Heterogeneous Von Neumann/Dataflow Microprocessors

By Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam

Abstract

General-purpose processors (GPPs), which traditionally rely on a Von Neumann-based execution model, incur burdensome power overheads, largely due to the need to dynamically extract parallelism and maintain precise state. Further, it is extremely difficult to improve their performance without increasing energy usage. Decades-old *explicit-dataflow* architectures eliminate many Von Neumann overheads, but have not been successful as stand-alone alternatives because of poor performance on certain workloads, due to insufficient control speculation and communication overheads.

We observe a synergy between out-of-order (OOO) and explicit-dataflow processors, whereby dynamically switching between them according to the behavior of program phases can greatly improve performance and energy efficiency. This work studies the potential of such a paradigm of heterogeneous execution models, by developing a specialization engine for explicit-dataflow (SEED) and integrating it with a standard out-of-order (OOO) core. When integrated with a dual-issue OOO, it becomes both faster (1.33×) and dramatically more energy efficient (1.70×). Integrated with an in-order core, it becomes faster than even a dual-issue OOO, with twice the energy efficiency.

1. INTRODUCTION

As transistor scaling trends continue to worsen, power limitations make improving the performance and energy efficiency of general purpose processors (GPPs) ever more intractable. The status quo approach of scaling processor structures consumes too much power to be worth the marginal improvements in performance. On top of these challenges, a series of recent microarchitecture level vulnerabilities (Meltdown and Spectre⁹) exploit the underlying techniques which modern processors already rely on for exploiting instruction-level parallelism (ILP).

Fundamental to these issues is the Von Neumann execution model adopted by modern GPPs. To make the contract between the program and the hardware simple, a Von Neumann machine logically executes instructions in the order specified by the program, and dependences are implicit through the names of storage locations (registers and memory addresses). However, this has the consequence that exploiting ILP effectively requires sophisticated techniques. Specifically, it requires (1) dynamic discovery of register/memory dependences, (2) speculative execution past unresolved control flow instructions, and (3) maintenance of the precise program state at each dynamic instruction should it be need to be recovered (e.g., an exception due to a context switch).

The above techniques are the heart of modern Von Neumann out-of-order (OOO) processors, and each technique

requires significant hardware overhead (register renaming, instruction wakeup, reorder-buffer maintenance, speculation recovery, etc.). In addition, the instruction-by-instruction execution incurs considerable energy overheads in pipeline processing (fetch, decode, commit, etc.). As for security, the class of vulnerabilities known as Meltdown and Spectre all make use of speculative execution of one form or another, adding another reason to find an alternative.

Interestingly, there exists a well-known class of architectures that mitigate much of the above called *explicit-dataflow* (e.g., Tagged Token Dataflow,¹ TRIPS,³ WaveScalar²⁰). Figure 1 shows that the defining characteristic of this execution model is how it encodes both control and data dependences explicitly, and the dynamic instructions are ordered by these dependences rather than a total order. Thus, a precise program state is not maintained at every instruction. The benefit is extremely cheap exploitation of instruction-level parallelism in hardware, because no dynamic dependence construction is required.

However, explicit-dataflow architectures show no signs of replacing conventional GPPs for at least three reasons. First, control speculation is limited by the difficulty of implementing efficient dataflow-based squashing. Second, the latency cost of explicit data communication can be prohibitive.² Third, compilation challenges for general workloads have proven hard to surmount.⁵ Although a dataflow-based execution model may help many workloads, it can also significantly hamper others.

Unexplored opportunity: What is unexplored so far is the fine-grained interleaving of explicit-dataflow with Von Neumann execution—that is, the theoretical and practical limits of being able to switch with low cost between an explicit-dataflow hardware/ISA and a Von Neumann ISA. Figure 2(a) shows a logical view of such a heterogeneous architecture, and Figure 2(b) shows the capability of this architecture to exploit fine-grain (thousands to millions of instructions) application phases. This is interesting now, as

Figure 1. Von Neumann vs. dataflow at a glance.

The original version of this paper is entitled “Exploring the Potential of Heterogeneous Von Neumann/Dataflow Execution Models” and was published in ISCA 2015.

Figure 2. Taking advantage of dynamic program behavior.

trends mean that on-chip power is more limited than area; this creates “dark-silicon,” portions of the chip that cannot be kept active due to power constraints. The two major implications are that energy efficiency is the key to improving scalable performance, and that it becomes rationale to add specialized hardware which is only in-use when profitable.

With such a hardware organization, many open questions arise: Are the benefits of fine-grained interleaving of execution models significant enough? How might one build a practical and small footprint dataflow engine capable of serving as an offload engine? Which types of GPP cores can get substantial benefits? Why are certain program region-types suitable for explicit-dataflow execution?

To answer these questions we make the following contributions. Most importantly, we identify (and quantify) the potential of switching between OOO and explicit-dataflow at a fine grain. Next, we develop a specialization engine for explicit-dataflow (SEED) by combining known dataflow-architecture techniques, and specializing the design for program characteristics where explicit-dataflow excels as well as simplifying and common program structures (loops/nested loops). We evaluate the benefits through a design-space exploration, integrating SEED into little (in-order), medium (OOO2), and big (OOO4) cores. Our results demonstrate large energy benefits over $>1.5\times$, and speedups of $1.67\times$, $1.33\times$, and $1.14\times$ across little, medium, and big cores. Finally, our analysis illuminates the relationship between workload properties and dataflow profitability: code with high memory parallelism, instruction parallelism, and control noncriticality is highly profitable for dataflow execution. These are common properties for many emerging workloads in machine learning and data processing.

2. UNDERSTANDING VON NEUMANN/DATAFLOW SYNERGY

Understanding the trade-offs between a Von Neumann machine, which reorders instructions implicitly, and a dataflow machine, which executes instructions in dependence order, can be subtle. Yet, the trade-offs have profound implications. We attempt to distill the intuition and quantitative potential of a heterogeneous core as follows.

2.1. Intuition for execution model affinity

The intuitive trade-off between the two execution models is that explicit-dataflow is more easily specializable for high issue width and instruction window size (due to lack of need to discover dependences dynamically), whereas an implicit-dataflow architecture is more easily specializable for speculation (due to its maintenance of precise state of all dynamic instructions in total program order).

Figure 3. Von Neumann and dataflow execution models.

The performance implications can be seen in an example in Figure 3(a), which has a single control decision labeled as `if`. In (b), we show the program instruction order for one iteration of this code, assuming the left branch was taken. Figure 3(c) shows the ideal schedule of these instructions on an ideal machine (one instruction per cycle). The key to the ideal execution is both the reordering of dependent instructions (c, d) before the control decision is resolved, as well as being able to execute many instructions in parallel.

A Von Neumann OOO machine has the advantage of speculative execution, but the disadvantage is the complexity of implementing hardware for issuing multiple instructions per cycle (issue width) when the dependences are determined dynamically. Therefore, (d) shows how a dual-issue OOO takes five cycles because there was not enough issue bandwidth for both (d) and (h) before the third cycle.

A dataflow processor can easily be designed for high issue width due to dependences being explicitly encoded into the program representation. However, we assume here that the dataflow processor does not perform speculation, because of the difficulty of recovering when a precise order is not maintained. Therefore, in Figure 3(e), the dataflow processor’s schedule, (c) and (d); must execute after the `if`.

Although the example suggests the benefits of control specialization and wide issue widths are similar, in practice, the differences can be stark, which we can demonstrate with slight modifications to the example. If we add several instructions to the critical path of the control decision (between (b) and (if)), the OOO core can hide these through control speculation. If instead we add more parallel instructions, the explicit-dataflow processor can execute these in parallel, whereas these may be serialized in the OOO Von Neumann machine. Explicit-dataflow can also be beneficial if the `if` is unpredictable, and the OOO is anyway serialized.

2.2. Quantitative potential

A natural next question is how much potential benefit could a heterogeneous Von Neumann/dataflow core provide. The potential benefits of an *ideal* hybrid architecture (ideal dataflow + four-wide OOO) relative to a standard OOO core are as shown in Figure 4(a), which where each speedup bar is labeled with the percentage of execution time that dataflow execution is profitable. Figure 4(b) shows the overall energy and performance trends for three different GPPs.

These results indicate that dataflow specialization has significant potential, up to 1.5 $\times$ performance for an OOO4 GPP (2 $\times$ for OOO2), as well as over 2 $\times$ average energy-efficiency improvement. Furthermore, the preference for explicit-dataflow is frequent, covering around 65% of execution time, but also intermittent and application-phase dependent. The percentage of execution time in dataflow mode varies greatly, often between 20% and 80%, suggesting that phase types can exist at a fine grain inside an application.

Overall, this suggests that a heterogeneous Von Neumann/explicit-dataflow architecture with fine-granularity switching can provide significant performance improvements along with power reduction, and thus lower energy.

Remaining challenge: Although many high-performance explicit-dataflow architectures have been proposed over the last several decades, the remaining challenge is how to achieve the same benefits while avoiding a more heavyweight general-purpose explicit-dataflow engine (for example, WaveScalar²⁰ or TRIPS,³ see Figure 5). The approach we will take is to combine known dataflow mechanisms, while simplifying and specializing for the common workload characteristics where dataflow excels.

3. SEED: AN ARCHITECTURE FOR FINE-GRAIN DATAFLOW SPECIALIZATION

Based on our previous analysis and insights, there are three primary requirements for a dataflow specialization engine: (1) low area and power, so integration with the GPP is feasible; (2) enough generality to target a wide variety of workloads; and (3) achieving the benefits of dataflow execution with few overheads.

The dataflow processor is only constrained by the program's control and data-dependencies, but retains the same memory system. Note that *no* nonlocal program modifications

are performed (no loop reordering/tiling/layout-transforms/etc.). It is also nonspeculative and incurs latency when transferring values between control regions. For energy, only functional units and caches are considered.

First, we propose that requirement 1, low area and power, can be addressed by focusing on a common, yet simplifying case: *fully-inlined loops and nested loops* with a limited total static instruction count. This helps limit the size of the dataflow tags and eliminates the need for an instruction cache; both of which reduce hardware complexity. In addition, ignoring recursive regions and only allowing in-flight instructions from a single context eliminates the need for tag matching hardware. Targeting nested-loops also satisfies requirement 2: these regions can cover a majority of real applications' dynamic instructions.

For low-overhead dataflow execution, requirement 3, communication must be lowered while maintaining parallelism. For this, we first use a *distributed-issue* architecture, which enables high-instruction throughput with *low-ported RAM* structures. Second, we use a *multibus network* for sustaining instruction communication throughput at low latency. Third, we use *compound instructions* to reduce communication overhead. The proposed design is SEED: specialization engine for explicit-dataflow, shown at a high level in Figure 6, and explained next.

Figure 5. Relationship to dataflow architectures.

Are prior dataflow architectures sufficient?

Two reasons motivate innovation beyond existing dataflow architectures for the heterogeneous core. First, most prior dataflow architectures have significant area and power overheads, because they are targeted at whole-program execution and must handle arbitrary code. For example, TRIPS³ uses a dynamically routed mesh network to exploit many different forms of parallelism. WaveScalar²⁰ uses large hierarchical interconnects and complex tag-matching, in part because it needs to disambiguate instructions from multiple function contexts. Second, their designs do not consider the costs of switching at low-overhead (for example, not relying on prediction state that requires warm-up). On the other hand, existing in-core accelerators that act as offload engines have much lower power and area, but are not general enough. None of them can offload entire loop regions in general—only the computation in CCA⁴ and DySER⁶ or hot loop-traces in BERET.⁷

How are prior dataflow techniques used?

SEED is highly inspired by previous decades of dataflow research. For example, Monsoon¹⁹ improves the efficiency of matching operands using an Explicit Token Store, a concept we borrow for SEED's explicit operand buffer. The mechanisms that SEED uses for efficient and general dataflow-based control are derived from WaveScalar,²⁰ the concept of a coprocessor,¹² and the concept of efficient compound FUs from BERET.⁷

Figure 4. Ideal dataflow specialization potential.

3.1. Von Neumann core integration

Adaptive execution: To adaptively apply explicit-dataflow specialization, we use a technique similar to bigLITTLE, except that we restrict the entry points of specializable regions to fully-inlined loops or nested loops. This simplifies integration with a different ISA. Targeting longer nested-loop regions reduces the cost of configuration and GPP core synchronization.

GPP integration: SEED uses the same cache hierarchy as the GPP, which facilitates fast switching (no data-copying

Figure 6. High-level SEED integration and organization (IMU: instruction management unit; CFU: compound functional unit; ODU: output distribution unit).

through memory) and preserves cache coherence. SEED adds architectural state, which must be maintained at context switches. Lastly, functional units (FUs) could be shared with the GPP to save area (by adding bypass paths); this work considers stand-alone FUs.

3.2. Dataflow execution model

SEED's execution model closely resembles prior dataflow architectures, but is restricted for loops/nested-loops, and adds the use of compound instructions.

We use a running example to aid explanation: a simple linked-list traversal where a conditional computation is performed at each node. Figure 7(a) shows the original program, (b) the Von Neumann control flow graph (CFG) representation, and (c) SEED's explicit-dataflow representation.

Data-dependence: Similar to other dataflow representations, SEED programs follow the dataflow firing rule: instructions execute when their operands are ready. To initiate computation, live-in values are sent from the host. During dataflow execution, each instruction forwards its outputs to dependent instructions, either in the same iteration (solid line in Figure 7(c)), or in a subsequent iteration (dotted line). For example, the `a_next` value loaded from memory is passed on to the next iteration for address computation.

Control-flow strategy: Control dependencies between instructions are converted into data dependencies. SEED uses a `switch` instruction, which forwards values to one of two possible destinations depending on the input control signal. In the example, depending on the `n_val` comparison, `v2` is forwarded to either the `if` or `else` branch. This strategy enables control-equivalent regions to execute in parallel.

Figure 7. (a) Example C loop; (b) control flow graph (CFG); (c) SEED program representation.

Enforcing memory-ordering: SEED uses a primarily software approach to enforce memory-ordering. When the compiler identifies dependent (or aliasing) instructions, the program serializes these through explicit tokens. In this example, the stores of `n_val` can conflict with the load from the next iteration (e.g., when the linked list contains a loop), and therefore, memory dependence edges are added.

Executing compound instructions: To mitigate communication overheads, the compiler groups primitive instructions (e.g., adds, shifts, switches, etc.) into subgraphs and executes them on compound functional units (CFUs). These are logically executed atomically. The example program contains four subgraphs, mapped to two CFUs.

3.3. SEED microarchitecture

SEED achieves high instruction parallelism and simplicity by using eight distributed computation units. Each of these *SEED units* is organized around one CFU, and units communicate together over a network, as shown in Figure 6.

Compound functional unit (CFU): CFUs are composed of a fixed network of primitive FUs (adders, multipliers, logical units, switch units, etc.), where unused portions of the CFU are bypassed when not in use. Long latency instructions (e.g., loads) can be buffered and passed by subsequent instructions. Our design uses the CFU mix from existing work,⁷ where CFUs contain 2–5 operations. CFUs which have memory units will issue load and store requests to the host’s memory management unit. Load requests access a store buffer for store-to-load forwarding.

Instruction management unit (IMU): The IMU has three responsibilities. First, it stores up to 32 compound instructions, each with a maximum of four operands each for up to four dynamic loop iterations (equivalent to a 1024-entry instruction window). Second, it selects instructions with ready operands for execution on the CFU, giving priority to the oldest instruction. Third, the IMU routes incoming values from the network to appropriate storage locations based on the incoming instruction tag.

Communication: The ODU is responsible for distributing the output values and destination packets (SEED unit + instruction location + iteration offset), to the bus network, and buffering them during bus conflicts. A bus interconnect forwards output packets from the ODU to SEED unit IMU’s which use the corresponding operands. Therefore, dependent instructions communicating over the bus cannot execute in back-to-back cycles, a limitation of distributed dataflow.

4. SEED COMPILER DESIGN

The two main responsibilities of the compiler are determining which regions to specialize and scheduling instructions into CFUs inside SEED regions.

Region selection: The compiler must find or create fully-inlined nested-loop regions, which are small enough to match SEED’s operand/instruction storage. Also, the inner loop should be unrolled for instruction parallelism. An Amdahl-tree based approach can be used to select regions.¹⁶ Also, we should avoid regions where the OOO core (through control speculation) or the SIMD units would have performed

better. One approach is to use simple heuristics, for example, avoid control-critical regions. A dynamic approach can be more flexible; for example, training online predictors to give a runtime performance estimate based on per-region statistics. Related work explores this in detail,^{16, 18} and this work simply uses a static oracle scheduler (see Section 5).

Instruction scheduling: The instruction scheduler forms compound instructions and assigns them to units. Its job is to balance communication cost by creating large compound instructions, while also ensuring that combining instructions does not artificially increase the critical path length. To solve this, we use integer linear programming, specifically extending a general scheduling framework for spatial architectures¹⁷ with the ability to model instruction bundling.

5. EVALUATION METHODOLOGY

For evaluating SEED, OOO core specialization techniques, and the other designs we compare to, we employ a TDG-based modeling methodology.¹⁵ We use Mc-PAT¹¹ with 22nm technology to estimate power and area. Von Neumann core configurations are given in Table 1.

The benchmarks we chose were from SPECint and Media-bench,¹⁰ representing a variety of control and memory irregularity, as well as some regular benchmarks. To eliminate compiler/runtime heuristics on when to use which architecture, we use an oracle scheduler, which uses previous runs to decide when to use the OOO core, SEED, or SIMD.

6. EVALUATING DATAFLOW SPECIALIZATION POTENTIAL

To understand the potentials and trade-offs of dataflow specialization, we explore the prevalence of required program structure, per-region performance, and overall heterogeneous core benefits.

6.1. Program structure

Nested loop prevalence: Figure 8 shows cumulative distributions of dynamic instruction coverage with varying dynamic region granularity, assuming maximum 1024 instructions. Considering regions with a duration of 8K dynamic instructions or longer (x-axis), nested loops can cover 60% of total instructions, whereas inner loops cover only 20%. Nested loops also greatly increase the region duration for a given percentage coverage (1K–64K for 40% coverage).

Compound instruction prevalence: Figure 9 is a histogram of per-benchmark compound instruction sizes which the compiler created, showing on average 2–3 instructions. This

Table 1. Von Neuman core configurations.

GPP	Characteristics
Little (IO2)	Dual issue, 1 load/store port.
Medium (OOO2)	64 entry ROB, 32 entry IW, LSQ: 16 ld/20 st, 1 ld/st ports, speculative scheduling.
Big (OOO4)	168 entry ROB, 48 entry IW, LSQ: 64 ld/36 st, 2 ld/st ports, speculative scheduling.
Common	x86 ISA, 256-bit SIMD, 2-way 32KiB I\$, 64KiB L1D\$ (4 cycle latency), 8-way 2MB L2\$ (22 cycle hit latency), 2GHz.

Figure 8. Cumulative % contribution for decreasing dynamic region lengths. Static region size £ 1024 insts.

Figure 9. Compound instruction size histogram.

is relatively high considering that compound instructions cannot cross control regions. Some singletons are necessary; however, either because control regions lack dependent computation, or because combining certain instructions would create additional critical-path dependencies.

6.2. Per-region performance analysis

First, we compare the speedups of SEED to our most aggressive design (OOO4) on a per-region basis. Figure 10 shows SEED's speedup for the recurring nested-loop program regions (each >1% total insts), where the region with the highest contribution to the execution time of the original program is shown in red. Overall, speedup varies dramatically due to the significant differences in program characteristics. Around $3\times$ – $5\times$ speedup is possible, and many regions show significant speedup. We next examine the reasons for performance differences of the highest-contribution regions in several categories, as follows.

Performance and energy benefit regions: Compared to the OOO4-wide core, SEED can provide high speedups by exploiting ILP in compute-intensive regions (e.g., `jpeg2000dec`, `cjpeg`, and `djpeg`) and from using an effectively larger instruction window for achieving higher memory parallelism (e.g., `181.mcf` and `429.mcf`). The latter have high cache miss rates and clog the instruction window

Figure 10. Per-region SEED speedups. Highest-contributing region shown in red.

of the OOO processor. Across these regions, indirect memory access is common, which precludes SIMD vectorization.

Energy benefit-only regions: These regions have similar performance to the OOO4, but are more energy efficient by $2\times$ – $3\times$. Here, ILP tends to be lower, but control is mostly off the critical path, allowing dataflow to compete (e.g., `djpeg-1` and `h264dec`). Although `gsmenc` and `164.gzip` actually have high potential ILP, they are burdened by communication between SEED units. Contrastingly, `473.astar` and `jpeg2000enc` have significant control, but still perform close to the OOO core. These benchmarks make up for the lack of speculation by avoiding branch mispredictions and relying on the dataflow-based control.

Performance loss regions: The most common reason for performance loss is communication latency on the critical path (e.g., `403.gcc`, `mpeg2dec`, and `mpeg2enc`), as well as predictable data-dependent control (e.g., `401.bz2`). These are fundamental dataflow limitations. In two cases, configuration overhead was burdensome (`464.h264ref` and `197.parser`). Finally, some of these regions are vectorized on the GPP, and SEED is not optimized to exploit data-parallelism. In practice, the above regions would not be executed on SEED.

In summary, speedups come from exploiting higher memory parallelism and instruction parallelism, and avoiding mispeculation on unpredictable branches. Slowdowns come from the extra latency cost on more serialized computations.

6.3. Overall performance/energy trade-offs

Finally, we consider the overall performance when integrated with a little, medium, and big core, and compare explicit-dataflow specialization with existing techniques for targeting irregular codes. In-place loop execution, similar to Revolver,⁸ locks looping instructions into the instruction window to prevent redundant pipeline overheads such as fetch/decode/execute, but does not otherwise change the OOO execution model. Conservation cores²¹ use software-defined region-specific accelerators, but they do not exploit dataflow-based control (only in-order control and memory). Figure 11 shows the relative performance and energy benefits, normalized to the in-order core alone.

SEED improves performance and energy efficiency across GPP cores types, significantly more than existing accelerator

Figure 11. Overall performance and energy benefit.

and microarchitectural approaches. For the little, medium, and big cores, SEED provides 1.65 $\times$, 1.33 $\times$, and 1.14 $\times$ speedup, and 1.64 $\times$, 1.7 $\times$, and 1.53 $\times$ energy efficiency, respectively. The energy benefits come primarily from the prevalence of regions where dataflow execution can match the host core's performance; this occurs 71%, 64%, and 42% of the time, for the little, medium, and big Von Neumann cores, respectively.

Understanding disruptive trade-offs: Perhaps more interesting is the disruptive changes that explicit-dataflow specialization introduces for computer architects. First, the OOO2+SEED is actually reasonably close in performance to an OOO4 processor on average, within 15%, while reducing energy 2.3 $\times$. Additionally, our estimates suggest that an OOO2+SEED occupies less area than an OOO4 GPP core. Therefore, a hybrid dataflow system introduces an interesting path toward a high-performance, low-energy microprocessor: start with an easier-to-engineer modest OOO core, and add a simple, nongeneral-purpose dataflow engine.

An equally interesting trade-off is to add a hybrid dataflow unit to a larger OOO core—SEED+OOO4 has much higher energy efficiency (1.54 $\times$) with additional performance improvements of 1.14 $\times$. This is a significant leap for energy-efficiency, especially considering the difficulty of improving the efficiency for complex, irregular workloads such as SpecINT.

Overall, all cores can achieve significant energy benefits, little and medium cores can achieve significant speedup, and big cores receive modest performance improvement.

7. DISCUSSION

Dataflow specialization is a broadly applicable principle for both general-purpose processors and accelerators. We outline our view on the potentially disruptive implications in these areas as well as potential future directions.

7.1. General purpose cores

In this work, we showed how a dataflow processor can more efficiently take over and execute certain phases of application workloads, based on their properties. This can be viewed visually, as shown in Figure 12, where we show architecture affinity for programs along dimensions of control and memory regularity. Figure 12(a) shows how prior programmable specialization techniques only focus

on a narrow range of workloads—for example, SIMD can speedup highly regular program phases ① only.

Figure 12(b) shows how dataflow specialization further cuts into the space of programs that traditional architectures are best at. Specifically, when the OOO processor's issue width and instruction window size limits the achievable ILP (region ③), explicit-dataflow processors can exploit this through distributed dataflow, as well as more efficient execution under control unpredictability (region ④). Beyond these region types, dataflow specialization can be applied to create engines that target other behaviors, such as repeatable control ⑤, or to further improve highly regular regions by combining dataflow with vector-communication ①.

Future directions: The disruptive potential of exploiting common program phase behavior using a heterogeneous dataflow execution model can have significant implications leading to several important directions:

- **Reduced importance of aggressive out-of-order:** Dataflow engines which can exploit high ILP phases can reduce the need for aggressive and power-inefficient out-of-order cores. As a corollary, the design of modest-complexity loosely coupled cores should in principle be less design effort than a complex OOO core. This could lower the cost-of-entry into the general-purpose core market, increasing competition and spurring innovation.
- **Radical departure from status quo:** The simple and modular integration of engines targeting different behaviors, combined with microarchitecture-level dynamic compilation for dataflow ISAs²² can enable such designs to be practical. This opens the potential of exploring designs with radically different microarchitectures and software interfaces, ultimately opening a larger and more exciting design space.
- **An alternative secure processor:** An open question is how to build future secure processors that are immune to attacks such as Meltdown and Spectre.⁹ One approach is to simply avoid speculation; this work shows that an in-order core plus SEED may only lose on average around 20% performance with respect to an OOO core alone, at much lower energy.

7.2. Accelerators

In contrast to general-purpose processors, accelerators are purpose-built chips integrated at a coarse grain with computing systems, for workloads important-enough to the market to justify their design and manufacturing cost. A persistent challenge facing accelerator design is that in order to achieve desired performance and energy efficiency, accelerators often sacrifice generality and programmability, using application or domain-specific software interfaces. Their architecture and microarchitecture is narrowly tailored to the particular domain and problem being solved.

The principle of heterogeneous Von Neumann/dataflow architectures can help to create a highly efficient accelerator without having to give up on domain-generality. Inspired by the insights here, we demonstrated that domain-specific accelerators rely on fundamentally common specialization principles: specialization of computation, communication,

Figure 12. Program phase affinity by application characteristics. Memory ranges from *regular* and data-independent, to *irregular* and data-dependent but with parallelism, to *latency bound* with no parallelism. Control can range from *noncritical* or not present, critical but *repeating*, not repeating but *predictable*, to *unpredictable* and data-dependent.

concurrency, data-reuse, and coordination.¹⁴ A dataflow model of computation is especially suitable for exploiting the first three principles for massive parallel computation, whereas a Von Neumann model excels at the coordination of control decisions and ordering. We further addressed programmable specialization by proposing a Von Neumann/dataflow architecture called stream-dataflow,¹³ which specifies memory access and communication as streams, enabling effective specialization of data-reuse in caches and scratchpad memories.

Future directions: The promise of dataflow specialization in the accelerator context is to enable freedom from application-specific hardware development, leading to two important future directions.

- **An accelerator architecture:** The high energy and area-efficiency of a Von Neumann/dataflow accelerator, coupled with a well-defined hardware/software interface, enables the almost paradoxical concept of an accelerator *architecture*. We envision that a dataflow-specialized ISA such as stream-dataflow, along with essential hardware specialization principles, can serve as the basis for future innovation for specialization architectures. Its high efficiency makes it an excellent baseline comparison design for new accelerators, and the ease of modifying its hardware/software interface can enable integration of novel forms of computation and memory specialization for challenging workload domains.
- **Compilation:** How a given program leverages Von Neumann and dataflow mechanisms can have tremendous influence on attainable efficiency, and some methodology is required to navigate this design space. The fundamental compiler problem remains extracting and expressing parallelism and locality. The execution model and application domains make these problems

easier to address. Applications for which accelerators are amenable are generally well-behaved (keeping to a minimum or avoiding pointers, etc.). The execution model and architecture provides interfaces to cleanly expose the application's parallelism and locality to the hardware. This opens up exciting opportunities in compiler and programming languages research to target accelerators.

8. CONCLUSION

This article observed a synergy between Von Neumann and dataflow processors due to variance in program behaviors at a fine grain and used this insight to build a practical processor, SEED. It enables potentially disruptive performance and energy efficiency trade-offs for general-purpose processors, pushing the boundary of what is possible given only a modestly complex core. This approach of specializing for program behaviors using heterogeneous dataflow architectures could open a new design space, ultimately reducing the importance of aggressive OOO designs and lead to greater opportunity for radical architecture innovation. C

References

1. Arvind, K., Nikhil, R.S. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput.* 39, 3 (1990), 300–318.
2. Budiu, M., Artigas, P.V., Goldstein S.C. Dataflow: A complement to superscalar. In *ISPASS '05 Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (March 20–22, 2005) IEEE Computer Society, Washington, DC, USA, 177–186.
3. Burger, D., Keckler, S.W., McKinley, K.S., Dahlin, M., John, L.K., Lin, C., Moore, C.R., Burrill, J., McDonald, R.G., Yoder, W., Team, T.T. Scaling to the end of silicon with edge architectures. *Computer* 37, 7 (July 2004), 44–55.
4. Clark, N., Kudlur, M., Park, H., Mahlke, S., Flautner, K. Application-specific processing on a general-purpose core via transparent instruction set customization. In *MICRO 37 Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture* (Portland, Oregon, December 04–08, 2004), IEEE Computer Society, Washington, DC, USA, 30–40.
5. Gebhart, M., Maher, B.A., Coons, K.E., Diamond, J., Gratz, P., Marino, M., Ranganathan, N., Robotmili, B., Smith, A., Burrill, J., Keckler, S.W., Burger, D., McKinley, K.S. An evaluation of the trips computer system. In *ASPLOS XIV Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA, March 07–11, 2009), ACM, New York, NY, USA, 1–12.
6. Govindaraju, V., Ho, C.-H., Nowatzki, T., Chhugani, J., Satish, N., Sankaralingam, K.,

- Kim, C. DYSE: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro* 32, 5 (Sept. 2012), 38–51.
7. Gupta, S., Feng, S., Ansari, A., Mahlke, S., August, D. Bundled execution of recurring traces for energy-efficient general purpose processing. In *ASPLOS XIV Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA, March 07–11, 2009). ACM, New York, NY, USA, 1–12.
 8. Hayenga, M., Naresh, V., Lipasti, M. Revolver: Processor architecture for power efficient loop execution. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* (Orlando, FL, USA, 2014). IEEE, 591–602.
 9. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)* (IEEE Computer Society, 2019).
 10. Lee, C., Potkonjak, M., Mangione-Smith, W. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *MICRO 30 Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (Research Triangle Park, North Carolina, USA, December 01–03, 1997), 330–335.
 11. Li, S., Ahn, J.H., Strong, R.D., Brockman, J.B., Tullsen, D.M., Jouppi, N.P. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO 42 Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York, December 12–16, 2009), 469–480.
 12. Liu, Y., Furber S. A low power embedded dataflow coprocessor. In *ISVLSI '05 Proceedings of the IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design* (May 11–12, 2005), 246–247.
 13. Nowatzki, T., Gangadhar, V., Ardalani, N., Sankaralingam, K. Stream-dataflow acceleration. In *ISCA '17 Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada, June 24–28, 2017), 416–429.
 14. Nowatzki, T., Gangadhar, V., Sankaralingam, K., Wright, G. Pushing the limits of accelerator efficiency while retaining programmability. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, (March 12–16, 2016), 27–39.
 15. Nowatzki, T., Govindaraju, V., Sankaralingam, K. A graph-based program representation for analyzing hardware specialization approaches. *Comput. Archit. Lett.* 14, 2 (July-Dec 2015), 94–98.
 16. Nowatzki, T., Sankaralingam, K. Analyzing behavior specialized acceleration. In *ASPLOS '16 Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA, April 02–06, 2016), ACM, New York, NY, USA, 697–711.
 17. Nowatzki, T., Sartin-Tarm, M., De Carli, L., Sankaralingam, K., Estan, C., Robotmili, B. A general constraint-centric scheduling framework for spatial architectures. In *PLDI '13 Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA, June 16–19, 2013), ACM, New York, NY, USA, 495–506.
 18. Padmanabha, S., Lukefahr, A., Das, R., Mahlke, S.A. Trace based phase prediction for tightly-coupled heterogeneous cores. In *MICRO-46 Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California, December 07–11, 2013), ACM, New York, NY, USA, 445–456.
 19. Papadopoulos, G.M. Monsoon: An explicit token-store architecture. In *ISCA '90 Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, USA, May 28–31, 1990), ACM, New York, NY, USA, 82–91.
 20. Swanson, S., Michelson, K., Schwerin, A., Oskin, M. WaveScalar. In *MICRO 36 Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture* (December 03–05, 2003), IEEE Computer Society, Washington, DC, USA, 291.
 21. Venkatesh, G., Sampson, J., Goulding, N., Garcia, S., Bryksin, V., Lugo-Martinez, J., Swanson, S., Taylor, M.B. Conservation cores: Reducing the energy of mature computations. In *ASPLOS XV Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA, March 13–17, 2010), ACM, New York, NY, USA, 205–218.
 22. Watkins, M.A., Nowatzki, T., Carno, A. Software transparent dynamic binary translation for coarse-grain reconfigurable architectures. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (March 12–16, 2016), 138–150.

Tony Nowatzki ([tjn]@cs.ucla.edu),
University of California, Los Angeles,
Los Angeles, CA, USA.

Some work performed at University
of Wisconsin - Madison.

Vinay Gangadhar and Karthikeyan Sankaralingam ([vinay, karu]@cs.wisc.edu),
University of Wisconsin - Madison,
Madison, WI, USA.

© 2019 ACM 0001-0782/19/6 \$15.00

Theory, Systems, and Applications

Declarative Logic Programming

Edited by Michael Kifer & Yanhong Annie Liu

ISBN: 978-1-970001-969 | DOI: 10.1145/3191315

<http://books.acm.org>

<http://www.morganclaypoolpublishers.com/acm>

ACM BOOKS