Pushing the Limits of Accelerator Efficiency While Retaining Programmability

Tony Nowatzki[*] Vinay Gangadhar[*] Karthikeyan Sankaralingam[*] Greg Wright[†]

[*]University of Wisconsin-Madison [†]Qualcomm
{tjn,vinay,karu}@cs.wisc.edu gwright@qti.qualcomm.com

ABSTRACT

The waning benefits of device scaling have caused a push towards domain specific accelerators (DSAs), which sacrifice programmability for efficiency. While providing huge benefits, DSAs are prone to obsoletion due to domain volatility, have recurring design and verification costs, and have large area footprints when multiple DSAs are required in a single device. Because of the benefits of generality, this work explores how far a programmable architecture can be pushed, and whether it can come close to the performance, energy, and area efficiency of a DSA-based approach.

Our insight is that DSAs employ common specialization principles for *concurrency, computation, communication, data-reuse and coordination*, and that these same principles can be exploited in a programmable architecture using a composition of known microarchitectural mechanisms. Specifically, we propose and study an architecture called LSSD, which is composed of many low-power and tiny cores, each having a configurable spatial architecture, scratchpads, and DMA.

Our results show that a programmable, specialized architecture can indeed be competitive with a domain-specific approach. Compared to four prominent and diverse DSAs, LSSD can match the DSAs' 10× to 150× speedup over an OOO core, with only up to 4× more area and power than a single DSA, while retaining programmability.

1. INTRODUCTION

For many years, general-purpose processor architectures and micro-architectures took advantage of Dennard scaling and Moore's Law, exploiting increased circuit integration to deliver higher performance and lower energy computation; as the proverb says, a rising tide raises all boats, and many diverse application areas benefited. As has been noted [1, 2, 3], those physical trends are slowing, and thus there has been a recent surge of interest in more narrowly-applicable architectures in the hope of continuing system improvements in at least some significant application domains.

A popular approach so far has been *domain specific accelerators* (DSAs): hardware engines capable of performing computations in one domain with high performance and energy efficiency. DSAs have been developed for machine learning [4, 5], cryptography [6], XML processing [6], regular expression matching [7, 8], H.264 decoding [9], databases [10, 11] and many others. Together, these works have demonstrated that, for many important workloads, accelerators can achieve between 10× to 1000× performance and energy benefits over high performance, power hungry general purpose processors.

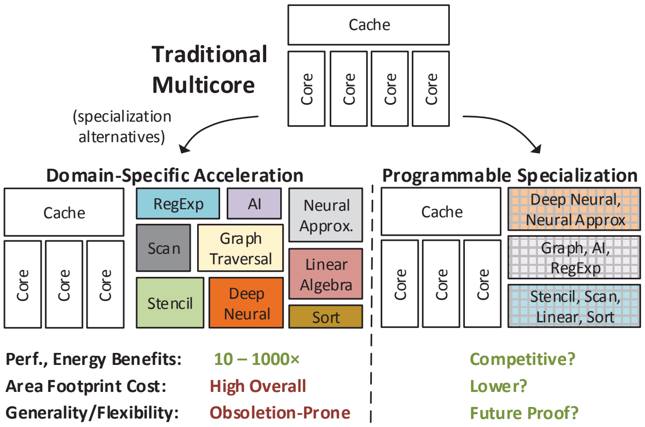


Figure 1: Specialization Paradigms & Tradeoffs

For all of their *efficiency* benefits, DSAs give up on *programmabiltiy* – a high price to pay. First, this makes DSAs prone to obsoletion – the domains which we need to specialize, as well as the best algorithms to use, are constantly evolving with scientific progress and changing user needs. Moreover, the relevant domains change between device types (server, mobile, wearable), and creating fundamentally new designs for each costs both design and validation time. More subtly, most devices run several different important workloads (e.g. mobile SOCs), and therefore multiple DSAs will be required – this may mean that though each DSA is area-efficient, a combination of DSAs may not be.

Critically, the alternative to domain specialization is not necessarily standard general-purpose processors, but rather programmable and configurable architectures which employ similar micro-architectural mechanisms for specialization. The promise of such an architecture is high efficiency *and* the ability to be flexible across workloads. Figure 1 depicts the two specialization paradigms at a high level, leading to the central question of this work: *how far can the efficiency of programmable architectures be pushed, and can they be competitive with domain-specific designs?*

To this end, this paper first makes the observation that DSAs, though differing greatly in their design choices, all employ a similar set of specialization principles:

1. Matching the hardware *concurrency* to the enormous parallelism typically present in accelerated algorithms.

2. Problem-specific functional units for *computation*.

978-1-4673-9211-2/16/$31.00 ©2016 IEEE

3. Explicit *communication* of data between units, as opposed to the arbitrary patterns of communication implied through shared (register and memory) address spaces in a general-purpose ISA.
4. Customized structures for *data reuse*[1].
5. *Coordination* of the control of the other hardware units using low power, simple hardware.

Our primary insight is that the above principles can be exploited by composing *known* programmable and configurable microarchitectural mechanisms. We explain the rationale of one such architecture, our proposed design, as follows:

To exploit the enormous *concurrency* typically present in specializable algorithms, we argue that the most intuitive strategy (while retaining programmability) is to use many *tiny*, low-power cores. Next, to improve the cores for handling the commonly high degree of computation in these workloads, we can add a spatial fabric to each core; the spatial fabric's network specializes the operand *communication*, and its functional units (FUs) can be specialized for algorithm-specific *computation*. Adding scratchpad memories enables the specialization of *data reuse*, and adding a DMA engine to feed these memories specializes the memory *communication*. Lastly is the specialization of the *coordination* of the various hardware units, for which we again employ the low-power core. Named after the components (Low-power cores, Spatial architecture, Scratchpad, and DMA), this architecture is called LSSD. While LSSD is not the first architecture to exploit the above principles, we are the first to combine its mechanisms in this way to target today's technology and application constraints.

In practice, the design and use of such an architecture has three phases (Figure 2). In the *synthesis* phase, the chosen domains' performance, power, and area constraints are identified. These properties determine hardware parameters (eg. FU types and count, scratchpad size). In the *programming* phase, standard programming languages with annotations are used to specify each algorithm. Finally, at *run-time*, the hardware is reconfigured as required for different workloads.

Though we arrived at this design by taking small cores and adding hardware for the various specialization principles, its possible to come to similar and similarly effective designs by starting from a different baseline architecture (eg. GPUs). We discuss these possibilities later.

In this paper, we study the tradeoffs of programmable specialization by comparing LSSD against DSAs from four workload domains, chosen because of their relevance and diversity, stressing its potential generality.

Our specific contributions are:

- Identifying the principles of specialization; showing their application to prominent accelerators. (Section 2)
- Composing a programmable specialization architecture from known mechanisms. (Sections 3 & 4)
- Demonstrating that LSSD can match DSAs' performance, limiting the overheads of programmability to 2× to 4×, as opposed to the 100× to 1000× inefficiency of large OOO cores. LSSD's programmability and efficiency imply that it could serve as an alternative, insightful baseline to extract out specialization benefits in future accelerator research. (Section 6)

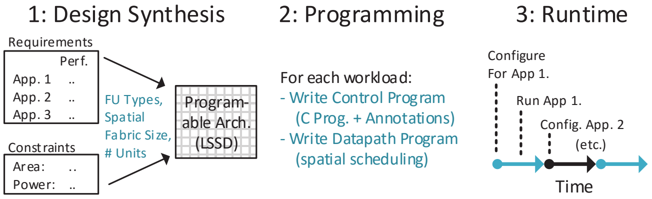


Figure 2: Phases of LSSD's Design & Use

- Analysis of potential economic and energy-efficiency tradeoffs of DSAs versus LSSD. We show that when LSSD and DSAs have equivalent performance, the potential further system-level energy savings by using a DSA is marginal. (Section 7)

The remaining sections describe the methodology (Section 5) and limitations (Section 8). Section 9 discusses related work, and Section 10 concludes.

2. COMMON PRINCIPLES OF ARCHITECTURAL SPECIALIZATION

Domain specific accelerators (DSAs) achieve efficiency through the employment of five *common* specialization principles, which we describe here in detail. We also discuss how four recent accelerator designs apply these principles.

Broadly, we see this as a counterpart to the insights from Hameed et al. [9]. While they describe the sources of inefficiency in a general purpose processor, we explain the sources of potential efficiency from specialization.

2.1 Defining the Principles of Specialization

Workload Assumptions: Before we define the specialization principles, we first characterize the types of workloads to which these principles apply. To do this, we list assumed workload characteristics, which we claim are typical of workloads commonly targeted by DSAs:

1. Workloads have significant parallelism, either at the data or thread level; 2. They perform some computational work (ie. not *just* data-structure traversals); 3. They have coarse grain units of work; 4. They have mainly regular memory access. Without the above characteristics, beneficial hardware specialization becomes much more challenging.

Below we define the five principles of architectural specialization and discuss the potential area, power, and performance tradeoffs of targeting each.

Concurrency Specialization: The concurrency of a workload is the degree to which its operations can be performed simultaneously. Specializing for a high degree of concurrency means organizing the hardware to perform work in parallel by favoring lower overhead structures. Examples of specialization strategies include employing many independent processing elements with their own controllers or using a wide vector model with a single controller.

[1] Though sometimes referred to as temporal locality specialization, reuse is actually most-often the *root cause* of the locality, i.e. algorithms/data structures are tuned so that reused-data exhibits locality.

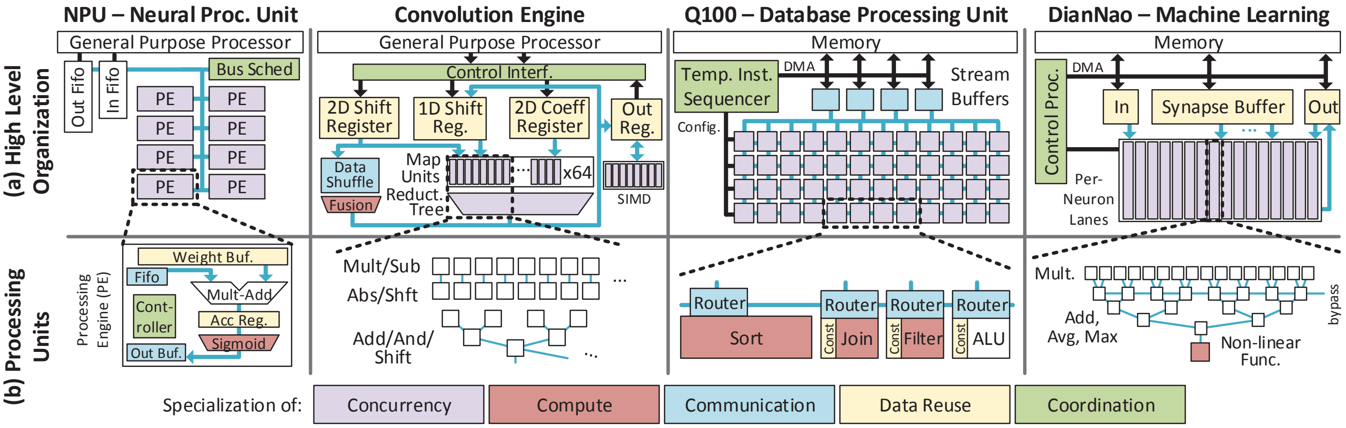


Figure 3: Application of Specialization Principles in Four DSAs.

Applying hardware concurrency increases the performance and efficiency of parallel workloads, while increasing power, at a close-to-linear ratio while parallelism exists.

Computation Specialization: Computations are individual units of work in an algorithm. The hardware which performs this work are functional units (FUs) – which typically take few inputs and produce a single output. Specializing *computation* means creating problem-specific FUs. For instance, a `sin` FU would much more efficiently compute the sine function than iterative methods on a general purpose processor[2]. Specializing computation improves performance and power by reducing the total work. We note some computation specialization can be problem-specific. However, commonality between and inside domains is almost guaranteed.

Communication Specialization: Communication is the means of transmission of values between and among storage and functional units. Specialized communication is simply the instantiation of communication channels, and potentially buffering, between hardware units to ultimately facilitate a faster operand throughput to the FUs. This reduces power by lessening access to intermediate storage, and potentially area if the alternative is a general communication network. One example is a broadcast network for efficiently sending immediately consumable data to many compute units.

Data Reuse Specialization: Data reuse is an algorithmic property where intermediate values are consumed multiple times. The specialization of data reuse means using custom storage structures for these temporaries[3]. Specializing reuse benefits performance and power by avoiding the more expensive access to a larger global memory.

A classic example of reuse specialization is caches. In the context of accelerators, access patterns are often known a priori, often meaning that low-ported, wide scratchpads are most effective (due to caches' energy spent in tag lookup and wires). Accelerators also specialize the reuse of constant values, either through read-only memories or FU-specific storage if the constants are specific to one static operation.

Indeed, the relationship between communication, computation, and reuse specialization is non-trivial. The presence of better or more computational resources can potentially obviate the need for reuse specialization, and the properties of specialized communication channels (e.g. width) often must be co-designed with memory structures for reuse.

Coordination Specialization: Hardware coordination is the management of hardware units and their timing to perform work. Instruction sequencing, control flow, signal decoding and address generation are all examples of coordination tasks. Specializing it usually involves the creation of small state machines to perform each task, rather than relying on a general purpose processor and (for example) out-of-order instruction scheduling. Performing more coordination specialization typically means less area and power compared to something more programmable, at the price of generality.

2.2 Relating Specialization Principles to Accelerator Mechanisms

Figure 3 depicts the block-diagrams of the four DSAs that we study; colors indicate the types of specialization of each component. The specialization mechanisms and algorithmic properties exploited are in Table 1, as explained below:

NPU [12] is a DSA for approximate computing using the neural network algorithm, integrated to the host core through a FIFO interface. NPU is designed to exploit the concurrency of each level of a neural network, using parallel processing entities (PEs) to pipeline the computations of eight neurons simultaneously. NPU specializes reuse with accumulation registers and per-PE weight buffers. For communication, NPU employs a broadcast network specializing the large fan-outs of the neural network, and specializes computation with sigmoid FUs. A bus scheduler and PE controller are used to specialize the hardware coordination.

Convolution Engine [13] accelerates stencil-like computations. The host core uses special instructions to coordinate control of the hardware through a control interface. It exploits concurrency through both vector and pipeline parallelism and relies heavily on reuse specialization by using custom memories for storing pixels and coefficients. In ad-

[2] It is arguable whether certain FUs can be considered *specialized*. There is no set answer here; in this paper we rely on the intuition that FUs that are not typically found in a general purpose processor should be considered specialized.

[3] To be more precise, algorithmic reuse can be exploited by direct communication when values can be directly communicated from producer to consumer.

Principle		NPU [12]	Conv. Engine [13]	Q100 [14]	DianNao [4]
Computation	Alg.	Sigmoid computation	Graph fusion	Sort, Partition, Join...	Transfer functions
	HW	Special Function Units			
Commun-ication	Alg.	All-to-all neuron communication b/t layers	Wide-vector + reduction	Streaming access between operators	Wide-vector + reduction
	HW	Broadcast network b/t processing entities	Wide buses + reduction network	Buffered inter-operator routers	Point-to-point links + reduction network
Data Reuse	Alg.	Neural weights	Reuse of shifted values	Constants	Synapse accumulation
	HW	Per-PE weight buffers	1D/2D Shift Registers	FU Const. Storage	Synapse scratchpads
Coordination	Alg.	Sequencing of algorithmic steps			
	HW	Bus Sched./PE Ctrl	Host + Control Interf.	Instruction Sequencer	Control Processor
Concurrency	Alg.	Inter-layer neuron independence	Independence of pixels + stencil computations	Streaming queries + operator independence	Independent neurons inside + across layers
	HW	Per-neuron Indep. PEs	Pipeline + wide-vector	Pipeline + FU Array	Pipeline + wide-vector

Table 1: Specialization Principles and Mechanisms. Alg: Algorithm insight, HW: Hardware implementation

dition, column and row interfaces provide shifted versions of intermediate values. These, along with other wide buses, provide communication specialization. Finally, it specializes reduction computation using a graph-fusion unit.

Q100 [14] is a DSA for streaming database queries, which exploits the pipeline concurrency of database operators and intermediate outputs. To support a streaming model, it uses stream buffers to prefetch the required database columns. Q100 specializes the communication by providing dynamically routed channels between FUs to prevent memory spills. Finally, Q100 relies on custom database FUs like Join, Sort, and Partition. Reuse specialization happens inside these FUs when storing constants (ALUs, filters, partitioners) and reused intermediates (aggregates, joins). The communication network configuration and stream buffers are coordinated using an instruction sequencer.

DianNao [4] is a DSA for deep neural networks. It achieves concurrency by applying a very wide vector computation model, and uses wide memory structures (4096-bit wide SRAMs) for reuse specialization of neurons, accumulated values and synapses. DianNao also relies on specialized sigmoid FUs. Point-to-point links between FUs, with little bypassing, specialize the communication. A specialized control processor is used for coordination.

3. AN ARCHITECTURE FOR PROGRAMMABLE SPECIALIZATION

Our primary insight is that well-understood mechanisms can be composed to target the same specialization principles that domain specific accelerators (DSAs) use, but in a programmable fashion. In this section we first explain the architecture of the design that we study, LSSD, highlighting how it performs specialization using the principles. We then discuss other possible programmable specialization architectures. Finally, we describe how to use such a design in practice over the phases of its life-cycle.

3.1 Design Requirements

To give insight into our design, we outline three intuitive requirements for a programmable specialization architecture:

1. **Is programmable:** While some DSAs provide limited programmability, this is usually applicable only for a specific domain. A programmable specialization architecture provides domain-agnostic programmability.
2. **Applies specialization principles:** For achieving energy and performance efficiency competitive with DSAs, they must apply their specialization principles.
3. **Design is parameterizable:** At design time, we must allow for customizing the aspects of the design to meet the combined requirements of the targeted domains.

3.2 LSSD Design

As we will describe, it is possible to construct an architecture which embodies the specialization principles by selecting a set of architectural mechanisms from well known techniques. This is not the only possible set of mechanisms, but they are a simple and effective set.

The most critical principle is exploiting *concurrency*, of which there is typically an abundant amount when considering specializable workloads. The requirement of high concurrency pushes the design towards simplicity, and the requirement of programmability implies the use of some sort of programmable core. The natural way to fill these combined requirements is to use an array of tiny low-power cores which communicate through memory[4]. This is a sensible tradeoff as commonly-specialized workloads exhibit little communication between the coarse-grained parallel units. Also, using many units, as opposed to a wide-vector model, has a flexibility advantage. When memory locality is not possible, parallelism can still be achieved through multiple execution threads. The remainder of the design is a straight-forward application of the remaining specialization principles.

Achieving *communication* specialization of intermediate values requires an efficient distribution mechanism for operands that avoids expensive intermediate storage like

[4] TLB's are a source of energy inefficiency in supporting virtual memory, which DSAs typically sidestep. We imagine that programmable specialization architectures will support limited virtual memory through large pages and small TLBs.

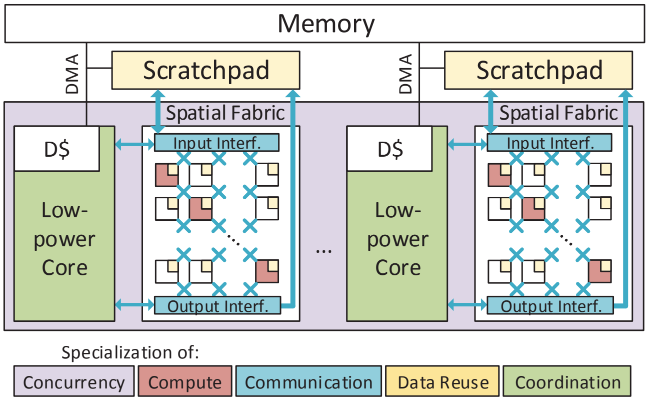


Figure 4: LSSD Architecture

multi-ported register files. Arguably, the best known approach is an explicit routing network, which is exposed to the ISA to eliminate the hardware burden of dynamic routing. This property is what defines spatial architectures (as well as the early explicit-dataflow machines that inspired them [15]), and therefore we add a spatial architecture as our first mechanism. This serves two additional purposes. First, it is an appropriate place to instantiate custom functional units, i.e. *computation* specialization. Second, it accommodates *reuse* of constant values associated with specific computations.

To achieve *communication* specialization with the global memory, a natural solution is to add a DMA engine and configurable scratchpad, with a vector interface to the spatial architecture. The scratchpad, configured as a DMA buffer, enables the efficient streaming of memory by decoupling memory access from the spatial architecture. When configured differently, the scratchpad can act as a *reuse* buffer. In either context, a single-ported scratchpad is enough, as access patterns are usually simple and known ahead of time.

Finally, we require an efficient mechanism for *coordinating* the above hardware units (e.g. configuring the spatial architecture or synchronizing DMA with the computation). Again, here we propose relying on the simple core, as this brings a huge programmability and generality benefit. Furthermore, the cost is low: if the core is low-power enough, and the spatial architecture is large enough, the overheads of coordination can be kept low.

To summarize, each unit of our proposed fabric contains a Low-power core, a Spatial architecture, Scratchpad and DMA **(LSSD)** as shown in Figure 4. This architecture satisfies the aforementioned requirements: programmability, efficiency through the application of specialization principles, and simple parameterizability.

3.3 Evolution towards Programmable Specialization

The approach we took in designing LSSD was to start with a concurrent architecture and evolve it by applying specialization principles. Here we explain how perhaps equally effective (and plausibly similar) designs could be arrived at by starting with different baseline architectures and applying the same principles.

Large Cores: One possible path forward is to begin with traditional, large out-of-order cores, and add mechanisms to achieve more concurrency and specialized execution. One example is MorphCore [16], which specializes for concurrency by adding an inorder SMT mode to an OOO core. WiDGET [17] is similar in that it decouples execution resources from threads, allowing it to specialize the concurrency and coordination of different threads. Another example is specializing for communication by adding reduction instructions to SIMD. While a valuable direction, this style of approach is arguably more difficult as it necessarily retains some of the overheads of the large core.

General Purpose Graphics Processing Units: GPGPUs are interesting as they already apply many specialization principles: concurrency through many independent units, computation through transcendental functions, and data reuse through programmable scratchpads. That said, they do not specialize for operand communication, instead relying on large register files. Some research architectures have even explored such specialization through the addition of spatial fabrics, like SGMF [18].

Making a GPGPU more suitable for programmable specialization would also mean removing or scaling back features geared towards more general computation, which are simply not needed if only targeting commonly accelerated workloads. Some candidates for simplification may be the hardware for coalescing memory, as access patterns are usually known a priori (specializing for communication), or the hardware that handles diverging control flow (specializing for coordination). We suspect that a straightforward application of specialization principles to the GPGPU may lead to a similar design as LSSD.

Field Programmable Gate Arrays: FPGAs are an interesting starting point as they already apply all specialization principles (DSP slices for computation, configurable networks for communication, block RAMs for data-reuse, configurable logic for coordination, and ample LUT resources for concurrency). They are also programmable, with a spectrum of language choices spanning HDL design, HLS and even OpenCL. Furthermore, they allow further specialization of the coordination using efficient finite state machines rather than a general purpose core.

However, FPGAs today lack in efficiency (both in frequency and power) because of the mechanisms they rely on: fine-grained programmability and global routing. Also, the workloads we specialize require large scratchpad/reuse buffers; and the small size of block RAMs (2 to 5KB each) means that many must be composed to support the needed access patterns, leading to further overheads. Furthermore, when considering the fact that many operations can occur in parallel, effectively a multi-ported register file must be constructed to pass intermediates between DSP slices. Indeed these challenges can all be addressed by constraining global routing (eg. innovations in Altera's 1GHz Stratix 10 [19]), performing physical layout to achieve the benefits of tiled architectures, and allowing customization of DSP slices to emerging workload needs. Exploring these is an alternative avenue for extending the generality and efficiency of accelerators.

	Synthesis-Time	Run-Time
Concurrency	Number of instantiated LSSD units	Power-gating unused LSSD units
Computation	Spatial architecture functional unit mix	Scheduling of spatial architecture and core
Commun-ication	Spatial datapath & SRAM interf. widths	Config. of spatial data-path and ports
Data Reuse	Scratchpad (SRAM) size	Use of Scratchpad as DMA or reuse buffer

Table 2: Configurable Aspects of LSSD.

3.4 Use of LSSD in Practice

Preparing the LSSD fabric for use occurs in two phases: 1. *design synthesis* – the selection of hardware parameters to suit the chosen workload domain(s)[5]; and 2. *programing* – the generation of the program and spatial datapath configuration to carry out the algorithm. Table 2 highlights this distinction by contrasting synthesis-time and run-time configuration parameters for the LSSD architecture. We describe each at a high level below, then give a brief example.

Design Synthesis: For specialized architectures, *design synthesis* is the process of provisioning an architecture for given performance, area and power goals. It involves examining one or more workloads or workload domains, and choosing the appropriate functional units, the datapath size, the scratchpad sizes and widths, and the degree of concurrency exploited through multiple core units.

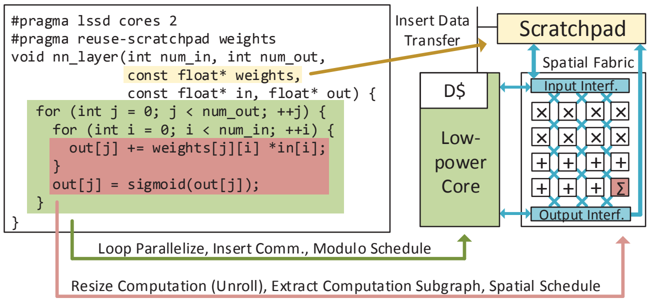
Though many optimization strategies are possible, in this work, we consider the primary constraint of the programmable architecture to be performance – i.e. there exists some throughput target that must be met, and power and area should be minimized, while still retaining some degree of generality and programmability. We outline a simple strategy in Figure 5.

Of course, if multiple workloads should be targeted, then the superset of the above hardware features should be chosen. Section 4 discusses the design points chosen to match the workload domains studied in this paper.

Programming: Programming an LSSD has two major components: creation of the coordination code for the low power core and generation of the configuration data for the spatial datapath to match available resources. Programming for LSSD in assembly may be reasonable because of the simplicity of both the control and data portions. In practice, using standard languages with #pragma annotations, or even languages like OpenCL would likely be effective.

1. Choose the most general single-output FUs which suit the algorithm.
2. Determine the total number of FU resources required to attain the desired throughput.
3. Divide FUs into groups, where each group accesses a limited amount of data per cycle, excluding per-instruction constants. We use 64 bytes maximum per-cycle, resulting in a reasonable scratchpad line usable by many algorithms. The number of these groups is the number of LSSD units.
4. If the algorithm has reuse within a small working set, then size the number of SRAM entries to match.

Figure 5: Procedure for provisioning LSSD



```
#pragma lssd cores 2
#pragma reuse-scratchpad weights
void nn_layer(int num_in, int num_out,
              const float* weights,
              const float* in, float* out) {
  for (int j = 0; j < num_out; ++j) {
    for (int i = 0; i < num_in; ++i) {
      out[j] += weights[j][i] *in[i];
    }
    out[j] = sigmoid(out[j]);
  }
}
```

Insert Data Transfer
Scratchpad
Spatial Fabric
D$
Low-power Core
Input Interf.
×
+
Σ
Output Interf.
Loop Parallelize, Insert Comm., Modulo Schedule
Resize Computation (Unroll), Extract Computation Subgraph, Spatial Schedule

Figure 6: Example Program & Compiler Passes

(Arrows labeled with required compilation passes.)

Though we do not implement a compiler in this work (See Section 5 for our modeling approach), for illustrative purposes, Figure 6 shows an example annotated code for computing a neural network layer, along with a provisioned LSSD. This code computes one neural approximation layer for NPU. The figure also shows the compilation steps to map each portion of the code to the architecture. At a high level, the compiler will use annotations to identify arrays for use in the SRAM, and how (either as a stream buffer or scratchpad). In the example, the weights can be loaded into the scratchpad, and reused across invocations.

Subsequently, the compiler will unroll and create a large-enough datapath to match the resources present in the spatial fabric, which could be spatially scheduled using known techniques [22]. Communication instructions would be inserted into the coordination code, as well as instructions to control the DMA access for streaming inputs. Finally, the coordination loop would be modulo scheduled to effectively pipeline the spatial architecture.

4. DESIGN POINT SELECTION

In this section we describe the design points that we study in this work, along with their rationale, and how workloads map to them. We begin by covering basic building blocks, then detail each design.

4.1 Implementation Building Blocks

We use several existing components, both from the literature and from available designs, as building blocks for the LSSD architecture. The spatial architecture we leverage is the DySER coarse grain reconfigurable architecture (CGRA) [23], which is a lightweight statically-routed mesh of FUs. Note that we will use CGRA and "spatial architecture" interchangeably henceforth. It belongs to a family of similar architectures like CCA [24], BERET [25], SGMF [18] and others. This choice is a good tradeoff between efficiency and configuration time. Because of DySER's network organization, we use only FUs which have single-outputs and low area footprints. It also supports a vector interface with configurable mapping between vector

[5] Our treatment of synthesis-time configurability is reminiscent of architectures like Smart Memories [20] and Custom Fit Processors [21], and we revisit these in the related work.

Name	Equiv. DSA	Concurrency	Computation	Communication	Reuse
LSSD$_N$	NPU	24-Tile CGRA (8 Add, 8 Mul, 8 Sigmoid); Single LSSD Unit	2048 x 32bit Sigmoid Lookup Table	32-bit CGRA; 256-bit SRAM interface	2048 x 32-bit Weight Buffer
LSSD$_C$	Convolution Engine	64-Tile CGRA (32 mul/shf, 32 add/logic); Single LSSD Unit	Standard 16-bit units	16-bit CGRA; 512-bit SRAM interface	512 x 16-bit SRAM for image inputs
LSSD$_Q$	Q100	32-Tile CGRA (16 ALU, 4Agg, 4 Join); 4 LSSD Units	Join Unit + Filter Unit	64-bit CGRA; 256-bit SRAM interface;	SRAMs for buffering
LSSD$_D$	DianNao	64-Tile CGRA (32 Mul, 32 Add); 8 LSSD Units	Piecewise linear sigmoid FU	16x2-bit CGRA; 512-bit SRAM interface;	2048 x 16bit Weight Buffer
LSSD$_B$	Balanced	32-Tile CGRA (combination of above); 8 LSSD Units	Combination of above FUs	64-bit CGRA; 512-bit SRAM interface;	4KB SRAM

Table 3: Configuration of LSSD for each domain.

elements and spatial architecture ports, which enables more efficient irregular memory access patterns.

The processor we leverage is a Tensilica LX3 [26], which is a simple VLIW design featuring a low-frequency (1GHz) 7-stage pipeline. This is chosen because of its very low area and power footprint (0.044 mm^2 and 14mW at 1GHz, 45nm), and is capable of running irregular code. An equally viable candidate is the similarly small Cortex M3 [27].

4.2 LSSD Synthesized Designs and Program Mapping

Here we describe the synthesized designs for the four domains that we target, and outline how their workloads are mapped to the provisioned design. For purposes of evaluation, we consider two different design scenarios: 1. domain-provisioned: programmable architecture's resources target a *single* domain; 2. balanced: programmable architecture resources target many different domains. Both sets of design points are attained using the procedure in Figure 5 and are characterized in Table 3. We discuss in detail below.

LSSD$_N$ for NPU's Neural Approximation Domain: We provision LSSD$_N$ for NPU's performance by including 8 32-bit multipliers, adders, and sigmoid units. As alluded to in the neural network layer example (Figure 6), the read-only weights are stored in scratchpad, and since we need to read a maximum of 8 32-bit words per cycle, the SRAM line size becomes 256-bit. For the coordination program, we consider different implementations of the neural network layer, which vectorize across either input or output neurons, depending on which is larger.

LSSD$_C$ for Convolution Engine's Domain: The primary operation in convolution engine is a map operation (on the image inputs) followed by reduce, with shifted inputs. We assume 32 16-bit multipliers/subtractors for the map stage, and 32 16-bit ALUs for the reduction. The input shift can be performed internally by the CGRA network, which allows input splitting. To supply these units, we only need 512-bits of data per cycle, meaning we can use one 512-bit wide SRAM and a single LSSD unit.

LSSD$_Q$ for Q100's Streaming Database Queries: To target Q100's streaming query workloads, we start with some of the same database query-oriented functional units, like `Filter` and `Aggregate`, for which we include the same number as the original. However, there are several functional units which do not fit well into the type of CGRA we propose using, which we outline next.

First, Q100's Join unit requires data-dependent data-consumption, and adding this capability is non-trivial as it complicates the CGRA itself as well as its communication with the DMA engine (increasing the CGRA's area). Also, Q100's Partitioner and Sorter, used in tandem for fast sorting[6], do not fit naturally into the paradigm of our spatial fabric. The Q100-Partitioner produces output-pairs (destination-bucket,value) and is quite large (0.94mm^2 at 32nm [14]). The Q100-Sorter sorts 1024 elements simultaneously and also has a large area footprint (0.19mm^2). Therefore, we do not include the above Q100 units, and instead employ the low-power core for executing the same algorithm. Unlike for the other domains, the mismatch in FUs means that we had to size the number of cores empirically to match Q100, which turned out to be 4.

Programming the Q100 unit in practice would likely be different than the others as well, as it would need to be integrated into some database management system (DBMS). The DBMS will need to generate a query plan which is composed of subgraphs containing streaming operators (or temporal instructions in Q100's terminology). Each one of these operator subgraphs can be converted to an LSSD program by applying standard code-generation techniques.

LSSD$_D$ for DianNao's Deep Learning Domain: DianNao's fundamental operation is a parallel multiply-reduce followed by an optional non-linear function. To match DianNao's performance, LSSD$_D$ includes 256 16-bit multipliers, 240 adders, and 16 non-linear sigmoid units, which are split up among 8 cores. Each core includes a 512-bit SRAM, primarily used for streaming in neural weights, and the neurons are stored in the core's cache.

LSSD$_B$ Balanced Design: To create a more balanced design with generality across the above four domains, we consider a design point which can achieve the same performance goals. Essentially, we create this design point by taking the largest design (LSSD$_D$ in this case), and distributing the necessary FUs and storage among the remaining units, creating a homogeneous LSSD. This design point enables the study of how increased programmability affects the area and power.

[6] We assume Q100 uses a "sample sort", which first range-partitions into 1024 size buckets, then sorts each bucket.

NPU	Layer Topology (# of in/out neurons)
Conv.	Image block & stencil size, map/reduce funcs
Q100	Query plans, database column formats/sizes
DianNao	Feature map & kernel sizes, tiling params

Table 4: Workload inputs to modeling framework.

5. METHODOLOGY

At a high level, our methodology attempts to fairly assess LSSD tradeoffs across workloads from four different accelerators through pulling data from past works and the original authors, applying performance modeling techniques, using sanity checking against real systems and using standard area/power models. Where assumptions were necessary, we made those that favored the benefits of the DSA. The remainder of this section provides details.

Workloads: For fair comparisons, we use the same algorithms where possible, and use the workloads from the original works. For NPU we use the neural network topologies from the original work [12]. Our results include the approximate regions *only* [7]. For Convolution Engine, we used the four basic convolution kernels [13]. As these do not use the DSA's graph fusion unit, we do not include it in its area calculation. We use all DianNao topologies [4] for comparison.

For Q100 we use 11 TPCH-queries [28] with the same TPCH scale factor (0.01) as the original Q100 paper [14]. We use the same query plan for each, though the LSSD version sometimes has more phases as its FUs differ (see Section 4).

LSSD Performance Estimation: The insight we use for estimating performance is that LSSD is quite simple and the targeted workloads have straightforward access and computation patterns, making execution time straightforward to capture. Our strategy uses a combined trace-simulator (PIN [29]) and application-specific modeling framework to capture the role of the compiler and the LSSD program. This framework is parameterizable for different FU types, concurrency parameters (SIMD width and LSSD unit counts), and reuse and communication structures.

Our framework considers execution stages, each corresponding to a particular computation or kernel. Kernels in NPU are individual neural network layers, while Convolution Engine has scratchpad-load and computation phases. Each Q100 stage is a compound database operation (a "temporal instruction"), and DianNao has convolution, classifier and pooling phases. All of these have significant parallelism, are generally long running and not-overlapping, and contain many pipelined computations. The latency of each phase is either bound by the memory bandwidth, instruction execution rate (considering FU contention and issue-width), cache or scratchpad ports, or the critical path latency. Table 4 describes workload-specific inputs.

LSSD Power & Area Estimation: For the LX3 core parameters, we rely on published datasheets [26]. For the integer FUs, we used the estimates from the DianNao work [4], and for floating point FUs we used estimates from the DySER implementation [23]. To estimate CGRA-network power, we synthesized a router using a 55nm library. SRAMs for data-reuse and buffering were estimated using CACTI 6.5 [30]. Specialized FU properties were taken from the original works.

DSA and Baseline Characteristics: Each DSAs' performance, area and power were obtained as follows:

	Execution Time	Power | Area
NPU	Authors Provided	MCPAT-based estim.
Conv.	Authors Provided	MCPAT-based estim.
Q100	Optimistic Model	In Original Paper
DianNao	Optimistic Model	In Original Paper

For the performance of the Q100 and DianNao DSAs, we constructed a model consistent with the LSSD framework, which is generally optimistic for the DSA. For Q100, we built a model which takes query-plans as inputs in our own python-based domain-specific language. We validated this against execution times provided by the authors, and our model is always optimistic, at most by a factor of 2×. We use our model for Q100 because it allowed us to make the same assumptions about the query plans. For DianNao, we used an optimistic performance model, based on its maximum computation throughput and memory bandwidth.

For NPU power and area, we used CACTI for each internal structure, since this was similar to the strategy used in the work itself. Though we did receive the power for Convolution Engine from the authors, we used our own estimates, as these were more consistent with our results (using their estimate would have been favorable to LSSD).

All area and power estimates are scaled to 28nm.

Integration of LSSD: LSSD_N and LSSD_C only required a single core for achieving the DSA's performance, meaning that these architectures could be directly integrated with a host core. Therefore we do not include the LX3 core's area in the estimation for these design points.

Comparison to Baseline OOO: Much of our results use an OOO core as a baseline, intuitive reference point. We use the Intel 3770K processor, and estimate the power and area based on datasheets and die-photos [31]. We scale its frequency to 2GHz, as this is similar to the baseline used in the other DSA works [12, 4]. An exception to this is the Q100 results, where we use their estimated performance of MonetDB (on a 2.2GHz Intel Xeon E5620), as this proved more consistent than the version of MonetDB we had available.

6. EVALUATION

Our evaluation is organized around three main questions:

Q1. Can LSSD match the performance of DSAs, and what are the sources of its performance?

Q2. What is the cost of general programmability in terms of area and power?

Q3. If multiple workloads are required on-chip, can LSSD ever surpass the area or power efficiency?

Our primary result is that LSSD is a viable and programmable alternative for DSAs, matching their perfor-

[7] If we used LSSD on the non-approximate regions as well, LSSD's performance would surpass NPU on several kernels.

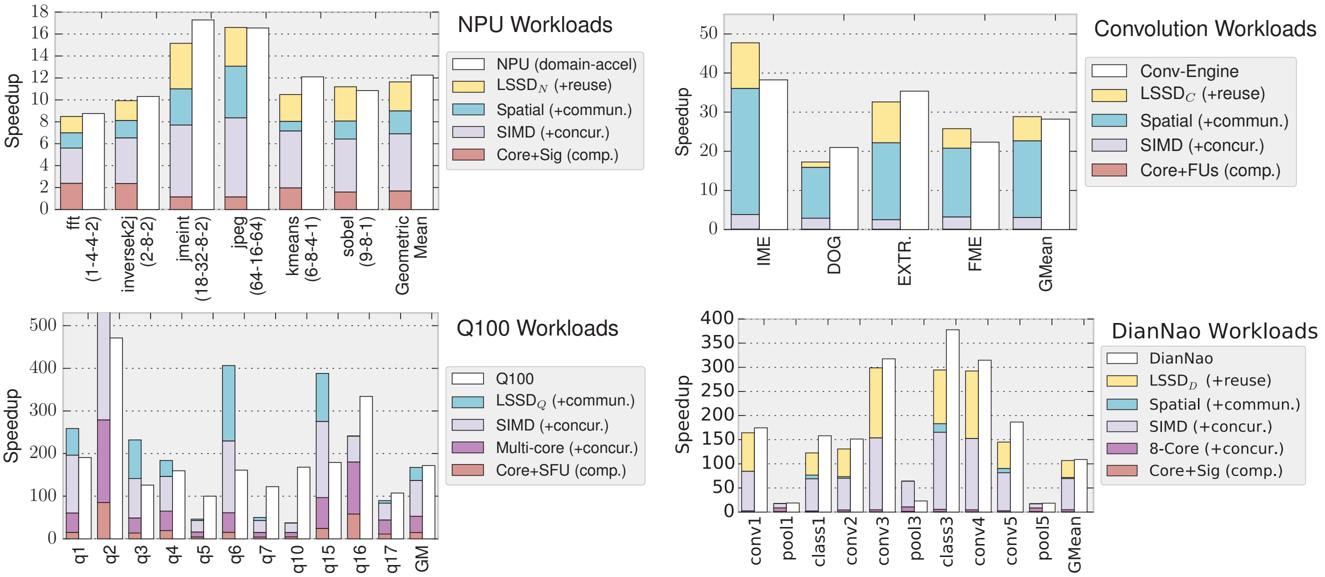


Figure 7: LSSD versus DSA Performance Across Four Domains

mance with only modest (max 2× to 4×) power and area overheads.

6.1 Performance Analysis (Q1)

In this section, we compare the performance of the DSAs and domain-provisioned LSSD designs in Figure 7, normalized to the OOO core. To elucidate the sources of benefits of each specialization principle in LSSD, we also include four additional design points, where each builds on the capabilities of the previous[8]:

1. **Core+SFU** The LX3 core with added problem-specific functional units (computation specialization).
2. **Multicore** LX3 multicore system (+concurrency).
3. **SIMD** An LX3 with SIMD, its width corresponding to LSSD's memory interface (+concurrency).
4. **Spatial** An LX3 where the spatial arch. replaces the SIMD units. (+communication).
5. **LSSD** Previous design plus scratchpad (+reuse)

Across workload domains, LSSD matches the performance of the DSAs, seeing performance improvements over a modern OOO core between 10× and 150×.

NPU versus LSSD$_N$: NPU's and LSSD's organizations differ greatly: independent processing elements in NPU versus the core+CGRA of LSSD. However, their performance is nearly the same. In terms of speedup sources, concurrency (SIMD) provides the most benefit: around 4× speedup. Communication specialization (Spatial) and reuse specialization (LSSD) together provide only 1.7× speedup.

Convolution Engine versus LSSD$_C$: Though LSSD$_C$ performs similarly to the DSA, it has 0.84× the clock-for-clock performance. This is because the DSA is running at a slower 800MHz frequency. For the DOG kernel, LSSD loses performance as the scratchpad size could not fit all of the images needed for computation. The major performance contribution among specialization principles is from concurrency (around 31× from SIMD[9]), and CGRA's spatial communication and reuse play a lesser role, 7.3× and 1.3× respectively.

Q100 versus LSSD$_Q$: Though the 4-core LSSD$_Q$ performs similarly with Q100 overall (only 2% difference), performance varies across queries. Specifically, Queries 5, 7 and 10 are sort-heavy (and 16 and 17 to a lesser extent), and Q100 benefits from the specialized Sort/Partition FUs. We also emphasize here the benefits of both LSSD and Q100 over MonetDB reduce by around a factor of 10 on larger databases (e.g. TPCH scale factor=1), but we report this scale factor, as it is what Q100 was optimized for. The major source of performance is concurrency (3.60× multicore, 2.58× SIMD). Some workloads benefit significantly from the CGRA, which is helpful when the bandwidth is not saturated and the workload is not dominated by sorts/joins.

Note that to be consistent with the Q100 work, the baseline here is an OOO core running MonetDB, while Q100 and LSSD model manually-optimized queries. This explains why the LX3 core can outperform the baseline. As an example, we ran an optimized Q1 (matching the LSSD/Q100 algorithm) on the OOO core, and it was 35x faster than the MonetDB version.

DianNao versus LSSD$_D$: Performance is similar on most workloads. LSSD$_D$ has minor instruction overhead (eg. managing the DMA engine), but gains back some ground on one of the pooling workloads, because the decoupled design of LSSD allows it to load neurons at higher bandwidth. Again, concurrency was most important for performance (8× multicore, 14.4× SIMD). The CGRA provides an additional small benefit by reducing instruction management, and adding a reuse buffer reduces cache contention; their

[8] These intermediate design points are not area or power-normalized, their purpose is to demonstrate the sources of performance.

[9] The LX3 Core+FUs bar is below 1, as it is slower than the baseline.

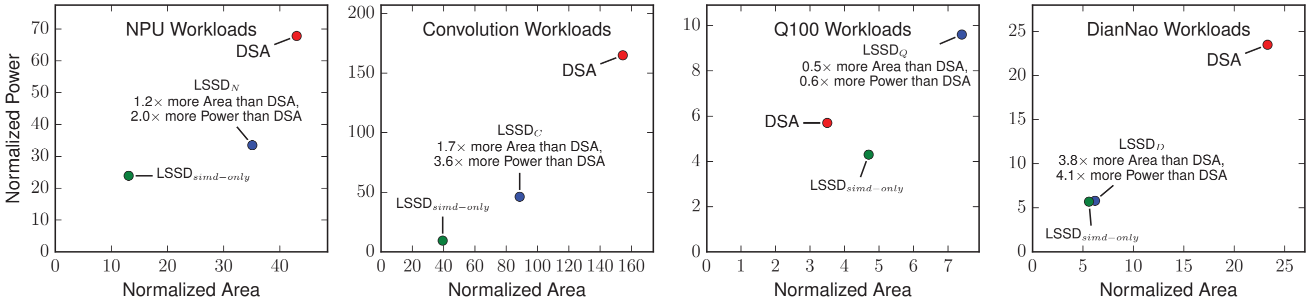


Figure 8: Area & Power tradeoffs using performance-equivalent designs (Baseline: Core+L1+L2 from I3770K processor)

	Area (mm^2)	LSSD$_N$	LSSD$_C$	LSSD$_Q$	LSSD$_D$
Breakdown	Core+Cache			0.09	0.09
	SRAM	0.04	0.02	0.04	0.04
	Functional Unit	0.24	0.02	0.09	0.02
	CGRA Network	0.09	0.11	0.22	0.11
	Unit Total	0.37	0.15	0.44	0.26
	LSSD Total Area	0.37	0.15	1.78	2.11
	DSA Total Area	0.30	0.08	3.69	0.56
	LSSD/DSA Overhead	1.23	1.74	0.48	3.76

Power (mW)	LSSD$_N$	LSSD$_C$	LSSD$_Q$	LSSD$_D$
Core+Cache	41	41	41	41
SRAM	9	5	9	5
Functional Unit	65	7	33	7
CGRA Network	34	56	46	56
Unit Total	149	108	130	108
LSSD Total Power	149	108	519	867
DSA Total Power	74	30	870	213
LSSD/DSA Overhead	2.02	3.57	0.60	4.06

Table 5: LSSD Power and Area Breakdown/Comparison (normalized to 28nm)**.**

combined speedup is 1.9×.

Takeaway: LSSD designs have competitive performance with DSAs. The performance benefits come mostly from concurrency *rather than any other specialization technique.*

6.2 LSSD Area/Power Overheads (Q2)

To elucidate the costs of more general programmability, Figure 8 shows the power and area efficiency for three performance-equivalent designs, using a single OOO core as the baseline. We also show the LSSD$_{simd-only}$, because it is a more standard reference point. It does not employ scratchpads or a CGRA network (but does have specialized FUs). We do not show the single-core or multi-core only points, as scaling up their cores to meet the performance target does not result in practical designs. Table 5 shows the power and area breakdowns for the LSSD designs.

Overall, LSSD has some, but not excessive overheads: up to 3.8× area and 4.1× power. Even so, the LSSD designs are between 6× to 90× less area than a single core, and between 5× and 40× less power. Also, the LSSD designs are generally better than the performance-normalized SIMD design point, implying that the spatial fabric and reuse buffers are effective for reducing overheads. An exception is for LSSD$_D$, as we discuss next.

In fact, LSSD$_D$ has the worst case area and power overheads of 3.76× and 4.06× respectively compared to DianNao. The CGRA network dominates the area and power, because it supports relatively tiny 16-bit FUs.

LSSD$_C$ also has significant overheads of 1.74× area and 3.57× power. Besides the CGRA network overhead, Convolution Engine optimizes for a non-standard datapath width (10-bit versus 16-bit in LSSD), and runs at a lower frequency (800MHz).

For the NPU workloads, the LSSD$_N$ is similar area and has a 2.02× power overhead. The reason for these relatively low overheads is the high contribution of floating point and sigmoid FUs, amortizing the overhead of the LX3 core and CGRA network in LSSD.

Surprisingly, LSSD$_Q$ has 0.48× the area and 0.6× the power of Q100. One reason for this is that LSSD does not embed the expensive Sort and Partition units, which did lead to performance loss on several queries, but was arguably a reasonable tradeoff overall. LSSD also uses a simple circuit-switched CGRA, whereas Q100 uses highly-buffered routers based on the Intel Teraflops chip [32].

Takeaway: With suitable engineering, the overheads of programmability can be reduced to small factors of 2× to 4×, as opposed to the 100× to 1000× inefficiency of large OOO cores.

6.3 Supporting Multiple Domains (Q3)

If multiple workload domains require specialization on the same chip, but do not need to be run simultaneously, it is possible that LSSD can be more area efficient than a Multi-DSA design. Figure 9 shows the area and geomean power tradeoffs for two different workload domain sets, comparing the Multi-DSA chip to the balanced LSSD$_B$ design.

The domain set [NPU/Conv/DianNao] excludes our best result (Q100 workloads). In this case, LSSD$_B$ still has 2.7× area and 2.4× power overhead. However, with Q100 added, LSSD$_B$ is only 0.6× the area.

Takeaway: If only one domain needs to be supported at a time, LSSD can become more area efficient than using multiple DSAs.

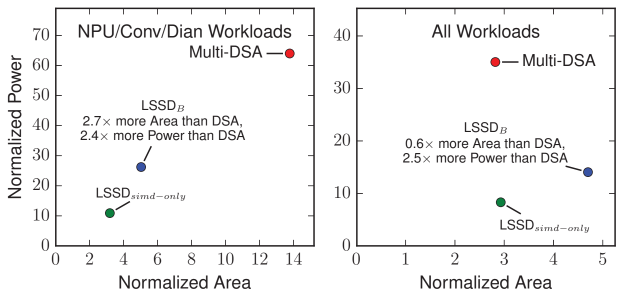


Figure 9: Area & Power of Multi-DSA vs LSSD$_B$.
(Baseline: Core+L1+L2 from I3770K processor)

7. SYSTEM-LEVEL TRADEOFFS

While this paper has explored the relative power, area, and speedup tradeoffs of using a programmable accelerator over a DSA, it is important to understand how this affects the overall decision of which architecture to employ. Specifically, while the speedups of DSAs and LSSD can be similar, an important question is how much the power and area overheads affect the energy benefits and economic costs when accelerating a general purpose chip. We use simple analytical reasoning in this section to explore the tradeoffs.

7.1 Energy Efficiency Tradeoffs

In this subsection, we will analytically bound the possible energy efficiency improvement of a general purpose system accelerated with a DSA versus an LSSD design, by considering a zero-power DSA.

We first define the overall relative energy, E, for an accelerated system in terms of S: the accelerator's speedup, U: the accelerator utilization as a fraction of the original execution time, P_{core}: general purpose core power, P_{sys}: system power, and P_{acc}: accelerator power. The core power includes components which are *not* used because the accelerator is invoked. The system power includes chip components that are active while accelerating, which could include higher level caches and DRAM, or other SOC components, for example. Total energy then becomes:

$$E = P_{acc}(U/S) + P_{\text{sys}}(1-U+U/S) + P_{\text{core}}(1-U) \qquad (1)$$

The energy efficiency improvement of a DSA versus LSSD system ($\text{Eff}_{\text{dsa/lssd}}$), given that their speedups are held equivalent, becomes:

$$\text{Eff}_{\text{dsa/lssd}} = \frac{P_{\text{lssd}}(U/S) + P_{\text{sys}}(1-U+U/S) + P_{\text{core}}(1-U)}{P_{\text{dsa}}(U/S) + P_{\text{sys}}(1-U+U/S) + P_{\text{core}}(1-U)} \qquad (2)$$

By setting the DSA power to zero (and rearranging):

$$\text{Eff}_{\text{dsa/lssd}} < \frac{P_{\text{lssd}}(U/S)}{P_{\text{sys}}(1-U+U/S) + P_{\text{core}}(1-U)} + 1 \qquad (3)$$

The best case for the DSA would be 100% utilization, and in that case we get the intuitive result that maximum energy efficiency improvement is $\text{Eff}_{\text{dsa/lssd}} < P_{\text{lssd}}/P_{\text{sys}} + 1$. Since LSSD's power is usually very low, perhaps a factor of 10 less than system power, the maximum DSA power savings is less than 10%, even with a perfect DSA.

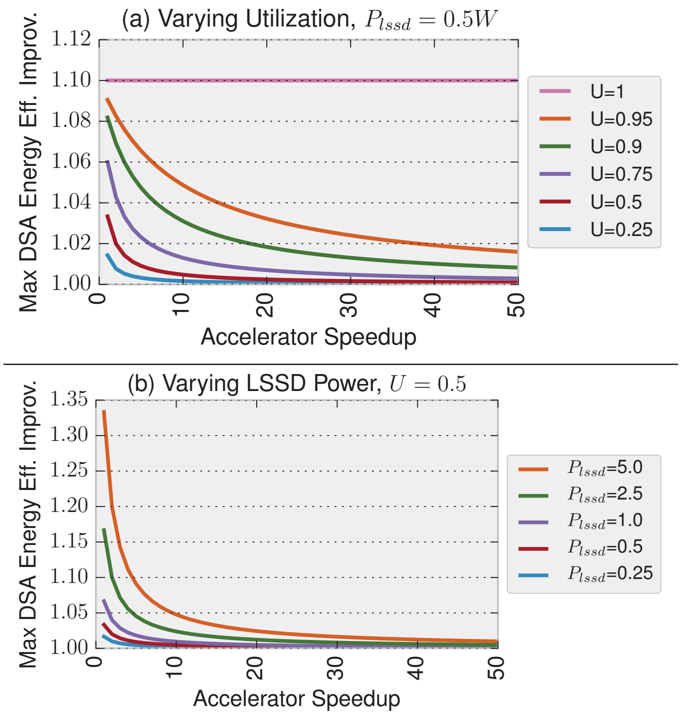


Figure 10: Energy Benefits of Zero-Power DSA
($P_{\text{core}} = 5W$,$P_{\text{sys}} = 5W$)

We characterize this tradeoff across different accelerator utilizations and speedups in Figure 10, using 5W as the core and system power. Figure 10(a) shows that the maximum benefits from a DSA reduce both as the utilization goes down (stressing core power), and when accelerator speedup increases (stressing both core and system power). For a reasonable utilization of U=.5 and speedup of S=10, the maximum energy efficiency gain from a DSA is less than 0.5%. Figure 10(b) shows a similar graph where LSSD's power is varied, while utilization is fixed at U=.5. Even considering an LSSD with power equivalent to the core, when LSSD has a speedup=10×, there is only 5% potential energy savings remaining for a DSA to optimize.

Putting the above together, we claim that when an LSSD can match the performance of a DSA, the further potential energy benefits of a DSA are usually very small. In other words, a 4× power overhead for LSSD versus a DSA is generally inconsequential.

7.2 Economic Tradeoffs

DSAs are a reality today, and without a significant reason to invest in programmable accelerators, its possible that LSSD would not ever become economically viable. That said, we argue that there are tangible reasons why an LSSD-based approach could be economically advantageous.

First, the fixed costs of targeting some domain may indeed be much lower for LSSD. Once the LSSD hardware is designed, targeting a domain is a matter of hardware parameterization and relatively straightforward software development. In some cases, LSSD may be able to target new domains without any hardware changes.

It is also true that for a given domain, LSSD's recurring area costs are factors higher. However, LSSD is relatively small compared to a modern general purpose multicore, and when targeting multiple domains, LSSD's area overheads are amortized.

8. LIMITATIONS

Accelerator/Workload Generality: In section 3, we made several assumptions about the type of workloads targeted. The implication is that there are workload varieties that LSSD will not be suitable for, as well as at least several existing accelerators which LSSD will not be able to match. One example would be packet classification, which would require too much irregular memory access, but Spitznagel et al. [33] is able to create effective specialized hardware with extended TCAMs. Another example is the unsuitable bit-level optimizations that Fowers et al. uses to design an accelerator for lossless compression on FPGAs [34].

It is possible that, when considering a broader or different set of workload properties, the specialization principles identified here may not be the most operative. Even so, an LSSD reference point could help identify what should be these new mechanisms, or perhaps even help to expose additional specialization principles.

Hardware and Compiler Implementations: The design and implementation of an LSSD compiler would be useful for understanding exactly what information is required from the programmer, as well as the difficulty or simplicity of programming such an architecture. Existing compilers from the DySER [35] and SGMF [18] projects could be leveraged as a starting point. Since we still ultimately envision targeting a relatively small number of domains, an alternative approach could be to adapt existing domain-specific languages [36, 37, 38, 39] to target LSSD.

A parameterizable hardware implementation of LSSD would be equally useful in attaining more accurate design space characteristics, as well as finding detailed phenomenon in its design. This is also future work.

9. RELATED WORK

Programmable Specialization Architectures: The literature contains many examples of flexible and efficient specialization architectures. Smart Memories [20], which when configured acts like either a streaming engine or a speculative multiprocessor. One of its primary innovations is mechanisms allowing SRAMs to act as either scratchpads or caches for reuse. Smart Memories is both more complex and more general than LSSD, though likely less efficient on the regular workloads we target.

Another example is Charm [40]: composable heterogeneous accelerator-rich microprocessor, which integrates coarse-grain configurable FU blocks and scratchpads for reuse specialization. A fundamental difference is in the decoupling of the compute units, reuse structures, and host cores, allowing concurrent programs to share blocks in complex ways. Camel [41] augments this with a fine-grained configurable fabric. These architectures provide more choice in mapping computation but are more complex.

The Vector-Thread architecture [42] supports unified vector+multithreading execution, providing flexibility across data-parallel and irregularly-parallel workloads.

The most similar design in terms of microarchitecture is MorphoSys [43]. It also embeds a low power TinyRisc core, integrated with a CGRA, DMA engine and frame buffer. Here, the frame buffer is not used for data reuse, and the CGRA is more loosely coupled with the host core.

There are also a number of related models for exploring energy tradeoffs in heterogeneous environments [44, 45].

Alternate Approaches: An alternate approach to enable further specialization when area is constrained is to reduce the footprint of DSAs themselves. Lyons et al. explore sharing SRAMs across accelerators [46], and their later work explores virtualizing computation components in FGPAs [47].

Our work leverages the notion of synthesis-time reconfigurability, where architectures can be tuned easily for particular workloads. A prior example of such an approach is Custom Fit Processors [21], which tunes the VLIW instruction organization to a workload set.

Principles of Specialization: As mentioned earlier, the work by Hameed et al. [9] studies the principles of specialization from the opposite perspective: in identifying the sources of *inefficiency* in a general purpose processor. While both our paper and their paper argue that the best way forward appears to be augmenting general purpose systems with specialization techniques, their proposed methods differ significantly. In contrast to their work, we argue that large (100-operation) fixed-function units are *not* necessary to bridge the performance gap between general purpose and ASICs, and that a specialized programmable architecture can come close.

10. CONCLUSION

This work has proposed a programmable accelerator, LSSD, that pushes the limits of efficiency while retaining generality. Its design leverages the observation that a common set of specialization principles governs DSA designs. By composing mechanisms for these principles, synthesized LSSD designs are competitive with DSAs, matching their performance with at most modest area and power (2× to 4×) overheads. Because of their large speedup over an OOO core (10× to 150×) and much lower area and power, LSSD design points could also serve as an alternative baseline for future accelerator research.

As our evaluation has shown, much of LSSD's performance comes from exploiting concurrency – in fact, it is more important than all remaining factors across domains. This speaks to why a specialization fabric like LSSD works: As long as a programmable specialization architecture can be tuned to match the parallelism present in the algorithm, most of the potential performance will be achieved. Straightforward mechanisms can reap the modest further benefits from computation, communication, and reuse specialization. Finally, when considering total system energy, LSSD's performance makes its power-overheads over a DSA inconsequential.

11. REFERENCES

[1] R. Colwell, “The chip design game at the end of Moore’s law,” Hot Chips, 2013. http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.15-keynote1-Chipdesign-epub/HC25.26.190-Keynote1-ChipDesignGame-Colwell-DARPA.pdf.

[2] B. Sutherland, “No Moore? a golden rule of microchips appears to be coming to an end,” The Economist, 2013. http://www.economist.com/news/21589080-golden-rule-microchips-appears-be-coming-end-no-moore.

[3] R. Courtland, “The end of the shrink,” *Spectrum, IEEE*, vol. 50, pp. 26–29, November 2013.

[4] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” in *ASPLOS*, 2014.

[5] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, “PuDianNao: A polyvalent machine learning accelerator,” in *ASPLOS*, 2015.

[6] J. Brown, S. Woodward, B. Bass, and C. Johnson, “IBM Power Edge of Network Processor: A wire-speed system on a chip,” *IEEE Micro*, 2011.

[7] B. C. Brodie, D. E. Taylor, and R. K. Cytron, “A scalable architecture for high-throughput regular-expression pattern matching,” in *ISCA*, 2006.

[8] J. V. Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu, “Designing a programmable wire-speed regular-expression matching accelerator,” in *MICRO*, 2012.

[9] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, “Understanding sources of inefficiency in general-purpose chips,” in *ISCA*, 2010.

[10] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, “Meet the walkers: Accelerating index traversals for in-memory databases,” in *MICRO*, 2013.

[11] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross, “Navigating big data with high-throughput, energy-efficient data partitioning,” in *ISCA*, 2013.

[12] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Neural acceleration for general-purpose approximate programs,” in *MICRO*, 2012.

[13] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, “Convolution engine: Balancing efficiency & flexibility in specialized computing,” in *ISCA*, 2013.

[14] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, “Q100: The architecture and design of a database processing unit,” in *ASPLOS*, 2014.

[15] Arvind and R. S. Nikhil, “Executing a program on the MIT Tagged-Token Dataflow Architecture,” *IEEE Trans. Comput.*, 1990.

[16] K. Khubaib, M. Suleman, M. Hashemi, C. Wilkerson, and Y. Patt, “Morphcore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP,” in *MICRO*, 2012.

[17] Y. Watanabe, J. D. Davis, and D. A. Wood, “Widget: Wisconsin decoupled grid execution tiles,” in *ISCA*, 2010.

[18] D. Voitsechov and Y. Etsion, “Single-graph multiple flows: Energy efficient design alternative for GPGPUs,” in *ISCA*, 2014.

[19] “Stratix 10 soc highest performance and most power efficient processing,” 2015. https://www.altera.com/products/soc/portfolio/stratix-10-soc/overview.html.

[20] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, “Smart Memories: A modular reconfigurable architecture.,” in *ISCA*, 2000.

[21] J. A. Fisher, P. Faraboschi, and G. Desoli, “Custom-fit processors: Letting applications define architectures,” in *MICRO*, 1996.

[22] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili, “A general constraint-centric scheduling framework for spatial architectures,” in *PLDI*, 2013.

[23] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, “DySER: Unifying functionality and parallelism specialization for energy efficient computing,” *IEEE Micro*, 2012.

[24] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, “Application-specific processing on a general-purpose core via transparent instruction set customization,” in *MICRO*, 2004.

[25] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, “Bundled execution of recurring traces for energy-efficient general purpose processing,” in *MICRO*, 2011.

[26] “Xtensa LX3 customizable DPU, high performance with lexible I/Os,” 2010. http://ip.cadence.com/uploads/pdf/LX3.pdf.

[27] J. Yiu, *The definitive guide to the ARM Cortex-M3*. Newnes, 2009.

[28] Transaction Processing Performance Council, “TPC-H benchmark specification,” *Published at http://www. tcp. org/hspec. html*, 2008.

[29] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *PLDI*, 2005.

[30] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, “Cacti 6.0: A tool to model large caches,” *HP Laboratories*, 2009.

[31] Intel, “Core i7-3770k processor.” http://ark.intel.com/products/65719/Intel-Core-i7-3770-Processor-8M-Cache-up-to-3_90-GHz.

[32] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, “An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS,” in *ISSCC*, 2007.

[33] E. Spitznagel, D. Taylor, and J. Turner, “Packet classification using extended tcams,” in *Network Protocols, 2003. Proceedings. 11th IEEE International Conference on*, 2003.

[34] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, “A scalable high-bandwidth architecture for lossless compression on fpgas,”

[35] V. Govindaraju, T. Nowatzki, and K. Sankaralingam, “Breaking SIMD shackles with an exposed flexible microarchitecture and the access execute PDG,” in *PACT*, 2013.

[36] H. Lee, K. Brown, A. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun, “Implementing domain-specific languages for heterogeneous parallel computing,” *IEEE Micro*.

[37] N. George, H. Lee, D. Novo, T. Rompf, K. Brown, A. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne, “Hardware system synthesis from domain-specific languages,” in *FPL*, 2014.

[38] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, “A heterogeneous parallel framework for domain-specific languages,” in *PACT*, 2011.

[39] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun, “Optiml: An implicitly parallel domain-specific language for machine learning,” in *ICML*, 2011.

[40] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, “Charm: A composable heterogeneous accelerator-rich microprocessor,” in *ISPLED*, 2012.

[41] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman, “Composable accelerator-rich microprocessor enhanced for adaptivity and longevity,” in *ISLPED*, 2013.

[42] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, “The vector-thread architecture.,” in *ISCA*, 2004.

[43] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho, “Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications,” *Computers, IEEE Transactions on*, 2000.

[44] D. H. Woo and H.-H. S. Lee, “Extending Amdahl’s law for energy-efficient computing in the many-core era,” *Computer*, 2008.

[45] A. Morad, T. Morad, Y. Leonid, R. Ginosar, and U. Weiser, “Generalized MultiAmdahl: Optimization of heterogeneous multi-accelerator soc,” *Computer Architecture Letters*, 2014.

[46] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks, “The accelerator store: A shared memory framework for accelerator-based systems,” *ACM Trans. Archit. Code Optim.*, 2012.

[47] M. Lyons, G.-Y. Wei, and D. Brooks, “Shrink-fit: A framework for flexible accelerator sizing,” *IEEE Comput. Archit. Lett.*, 2013.