A Heterogeneous Von Neumann/ Explicit Dataflow Processor

Decades-old explicit dataflow architectures eliminate many of the overheads of general-purpose processors but have not been successful because of their lack of sufficient control speculation and the latency overhead of explicit communication. This article observes a synergy between out-of-order and explicit dataflow processors, in which dynamically switching between them according to program phases can greatly improve performance and energy efficiency.

Tony Nowatzki
Vinay Gangadhar
Karthikeyan Sankaralingam
University of Wisconsin—Madison

As transistor scaling trends worsen, improving the performance and energy efficiency of general-purpose processors (GPPs) has become ever more challenging. Great strides have been made in targeting regular codes through the development of single-instruction, multiple-data (SIMD) architectures and GPUs. However, codes with either irregular control (divergent or unpredictable branches) or irregular memory (noncontiguous or indirect access) still remain problematic.

Primarily, irregular codes are executed on GPPs, which incur considerable overhead in per-instruction processing, both in extracting instruction-level parallelism (ILP) and maintaining instruction-precise state. See the sidebar, "Existing Approaches to Improve General-Purpose Cores' Energy Efficiency," for more information.

However, there exist well-known architectures that eschew complex out-of-order (OoO) hardware structures, yet can extract significant ILP; these are called *explicit dataflow architectures*. These include the early Tagged-Token Dataflow,[1] as well as the more recent Trips,[2] WaveScalar,[3] and Tartan[4] architectures. But explicit dataflow architectures show no signs of replacing conventional GPPs, for at least three reasons. First, control speculation is limited by the difficultly of implementing dataflow-based squashing. Second, the latency cost of explicit data communication can be prohibitive.[5] Third, compilation challenges have proven hard to surmount.

Dataflow machines researched and implemented thus far have failed to provide higher instruction-level parallelism, and their theoretical promise of both low power and high performance remains unrealized for irregular codes.

What remains unexplored thus far is the fine-grained interleaving of explicit dataflow with Von Neumann execution to adapt to changing program behavior—that is, the theoretical and practical limits of being able to switch, with low cost, between an explicit dataflow hardware instruction set architecture (ISA) and a Von Neumann ISA. Figure 1a shows a logical view of such a heterogeneous architecture, and Figure 1b shows this architecture's

Published by the IEEE Computer Society

Existing Approaches to Improve General-Purpose Cores' Energy Efficiency

Two broad specialization approaches have arisen to address general-purpose inefficiencies.

The first is to use simple and serial low-power hardware in commonly used low instruction-level parallelism (ILP) code regions for better energy efficiency. Examples architectures include big.LITTLE[1] and Composite Cores,[2] which switch to an in-order core when ILP is unavailable, and "accelerators" such as Bundled Execution of Recurring Traces (BERET)[3] and Conservation Cores.[4] The latter approaches are effective when integrated to small in-order cores, but they curtail performance too much to be useful for out-of-order (OoO) general-purpose processors (GPPs).

The other approach is to enhance the GPP for energy efficiency—for example, by adding micro-op caches, loop caches, and in-place loop execution techniques such as Revolver.[5] These approaches achieve only modest improvements because they fundamentally retain complex OoO structures such as the instruction window, reorder buffer, and multiported register files.

References

1. P. Greenhalgh, *big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7*, white paper, ARM, 2011.
2. A. Lukefahr et al., "Composite Cores: Pushing Heterogeneity into a Core," *Proc. 45th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2012, pp. 317–328.
3. S. Gupta et al., "Bundled Execution of Recurring Traces for Energy-Efficient General Purpose Processing," *Proc. 44th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2011, pp. 12–23.
4. G. Venkatesh et al., "Conservation Cores: Reducing the Energy of Mature Computations," *Proc. 15th Conf. Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 205–218.
5. M. Hayenga, V.R.K. Naresh, and M.H. Lipasti, "Revolver: Processor Architecture for Power Efficient Loop Execution," *Proc. IEEE 20th Int'l Symp. High Performance Computer Architecture*, 2014, pp. 591–602.

capability to exploit fine-grained (thousands to millions of instructions) application phases.

In this article, we explore the microarchitecture of such a design, and the many questions that 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 substantially benefit? Why are certain program region-types suitable for explicit dataflow execution?

To answer these questions, we first analyze the potential benefits of ideal dataflow heterogeneity. We then design a specialization engine for explicit dataflow (SEED), which is simple but still widely applicable. We perform design space exploration by integrating SEED with little (in-order), medium (OoO2), and big (OoO4) cores, achieving 1.67, 1.33, and 1.14 times speedup, respectively, always with more than 1.5 times energy benefit. Our analysis shows that code with high memory parallelism, instruction parallelism, and branch unpredictability is highly profitable for dataflow execution.

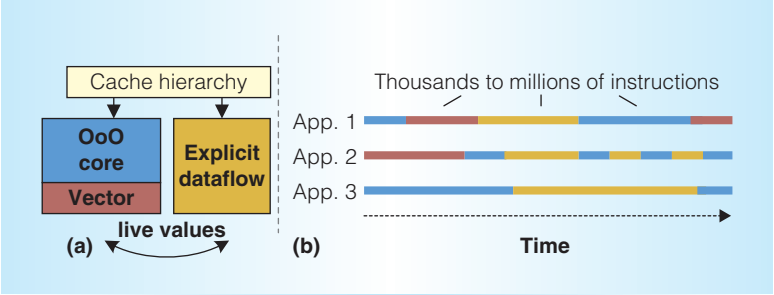

Figure 1. Exploiting dynamic program behavior. (a) The logical view of a heterogeneous architecture, where Von Neumann and dataflow are substrates that are integrated to the same cache hierarchy. (b) The architecture preference changes over time during application execution.

The Potential of Ideal Hybrid Dataflow

In this section, we consider an ideal hybrid dataflow system to understand the potential and the underlying reasons for the benefits. We begin with Figure 2a, which shows the potential speedup of such an architecture (ideal dataflow plus four-wide OoO). Above each bar is the percentage of execution time in dataflow mode. Figure 2b shows the overall energy and performance trends for three GPPs.

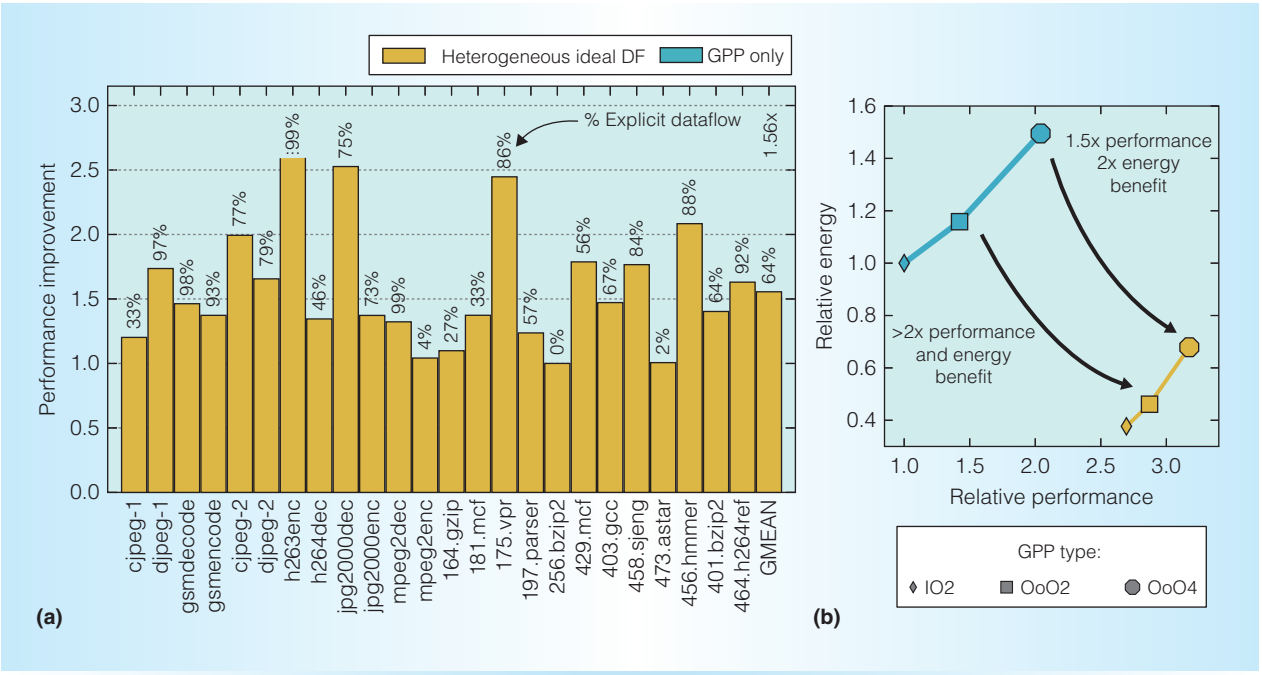

Figure 2. Potential of ideal explicit dataflow specialization: (a) heterogeneous ideal dataflow performance and (b) overall tradeoffs. The "ideal" dataflow processor is constrained only by the program's control and data dependencies, but it is non-control-speculative. For its energy model, only functional units and caches are considered.

These results indicate that hybrid dataflow has significant potential, up to 1.5 times performance for an OoO4 GPP (two times for OoO2), as well as more than two times average energy-efficiency improvement. Furthermore, the preference for explicit dataflow is frequent, covering about 65 percent of execution time, but it is also intermittent and application-phase dependent. The percentage of execution time in dataflow mode varies greatly, often between 20 to 80 percent, suggesting that phase types can exist at a fine grain inside an application.

To understand when and why explicit dataflow can provide benefits, we consider the program space along two dimensions: control regularity and memory regularity. Figure 3 shows our view on how different programs in this space can be executed by other architectures more efficiently than with an OoO core. Naturally, vector architectures are the most effective when memory access and control is highly regular (see label 1 in Figure 3). When programs are memory-latency bound (label 2 in Figure 3), little ILP will be available, and the simplest possible hardware will be the best (for example, a low-power engine like BERET[6] or Conservation Cores[7]). An explicit dataflow engine could also fill this role.

There are two remaining regions in which explicit dataflow has advantages over OoO. First, when the OoO processor's issue width and instruction window size limit the achievable ILP (label 3 in Figure 3), explicit dataflow processors can exploit this through more efficient hardware mechanisms, achieving higher performance and energy efficiency. Second, when control is not predictable, which would serialize the execution of the OoO core (label 4 in Figure 3), explicit dataflow can execute the same code with higher energy efficiency.

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

SEED: An Architecture for Fine-Grained Dataflow Specialization

Our primary observation is that there is potential for exploiting the heterogeneity of execution models between Von Neumann

and dataflow at a fine grain. Attempting to exploit this raises this article's main concern: how can we exploit dataflow specialization with simple, efficient hardware? We argue that any solution requires three properties:

1. It must have low area and power so that integration with the GPP is feasible.
2. It must be general enough to target a wide variety of workloads.
3. It must achieve the benefits of dataflow execution with few overheads.

Our codesign approach involves exploiting properties of frequently executed program regions, using a combination of power-efficient hardware structures and employing a set of compiler techniques.

First, we propose that requirement 1, low area and power, can be addressed by focusing on a common yet simplifying case: fully inlined nested loops with a limited total static instruction count. Limiting the number of per-region static instructions limits 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 allowing only in-flight instructions from a single context eliminates the need for tag matching hardware—direct communication can be used instead. Targeting nested loops also satisfies requirement 2: these regions can cover a majority of real applications' dynamic instructions.

To achieve low-overhead dataflow execution, requirement 3, we must lower the communication costs. We achieve this through a judicious set of microarchitectural features. First, we 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 the data communication overhead.

Using these insights creates two new compiler requirements: to create appropriately sized inlined nested loops matching the hardware constraints, and to create compound instruction groupings that minimize the communication overhead. For the first requirement, we can use aggressive inlining and loop nest analysis, and for the second, we can employ integer linear programming models.

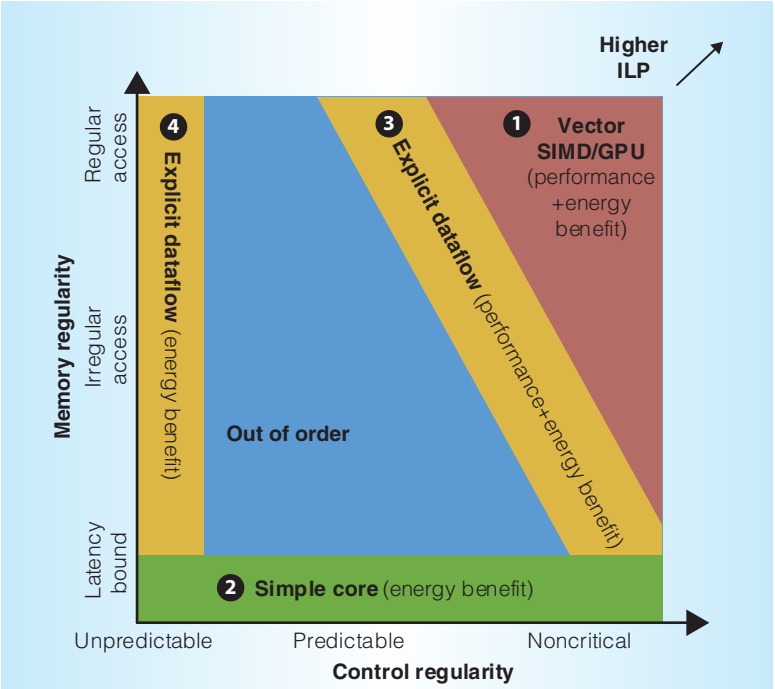

Figure 3. Architectures' effectiveness based on application characteristics of memory and control regularity. Regular program regions (1) are most suited to vector processors, and latency-bound regions (2) are suited to simple processors. In addition to (2), an explicit-dataflow processor is especially apt for regions with high ILP (3) or significant unpredictable control (4).

Execution Model and Core Integration

To address the architecture and compiler challenges, we propose SEED (shown in Figure 4).[8] In this article, we give an overview of SEED's execution model and integration.

Adaptive execution. The model we use for adaptively applying explicit dataflow specialization is similar to big.LITTLE, except that we restrict the entry points of acceleratable regions to fully inlined or nested loops. This lets us target custom hardware with a different ISA, using statically generated instructions. Targeting longer nested-loop regions also leads to a reduced overall cost of configuration and GPP core synchronization.

GPP integration. We integrate the SEED hardware with the same cache hierarchy as the GPP, as shown in Figure 4. This approach facilitates fast switching (no data copying through memory), maintains cache coherence, and eliminates the area of scratchpad

Table 1. Region-wise comparison of OoO4 to SEED,

	Benchmark	SEED region function	Percent of program executed by SEED	Region vectorized by the GPP	OoO4 IPC	Effective IPC of SEED
Perf. > OoO4	jpg2000dec	jas_image_encode	50		2.5	12.8
	429.mcf	primal_bea_mpp	37		0.8	2.8
	cjpeg-1	encode_mcu_AC_refine	24		2.5	5.9
	181.mcf	primal_bea_mpp	31		0.9	1.8
	djpeg-2	ycc_rgb_convert	33		2.7	5.4
	456.hmmer	Viterbi*	73		2.9	5.4
	458.sjeng	std_eval	5		2.4	3.7
	gsmdecode	Gsm_Short_Term_Syn...	61		2.4	3.1
	cjpeg-2	compress_data	48	✓	2.2	2.7
Perf. ≈ OoO4	gsmencode	Gsm_Long_Term_Pred...	49		1.9	2.2
	djpeg-1	decompress_onepass	39		2.6	2.7
	h263enc	MotionEstimation	98		2.0	1.9
	164.gzip	inflate	23		1.9	1.7
	473.astar	wayobj::fill	96		1.1	1.0
	h264dec	decode_one_macroblock	21		0.4	0.4
	jpg2000enc	jpc_enc_encpkt	3		2.1	1.8
Perf. < OoO4	403.gcc	ggc_mark_trees	4		0.5	0.4
	464.h264ref	SetupFastFullPelSearch	29		1.5	1.3
	175.vpr	try_swap	49		1.4	1.2
	mpeg2enc	fullsearch.constprop.3	93		1.9	1.5
	mpeg2dec	conv422to444	31		2.7	2.1
	197.parser	restricted_expression	17		3.3	1.6
	401.bzip2	BZ2_compressBlock	31	✓	4.3	1.5
	2560bzip2	compressStream	99	✓	13.5	2.0

memories and the associated need for programmer intervention. SEED also adds architectural state, which must be maintained at context switches. Moreover, functional units (FUs) could be shared with the GPP to save area (by adding bypass paths); this article considers stand-alone FUs.

Dataflow style. Similar to dataflow architectures like WaveScalar,[3] control dependencies in the original program become data dependencies. The control flow is implemented by forwarding values to the appropriate location, depending on the branch outcomes.

Dataflow Execution

Here, we discuss SEED's internal execution model, which resembles those of previous dataflow architectures,[2–4] in which the primary difference is that SEED's instructions are grouped into subgraphs.

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 or in a

showing only the top region per benchmark, highest to lowest relative performance.

IPC of ideal dataflow	SEED's energy reduction	Branches per 1,000 μops (BPKI)	Branch misprediction per 1,000 μops (BMPKI)	Cache misses per 1,000 μops	Explanation
21.8	9.1	101	0	0	High exploitable ILP
8.3	4.6	152	10	96	Higher memory parallelism
6.2	4.2	48	0	2	Indirect memory and high ILP
9.6	3.0	170	8	106	Higher memory parallelism
12.0	3.5	29	0	0	Indirect memory and high ILP
7.3	4.5	32	0	4	High exploitable ILP
4.1	3.8	126	5	0	High exploitable ILP
3.4	4.5	92	0	0	High exploitable ILP
4.9	3.5	58	8	0	High exploitable ILP
2.7	3.5	5	0	0	Moderate ILP, communication overhead
3.6	3.6	18	1	0	Indirect memory and moderate ILP
8.7	3.2	18	0	0	Comparable performance
2.3	2.0	81	0	10	Moderate ILP, communication overhead
1.1	3.3	114	31	2	Avoids branch misses, modest ILP
0.4	1.9	39	0	0	Comparable, low ILP
2.0	1.3	135	6	2	Comparable performance
0.4	1.0	66	2	2	Comparable, low ILP
1.7	2.7	40	0	0	Short region (340 dynamic instructions)
6.9	2.1	88	17	5	Avoids B-misses, communication overhead
2.9	3.9	17	0	0	Moderate ILP, communication overhead
3.0	2.8	68	0	2	Moderate ILP, communication overhead
3.7	1.4	108	0	0	Short region (300 dynamic instructions)
1.5	0.8	97	3	3	Region vectorized
2.0	0.4	83	0	0	Region vectorized

subsequent iteration. Control dependencies between instructions are converted into data dependencies. SEED uses a switch instruction that forwards values to different instructions. Memory dependencies (aliasing memory instructions) are serialized in SEED's program representation through explicit tokens.

Executing compound instructions. To mitigate communication overheads, the compiler groups primitive instructions (such as adds, shifts, and switches) into subgraphs and executes them on compound functional units (CFUs). These are logically executed atomically.

SEED Microarchitecture

Our microarchitecture achieves high instruction parallelism and simplicity by using distributed computation units. The overall design comprises eight SEED units, wherein each SEED unit is organized around one CFU. The SEED units communicate over a network, as shown in Figure 4.

Compound functional unit. CFUs comprise a fixed network of primitive FUs (such as adders, multipliers, logical units, and switch units) in which unused portions of the CFU are bypassed when not in use. Long-latency instructions (for example, loads) can be

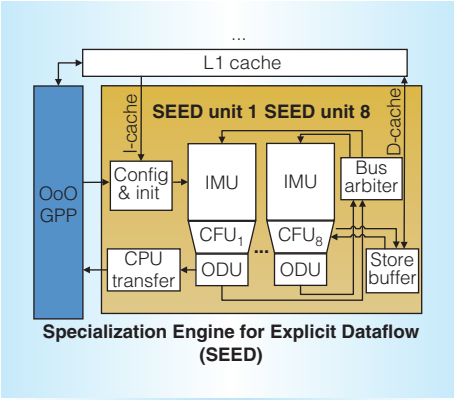

Figure 4. High-level view of integration and organization of SEED. (CFU: compound functional unit; IMU: instruction management unit; ODU: output distribution unit).

buffered and passed by subsequent instructions. Our design uses a CFU mix from existing work[6] in which CFUs contain two to five operations.

CFUs with memory units will issue load and store requests to the host's memory management unit, which is still active while SEED is executing. Load requests access a store buffer for store-to-load forwarding.

Instruction management unit. The IMU has three responsibilities. First, it stores instructions, operands, and destinations. The IMU has storage locations for 32 compound instructions, each with a maximum of four operands, and we keep operand storage space for four concurrent loop iterations. The static instruction storage is roughly equivalent to a maximum of 1,024 noncompound instructions.

Second, the IMU fires instructions. Ready logic monitors the operand storage unit and picks a ready instruction (when all operands are available), with priority to the oldest instruction. Then, the compound instruction and its operands and destinations are sent to the CFU.

Third, the IMU directs incoming values. The input control pulls values from the network to appropriate storage locations on the basis of the incoming instruction tag.

Output distribution unit. The ODU distributes the output values and destination packets (SEED unit, instruction location, and iteration offset) to the bus network and buffers them during bus conflicts.

Bus architecture and arbiter. SEED uses a parallel bus interconnect to forward the output packets from the ODU to a data-dependent compound instruction that's present in either the same or another SEED unit. This means dependent compound instructions cannot execute in back-to-back cycles.

Evaluation Methodology

We employed a modeling methodology based on the transformable dependence graph[9] with 22-nm technology to evaluate SEED.

Benchmark Selection

We chose benchmarks from SPECint and MediaBench that represent a variety of control and memory irregularity.

GPP Characteristics

All cores are x86, have 256-bit SIMD, and are configured with the same cache hierarchy: a two-way 32-Kbyte instruction cache and a 64-Kbyte level-1 data cache, both with four cycle latencies, and an eight-way 2-Mbyte level-2 cache with a 22-cycle hit latency. Also, in order to exclude the effects of frequency scaling, all cores run at 2 Ghz.

Evaluating Dataflow-Specialization Potential

To understand dataflow specialization's potential and tradeoffs, we explore the prevalence and duration of nested loop regions, their performance with dataflow execution, and their interaction with different host cores.

Nested Loop Prevalence

Figure 5 shows cumulative distributions of dynamic instructions coverage with varying dynamic region granularity, assuming a maximum static size of 1,024 instructions. Considering regions with a duration of 8K dynamic instructions or longer (*x*-axis), nested loops can cover 60 percent of the total instructions, whereas inner loops cover only 20 percent. Nested loops also greatly increase the region duration for a given percentage of

coverage (1K to 64K for 40 percent coverage).

Dataflow-Performance Analysis

First we compare the speedups of SEED to our most aggressive design (OoO4) on the most frequent nested-loop regions of programs (each greater than 1 percent of total instructions). The results show that different regions have vastly different performance characteristics, and some are favored heavily by one architecture (see Figure 6). Around three to five times speedup is possible, and many regions show significant speedup.

We next examine the reasons for performance differences. Table 1 presents details on the highest-contributing region from each benchmark. SEED IPC is an effective IPC that uses the GPP's instructions as total instructions.

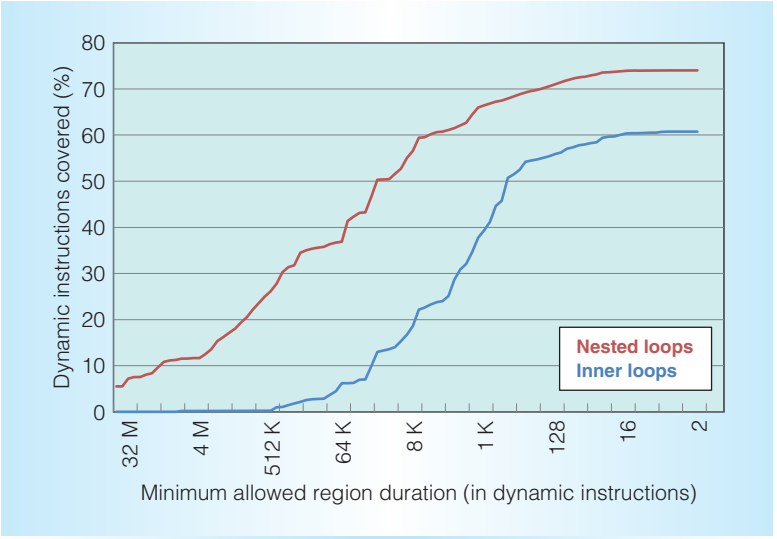

Figure 5. Cumulative contribution percentage for decreasing dynamic region lengths. The static region size is a maximum of 1,024 instructions.

Performance and energy benefit regions. Compared to the OoO4 wide core, SEED can provide high speedups for certain applications, coming from the ability to exploit higher ILP in computationally intensive regions and from the breaking of the instruction window barrier in order to achieve higher memory parallelism.

In the first category are `jpg2000dec`, `cjpeg`, and `djpeg`, which can exploit ILP past the processor's issue width while simultaneously saving energy by using less-complex structures. Often, these regions have indirect memory access that precludes SIMD vectorization. In the second category are `181.mcf` and `429.mcf`, which experience high cache miss rates and clog the OoO processor's instruction window. SEED is limited only by the store buffer size on these benchmarks.

Energy-benefit-only regions. These regions have similar performance to the OoO4, but are more energy efficient by two to three times. Overall, ILP tends to be lower, but control is mostly off the critical path, allowing dataflow to compete. This is the case for `djpeg-1` and `h264dec`. Benchmarks like `gsmencode` and `164.gzip` actually have some potential ILP advantages but are burdened by communication overhead. Benchmark `h263enc` has a high potential ILP but requires multiple instances of the inner loop (not just iterations) in parallel, which SEED does not support.

Contrastingly, benchmarks `473.astar` and `jpg2000enc` have significant control but still perform close to the OoO core. These benchmarks make up for the lack of speculation by avoiding branch misses and relying on the control-equivalent spawning that dataflow provides.

Performance loss regions. Several SEED regions lose performance versus the OoO4 core, as shown in the last set of rows in Table 1. The most common reason is additional communication latency on the critical path, affecting regions in `403.gcc`, `mpeg2dec`, and `mpeg2enc`. Also, certain benchmarks have load-dependent control (for example, `401.bzip2`), which causes low potential performance for dataflow. These are fundamental dataflow limitations. In two cases, configuration overhead hurt the benefit of a short-duration region (`464.h264ref` and `197.parser`). In practice, these regions would not be executed on SEED. Finally, some of these regions are vectorized on the GPP, and SEED is not optimized to exploit data parallelism. This affects `401.bzip2` and `256.bzip2`.

In summary, speedups come from exploiting higher memory parallelism and

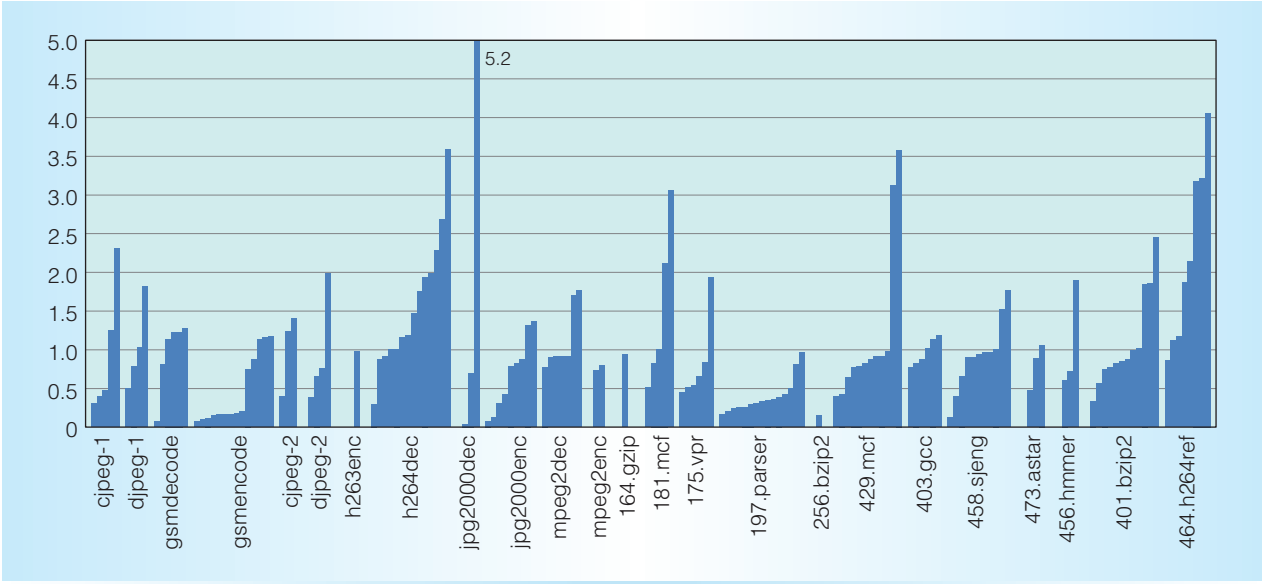

Figure 6. Per-region SEED speedups. Large slowdowns and speedups, three to five times, are possible.

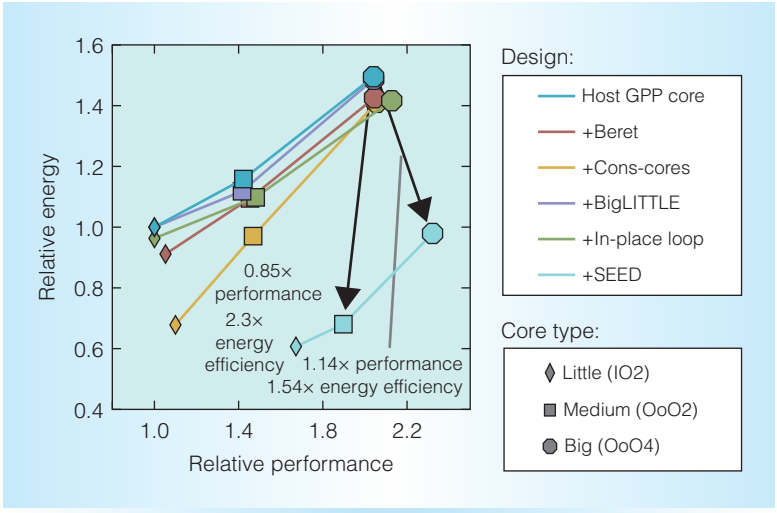

Figure 7. Comparison of SEED with other specialization techniques for targeting irregular codes. Dataflow specialization with SEED significantly pushes the performance and energy frontier.

instruction parallelism and avoiding misspeculation overheads. Slowdowns come from the extra latency cost on more serialized computations.

Integration with GPPs

We consider integration with a little, medium, and big core. To eliminate compiler and runtime heuristics on when to use which architecture, we consider using an oracle scheduler, which uses advance knowledge of region execution time to decide when to use the OoO core, SEED, or SIMD. Figure 7 and Table 2 show the summary data.

For the little, medium, and big cores, SEED provides 1.65, 1.33, and 1.14 times speedup and 1.64, 1.70, and 1.53 times energy efficiency, respectively. This is primarily due to the prevalence of regions in which dataflow execution can match the host core's performance 71, 64, and 42 percent of the time, respectively.

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

This article has demonstrated the potentially disruptive tradeoffs for heterogeneous cores, raising the bar of what is possible given only a modestly complex core. Beyond heterogeneous architectures, its insights suggest opportunities for significant advancement on a purely microarchitectural level. Finally, it broadly contributes to the focus of future research in hardware specialization: that specialization could more aptly be performed along dimensions of program behaviors, rather than just low or high potential ILP.

Table 2. GPP core type details for the comparison in Figure 7.

GPP core type	Characteristics
Little (IO2)	Dual issue, 1 load/store port
Medium (OoO2)	64-entry reorder buffer, 32-entry instruction window (IW), load/store queue (LSQ): 16 loads/20 stores, 1 load/store port, speculative scheduling
Big (OoO4)	168-entry reorder buffer, 48-entry IW, LSQ: 64 loads/36 stores, 2 load/store ports, speculative scheduling

The potential for disruptive design tradeoffs is highlighted in Figure 7, which compares SEED to existing techniques for targeting irregular codes. SEED improves performance and energy efficiency across GPP cores types, significantly more than existing accelerator and microarchitectural approaches do.

Perhaps more interesting are the disruptive changes that a heterogeneous dataflow system introduces for computer architects. First, the OoO2+SEED is actually reasonably close in performance to an OoO4 processor on average, within 15 percent, while reducing energy 2.3 times. Additionally, our McPAT-based estimates suggest that an OoO2+SEED occupies less area than an OoO4 GPP core. Therefore, a heterogeneous 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, non-general-purpose dataflow engine.

An equally interesting tradeoff is to add a dataflow unit to a larger OoO core—this improved the energy efficiency of the OoO4 core by 1.54 times, while improving the performance by 1.14 times. This is a huge leap for energy efficiency, especially considering the difficulty of improving the efficiency for complex, irregular workloads like SpecINT. We intentionally chose SpecINT because it is challenging, and we expect our results on emerging workloads to be even better. Furthermore, we argue that adding a dataflow engine is not actually an exorbitant amount of additional effort, because the design is almost completely decoupled.

In addition, many of our observations apply to more than just heterogeneous architectures—they can also be applied to the core microarchitecture itself. For example, our work demonstrated the energy efficiency of explicit dataflow execution. Other microarchitecture techniques have tried to exploit this, such as ForwardFlow,[10] which uses pointer-based communication of instruction operands. However, ForwardFlow still suffers fetch and decode overheads and still must dynamically build the dependence graph. Thus, applying our configurable-dataflow principles would enable the retention of the generated dataflow graph across loop iterations and could dramatically improve energy efficiency.

Others have looked at in-place loop execution techniques in the context of OoO hardware; for example, Revolver locks looping traces inside the OoO back end to eliminate front-end energy.[11] The insight we uncover is that although looping traces comprise about half of the dynamic instructions in our workloads, nested loops are much more prevalent, comprising nearly 75 percent of workloads even at a small static instruction size. Creating microarchitectural mechanisms for locking in these instructions could provide large benefits.

Beyond the particular microarchitecture and insights uncovered, this article makes a broader contribution to recent efforts in hardware specialization. Most of the research in general-purpose accelerators thus far has focused on creating simple hardware substrates that are more energy efficient when there is low ILP available. In other words, they specialize only along the dimensions of potential instruction parallelism, and thus manifest specialized architectures that differ primarily in terms of their hardware complexity. In contrast, this article supports the fact that there

are other properties besides performance with which to specialize program regions. We differentiate the underlying architecture in this work mostly by the region's control-flow criticality, which leads to large benefits. The principles here suggest a paradigm of behavior specialization in which different specialized hardware substrates target codes with certain program behaviors. This paradigm could eventually help merge the concepts of specialization and general-purpose computing by targeting a wide spectrum of program behaviors. If successful, it could ultimately obviate the need for large general-purpose processors altogether. MICRO

Acknowledgments

Support for this research was provided by the US National Science Foundation under grant CNS-1228782 and by a Google US/Canada PhD Fellowship.

References

1. K. Arvind and R.S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Trans. Computers*, vol. 39, no. 3, 1990, pp. 300–318.
2. D. Burger et al., "Scaling to the End of Silicon with EDGE Architectures," *Computer*, vol. 37, no. 7, 2004, pp. 44–55.
3. S. Swanson et al., "WaveScalar," *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2003, pp. 291–302.
4. M. Mishra et al., "Tartan: Evaluating Spatial Computation for Whole Program Execution," *Proc. 12th Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 163–174.
5. M. Budiu, P.V. Artigas, and S.C. Goldstein, "Dataflow: A Complement to Superscalar," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software*, 2005, pp. 177–186.
6. S. Gupta et al., "Bundled Execution of Recurring Traces for Energy-Efficient General Purpose Processing," *Proc. 44th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2011, pp. 12–23.
7. G. Venkatesh et al., "Conservation Cores: Reducing the Energy of Mature Computations," *Proc. 15th Conf. Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 205–218.
8. T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Exploring the Potential of Heterogeneous Von Neumann/Dataflow Execution Models," *Proc. 42nd Ann. Int'l Symp. Computer Architecture*, 2015, pp. 298–310.
9. T. Nowatzki, V. Govindaraju, and K. Sankaralingam, "A Graph-Based Program Representation for Analyzing Hardware Specialization Approaches," *IEEE Computer Architecture Letters*, vol. 14, no. 2, 2015, pp. 94–98.
10. D. Gibson and D.A. Wood, "ForwardFlow: A Scalable Core for Power-Constrained CMPs," *Proc. 37th Ann. Int'l Symp. Computer Architecture*, 2010, pp. 14–25.
11. M. Hayenga, V.R.K. Naresh, and M.H. Lipasti, "Revolver: Processor Architecture for Power Efficient Loop Execution," *Proc. IEEE 20th Int'l Symp. High Performance Computer Architecture*, 2014, pp. 591–602.

Tony Nowatzki is a PhD student in the Department of Computer Sciences at the University of Wisconsin–Madison. He received an MS in computer science from the University of Wisconsin–Madison. He is a student member of IEEE. Contact him at tjn@cs.wisc.edu.

Vinay Gangadhar is a PhD student in the Department of Electrical and Computer Engineering at the University of Wisconsin–Madison. He received an MS in electrical and computer engineering from the University of Wisconsin–Madison. He is a student member of IEEE. Contact him at vinay@cs.wisc.edu.

Karthikeyan Sankaralingam is an associate professor in the Department of Computer Sciences and the Department of Electrical and Computer Engineering at the University of Wisconsin–Madison. He received a PhD in computer science from the University of Texas at Austin. He is a senior member of IEEE. Contact him at karu@cs.wisc.edu.