all memory accesses and synchronization happen only on the DDS side.

Mercury-Shared: A shared DDS approach for better resource utilization. Although the Mercury-N configuration is effective, it can still suffer from the capability underutilization issue when there is a mismatch between the data supply rate and the data consumption rate, as shown in Figure 4(a). In this example, there are four different applications (App0 to App3) running on four CUs, each having different data consumption rates, as illustrated by their different widths. Each CU is paired with a different DDS, all designed to have the same data supply rate, as illustrated by their equal width. In particular, for App1 and App3, the supply rate of their respective DDS units exceeds the consumption rate of the respective CUs, and thus both DDS units become underutilized. For App2, however, the CU's consumption rate is higher than its peer DDS supply rate, and so the CU becomes underutilized. Finally, for App0, the supply and consumption rate match, and thus there is no underutilization.

Mercury-Shared is a system consisting of a single big DDS and multiple CUs (Figure 4(b)). Unlike a single pair DeSC or a replicated Mercury-N configuration, Mercury-Shared breaks the convention of the one-to-one pairing between a DDS unit and a CU. Instead, a single Mercury-Shared DDS unit supplies data for multiple CUs by adopting an SMT approach. With an SMT-based design, the Mercury-Shared DDS unit runs multiple access threads simultaneously, and each of those threads interacts with a different CU in the system. The threads running on the shared DDS can be part of a single multithreaded application or be individual applications.

The key benefit of a shared DDS design is that it allows for a more efficient use of resources, particularly in the case where each thread has different supply/compute demand needs. For example, a shared DDS unit design allows resources not utilized for supplying data to one application (e.g., App1, App3 in Figure 4) to be *dynamically* used for supplying data to another application with higher data supply demands (e.g., App2 in Figure 4). There are two different ways to utilize the Mercury-Shared architecture. First, a shared DDS unit can be configured to have the same amount of resources as multiple DDS units. In such a scenario, Mercury-Shared can achieve better overall throughput compared to Mercury-N by letting a particular data supply thread (i.e., App2 in Figure 4(b)) utilize more resources within the DDS if the CU executing App2 has a higher data demand need. Second, a shared DDS can be configured to have fewer resources compared to multiple DDS units. In such a case, Mercury-Shared can improve or maintain the performance of Mercury-N while using fewer resources. Overall, Mercury-Shared is capable of dynamically tailoring data supply rates to the needs of multiple CUs. Section 3.3 further explains the Mercury-Shared design.

3.3 Designing a Shared DDS

There are various possible shared DDS unit designs based on an SMT approach. Our work particularly focuses on a shared DDS unit that shares two key resources in out-of-order cores, namely, the instruction window (IW) and the instruction bandwidth (e.g., fetch/decode/issue bandwidth). It is important to note that in a traditional SMT design, the sharing of such resources might easily result in resource contention leading to performance degradation. However, the situation is different on a decoupled system. This section explains how the unique nature of the access threads that run on the shared DDS allows the effective sharing of resources with substantially less contention compared to the conventional (nondecoupled) SMT scenario. Additionally, it explains how sharing memory system resources brings even further benefits.

Sharing the instruction BW. In a nondecoupled SMT sharing the instruction BW (and ALUs) is quite a common bottleneck. For example, if four threads are running on a four-way SMT core that can process up to four instructions per cycle, each thread, on average, can process a single instruction per cycle. Although there are communication-intensive workloads whose IPC does not exceed 1, many compute-intensive workloads often reach a higher IPC when provided with enough resources. If such workloads are run on this SMT core, the fetch/decode/issue BW will work as a bottleneck, degrading overall performance.

However, the situation is different in a decoupled scenario. A shared DDS unit runs sliced access (supply) threads that are in charge of calculating addresses, accessing data from/to the memory and supplying the data to the data buffer. Naturally, due to its frequent data accesses, a decoupled access thread often has lower IPC compared to a nondecoupled code or the execute (compute) thread. Furthermore, note also that decoupling reduces the number of instructions to be executed on the DDS (since computation instructions are offloaded to the CU), and thus a decoupled access thread requires a lower IPC to achieve the same performance compared to the nondecoupled (original) thread. Therefore, given that access threads' IPC is relatively low, even though multiple of such low-IPC threads share the instruction bandwidth, overall throughput degradation can be minimal or even nonexistent. Still, there are cases where sharing the instruction BW can work as a bottleneck. To minimize the negative impact in such cases, we introduce a fetch prioritization policy that further minimizes the negative impact of instruction bandwidth sharing.

Decoupling-aware fetch policy. In SMT cores, the fetch policy dictates how resources are allocated to different threads. If a fetch policy favors one thread, it will utilize the fetch bandwidth for the cycle, leading to more IW usage for that thread. The most commonly used fetch policy is

ICOUNT [50], which favors threads with the least number of instructions in the decode, rename, and IW. This policy is based on the intuition that providing more resources to underutilized threads brings overall efficiency. However, this simple intuition does not always hold true in DAE-based architectures where the system performance is not solely dictated by the capability of the DDS unit. In fact, increasing the data supply rate further does not improve performance once it reaches the paired CU's consumption rate.

In a scenario where the data supply rate exceeds the data consumption rate, the access thread will frequently stall because the corresponding data buffer is full. However, a conventional policy like ICOUNT will identify this thread as an underutilized thread (since it often has a very low number of instructions in decode, rename, and IW due to the frequent stalls) and will prioritize it. We propose a simple yet effective variation of the ICOUNT policy that takes the decoupling scenario into account. Our *decoupling-aware fetch policy* simply inspects the current occupancy of the data buffer to estimate the decoupling distance. If the current occupancy exceeds a certain threshold (e.g., 75% of the queue size), it indicates that the data consumption rate is likely to be lower than the data supply rate, and thus we assign this thread a low priority. Otherwise, if the current occupancy is below a certain threshold (e.g., 25% of the queue size), this thread will soon need more data, and thus we assign the thread a high priority. Otherwise, the thread is categorized as normal priority. Then, for every cycle, the fetch thread selection logic selects the thread with the highest priority. If the selected thread does not have any instruction to fetch, another thread is given the chance. If there are multiple threads in the same priority class, the normal ICOUNT policy is used. This approach preserves the benefits of ICOUNT while preventing it from prioritizing the wrong threads.

Sharing the IW. The IW is one of the essential resources in an OoO core with a large impact on performance. Thus, sharing it can result in significant performance degradation. For example, when four threads share the same IW, this decreases the effective IW size for each thread to one-fourth. Furthermore, it is also possible for a single thread to clog the IW, leaving even less effective IW size for other threads. Typically, the IW is clogged when a thread has a long latency instruction with many dependents. Dependents of the long latency instruction (and their respective dependents) will clog the IW space until the long latency instruction fully executes.

However, a Mercury's DDS unit is relatively free from this issue. First, its dependency chains are substantially shorter. A nondecoupled thread's dependency chain commonly starts with a load instruction, continues with a number of computing instructions, and eventually ends with a store instruction. However, a decoupled access thread's dependency chain starts with a load instruction and ends with a PRODUCE instruction that supplies data to the CU. Second, all of the instructions in an access thread are low-latency instructions except for loads. Furthermore, most of those loads are *terminal load* instructions that do not have any dependents. As a result, sharing the IW among access threads, which are naturally short-dependency chained and less prone to dependency-related stalls, reduces or even eliminates the potential performance loss from sharing the IW as it would have been the case for a traditional SMT processor executing normal threads.

Sharing memory system resources. There are two types of memory system resources: those shared across the chip (e.g., main memory, shared caches) and those private to a core (L1 cache, MSHRs). A shared, SMT-based DDS unit has an advantage in that all threads running on the core share the per-core memory system resources that can be easily aggregated. For example, if the baseline single DDS unit has a 16KB L1 cache, a four-way SMT DDS unit will have a 64KB L1 cache. In a scenario where access threads from different applications are supplying data to different CUs, this sharing allows one thread to utilize memory resources that other threads are not utilizing (if any). In addition, a multithreaded application can benefit from inexpensive interthread communication through a (larger) L1 cache instead of a shared LLC that incurs coherence overheads.

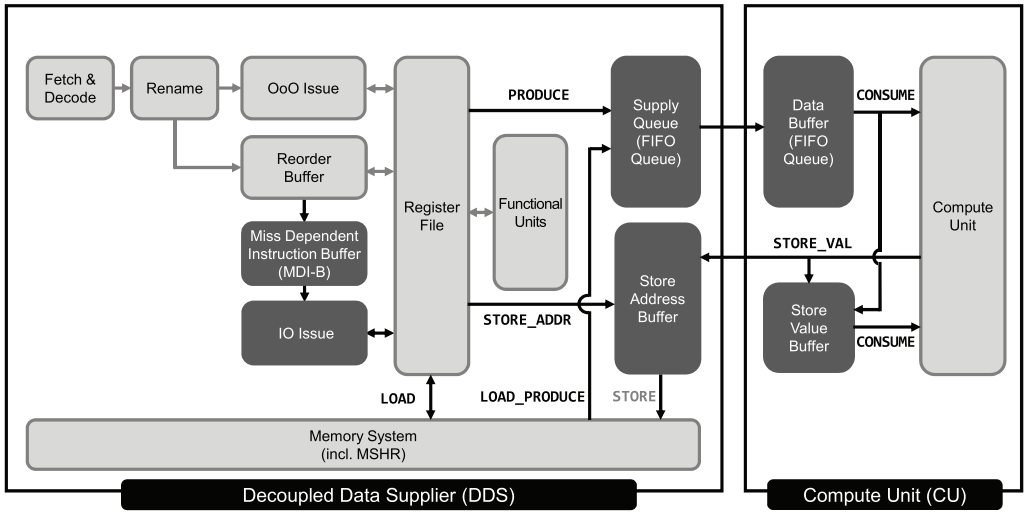


Fig. 5. Improved microarchitecture for a DeSC-style DDS unit that builds on top of an OoO core. Darker structures are the new additions for Mercury.

4 IMPROVING THE DDS MICROARCHITECTURE

In addition to the parallel DDS unit configurations we presented in the previous section, this work identifies some limitations for the supplier unit of DeSC and presents two microarchitectural techniques to overcome such limitations. In fact, these improvements can be applied to any DDS unit including the conventional DeSC (one-to-one configuration), Mercury-N, and Mercury-Shared. Still, these improvements are more beneficial for parallel systems containing multiple DDS or a shared DDS with a tighter resource (i.e., the IW).

Figure 5 shows the microarchitecture of the DDS unit and the CU. The DDS uses `PRODUCE` instructions to push data into the data buffer in the CU. For each produced data item, there is a corresponding `CONSUME` instruction (or its equivalent) executed on the CU to retrieve it. Similarly, the DDS executes a `STORE_ADDR` instruction to update the *store address buffer* with a calculated address for every original store instruction, whereas the CU executes a `STORE_VAL` instruction to pass the value generated (on the CU) to the DDS unit. The *store value buffer* is a structure that allows the reuse of the computed data as determined by a decoupled store-to-load forwarding technique described in Ham et al. [15]. The following sections explain the DDS microarchitecture in more detail and propose two improvements.

4.1 Supply Queue for Terminal Loads

A DDS unit is latency tolerant by nature, which allows for a near-zero latency exposed to the CUs. However, for memory-intensive applications, or when high-performance CUs are used, the DDS cannot cope with the higher data consumption rate, resulting in a bottleneck of the entire system. One key difference between a DDS unit and a conventional core is that their workloads are different. Although a conventional core runs general-purpose code, a DDS unit only runs access threads that contain just simple address calculation and data access instructions. Therefore, the only remaining long-latency instructions that can potentially hurt data supply throughput is a load instruction itself. This section explores how a specialized DDS design can ameliorate the effect of long latency loads on the data supply throughput.

In an access slice, there are two types of loads: loads whose results are only used in the execute slice and loads whose results are later used within the access slice. Specifically, the first type of loads are called *terminal loads* [15]. DeSC identifies such terminal loads with its compiler framework and marks them using a `LOAD_PRODUCE` instruction. DeSC presents an optimization for such loads to prevent them from blocking the head of the ROB. The key intuition is that since terminal loads' values will not be reused within the DDS unit, they can be retired early from the ROB (out-of-order) and be moved to a CAM-structured buffer (named the *terminal load buffer*), where they wait until their values are returned from memory that are then communicated, out-of-order, to the CU.

Although DeSC's proposed solution does the job, it utilizes relatively expensive and not scalable CAM structures on both the DDS units and CUs. Furthermore, since DeSC's out-of-order communication may introduce a deadlock, it also requires a deadlock prevention mechanism whose implementation can be expensive. This article proposes a cost-effective mechanism to manage terminal loads without using a CAM structure.

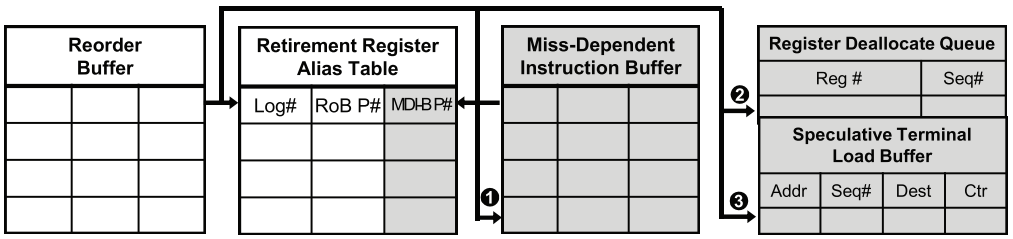
For every instruction that sends data to the CU (i.e., `PRODUCE`), an entry is allocated in a RAM structure named the *supply queue* (depicted in Figure 5). Then, when a terminal load executes and misses in the cache, an MSHR will be assigned for it. Unlike a conventional core, however, this MSHR records this terminal load's position in the supply queue (instead of a destination register or a ROB entry—positions in the queue are assigned in-order at decode time) to indicate where to buffer the value once it is serviced by the memory. This mechanism allows a terminal load that got its MSHR assigned and reaches the head of the ROB to safely commit, even if its data is not ready, since the assigned MSHR will provide the data directly to the corresponding entry in the supply queue. Data from the head of this supply queue is passed over to the data buffer on the CU *in-order*. This in-order communication allows the data buffer to be implemented as a RAM structure, and so any `CONSUME` instruction in the CU can access its matching data without an expensive associative search. Since our supply queue is much simpler than the terminal load buffer in Ham et al. [15], it is easier to enlarge to enhance performance.

4.2 Attacking Indirect Loads: The Miss-Dependent Instruction Buffer

No decoupling approach will be broadly useful without addressing long-latency *indirect loads* (i.e., loads whose outcome is used to compute another load's address). It is worth noting that indirect loads have a high potential to limit the system performance for some classes of applications (e.g., graph analytics, sparse matrix computation) that include many of such loads. The *miss-dependent instruction buffer* (MDI-B) is our novel microarchitectural approach to address this issue.

Miss-dependent instruction buffer. By attacking the long-latency indirect loads that can appear in an access stream, the MDI-B eliminates the ultimate bottleneck on the DDS performance. The result is similar to an access stream comprised only of short-latency instructions.

The main intuition is simple: migrate any long-latency indirect load and its dependent instructions to the MDI-B (a FIFO buffer) when they reach the head of the ROB. This allows other newer instructions to reach the head of the normal ROB and commit earlier than expected, bypassing these older miss-dependent streams. The miss-dependent instructions (migrated to the MDI-B) will execute in-order and commit when they reach the head of the MDI-B. If a conventional core running nondecoupled code implemented the MDI-B, it would be ineffective. In such case, load instructions often have deep dependency chains that would cause many instructions to be migrated to the MDI-B. However, we take advantage of the DDS running decoupled access code, which has very short dependency chains. Unlike conventional threads, the only dependents of load instructions in an access thread are address calculation instructions or instructions to compute branch conditions. For this reason, a small FIFO-based MDI-B buffer (32 entries in our experiments) can effectively house the dependents of indirect loads in a decoupled scenario.



❶ **Migrate** if long-latency nonterminal load or its dependents reaches head of ROB.
❷ **Allocate** if terminal load is ready to retire but there is nonzero unresolved STORE_ADDR in MDI-B.
❸ **Allocate** if committed architectural register's previously mapped physical register is poisoned.

Fig. 6. Hardware structures to support MDI-B.

```
i1: idx = LOAD(a[i])
i2: STORE_ADDR(b[idx])
i3: LOAD_PRODUCE(c[i])
```

Fig. 7. Indirect store address example.

The MDI-B approach is similar to previous literature on latency-tolerant out-of-order core designs [10, 21, 31, 46]. However, the MDI-B approach is considerably less complex and utilizes substantially fewer resources compared to such schemes. This is because we exploit the characteristic of a decoupled access thread and target a smaller problem—mitigating the impact of indirect loads in decoupled access threads. Designers can choose to exclude this extension if a target application is known not to be heavily reliant on indirect loads.

Migrating instructions to the MDI-B. Figure 6 shows the MDI-B structures. When an indirect load reaches the head of the ROB and misses in the LLC, it is moved to the MDI-B and its destination register is marked as poisoned. Later, if any instruction reaches the head of the ROB whose input register is poisoned, it is also moved to the MDI-B and its destination register is poisoned. This poisoning mechanism moves miss-dependent instructions to the MDI-B.

Branches and other exception-prone instructions (e.g., system calls, OS-related or other privileged instructions) are not migrated to the MDI-B to avoid a complicated recovery. In addition, nonterminal loads that depend on other nonterminal loads are neither migrated to prevent them from blocking the in-order (FIFO) MDI-B structure. When such instructions reach the head of the ROB with a poisoned input register, they simply wait until all other instructions in the MDI-B complete. Any other instruction reaching the head of the ROB just commits from there when its execution finishes.

Early retirement of terminal loads with MDI-B. With the MDI-B optimization, a terminal load at the head of the ROB often cannot commit due to the presence of unknown STORE_ADDR instruction migrated to the MDI-B. Figure 7 shows a code example for such case, where i2 is a STORE_ADDR instruction that depends on the load (i1) preceding it. If the MDI-B is enabled and i1 misses in the LLC, i1 is migrated to the MDI-B. Since i2 depends on i1, it is migrated as well. At that point, i3 is at the head of the ROB. However, since i2's address is unknown, and it may alias i3, it cannot retire. In this case, i3 should have to wait until i2 retires from the MDI-B, then nullifying its potential benefit.

To attack this issue, our approach allows speculatively executed terminal loads to *retire* from the ROB even when there are preceding unknown store address instructions. When a terminal

load reaches the head of the ROB *with* a preceding unknown STORE_ADDR, it removes itself from the ROB and retires to a CAM structure named the *speculative terminal load buffer* (STLB). As shown in Figure 6, an STLB entry is a tuple of address, seq#, location in the supply queue, and a counter that is initialized to the number of unresolved store addresses in the MDI-B. In addition, for each new STLB entry, the entry in the supply queue corresponding to the terminal load moved to the STLB entry is marked as *not ready*. This prevents data from being sent to the CU while the terminal load is still possibly dependent on a STORE_ADDR in the MDI-B. Every time a STORE_ADDR instruction executes from the MDI-B, its address is checked against the loads in the STLB. In the case of a match, the decoupled store-to-load forwarding mechanism proposed in Ham et al. [15] is triggered. Otherwise, every younger (in terms of seq#) instruction's counter field is decremented by 1. Whenever an STLB entry's counter becomes zero, it is removed from the STLB and its matching supply queue entry is marked as *ready* again. Our experiments show that a tiny STLB (8-entries) is sufficient.

4.3 Potential Issues and Solutions for the MDI-B

Since the MDI-B extension allows instructions to bypass earlier indirect load instructions and their dependents, few aspects of the processor microarchitecture should be changed. In the following, we discuss such changes.

Register management. In a conventional unified register file architecture, when an entry from the ROB retires, the physical register that corresponds to the previous mapping of the just committed instruction's destination architectural register is freed. However, in our proposed design, since ROB entries can commit earlier than MDI-B entries, the physical register cannot be freed when the target physical register is currently poisoned. In such a case, instead of freeing the physical register, we allocate an entry in a RAM structure named the *register deallocate queue* (RDQ) (see Figure 6) with the current instruction's sequence number (seq#). Whenever an instruction in the MDI-B commits, it compares its sequence number to that of the head of the RDQ. If the former is higher, the corresponding register of the RDQ's head entry can be freed.

Exception/misspeculation recovery. Mercury uses a retirement register alias table (RRAT) (see Figure 6) to recover from an exception or a branch misspeculation. For supporting the MDI-B extension, the RRAT is extended with one extra column that represents the register state seen by the MDI-B (in addition to keeping the register state seen by the ROB). Now, when an instruction retires from the ROB (or it is migrated to the MDI-B), it updates the ROB column of the RRAT with its physical register number. However, when an instruction retires from the MDI-B, it updates the MDI-B column of the RRAT. In addition, when the MDI-B frees a register, it clears the entry in the MDI-B's column. Therefore, when a branch misprediction or an exception occurs on instructions in the ROB, the core simply waits until (a) all the MDI-B instructions commit and (b) all preceding instructions in the ROB commit, then flushes the pipeline. By doing so, the core rolls back to the register state in the ROB's column of the RRAT. Analogously, when an exception occurs on instructions in the MDI-B (the only possible exception is a page fault), the core waits until the instruction reaches the head of the MDI-B and flushes the entire pipeline. Then, the core rolls back to the register state in the MDI-B's column (or the ROB's column counterpart if the MDI-B's column for a particular row is empty). Note that branch instructions are not migrated to the MDI-B, and thus there is no branch misprediction happening in the MDI-B to recover from.

Synchronization instructions. The Mercury DDS unit has a weak consistency model similar (or little stronger) to that of ARMv7's that requires programmers to use appropriate synchronization instructions such as fences when communicating between threads. With the proposed MDI-B

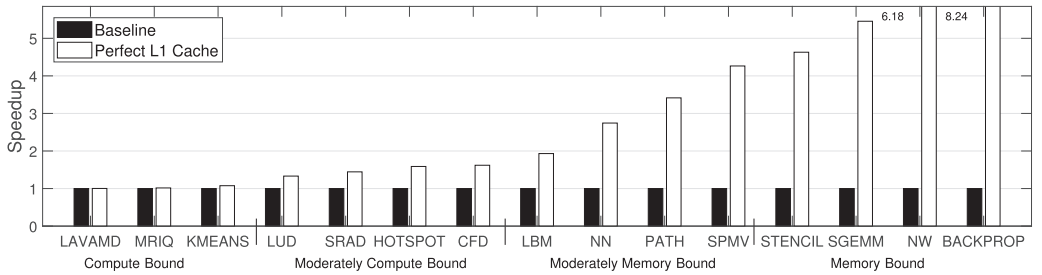


Fig. 8. Workload categorization.

extension, fences or any other synchronization instruction (a) do not commit when there are preceding terminal loads or (b) the MDI-B is not empty. Instead, such instructions simply wait at the head of the ROB until such conditions are cleared.

5 MERCURY EVALUATION

5.1 Methodology

We use a heavily modified version of the Sniper simulator [3] for the performance evaluation. Specifically, we extend Sniper's cycle-level out-of-order processor model [4] so that it can model Mercury's ISA and hardware components.

Workloads. Our workloads consists of 15 parallel kernels from the Parboil [47] and Rodinia [7] suites (mostly OpenMP versions suited for a CPU execution). Note that benchmark suites include more than 15 kernels; however, we excluded a few kernels for our experiments because they are extremely communication bound (e.g., bfs, b+tree). These kernels benefit little from an accelerator-based implementation or a decoupled architecture and thus are not considered as our targets. Similarly, few extremely compute-bound (e.g., cutcp) kernels are also excluded to avoid redundancy while keeping some as representative cases. These extreme benchmarks can be easily identified by compilers and help one to employ a decoupled architecture only when it is expected to be beneficial. We excluded three benchmarks because of our evaluation framework's incompatibility. Note also that some of these OpenMP kernels provided by the official benchmark suites are not tightly optimized for CPU execution, so its behavior may be different from that expected for highly optimized kernels (e.g., BLAS for matrix multiplication).

To identify applications' sensitivity to memory latency, we run the 15 benchmarks on four baseline OoO cores and measure their speedup on a perfect L1 cache system. Based on this result (Figure 8), we classify our workloads into four categories: compute intensive (Category 1), moderately compute intensive (Category 2), moderately memory intensive (Category 3), and memory intensive (Category 4). Utilizing these four categories, we construct eight multiprogrammed workloads with varying memory intensity (Table 1). To evaluate their performance, we synchronize all applications at their entrance point to the region of interest and run until one application finishes. For the performance metric, we measure system throughput (STP) as suggested in Eyerman and Eeckhout [12].

Configurations. Table 2 summarizes the architectural parameters used to model the baseline and evaluated Mercury systems. For the baseline case, four conventional OoO cores are utilized. We only report the baseline with the base memory system, as it is not limited by memory BW and it does not get any benefit from the more aggressive memory system. For Mercury systems, simplified baseline cores (with no memory hierarchy nor LSQs) are utilized as the CUs. For Mercury-N we evaluate a case with four DDS units and four CUs. For Mercury-Shared, we evaluate a system

Table 1. Evaluated Multiprogrammed Mixes

MP1	mri-q, kmeans, cfd, hotspot	2x Cat1, 2x Cat2
MP2	mri-q, kmeans, pathfinder, nn	2x Cat1, 2x Cat3
MP3	mri-q, kmeans, backprop, nw	2x Cat1, 2x Cat4
MP4	cfd, hotspot, pathfinde, nn	2x Cat2, 2x Cat3
MP5	cfd, hotspot, backprop, nw	2x Cat2, 2x Cat4
MP6	pathfinder, nn, backprop, nw	2x Cat3, 2x Cat4
MP7	mri-q, cfd, pathfinder, backprop	Cat1, Cat2, Cat3, Cat4
MP8	kmeans, hotspot, nn, nw	Cat1, Cat2, Cat3, Cat4

Table 2. Architectural Simulation Parameters

	Baseline Cores	**Mercury-N DDS Unit**	**Mercury-Shared DDS Unit**
Core	4x OoO cores	4x OoO cores	4-way SMT core
	64-entry ROB	64-entry ROB	4 x 64-entry ROB
	32-entry IW / 2.0GHz		
Fetch/Decode	4-way	4-way	2 x 4 way
Issue Width	4-way	4-way	8-way
Mercury Structures	N/A	4 x 256-entry supply queue/data buffer 4 x 128-entry store address/value buffer 4 x [32-entry MDI-B] (with 32-entry RDQ, 8-entry STLB)	
L1 Cache	32KB/core	32KB/core	128 KB
	4-way, 2ns latency, 64B cacheline		
L2 Cache	1MB, 8-way, 10ns latency, 64B cacheline		
Main Memory (Base)	16 MSHRs/core	16 MSHRs/core	64 MSHRs
	51.2GB/s BW, 100ns base latency		
Main Memory (Aggressive)	N/A	64 MSHRs/core	256 MSHRs
	N/A	204.8GB/s BW, 100ns base latency	

consisting of a single, four-way SMT DDS unit and the same four CUs. Note that we evaluate the Mercury-Shared DDS unit with 2x larger fetch/decode/issue width than the Mercury-N DDS to avoid Mercury-Shared performance severely limited by its peak instruction BW.

Area. We use McPAT [32] and CACTI [33] with a 22nm technology node to compare the area and static power consumption of both Mercury configurations. Experiments show that a four-way SMT DDS for Mercury-Shared consumes 2.50x less area and 3.09x less static energy compared to the four DDS units used for Mercury-N. Note that the baseline Mercury-N system utilizes about 2x or slightly more than 2x area (and static power) compared to the baseline, as it requires an additional core (i.e., DDS) in addition to the CU. This implies that Mercury-Shared requires about 40% more area and 33.3% more static power compared to the baseline.

5.2 Overall Performance Evaluation

Figures 9 and 10 show the effectiveness of Mercury at improving system performance. There are three key sources of Mercury speedup: hiding long latency memory accesses (i.e., like a perfect cache), improving the access time of on-chip storage, and parallelization of address computation

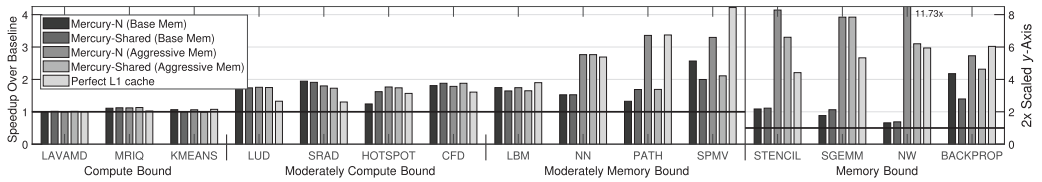


Fig. 9. Performance of Mercury running multithreaded workloads. The four memory-bound workloads use a right-side 2x scaled *y*-axis. Mercury-N offers 3.7x speedup and Mercury-Shared offers 2.9x speedup over a baseline four-core CMP. With a more aggressive memory system, they often outperform the baseline CMP with a perfect L1 cache (i.e., always hit). Note that Mercury-Shared takes only 40% of the area for the DDS compared to Mercury-N.

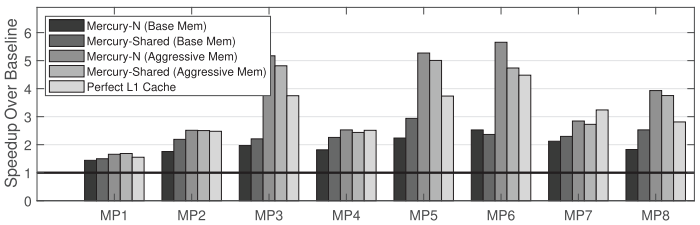


Fig. 10. Performance of Mercury running multiprogrammed workloads. Mercury achieves substantial speedup (Mercury-N: 3.7x, Mercury-Shared: 3.5x) over the baseline CMP and often outperforms the perfect L1 with the aggressive memory system.

and value computation. First, Mercury avoids exposing the memory latency to the CUs because DDS units access data ahead of time and supply data to them. Second, Mercury CUs retrieve data from a smaller data buffer instead of accessing a larger L1 cache, and thus their data access latency is smaller (one cycle instead of four cycles). Finally, unlike in a conventional architecture, address computations (in the DDS) and value computations (in the CUs) happen in a truly parallel manner. The second and third benefits allow Mercury to outperform a perfect L1 cache case (i.e., a baseline with an L1 cache that always hits), which only gets the first benefit.

Performance results (multithreaded case). Figure 9 shows Mercury's performance normalized to the baseline CMP (4x OoO cores). For each workload, the first two bars represents Mercury configurations, and the next two bars represents the same configuration with the aggressive memory system, which has higher BW limit and MSHR counts. Here, the first two bars are normalized to the baseline system with a base memory system, whereas the next two bars are normalized to the baseline system with the baseline system with an aggressive memory system. The rightmost bar represents a baseline system with a perfect L1 cache that models an ideal latency-tolerant OoO core or an ideal prefetcher (i.e., which loads all data ahead of time without being limited by the cache size or the available bandwidth). As shown, Mercury with the base memory system achieves significant speedup across all workloads except for compute-bound ones. There are many workloads where Mercury equals or even outperforms the perfect L1 cache results. Furthermore, the Mercury-Shared configuration often matches or even exceeds the performance of Mercury-N, which has 2x more instruction BW and 4x larger IW, and uses 2.5x more area. The Mercury-Shared configuration can achieve even higher performance than Mercury-N when there is interthread data sharing (e.g., `path`, `sgemm`). In `spmv` and `backprop`, Mercury-N outperforms Mercury-Shared by a substantial margin. This is because these two workloads contain indirect memory access patterns that pressure Mercury-Shared's smaller IW.

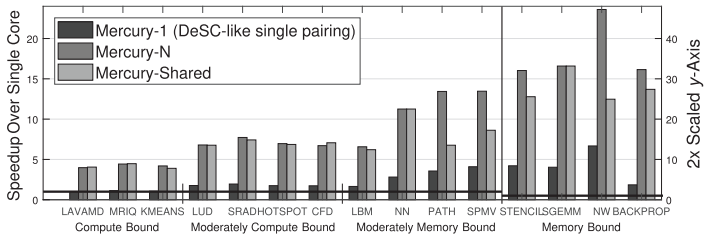


Fig. 11. Performance of Mercury (with an aggressive memory system) over a *single core* for multithreaded workloads. Mercury combines the benefits of parallelism and decoupling (average speedup of 3.73x) to achieve multiplicative speedups (Mercury-N: 15.3x, Mercury-Shared: 12.3x).

When Mercury operates on the more aggressive memory system, Mercury's throughput is not held back by the limited memory BW, and it achieves substantial speedup, especially for memory-bound workloads, with an average speedup of 3.7x on Mercury-N and 2.9x on Mercury-Shared. In almost all workloads, Mercury systems achieve equivalent or even higher performance than what an ideal latency-tolerant core (i.e., perfect L1) can achieve for the reasons outlined at the beginning of this section. An impressive 7.8 to 11x speedup in memory-intensive workloads shows that Mercury has the potential to deliver very high performance when given enough external resources. Considering that recent emerging memory technologies such as HBM or HMC deliver high bandwidth [20, 23], we argue that this is a practical scenario.

Performance results (multiprogrammed case). Figure 10 shows Mercury's performance with multiprogrammed workloads. As in the multithreaded case, Mercury with the base memory system achieves notable speedup across all mixes. Note that overall speedup is larger in multiprogrammed workloads compared to multithreaded ones because there is no synchronization point (or barriers) across threads, which makes Mercury temporarily lose the decoupled distance. In addition, multiprogrammed workloads inherently exhibit imbalance across threads, from which Mercury-Shared benefits. For this reason, despite having more resources and larger area, Mercury-Shared often outperforms the Mercury-N configuration (e.g., `MP3`, `MP5`, `MP7`, `MP8`).

When Mercury is operating on the more aggressive memory system, it achieves significant speedup (i.e., average of 3.7x on Mercury-N and 3.5x on Mercury-Shared), which is even higher than perfect L1 cache speedup in most workloads, as in multithreaded workloads. However, unlike in the base memory system setup, the Mercury-N configuration generally outperforms Mercury-Shared. With its ability to tolerate a large memory latency, a Mercury-Shared DDS unit is now limited by its smaller instruction BW and IW size when operating with the more aggressive memory system.

Comparison with a single core. Finally, with the aim of showing the multiplicative effect of decoupling and parallelism, Figure 11 compares Mercury systems' throughput over a single out-of-order core on aggressive memory systems. In addition, we show the speedup of Mercury-1 to demonstrate how parallelism brings additional benefit in addition to the benefits of decoupling. As shown here, both Mercury-N and Mercury-Shared can achieve greater than 25x throughput improvement compared to a single, conventional core on memory-bound workloads. In such case, around 4x (or more when there is data sharing across threads on read-only data) of the speedup comes from parallelization, and more than 6x speedup comes from decoupling (i.e., memory latency hiding and improving access time for on-chip storage). This shows that Mercury effectively combines the benefits of parallelism along with the use of a decoupled data supply system.

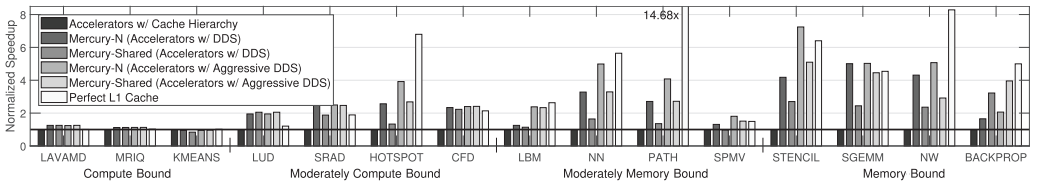


Fig. 12. Speedup of Mercury with accelerator CUs over a set of four accelerators with direct cache hierarchy access. Both Mercury systems (with the aggressive DDS) significantly improve the performance of the baseline four-accelerator system (Mercury-N: 3.11x, Mercury-Shared: 2.61x) by providing memory latency tolerance to their paired accelerator CUs.

Accelerator system results. Figure 12 explores the performance of Mercury configurations (on aggressive memory systems) when the CUs are hardware accelerators. For this experiment, we utilize the approximate model for accelerators proposed in Ham et al. [15], which models accelerators as idealized OoO cores (e.g., large resources, perfect instruction cache and branch predictor) with the ability to execute applications' key loops in parallel. Now, the baseline configuration utilizes four accelerator CUs paired with the cache hierarchy. Mercury configurations use four accelerator CUs but without direct access to memory, paired either with four DDSs (in Mercury-N) or with a single, shared DDS (in Mercury-Shared). The figure also reports each Mercury configuration with a more aggressive DDS design, in the sense that they have 2x larger design parameters (e.g., larger IW, larger ROB, larger fetch/decode/issue BW). Finally, the perfect L1 cache configuration is shown, which represents the baseline system with four accelerator CUs operating with perfect L1 caches.

On average, Mercury-N improves the performance of accelerator CUs by 2.42x, whereas Mercury-Shared improves performance by 1.77x. Particularly, both Mercury configurations are much more effective in memory-bound workloads when compared to stand-alone accelerators directly paired with a cache hierarchy. However, in many workloads, speedup from the Mercury-Shared configuration is limited because the shared DDS has a limited data supply rate due to its limited resources (i.e., fetch/issue BW, IW). One interesting exception is `backprop`. This workload has frequent accesses to shared data across threads and Mercury-Shared benefits from its shared L1 cache and achieves better performance than Mercury-N. With the more aggressive DDS designs, the overall speedup of Mercury configurations improves greatly. On average, the Mercury-N configuration achieves 3.11x speedup and the Mercury-Shared configuration achieves 2.62x speedup over accelerators directly paired with the cache hierarchy. In particular, utilizing the aggressive DDS design improves the performance of Mercury-Shared since it was bottlenecked by the limited data supply throughput of the shared DDS. In most applications, both Mercury-N and Mercury-Shared achieve the performance comparable to accelerators with the perfect L1 cache. Still, in some applications (i.e., `hotspot`, `path`), accelerators with a perfect L1 cache perform better than Mercury configurations because such applications include a substantial amount of address computation that is better handled in accelerators than in DDSs.

5.3 Comparing Mercury-N and Mercury-Shared

Overall, the previous results show that Mercury-N and Mercury-Shared can both work as a building block for larger parallel systems. Such systems can contain multiple instances of Mercury-N and Mercury-Shared together to achieve even larger heterogeneous parallelism. Rather than relying on one particular design, it is important to judiciously utilize both configurations depending on the target since both Mercury-N and Mercury-Shared have advantages and disadvantages as we discuss next.

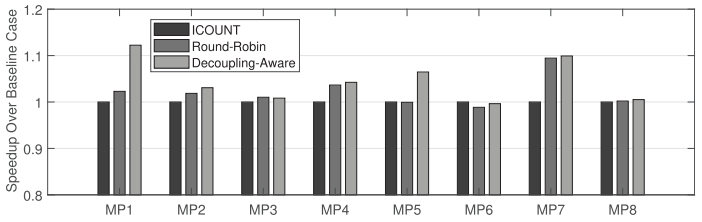


Fig. 13. Effect of fetch policies for Mercury-Shared. The proposed fetch policy provides moderate speedup (e.g., 10%) on workloads with a mismatch between the data supply rate and the data consumption rate.

Mercury-N's main advantage is that it provides a simpler, modular design that can deal with a variety of scenarios. Its intuitive nature allows for easy deployment, and it scales relatively well as long as an application has enough TLP. Still, this approach can result in capability underutilization when there is a mismatch between a DDS unit's data supply rate and its paired CU's data consumption rate. As a result, it utilizes more resources compared to Mercury-Shared while achieving similar performance in many workloads or circumstances, such as when it is limited by system memory bandwidth (e.g., Mercury performance with the base memory system in Figures 9 and 10). In addition, this approach is not well suited for the case when CUs cannot be designed at a finer granularity.

However, Mercury-Shared avoids the drawbacks of Mercury-N by utilizing an SMT-based shared DDS unit. In many cases, particularly in scenarios where each access thread is running different applications with varying data consumption rates, Mercury-Shared can perform equal to or better than Mercury-N (see Figure 10 with the base memory system). Of course, Mercury-Shared can perform worse in certain situations (i.e., no data supply/consumption rate mismatch, few DDS stalls, or aggressive memory systems) where its shared instruction BW or IW can work as bottlenecks (e.g., the Mercury-Shared results with the more aggressive memory system in Figure 9).

5.4 Effects of Optimizations

Fetch policy effectiveness. Figure 13 highlights how Mercury-Shared performance changes with varying fetch policies. Performance is normalized to the same system's performance case with the `ICOUNT` policy as the baseline. The decoupled data supply system performance is not only dependent on the amount of resources a thread has but also depends on the data supply rate and the data consumption rate. When the CU's data consumption rate is low and thus the data buffer is almost full, since access threads have already supplied many data, allocating more resources to this access thread does not achieve any speedup. Our proposed decoupling-aware fetch policy (Section 3.3) prioritizes those threads having low data buffer occupancy and deprioritizes threads having a high data buffer occupancy. The proposed decoupling-aware policy works better than the base `ICOUNT` when the workload has applications whose data buffer occupancy goes above or below those thresholds during the execution. Overall, it achieves over 10% speedup in MP1 and MP7. The policy almost always performs better than `ICOUNT` because it conservatively falls back to the `ICOUNT` policy when there is no thread having high or low data buffer occupancy.

MDI-B effectiveness. Figure 14 shows how the use of the MDI-B improves performance on workloads that suffer from indirect memory accesses. In our evaluated workloads, there are two that exhibit frequent indirect accesses: `backprop` and `spmv`. However, the fact that we only have two workloads that show an indirect access pattern does not mean that indirect load patterns are not popular in the real world. In fact, they are one of the *key access patterns* in important application

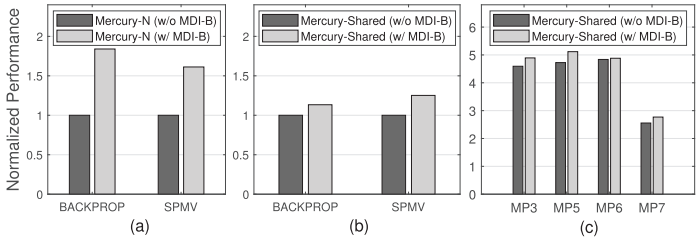


Fig. 14. Effect of the MDI-B on parallel workloads with indirect loads: Mercury-N (a), Mercury-Shared (b), and Mercury-Shared (multiprogrammed) (c). The MDI-B provides *additional* 61% to 83% speedup to Mercury-N systems.

domains that include data mining, machine learning, graph analytic, and so forth. As shown in Figure 14(a), the use of the MDI-B improves `backprop`'s performance by around 83% and `spmv`'s performance by 61% for the Mercury-N configuration when compared to Mercury-N without the MDI-B extension (but with all other optimizations). However, for Mercury-Shared (Figure 14(b)), the improvement is less. This is because Mercury-Shared has a natural ability to still run other threads while one is suffering from indirect load access latency. Figure 14(c) supports this by showing that multiprogrammed workloads, including `backprop`, do not suffer a noticeable performance degradation without the MDI-B approach.

6 RELATED WORK

Latency-tolerant architectures. The kilo-instruction processor [10], Bolt [21], waiting instruction buffer [31], continual flow pipeline [46], EMC [17], and several other previous works [1, 6, 40, 41] explored the potential of migration or early retirement of long latency loads and their dependents from the ROB or issue queue. While sharing the same motivation, our proposed MDI-B extension exploits a unique opportunity presented in a DAE architecture to achieve a similar benefit at a much lower complexity.

Execution-based prefetching techniques. Execution-based prefetching techniques [5, 13, 18, 28, 30, 34, 35, 39, 42, 48, 54–56] are often related to decoupled execution. The common key idea of such schemes is simple: construct a thread by hand, compiler, or hardware, and then let this constructed thread run on a separate processor, core, or even on the same core to let this helper (or precomputation) thread fetch data into nearer storage (e.g., cache) ahead of compute time. Although the aforementioned prior work is effective for providing extra latency tolerance for conventional, general-purpose cores, such approaches are not suitable for accelerator-oriented heterogeneous systems for a few reasons: (a) some techniques only provide a subset of data that a loosely connected accelerator without direct access to main memory needs, (b) some approaches are speculative and thus waste the limited on-chip storage by supplying excessive data that ends up not being used, and (c) some of such proposals are designed for multicore systems where all cores have the same capabilities (e.g., all of them with access to the memory system). Alternatively, Mercury envisions efficient data supply for parallel, heterogeneous architectures where the CUs can be accelerators without the ability to directly access the memory hierarchy.

Other prefetching techniques. Stride-based prefetchers [14, 27, 38] and correlation-based prefetchers [19, 24, 26, 36] are widely used to predict and prefetch the next data to be accessed with a minimal amount of computation. Such approaches often perform the less amount of computation when compared to the execution-based prefetching techniques, and thus their implementation tends to be simpler; however, they tend to be much more speculative and less accurate. In

addition, they are not suitable for accelerator-oriented heterogeneous systems because (a) they only fetch a subset of the necessary data and (b) likely to fetch unnecessary data and waste the limited on-chip storage and the off-chip bandwidth.

Hybrid core design. Some other recent works propose to utilize multiple different core microarchitectures for higher performance or energy efficiency. For example, MorphCore [29] can operate as both an OoO core or a SMT core. Shelf [43] utilizes an in-order pipeline within an OoO core for higher efficiency, whereas Outrider [9] utilizes a SMT core and decoupled execution to achieve higher memory latency tolerance.

Parallel configurable heterogeneous architectures. The Widget architecture [51] proposes utilizing a sea of fine-grain resources for power-proportional computing. The key intuition is that allowing finer-grain hardware elements to work together to achieve higher performance allows for more efficient computing. Although at a different scale, Mercury shares the same insight and advocates configurable parallelism.

7 CONCLUSION

To summarize, this work has taken promising decoupled data supply work from the single-threaded one-to-one pairing world into the parallel world. In doing so, it offers an opportunity for parallel workloads to gain speedup from reducing or mitigating exposed memory latency *in addition to* speedup from parallelism itself. Our approaches offer greater than 3.7x average speedup with Mercury-N and 2.9x speedup with Mercury-Shared, but this effect is multiplied by other forms of parallelism achieved. As a result, a Mercury-N or Mercury-Shared configuration, where four OoO cores work as CUs, can accelerate memory-bound algorithms (i.e., `stencil`, `sgemm`, `nw`, `backprop`) by more than 25x compared to a single core. Mercury's decoupled data supply offers the advantages of high programmability, modular design, and good speedup potential for many accelerator-oriented systems. Going forward, as heterogeneous approaches become even more common, decoupled data supply approaches must play an important role in managing the challenges of an effective data supply to accelerators.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and suggestions.

REFERENCES

[1] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. 2003. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO'03)*. http://dl.acm.org/citation.cfm?id=956417.956554

[2] Peter Bird, Alasdair Rawsthorne, and Nigel Topham. 1993. The effectiveness of decoupling. In *Proceedings of the 7th International Conference on Supercomputing (ICS'93)*. http://doi.acm.org/10.1145/165939.165952

[3] 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 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'11)*. http://doi.acm.org/10.1145/2063384.2063454

[4] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization* 11, 3 (2014), 23.

[5] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. 1999. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*. 10. https://doi.org/10.1145/300979.300995

[6] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, et al. 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)*. http://doi.acm.org/10.1145/1555754.1555814

[7] 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 International Symposium on Workload Characterization (IISWC'09)*. http://dx.doi.org/10.1109/IISWC.2009.5306797

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

[9] 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'11)*. http://doi.acm.org/10.1145/2000064.2000079

[10] Adrián Cristal, Oliverio J. Santana, Mateo Valero, and José F. Martínez. 2004. Toward kilo-instruction processors. *ACM Transactions on Architecture and Code Optimization* 1, 4 (2004), 389–417. http://doi.acm.org/10.1145/1044823.1044825

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

[12] Stijn Eyerman and Lieven Eeckhout. 2014. Restating the case for weighted-IPC metrics to evaluate multiprogram workload performance. *IEEE Computer Architecture Letters* 13, 2 (July 2014), 93–96. https://doi.org/10.1109/L-CA.2013.9

[13] Alok Garg and Michael C. Huang. 2008. A performance-correctness explicitly-decoupled architecture. In *Proceedings of the 41st Annual International Symposium on Microarchitecture (MICRO'08)*. http://dx.doi.org/10.1109/MICRO.2008.4771800

[14] J. D. Gindele. 1977. *Buffer Block Prefetching Method*. IBM.

[15] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *Proceedings of the 48th Annual International Symposium on Microarchitecture (MICRO'15)*. http://doi.acm.org/10.1145/2830772.2830800

[16] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2017. Decoupling data supply from computation for latency-tolerant communication in heterogeneous architectures. *ACM Transactions on Architecture and Code Optimization* 14, 2 (June 2017), Article 16, 27 pages. https://doi.org/10.1145/3075620

[17] Milad Hashemi, Khubaib, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2016. Accelerating dependent cache misses with an enhanced memory controller. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*.

[18] Milad Hashemi, Onur Mutlu, and Yale N. Patt. 2016. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 12. http://dl.acm.org/citation.cfm?id=3195638.3195712

[19] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, et al. 2018. Learning memory access patterns. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*.

[20] AMD. 2015. High-Bandwidth Memory (HBM). Retrieved March 22, 2019 from https://www.amd.com/Documents/High-Bandwidth-Memory-HBM.pdf.

[21] A. Hilton and A. Roth. 2010. BOLT: Energy-efficient out-of-order latency-tolerant execution. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA'10)*. DOI: https://doi.org/10.1109/HPCA.2010.5416634

[22] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. 2015. Efficient execution of memory access phases using dataflow specialization. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. 13. https://doi.org/10.1145/2749469.2750390

[23] Hybrid Memory Cube Consortium. 2018. Hybrid Memory Cube (HMC). Retrieved March 22, 2019 from http://hybridmemorycube.org.

[24] Akanksha Jain and Calvin Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*.

[25] 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 Annual International Symposium on Code Generation and Optimization (CGO'14)*. Article 262, 11 pages. https://doi.org/10.1145/2544137.2544161

[26] Doug Joseph and Dirk Grunwald. 1997. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*.

[27] N. P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*. 364–373.

[28] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. 2011. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. https://doi.org/10.1145/1950365.1950411

[29] Khubaib, M. Aater Suleman, Milad Hashemi, Chris Wilkerson, and Yale N. Patt. 2012. MorphCore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP. In *Proceedings of the 45th Annual International Symposium on Microarchitecture (MICRO'12)*. 12. https://doi.org/10.1109/MICRO.2012.36

[30] Dongkeun Kim and Donald Yeung. 2002. Design and evaluation of compiler algorithms for pre-execution. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*. https://doi.org/10.1145/605397.605415

[31] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. 2002. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*. 12. http://dl.acm.org/citation.cfm?id=545215.545223

[32] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual International Symposium on Microarchitecture*.

[33] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'11)*.

[34] 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 International Symposium on Microarchitecture (MICRO'05)*. http://dx.doi.org/10.1109/MICRO.2005.18

[35] 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 Annual International Symposium on High-Performance Computer Architecture (HPCA'03)*. http://dl.acm.org/citation.cfm?id=822080.822823

[36] Kyle J. Nesbit and James E. Smith. 2004. Data cache prefetching using a global history buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA'04)*.

[37] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. https://doi.org/10.1145/3079856.3080255

[38] S. Palacharla and R. E. Kessler. 1994. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*.

[39] R. Parihar and M. C. Huang. 2017. DRUT: An efficient turbo boost solution via load balancing in decoupled look-ahead architecture. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT'17)*. 91–104. https://doi.org/10.1109/PACT.2017.35

[40] 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)*.

[41] Miquel Pericas, Adrian Cristal, Ruben González, Daniel Jiménez, and Mateo Valero. 2006. A decoupled KILO-instruction processor. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture (HPCA'06)*.

[42] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. 2004. Decoupled software pipelining with the synchronization array. In *Proceedings of 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*. http://dx.doi.org/10.1109/PACT.2004.14

[43] Faissal M. Sleiman and Thomas F. Wenisch. 2016. Efficiently scaling out-of-order cores for simultaneous multithreading. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. 13. https://doi.org/10.1109/ISCA.2016.45

[44] James E. Smith. 1982. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture (ISCA'82)*. 8. http://dl.acm.org/citation.cfm?id=800048.801719

[45] James E. Smith. 1984. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems* 2, 4 (1984), 289–308. http://doi.acm.org/10.1145/357401.357403

[46] 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)*. http://doi.acm.org/10.1145/1024393.1024407

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

[48] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. 2000. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*. http://doi.acm.org/10.1145/378993.379247

[49] Nigel Topham, Alasdair Rawsthorne, Callum McLean, Muriel Mewissen, and Peter Bird. 1995. Compiling and optimizing for decoupled architectures. In *Proceedings of the Conference on Supercomputing (SC'95)*. 40. http://doi.acm.org/10.1145/224170.224301

[50] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA'96)*. 12. https://doi.org/10.1145/232973.232993

[51] Yasuko Watanabe, John D. Davis, and David A. Wood. 2010. WiDGET: Wisconsin decoupled grid execution tiles. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. 12. https://doi.org/10.1145/1815961.1815965

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

[53] William A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News* 23, 1 (March 1995), 20–24.

[54] Weifeng Zhang, Dean M. Tullsen, and Brad Calder. 2007. Accelerating and adapting precomputation threads for effcient prefetching. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA'07)*. http://dx.doi.org/10.1109/HPCA.2007.346187

[55] 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)*. http://dx.doi.org/10.1109/PACT.2005.18

[56] Craig Zilles and Gurindar Sohi. 2001. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'56)*. 12. https://doi.org/10.1145/379240.379246

Received September 2018; revised December 2018; accepted January 2019