Performance Evaluation of a DySER FPGA Prototype System Spanning the Compiler, Microarchitecture, and Hardware Implementation

Chen-Han Ho[†,*], Venkatraman Govindaraju[‡,*], Tony Nowatzki[*], Ranjini Nagaraju[β,*], Zachary Marzec[†,*], Preeti Agarwal[$,*], Chris Frericks[+,*], Ryan Cofell[*], Karthikeyan Sankaralingam [*]
vertical@cs.wisc.edu
[†] Qualcomm. [‡] Oracle. [$] Intel. [+] Samsung. [β] ARM.
[*] University of Wisconsin-Madison
All work performed by all authors when at UW-Madison

***Abstract*—Specialization and accelerators are being proposed as an effective way to address the slowdown of Dennard scaling. DySER is one such accelerator, which dynamically synthesizes large compound functional units to match program regions, using a co-designed compiler and microarchitecture. We have completed a full prototype implementation of DySER integrated into the OpenSPARC processor (called SPARC-DySER), a co-designed compiler in LLVM, and a detailed performance evaluation on an FPGA system, which runs an Ubuntu Linux distribution and full applications. Through the prototype, this paper evaluates the fundamental principles of DySER acceleration.**

Our two key findings are: i) the DySER execution model and microarchitecture provides energy efficient speedups and the integration of DySER does not introduce overheads – overall, DySER's performance improvement to OpenSPARC is $6\times$, consuming only 200mW ; ii) on the compiler side, the DySER compiler is effective at extracting computationally intensive regular and irregular code.

I. Introduction

Accelerators [26], [36], [72], [34], [2], [29], [19] are designed to push their baseline architectures across the established energy and performance frontier, a trend we have depicted in Figure 1. Accelerators, which are shown as vectors (arrows), move the baseline processor to a new point on the graph that has a better performance and/or energy tradeoff. Many coarse-grain reconfigurable accelerators have been proposed to achieve this goal, each exploiting different program properties, and each designed with their own fundamental principles [29], [72], [34], [19], [44]. These principles ultimately decide the magnitude and direction of the benefit vector components, which intuitively quantifies accelerator effectiveness. While early stage results from simulation and modeling provide good estimates, performance prototyping on a physical implementation uncovers the fundamental sources of improvement and bottlenecks. This paper undertakes such a prototype evaluation of DySER, which is based on three principles:

1) Exploit frequent, specializable code regions
2) Dynamically configure accelerator hardware and therefore accelerate code regions
3) Integrate the accelerator tightly, but non-intrusively, to a processor pipeline

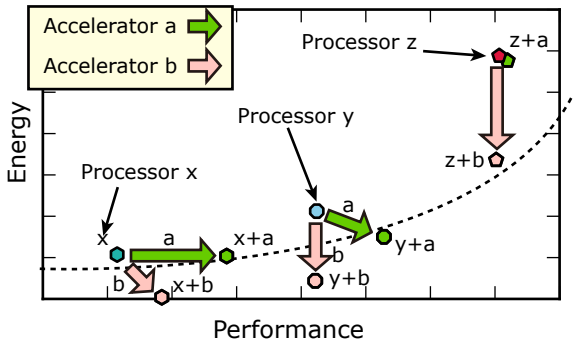


Fig. 1. Energy/Performance Frontier & Accelerators

Prior works have presented DySER's architecture and early stage results [26], [22], ISA design and proof-of-concept integration into OpenSPARC [6], compiler [27] and scheduler [47]. In this paper, we use a performance-capable FPGA-based prototype, its compiler, and meaningful workloads and undertake an end-to-end evaluation of what we call the SPARC-DySER system. Like any full system prototype evaluation, our end goal is to elucidate the merit of the underlying principles using detailed quantitative measurements and analysis of a physical prototype. To this end, we perform the following analyses for the three principles:

1) Compiler analysis: to measure the feasibility and effectiveness of exploiting specializable regions.
2) Performance, power and energy analysis: to quantify the effectiveness of dynamic configuration and its impact on performance and energy.
3) Area analysis: to assess the tight integration of DySER to a processor pipeline.

Our paper's main contribution is in demonstrating that dynamic specialization in a coarse-grained substrate like DySER is achievable by building the end-to-end system: compiler, microarchitecture, core integration. Our findings from this end-to-end evaluation are: First, the DySER compiler can extract frequent specializable regions, and achieves $3\times$ speedup over the original code. Second, DySER hardware speeds up the program through concurrently active functional units, and is limited by the data delivery rate of the integrated processor. Third, irregularity in the code does not limit the compiler, but the overly decoupled nature of the current DySER imple-

mentation limits opportunities. In particular, having conditional loads/stores whose condition is computed inside DySER, or the capability for some load/store execution inside the array would be beneficial. Fourth, DySER can be tightly integrated into a processor with small power and area overhead. Finally, we can compare performance, power, and energy of the DySER prototype to various other platforms like a Cortex-A8, A9, A15 and show DySER provides a compelling alternative to improve performance and reduce power.

Our paper is organized as follows. Section II gives a background of DySER and describes our performance-capable FPGA-based implementation. Section III presents methodology for the quantitative measurements. Sections IV-VII cover the compiler, performance, power/energy, and area analysis respectively. Section IX concludes with lessons learned.

II. DySER Design

DySER's microarchitecture and compiler concepts have been reported in previous publications. This includes the architecture description [26], [22], detailed compiler description [27], scheduler [47], and a proof-of-concept integration [6]. This section first provides an overview and background, and then lists the necessary work to bring the DySER prototype to design that is capable of performance evaluation. Chen-Han [35] describes the detailed microarchitecture and physical design and Govindaraju [23] describes in detail the compiler. We refer the reader to those publications for details.

A. Background

Figure 2 shows an overview of the DySER architecture. First, the original code is processed by the DySER compiler, a process we term "DySERizing." DySERization splits a code region into two components: the computation component and the memory component. The computation is mapped onto the DySER hardware, and the memory access is transformed to include communication instructions which are from DySER's instruction set extensions. One example of a DySER communication instruction is the DySER vector load, shown as *dld_vec* in Figure 2. Further ISA extension details are in [6], [22]. The computation component is executed on DySER by the configuration shown in the blue circle of Figure 2. Through `dconfig` DySER instruction, we can set up the functional units (computation) and switches (interconnection) prior to the accelerated region.

The processor-DySER interface is shown in Figure 2 as the striped boxes between D$ and DySER (described in [22]). The two vector DySER access interfaces are: i) to a single DySER port (*deep* communication), or ii) across multiple ports (*wide* communication). To explain the utility of this feature, we introduce the term *invocation*, which means one instance of the computation for a particular configuration. This deep and wide flexibility allows DySER to vectorize loops via inter- or intra-invocation parallelism, which is intractable for traditional SIMD techniques [27].

The DySER approach relies on the compiler to identify and transform amenable program regions. The DySER compiler, implemented on top of LLVM, creates the aforementioned computation component and memory access component, and represents them with the Access Execute Program Dependence Graph (AEPDG) [27]. The DySER compiler performs transformations on the AEPDG to optimize the memory accesses and vectorize them if possible.

OpenSPLySER is an integration of DySER and OpenSPARC [6], built to demonstrate that non-intrusive integration is possible. It includes many simplifications, including modified switch microarchitecture, flow-control, DySER configuration, output retrieving mechanisms, and DySER size. The largest DySER configuration possible was a 2×2 configuration, or an 8×8 configuration with only 2-bit datapath. Hence, only peak performance was quantified with simple microbenchmarks.

B. From Prototype to Performance Evaluation

The OpenSPLySER design provides support for the claim that DySER is a non-intrusive approach, however, it is not a feasible platform for performance evaluation because of the following reasons: 1) simplifications in integration break the precise state of the processor; 2) it lacks the performance critical vector interface, and other optimizations; and 3) it has limited resources for DySER, due to FPGA size constraints. We describe novel implementations that overcome these hurdles, and they are contributions of this paper.

1) Retire Buffer and Stall-able Design: For simplicity, the OpenSPLySER prototype does not consider OpenSPARC T1 traps and exceptions and hence cannot deploy real workloads. Moreover, because the DySER FIFO resides in a prior stage than the register file, utilizing the existing OpenSPARC rollback and re-execute mechanisms that preserve RF states will lose DySER FIFO states. Therefore, we modify the existing OpenSPARC trap logic to support DySER instructions, and add a three-entry retire buffer at the DySER output, which is shown in Figure 3(a). The retire buffer discards DySER outputs only after all exceptions are resolved.

2) Enhancements for performance: Since OpenSPLySER was not designed for performance evaluation, it did not include the vector interface [22]. To achieve a performance-accurate design and implementation without significantly increasing design complexity, we implemented a simplified vector interface, as shown in Figure 3(b). Essentially, the vector load is emulated by performing a scalar load, and duplicating the data for each appropriate DySER input FIFO. This mechanism is performance-equivalent to *wide* or *deep* loads, and we verify that we do not affect the benchmark's execution path.

3) Coping with FPGA limitations: As previously mentioned, the OpenSPLySER prototype can only fit a small (2X2) DySER or a DySER with a 2-bit datapath. The previously mentioned optimizations reduce the area required for DySER, but are still insufficient to fit a full DySER prototype on the Virtex-5 evaluation board. To mitigate this problem and achieve a performance capable system, our strategy is to remove the unused functional units and switches in DySER, and perform FPGA synthesis for each configuration. Though this means that the prototype does not retain reconfigurability, it is still performance-equivalent and emulates the generic 8x8 DySER. This is because we keep the specialized datapath intact, and we continue to issue `dconfig` instructions, even though they do not actually reconfigure DySER.

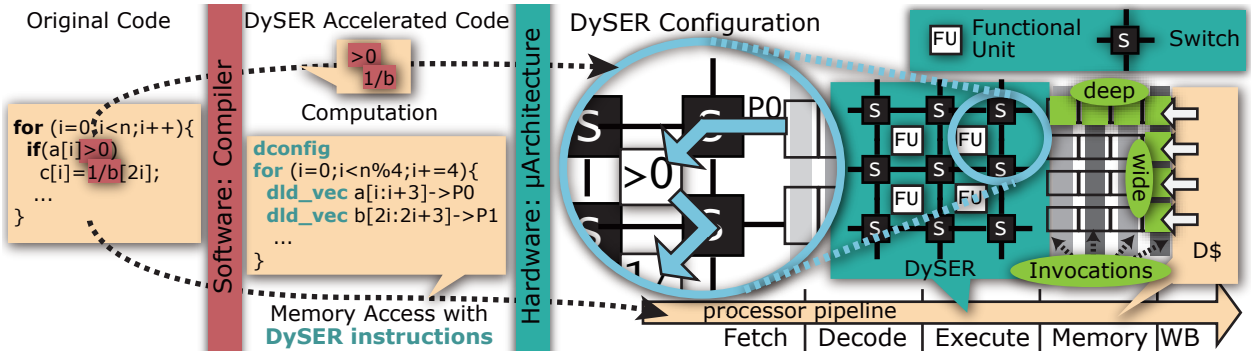


Fig. 2. Overview of the DySER Architecture

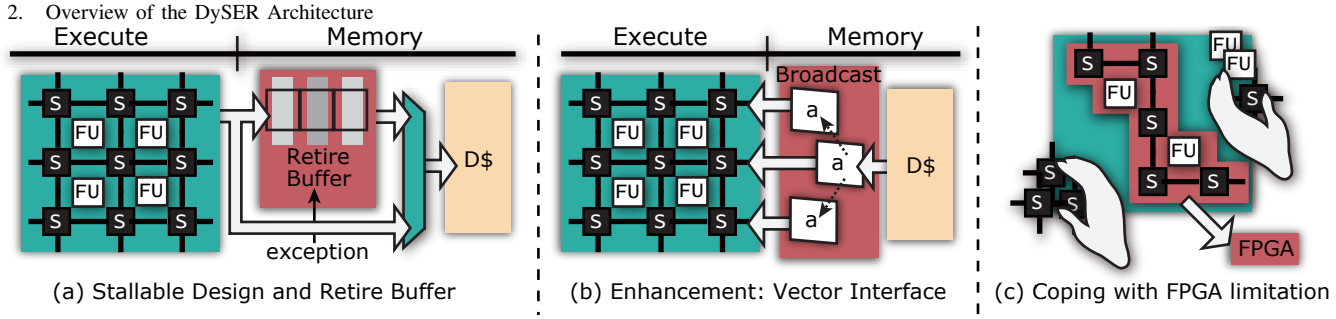


Fig. 3. Summary of Works towards Performance Evaluation

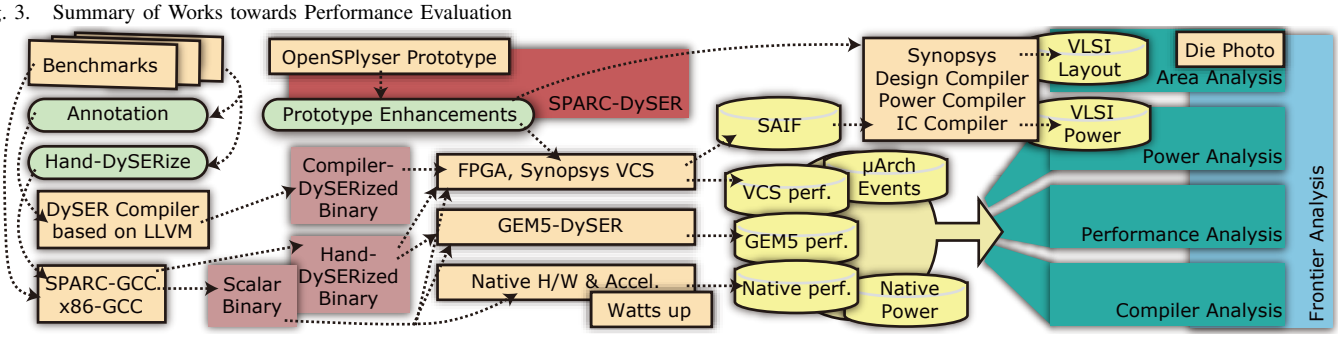


Fig. 4. Overview of the DySER Performance Evaluation Flow

III. Evaluation Methodology

a) Evaluation Flow Outline: Figure 4 gives an overview of our evaluation method. First, we run annotated and DySERized benchmarks on the three implementations of SPARC-DySER: RTL-level Synopsys VCS simulation, FPGA, and gem5-based cycle accurate simulator. To control the experiment, we use VCS simulation to analyze the compiler, performance, power/energy and area. We verified the VCS performance numbers and the architecture events with FPGA numbers and performance counter outputs. The gem5 simulation is only used in Section IX for sensitivity analysis.

Second, the power, area and layout metrics are collected with Synopsys Design Compiler and IC Compiler under VLSI design flow. Third, the performance and area metrics are in addition compared with other state-of-the-art processors. To this end, we run the non-DySERized binaries on different native platforms for performance comparison. The area of other microprocessors are gathered from the literature.

Finally, we select some representative benchmarks in the Parboil [51] suite, throughput kernels from [58], and SPECINT [62]. The Parboil and throughput kernels serve as emerging workloads, and the SPEC benchmarks help us to understand DySER's effectiveness on "code in the wild" or legacy code. The analysis metrics, measurements and benchmarks used are summarized in Table I, and the state-of-the-art processors we used for comparison are in Table II. They represent a wide spectrum ranging from low-power to high-performance processors and also from general purpose to specialized architecture. Overall, our goal is to understand the effectiveness of DySER's three driving principles through quantitative analysis.

b) Role of the FPGA: We elaborate on the role of the FPGA and what it means for performance evaluation on an FPGA. Like any FPGA, ours runs at a much lower frequency than ASIC (50 MHz vs. GHz and higher for the comparison platforms). So our goal in developing the FPGA is to demonstrate that the software runs end to end, that all of the microarchitecture pieces are modeled correctly, to create an emulation platform for running long workloads which aids compiler development. Our performance comparison use cycle counts instead of execution time in seconds for this reason. In all cases, we checked our workloads for correctness by running on our FPGA and comparing the outputs produced. An FPGA prototype by itself is not useful for power measurements. Instead we use VCS simulation with activity from benchmarks for power estimation.

Since our goal is to specifically isolate individual microarchitecture performance sources and bottlenecks, in this paper we report data obtained from VCS simulation instead

Benchmarks	Parboil [51], two throughput kernels, and SPECINT [62].
Metrics	Performance, energy and power.
Performance Measurements	Dynamic instruction counts, cycle counts, and μarch. events.
Energy Measurements	VLSI-based Power (55nm standard-cell library) from Synopsys Power Compiler, annotated with SAIF file. Watts from Watts-up meter for native platforms.
Area Measurements	Area from Synopsys Design Compiler and IC Compiler in 32nm standard-cell library[1]

TABLE I. METRICS AND MEASUREMENTS

	Cortex-A8	Cortex-A9	Cortex-A15	Ivy Bridge	GPU(Tesla)
Proc.	OMAP4430	OMAP3530	Exynos 5	i7-3770k	NVS 295
Freq.	0.6GHz	1GHz	1.7GHz	3.5GHz	540MHz
Board	Begalboard	Pandaboard	Arndaleboard	Desktop	Desktop

TABLE II. SUMMARY OF THE NATIVE PLATFORMS

of reporting coarse-grained performance from the FPGA. This does *not* mean that we are not evaluating the FPGA - it is simply stating that for particular fine-grained measurements to analyze microarchitecture events, VCS cycle traces are easier. By definition, our FPGA execution is simply an accelerated execution of VCS simulation.

IV. COMPILER ANALYSIS

To show that the DySER compiler can exploit the frequent specializable regions (the first DySER principle), we describe the compiler generated regions, and compare the performance of the compiler generated code with the scalar and with the hand DySERized code. We first focus on the emerging workloads and conclude this section describing the generality of the current compiler implementation by compiling SPECINT for DySER. In the remainder of the paper, the term "*DyVec*" refers to vectorized DySER codes and the term "*DySER*" refers to unvectorized codes.

A. Benchmark Characterization

Table III shows the characterization of the most frequently executing regions as determined by the compiler: on average, specialization regions are of length 45 instructions. Of those, the compiler can offload 11 instructions to DySER with an average of 8 communication instructions. With vectorization (DyVec), these communication instructions are amortized and on average 48% of code is off-loaded to DySER.

[1]Because of the lack of 55nm back-end technology library

Benchmark	Scalar	DyAccess	DyOps	DyVec Access	DyVec Ops
fft	58	48	10	17	20
kmeans	43	33	12	24	24
mm	13	13	2	5	16
mriq	24	21	10	14	20
spmv	45	37	8	37	8
stencil	34	27	7	5	14
tpacf	40	30	29	23	29
conv	133	150	16	68	16
radar	20	18	6	8	24
Average	45.6	41.9	11.1	22.3	19

Scalar - # instr. in region, DyAccess- # instr. in access component
DyOps - # ops. in DySER, DyVec Access - # instr. after vectorized
DyVec Ops - # operations in DySER after vectorized

TABLE III. CHARACTERIZATION OF TOP REGIONS

Benchmark	Scalar	DyAccess	DyOps	CFG shape
401.bzip2	21	19	9	Mult-exit
429.mcf	56	61	10	Ctrl-dep-mem
456.hmmer	106	110	7	Ctrl-dep-mem
462.libquantum	16	19	5	Ctrl-dep-mem
464.h264ref	9	9	0	Mult-exit
473.astar	224	224	0	Mult-exit

TABLE IV. CHARACTERIZATION OF TOP REGIONS: SPECINT

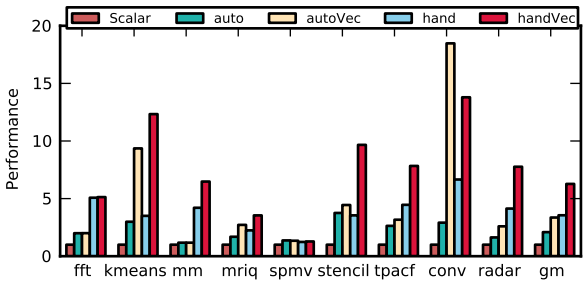


Fig. 5. Compiler Performance

Finding 1: The DySER compiler can extract frequently executed specializable regions.

B. Compiler Performance and Bottlenecks

Figure 5 compares the speedup (in RTL-level simulation) of the compiler and hand DySERized code over the scalar baseline. The `autoVec` and `handVec` shows the compiler (auto) vectorized and hand vectorized performance. On average, compiler generated code performs $1.9\times$ faster than the scalar version. With vectorization, the geometric mean speedup of the DySERized code is $3\times$ over the scalar code.

Compared to compiler DySERized code, hand DySERized `fft`, `mm`, `stencil` and `tpacf` shows significantly better speedup. For `conv`, the compiler generated code actually performs better than the manual version because the hand DySERized code inadvertently created register pressure, causing more registers spilling than necessary. Overall, the major limitation in the DySER compiler compared to hand DySERization is the lack of support for software-pipelining.

Finding 2: The DySER compiler can exploit benchmark characteristics and generate binaries which execute $3\times$ faster than the scalar version.

Finding 3: On average, compiler generated code performs within 50% of the hand optimized code and reduces the number of dynamic instructions with vectorization.

C. Compiler Generality

To understand DySER's effectiveness on legacy codes, we analyzed the SPECINT benchmarks compiled by our compiler.

On a positive note, our compiler produces correct code for all benchmarks and most times finds large specializable regions. However, all of them report slowdowns. We discuss the reasons here. Table IV shows code characteristics produced by our compiler (this data is for the dominant function in the benchmark and is representative of overall behavior). For most cases, the candidate regions are quite large. However, the problem is that the compiler is unable to off-load much to DySER - the DyOps are in single digits, and excess communication instructions create slowdowns.

Overall, these legacy codes have significantly more irregular control-flow graph shapes interacting with memory accesses

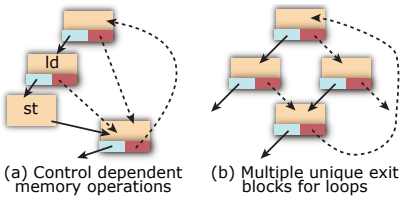


Fig. 6. Compiler on irregular benchmarks

that are not amenable to our current compiler's heuristics. Another reason is an artifact of how LLVM operates—it sometimes creates internal arbitrary precision integers and uses structures directly. This requires sophisticated analysis to be correctly lowered into DySER, which we have not yet implemented. Figure 6 shows the shapes of control flow that are the source of the slowdown and Table IV shows the characterization of selected SPEC benchmarks. In brief, our DySER compiler does not offload the control instructions to DySER because of few control-flow patterns such as the control dependent memory operations (`Ctrl-dep-mem`) and multiple exit blocks (`Mult-exit`).

a) Control dependent memory ops and multiple exits: The current implementation of the compiler schedules all control instructions that have dependent loads, stores or region exit branches into the main processor pipeline along with their backward slices. This limits the number of DySERizable instructions for the two control flow graph shapes shown in Figure 6(a) and 6(b), the control-dependent memory operations and the exit branches. One solution is to use finer-granularity control heuristics that schedule the computation of control instructions to DySER, and only schedule the first branch instruction before the memory operations in the main processor.

b) Multiple small loops:: When a region has multiple small inner loops as shown in Figure 6(c), our compiler treats each loop as a region. To eliminate the need to switch configurations between the loops, it could either schedule computation from multiple small loops to the same configuration, or it could coalesce the inner loops, creating a larger computation region for DySER.

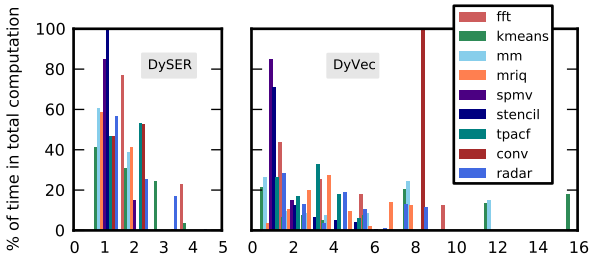
Finding 4: The DySER compiler finds acceleratable regions even on highly irregular legacy codes. However, our DySER implementation cannot accelerate specific irregular patterns such as control-flow dependent memory operations.

V. Performance Analysis

This section quantitatively shows the second DySER principle: DySER can dynamically specialize and accelerate frequent regions in a program. For this purpose, we examine the performance of SPARC-DySER in three perspectives: i) the overall performance compared with other state-of-the-art processors, ii) the source of the performance, and iii) the bottlenecks. To eliminate the aforementioned compiler effects, in this section (and all remaining sections) we use hand DySERized benchmarks for evaluation.

A. Performance Comparison

Figure 7 shows the overall speedup over the baseline (OpenSPARC T1) in terms of cycle counts. We classify the processors into two categories: low-power processors and high performance processors. Among low-power processors,



bench.	fft	km	mm	mriq	spmv	stencil	tpacf	conv	radar
DySER	4	4	2	3	2	1	3	3	4
DyVec	6	16	9	7	2	6	10	8	8
Total	10	23	15	11	14	16	20	16	22

Maximum concurrent active FUs during execution

Fig. 8. Distribution and maximum of concurrent Active FUs

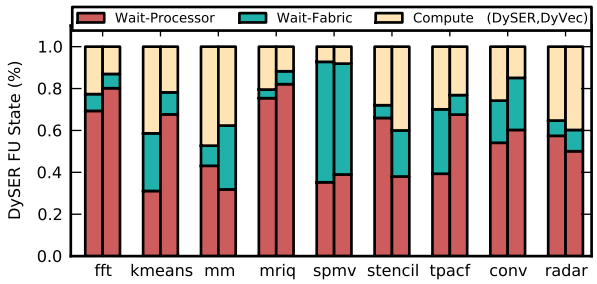


Fig. 9. DySER internal utilization while active

SPARC-DySER is slightly behind the Cortex A9, but outperforms the Cortex A8 on average. This is an important result, as the A8 is a sophisticated, dual issue in-order processor. SPARC-DySER and other low-power processors are far behind (more than 10×) the performance of high-performance processors. GPU acceleration exhibits extraordinary performance in certain benchmarks because the scalar version and the accelerator version use very different algorithms to exploit GPU memory. It is shown for reference, and we did not further analyze and modify the GPU programs.

Finding 5: Equipped with DySER, a single-issue in-order OpenSPARC processor can surpass dual-issue in-order A8, and becomes comparable to a dual-issue out-of-order A9 in performance. Dynamic specialization can energy efficiently provide performance benefits of out-of-order and multi-issue execution.

B. Performance Sources & Bottlenecks

Figure 8 summarizes our observations in the selected Parboil benchmarks and throughput kernels. First, the histogram reports the percentage of time that a given number of FUs are concurrently activated. The unvectorized version frequently has 2 FUs activated in parallel, and the `DyVec` has a wider distribution of parallel activated FUs (from 3 to 8). Second, the table in Figure 8 shows the maximum concurrent activated FUs observed during execution. Except `spmv` and `radar`, most benchmarks could activate more than 40% of the total functional units in parallel. From the above, the major source of speedup is DySER's ability to extract more ILP than the 1-issue SPARC baseline processor. Also, we observed low average/maximum utilization of DySER's functional units. Figure 9 explains this trend, by categorizing the state of DySER functional unit into: i) Wait-Processor,

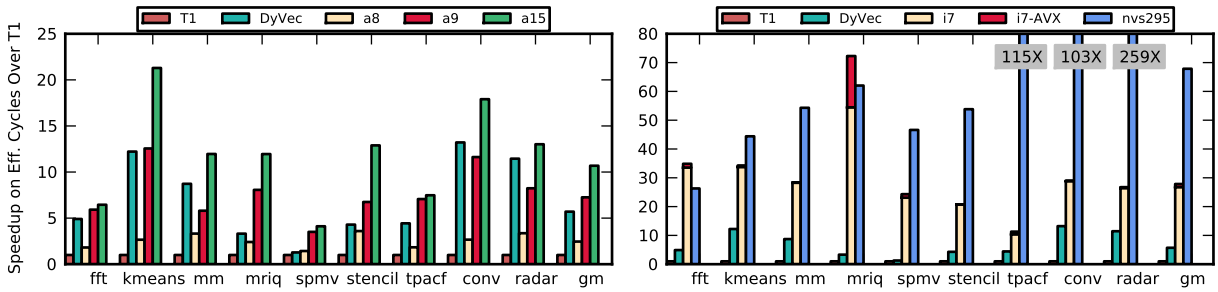


Fig. 7. Architecture Comparison: Performance in cycles

benchmarks	fft	km	mm	mriq	spmv	stencil	tpacf	conv	radar
Stall %	11.32	0	4.50	9.74	0	18.34	2.59	10.31	7.30
Stall % w/ Vec	4.59	3.02	7.94	1.14	0	7.92	5.65	23.21	0

TABLE V. PERCENTAGE OF STALLS ATTRIBUTABLE TO DYSER

Benchmarks	astar	bzip2	h264ref	hmmer	libquantum	mcf
Speedup	1.01	1.00	0.97	1.11	0.75	1.02

TABLE VI. DYSER CHALLENGE BENCHMARKS: SPEC

bench.	astar	bzip2	h264ref	hmmer	libquantum	mcf
DySER	1	2	1	1	1	1
Total	8	5	6	14	6	2

TABLE VII. MAXIMUM OF CONCURRENT ACTIVE FUS: SPEC

which means the functional unit is stalling because at least one of its input data is not fetched by processor pipeline, ii) Wait-Fabric, which means the functional unit is waiting for switches to pass the input data, and iii) Compute, which means it is computing. We observed that the 1-issue in-order SPARC processor is a major bottleneck because of the low memory access performance. When executing the memory access component of the specialized region, the data delivery rate is considerably low, such that DySER is idling and waiting for the data from processor. Furthermore, DySER cannot pipeline iterations because of lacking in data. The functional units often wait for full delay from input to itself, instead of the delay from previous functional unit in the case of pipelining.

The last potential bottleneck comes from the interaction between DySER and the processor pipeline. Table V shows the stall statistics of the SPARC-DySER pipeline. The second and third row shows the stalling ratio in execution. We list the reasons for stalling behavior below:

- `fft, mriq, stencil, and radar`: If the DySER code uses deep vectorization (where multiple instances become pipelined), less stalling is expected because the pipeline stalls due to DySER computation latency are amortized across multiple invocations.
- `kmeans, mm, tpacf, and conv`: If the DySER code uses wide vectorization (where each instance is not pipelined), the computation time cannot be hidden by the long and not-pipelined load latency in OpenSPARC. The processor pipeline now perceives a higher DySER latency, which results in higher stalling time.

Finding 6: DySER provides performance though parallel computation. The data it computes, however, is fetched from the host processor, whose performance is a major bottleneck to DySER.

C. DySER challenge benchmarks: SPEC

In this section, we describe the SPEC benchmarks alone and discuss the performance sources and bottlenecks. First, Table VI shows the overall speedup (slowdown) of SPEC benchmarks accelerated with DySER. Similar to the compiler analysis, the SPEC benchmarks reports negligible speedup over the OpenSPARC baseline with our hand-DySERized codes. Figure VII also shows that the FU activation maximum of our SPECINT implementation is only 1-2. We characterize the reasons as follows:

- Control-flow: Because of the lack of the conditional DySER access instructions (conditional DySER load and stores) in our prototype, more instructions have to be used in the memory access component to check the validity of DySER generated values (`astar` and `libquantum`). Also, `bzip2` has significant control-flow, and hence the DySER FU utilization is low.
- Loop carried dependence: `hmmer` has loop-carried dependence such that our DySER prototype cannot unroll loops and create larger computation components.
- Small computation component: `h264ref` and `mcf`'s frequent executed region has much more memory operations than computation.

Finding 7: DySER provides negligible benefit on programs with low computation to memory ratio and irregularity. The current DySER implementation lacks a conditional interface to accelerate control-heavy programs.

VI. POWER AND ENERGY ANALYSIS

In this section, we only show the results for selected throughput kernels and Parboil benchmarks, since our SPECINT code implementation cannot offload much work onto DySER, as stated in the performance analysis. The goal of this section is to quantify the second principle from a power/energy perspective.

A. Overall Energy and Power

Figure 10 shows the normalized energy of SPARC-DySER over the OpenSPARC baseline, per benchmark, based on cycle counts and the Synopsys power report. On average, *DySER* offers 2× better energy consumption and *DyVec* can achieve 4× energy improvement. Figure 11 shows the per-benchmark power consumption. While the scalar code consumes 4 Watts on average, SPARC-DySER accelerated code consumes between 5 and 6 watts. DySER itself contributes 200mW.

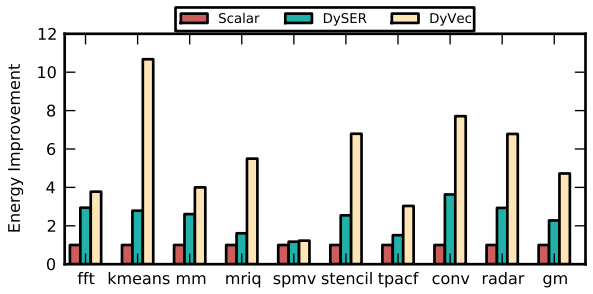


Fig. 10. Energy Improvement: Throughput Kernels & Parboil

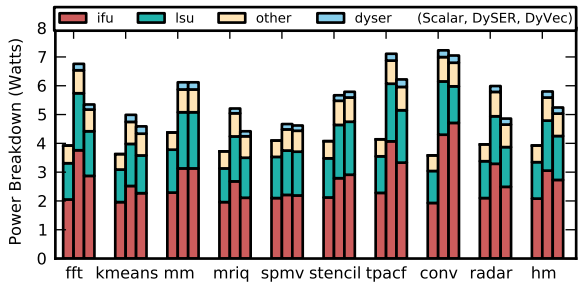


Fig. 11. SPARC-DySER Power Breakdown

Finding 8: DySER's dynamic specialization provides energy efficiency.

B. Energy/Power Sources & Bottlenecks: Throughput Kernels

Figure 11 also shows the breakdown of power sources for throughput kernels, where the components are: *ifu* (instruction fetch unit including I$), *lsu* (load-store unit including D$), *other* (remainder of pipeline, including execution units), and *dyser* (DySER accelerator power). The three bars for each benchmark are the original code, unvectorized DySER code, and vectorized DySER code. From the breakdown, we can observe that DySER (which consumes around 200 mW) is not the major source of power consumption compared to other components. Most of power comes from memory accesses in the *lsu* and *ifu*, though this might be partly attributable to the under-optimization of these units in our synthesis tool. This *lsu* and *ifu* power increases can be explained by examining the actual IPC in Table VIII (the real instructions issued per cycle, in contrast to effective IPC based on scalar instructions). If more instructions are issued per cycle, we naturally consume more power in the instruction fetch and load store units. Also, since we need more DySER instructions to communicate data in the non-vectorized versions, we observe higher power consumption for *DySER* compared to *DyVec*.

Finding 9: The major source of energy improvement is the speedup. SPARC-DySER consumes slightly more power because it executes more instructions in a shorter time period. DySER itself is not a major factor in the power consumption.

VII. Area Analysis

In this section, we compare the area of SPARC-DySER to commercial processors. Through the analysis, we evaluate the third principle that DySER can be tightly integrated into a processor pipeline with negligible complexity.

Figure 12 shows the hierarchical view of the SPARC-DySER layout with the Synopsys 32nm generic library. For this synthesis we used RAM blackboxes based on area estimates from CACTI. The SPARC-DySER core occupies 3.09 mm^2 and DySER occupies about 8% of the area. The wiring overhead is 44% excluding the SRAMs (this is on the higher side but is expected because UltraSPARC's register files are implemented with flip-flops and DySER routers introduce wiring overheads). Within DySER, the internal breakdown is 70% functional units and switch fabric, and 15% each for input and output interface.

bench.	fft	km	mm	mriq	spmv	stencil	tpacf	conv	radar	gm
DySER	0.31	0.20	0.25	0.15	0.10	0.26	0.41	0.46	0.19	0.23
DyVec	0.18	0.12	0.09	0.11	0.09	0.24	0.23	0.50	0.12	0.16

TABLE VIII. SPARC-DySER Actual IPC

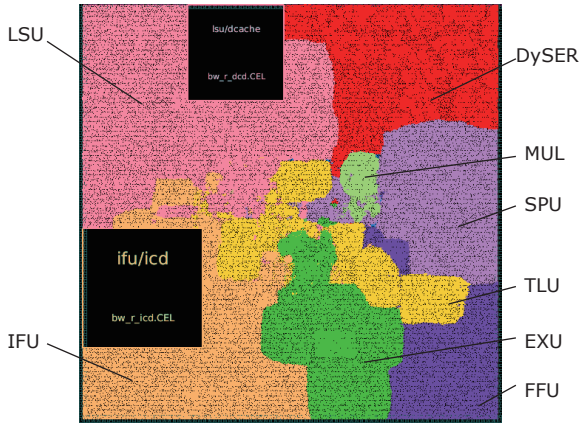


Fig. 12. SPARC-DySER Layout

We show the comparison of core areas in Figure 13. From die photos [1], Intel Atom Bonnell core occupies around $9mm^2$ at 45 nm, AMD Bobcat core occupies $5mm^2$ at 40nm, and ARM Cortex A9 occupies $3.25mm^2$ and $2.45mm^2$ for speed and power optimized versions at 40nm. The comparison shows that the SPARC-DySER is relatively larger in area, and this because of i) the physical implementation is under-optimized, ii) the logical implementation is aimed at FPGA synthesis instead of a low-power VLSI chip, and iii) our functional-unit modules are inefficient. For DySER itself, the simple vector interface can also be improved for lower area.

Finding 10: A SPARC-DySER core occupies $3mm^2$ in 32nm, and DySER itself occupies around 8% of the total area. SPARC-DySER and DySER itself can be further improved by physical-design optimizations in size.

VIII. Related Work

Using specialized architectural designs to improve performance and energy has been an active area of research for

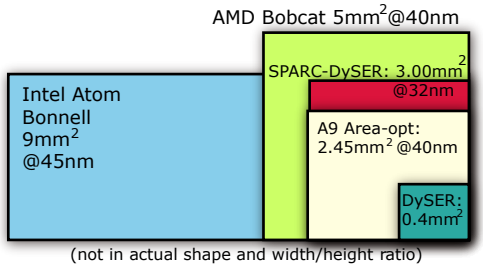


Fig. 13. Architecture Comparison: Area

decades. We discuss below architecture and compiler works related to DySER.

A. Hardware

One of the closest works to DySER from the classical era of supercomputing is the Burroughs Scientific Processor (BSP) [39]. BSP uses highly specialized arithmetic elements that are fully pipelined to accelerate vectorized FORTRAN code. The evolution of three important insights from BSP leads to the DySER architecture. First, to achieve generality, both BSP and DySER utilize compiler support to generate configurations that map computations to an execution substrate. DySER further expands the flexibility and efficiency by introducing a circuit-switched network in the execution substrate. This improvement needs several additional supporting mechanisms in the architecture such as flow control and reconfigurability. Second, both BSP and DySER identify the critical role of intermediate value storage for performance and efficiency. The arithmetic elements in the BSP have dedicated register files which are not part of the architectural state. Unlike this "centralized" design, which is not energy efficient, DySER provides distributed storage in its network using pipeline registers. Third, to generate useful code, the BSP compiler maps vectorized FORTRAN code to a set of prebuilt templates called *vector forms*, which in turn have an efficient mapping to the pipelined arithmetic elements. In contrast, DySER uses a co-designed compiler that can identify arbitrary code-regions and map them to DySER's hardware substrate. A final difference is in the implementation. While the BSP spends much effort on building a fast storage system (register, I/O, special memory), DySER uses a conventional core for efficient data management to achieve the same goal.

We broadly classify the related work from the recent literature into four categories: application specific accelerators, coarse grain reconfigurable accelerators, tiled architectures, and data parallel architectures. Table IX lists the related works and their characteristics compared to DySER. Column 6 of Table IX lists the characteristics of DySER. Govindaraju [23] describes the related work in detail which we summarize here in the interest of space. The design of DySER achieves three goals simultaneously: generality in software, low design complexity in hardware, and efficiency. Basically, it strives to provide efficiency on diverse sets of workloads written in the traditional programming model without increasing design complexity.

Recent Proposals In the last few years, several works have been published on using specialized hardware to achieve energy efficiency or performance with an execution model similar to DySER. Examples include HARP [69], NPU [16], BERET [30], Convolution engine [55], Index Traversal Acceleration [37], Q100 [70], LEAP [32] and SGMF [67]. We describe the connections and influence of DySER in their principles.

HARP [69] seeks to improve the throughput and energy efficiency of large scale data partitioning, especially range partitioning, with a domain specific accelerator and stream buffers. Similar to DySER decoupled access/execute architecture, the HARP accelerator is decoupled from the rest of the microarchitecture with an input and an output stream buffer. Like DySER, ISA extensions are used to manage data transfers from memory to the stream buffers. The accelerator pulls its data from the input stream buffer and delivers its output to the output stream buffer. We can configure DySER to partition the data and use its flexible vector interface to achieve efficiency similar to HARP. However, HARP's dedicated data path to memory, dedicated stream buffers and dedicated hardware is more energy efficient than DySER's general purpose circuit switched interconnect.

NPU [16] proposes an accelerator, called the Neural Processing Unit, which accelerates applications with inexact computation. Many modern applications such as image rendering, signal processing, augmented reality, and data mining have approximatable computation, i.e., they can tolerate a certain degree of error in their outputs. The NPU approach exploits these characteristics by replacing a large code region with an invocation of neural network in the NPU. Similar to a DySER invocation, the main processor communicates with the NPU through input and output FIFOs. Unlike DySER, which creates specialized data paths for the exact computation, NPU accelerates the learned model of the neural network with a specialized sigmoid functional unit and dedicated constant broadcast network. DySER can be adapted to accelerate the neural network model instead of the computation to mimic NPU. However, NPU's dedicated sigmoid functional unit and constant broadcast network provide more efficient support for computing the neural network than the resources available in DySER.

BERET [30] specializes only code-regions with repeated control-flow traces using its subgraph execution blocks (SEB), which are customizable cluster of functional units. Lack of divergent control-flow support limits the number of potential code-regions that can mapped to SEBs. DySER's ability to map control-flow natively helps more code regions to be accelerated with DySER. BERET is integrated with an inorder processor as a coprocessor and does not lend itself to integrate with an out-of-order processor, as it does not have mechanisms to rollback misspeculated computation. Also, implementing SEBs and integrating them to an existing microarchitecture pipeline is hard, since the BERET architecture allows memory operations to be performed from SEBs themselves. In contrast, DySER's decoupled access execute model makes it easier to integrate with an out-of-order processor.

The convolution engine [55] targets image processing kernels and stencil computations by exploiting the key data flow patterns in the kernels. It uses custom load/store units, custom shift registers, map and reduce logic, a complex graph fusion unit, and custom SIMD registers to accelerate convolution and other filter kernels. The programming for the convolution engine is done through compiler specific intrinsics unlike DySER. Since convolution type kernels have more fine grain data level parallelism, we can specialize these kernels with DySER and use its vectorized instruction to achieve high throughput and efficiency.

Meet the Walkers [37] presents an on-chip accelerator, called Widx, for indexing operations in big data analytics. Widx uses a set of programmable hardware units to achieve high performance by accessing multiple hash buckets concurrently and hashing input keys in advance, removing hashing from the critical path. Widx itself is implemented with a

	Application Specific accelerators [41], [71], [31], [50], [20], [48]	CGRAs [15], [33], [18], [60], [73], [11], [13], [12], [42], [10], [45], [43], [14].	Tiled [64], [63], [7]	DLP [61], [21], [46], [38], [40], [56], [59], [52]	DySER [26], [28], [24], [25], [5], [4]
Software					
Generality	Application specific	Loop specific	General Purpose	Loop specific	General Purpose
Scope	Application	Inner Loop	Full	Kernels/ Loops	Code Regions
Flexibility	None	Limited	Yes	Limited	Yes
Hardware					
Overall Complexity	High	Medium	High	Low	Low
Integration	Dedicated	coprocessor/ in-core	Dedicated	coprocessor/ in-core	incore
Area	Large	Large/ Small	Large	Large/ Medium	Medium
Performance	High	Low	Medium	High/ Medium	Medium
Mechanisms					
ISA	New	Co-designed	New	New/ Extension	Extension
Compute Elements	Custom Logic	Functional Units	Cores, RF, buffers	SIMD Units	FU/Switches
Network	Custom	Custom	Packet Switch	Custom	Circuit Switch

TABLE IX. RELATED WORK ON SPECIALIZED ARCHITECTURE

custom RISC processor that supports fused instructions to accelerate hash functions. The accelerator is programmed with a limited subset of C, without any dynamic memory allocation, no stack and with one output. DySER can specialize the indexing operations using its substrate. However, the "Walkers" architecture achieves high throughout by decoupling hashing and hash table walking with a dedicated buffer. Without this dedicated buffer, the DySER architecture stores and loads from memory and consumes memory bandwidth, which may lead to loss of efficiency. As with other domain specific accelerators, the applicability of Walkers outside its chosen domain is limited. In contrast, DySER accelerates a variety of workloads.

The Q100 [70] architecture accelerates database processing tasks with a collection of heterogeneous ASIC tiles that can efficiently perform database primitives like sort and scan. As described by Govindaraju [23], DySER can specialize database primitives and achieve significant energy efficiency. Compared to Q100, which needs separate ASIC tiles for each primitive, DySER can dynamically specialize for each primitive and hence be more area efficient. However, for each specific primitive, Q100 is more energy efficient than DySER because DySER uses its general purpose circuit-switched network.

B. Compiler

In order to achieve high efficiency with coarse grain reconfigurable architectures, a good compiler is essential to manage and exploit the available heterogeneous computing resources available. In addition to being a CGRA compiler, the DySER compiler also borrows vectorization techniques to generate DySER vector instructions to exploit fine grain data level parallelism. We refer the reader to Govindaraju [23] for details and discuss the most relevant works below.

The ispc compiler tries to solve the challenges with SIMD by adopting a new language semantics and trying to overcome compiler problems [53], whereas the DySER approach operates on C/C++ source code and makes the architecture more flexible. Intel Xeon Phi [59], a recent SIMD architecture, and its compiler help programmers tackle the challenges of SIMD through algorithmic changes such as struct-of-arrays to array-of-structs, blocking, and SIMD friendly algorithms, compiler transformations such as parallelization, vectorization, and with scatter/gather hardware support [57]. However, to successfully use them, these changes require heavy programmer intervention and application specific knowledge. The recent HELIX-RC [8] framework is tangentially related to DySER. In short HELIX-RC goes after "irregular" programs with ideas inspired by thread-level-speculation. DySER goes after more "regular" programs with ideas inspired by decoupled access execute and dataflow.

IX. LESSONS LEARNED

A. Breaking the Frontier

Using the analysis of SPARC-DySER's performance, energy, and area, we revisit the performance-energy frontier and position SPARC-DySER. Figure 14 shows SPARC-DySER and other state-of-the-art processors, as well as two simulated OOO-DySER integration points. In this comparison, we take the frequency and the technology node of each processor into account, showing the energy in millijoules and instructions per second. (*DySER* and *DyVEC* use scalar instruction count).

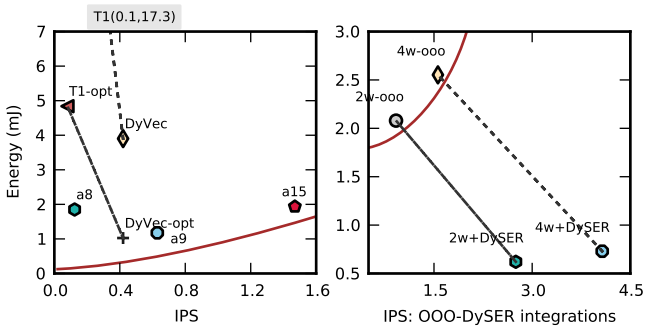


Fig. 14. Energy/Performance Frontier

The *DyVec* point represents SPARC-DySER with the vector interface, and the accelerator benefit vector of DySER is shown in dotted lines that connect *T1* and *DyVec*. From the figure, DySER successfully brings down the energy cost and improves the performance of the OpenSPARC core. The DySER benefit vector tuple in (IPS, mJ) is $(0.34, 13.3)$.

In our Synopsys tools, the OpenSPARC T1 consumes much more power than an ARM-processor. This may be because of the technology library we used, and also our physical implementation is not optimized. As as result, we show two model-based points: *T1-opt* and *DyVec-opt*. These two points represent the projected energy (the performance remains the same) of T1 and SPARC-DySER if the OpenSPARC core consumes the same power as A9. Recall A9 is out-of-order, so this is a conservative projection, and this provides a conservative view of SPARC-DySER's energy/performance behavior. Overall, the energy of DyVEC is above both A8 and A9 even though the performance of DyVEC is in-between. For the projected points *DyVEC-opt*, SPARC-DySER can achieve slightly lower energy than A9, with lower performance.

From the microarchitecture evaluation, we have observed that the OpenSPARC processor pipeline is the major performance bottleneck. To understand DySER's suitability for next generation processors, we show DySER's benefit vector with out-of-order processors on the energy/performance frontier in Figure 14. We use the GEM5 simulator to simulate DySER with 2-wide and 4-wide out-of-order processors. The graph elicits three observations: First, the 4-wide OOO processor has around 2× better performance and 25% lower energy than the 2-wide. Second, DySER improves the performance and energy of the 4-wide processor more than the 2-wide, because DySER is more effective when it can be fed data faster.

Overall, we conclude that DySER integration is one approach that can increase energy efficiency and performance, and could be considered when developing the next generation of an architecture.

B. Prototype Evaluation

The findings of some other prototype evaluations are summarized in Table X. Although quantitative results have sometimes been lower in early stage results because of features eliminated from the prototype compared to design proposals, the studies have lasting impact by establishing the fundamental merit of their underlying principles. For DySER, the early results showed 2.1× speedup across workloads and 10% to 50% on SPECINT. Our current prototyping results show compilation for SPECINT is quite challenging, but establish 6× manually-optimized and 3× compiler-optimized performance improvements on emerging workloads. Qualitatively, the key features between the early-stage design that proved overly complex for the SPARC-DySER prototype are: i) performing speculative loads and stores, and ii) address aliasing within DySER. To some extent, the simple design of OpenSPARC eliminates the potential benefit of these features.

Work	Quantitative results	Demonstrated insights
DySER	*Early-stage*: 2.1 × AVG on workloads [26] *Prototype*: improvement on irregular workloads requires further compiler work, 3× compiler, 6.2× hand,on data-parallel workloads	Dynamic specialization
TRIPS	○ 1 IPC in most SPEC benchmarks ○ best case 6.51 [17]	Dataflow efficiency
RAW	○ up to 10× on ILP workloads ○ up to 100× on stream workloads [65]	Tiled architecture
Wave Scalar	○ 0.8 to 1.6 AIPC on SPEC ○ 10 to 120 AIPC with mutli-threading [54]	Dataflow efficiency
Imagine	IPC from 17 to 40, GFLOPS from 1.3 to 7.3 [3]	Streaming

TABLE X. SUMMARY OF THE PERFORMANCE EVALUATION WORKS

Most prototyping tasks, including RTL implementation, verification, FPGA mapping, compiler implementation, and hand-DySERing code are proved manageable, except for debugging the full system FPGA which was excessively tedious. Reflecting on our experiences, we believe two main things would help future accelerator prototype work:

- High-performance Open-source Processor: It would be advantageous to have open-source implementations of high-performance baseline processors reflecting state-of-the-art designs. Among what is available, OpenRISC [49] and Fabscalar [9] have low performance (OpenRISC's average IPC is 0.2) — and this could impede the prototyping of accelerators.
- Compiler Transformation Framework: Though it was relatively straightforward to design compiler transformations and heuristics, the most time consuming part was in implementation. A tool that took a declarative specification of compiler optimizations and manifested actual compiler transformations could be useful. From almost two decades ago, Sharlit [66] and the Gospel [68] systems provided ideas along these lines. Such frameworks, in a readily usable form, in a production compiler like LLVM or GCC, would be immensely useful for future prototyping works.

The most limiting component we observed in the DySER execution model is the reliance on the processor pipeline for providing data. This is true for both performance and power. Therefore, future developments must be for DySER's data fetching and retrieval engine. The conventional processor has many sophisticated mechanisms to perform memory access. Specializing these mechanisms for DySER would bring further improvement on performance and energy. In all, we think specialization is a promising solution to break the energy/performance frontier.

ACKNOWLEDGMENTS

Support for this research was provided by NSF under the following grants: CCF-0845751, CCF-0917238, and CNS-0917213.

REFERENCES

[1] "ARM Cortex-A7/A9 vs Bobcat, Atom [Online]. Available http://pc.watch.impress.co.jp/video/pcw/docs/487/030/p9.pdf."

[2] "GPGPU: www.gpgpu.org."

[3] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das, "Evaluating the imagine stream architecture."

[4] J. Benson, R. Cofell, C. Frericks, V. Govindaraju, C.-H. Ho, Z. Marzec, T. Nowatzki, and K. Sankaralingam, "Prototyping the DySER Specialization Architecture with OpenSPARC," in *Hot Chips 24*, August 2012.

[5] J. Benson, R. Cofell, C. Frericks, C.-H. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Design Integration and Implementation of the DySER Hardware Accelerator into OpenSPARC," in *Proceedings of 18th International Conference on High Performance Computer Architecture (HPCA)*, 2012.

[6] Benson et al., "Design, integration and implementation of the dyser hardware accelerator into opensparc," in *HPCA '12*.

[7] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and t. T. Team, "Scaling to the End of Silicon with EDGE Architectures," *Computer*, vol. 37, no. 7, pp. 44–55, Jul. 2004. [Online]. Available: http://dx.doi.org/10.1109/MC.2004.65

[8] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks, "Helix-rc: An architecture-compiler co-design for automatic parallelization of irregular programs," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 217–228. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665705

[9] Choudhary et al., "Fabscalar: Composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template," in *ISCA '11*.

[10] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The reconfigurable streaming vector processor (rsvptm)," in *MICRO 36*, 2003, p. 141.

[11] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner, "An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 272–283. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2005.9

[12] N. Clark, A. Hormati, and S. Mahlke, "VEAL: Virtualized Execution Accelerator for Loops," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 389–400. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.33

[13] 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 37*, 2004, pp. 30–40.

[14] A. Deb, J. M. Codina, and A. González, "SoftHV: A HW/SW Co-designed Processor with Horizontal and Vertical Fusion," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ser. CF '11. New York, NY, USA: ACM, 2011, pp. 1:1–1:10. [Online]. Available: http://doi.acm.org/10.1145/2016604.2016606

[15] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD - Reconfigurable Pipelined Datapath," in *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, ser. FPL '96. London, UK, UK: Springer-Verlag, 1996, pp. 126–135. [Online]. Available: http://dl.acm.org/citation.cfm?id=647923.741212

[16] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 449–460. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.48

[17] Gebhart et al., "An evaluation of the trips computer system," in *ASPLOS '09*.

[18] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *Computer*, vol. 33, no. 4, pp. 70–77, Apr. 2000. [Online]. Available: http://dx.doi.org/10.1109/2.839324

[19] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *IEEE Computer*, vol. 33, no. 4, pp. 70–77, April 2000.

[20] R. E. Gonzalez, "Xtensa: A Configurable and Extensible Processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, Mar. 2000. [Online]. Available: http://dx.doi.org/10.1109/40.848473

[21] C. Gou, G. Kuzmanov, and G. N. Gaydadjiev, "SAMS Multi-layout Memory: Providing Multiple Views of Data to Boost SIMD Performance," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 179–188. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810111

[22] 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," *Micro, IEEE*, vol. 32, no. 5, pp. 38–51, 2012.

[23] V. Govindaraju, "Energy Efficient Computing Through Compiler Assisted Dynamic Specialization, PhD Dissertation, UW-Madison, 2014."

[24] 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*, vol. 32, no. 5, pp. 38–51, Sep. 2012. [Online]. Available: http://dx.doi.org/10.1109/MM.2012.51

[25] V. Govindaraju, C.-H. Ho, T. Nowatzki, and K. Sankaralingam, "Mechanisms for Parallelism Specialization for the DySER Architecture," UW-Madison, Tech. Rep. TR-1773, June 2012.

[26] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *HPCA '11*.

[27] V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Breaking simd shackles: Liberating accelerators by exposing flexible microarchitectural mechanisms," in *PACT '13, To appear.*

[28] ——, "Breaking SIMD Shackles with an Exposed Flexible Microarchitecture and the Access Execute PDG," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 341–352. [Online]. Available: http://dl.acm.org/citation.cfm?id=2523721.2523767

[29] 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*.

[30] ——, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44 '11. New York, NY, USA: ACM, 2011, pp. 12–23. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155623

[31] 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 *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 37–47. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815968

[32] E. Harris, S. Wasmundt, L. D. Carli, K. Sankaralingam, and C. Estan, "LEAP: Latency- Energy- and Area-optimized Lookup Pipeline," in *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, October 2012.

[33] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, ser. FCCM '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 12–. [Online]. Available: http://dl.acm.org/citation.cfm?id=549928.795741

[34] ——, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997.

[35] C.-H. Ho, "Mechanisms Towards Energy-Efficient Dynamic Hardware Specialization, PhD Dissertation, UW-Madison, 2014."

[36] Kelm et al., "Rigel: an architecture and scalable programming interface for a 1000-core accelerator," in *ISCA '09*.

[37] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the Walkers: Accelerating Index Traversals for In-memory Databases," in *Proceedings of the 46th Annual IEEE/ACM*

International Symposium on Microarchitecture, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 468–479. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540748

[38] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The Vector-Thread Architecture," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 52–. [Online]. Available: http://dl.acm.org/citation.cfm?id=998680.1006736

[39] D. J. Kuck and R. A. Stokes, "The Burroughs Scientific Processor (BSP)," *IEEE Trans. Comput.*, vol. 31, no. 5, pp. 363–376, May 1982. [Online]. Available: http://dx.doi.org/10.1109/TC.1982.1676014

[40] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the Tradeoffs Between Programmability and Efficiency in Data-parallel Accelerators," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 129–140, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024723.2000080

[41] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "SODA: A Low-power Architecture For Software Radio," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ser. ISCA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 89–101. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2006.37

[42] B. Mathew and A. Davis, "A loop accelerator for low power embedded vliw processors," in *CODES+ISSS '04*, pp. 6–11.

[43] M. Mishra and S. Goldstein, "Virtualization on the Tartan Reconfigurable Architecture," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, Aug 2007, pp. 323–330.

[44] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, "Tartan: evaluating spatial computation for whole program execution," in *ASPLOS-XII*, pp. 163–174.

[45] ——, "Tartan: evaluating spatial computation for whole program execution," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-XII. New York, NY, USA: ACM, 2006, pp. 163–174. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168878

[46] J. Nickolls and W. Dally, "The GPU Computing Era," *Micro, IEEE*, vol. 30, no. 2, pp. 56–69, March 2010.

[47] T. Nowatzki, M. Sartin-Tarm, L. D. Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," in *PLDI '13*.

[48] "Opencores project home. http://opencores.org/."

[49] "OpenRISC, [Online]. Available http://opencores.org/or1k/Main_Page."

[50] M. Papadonikolakis, V. Pantazis, and A. P. Kakarountas, "Efficient High-performance ASIC Implementation of JPEG-LS Encoder," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '07. San Jose, CA, USA: EDA Consortium, 2007, pp. 159–164. [Online]. Available: http://dl.acm.org/citation.cfm?id=1266366.1266402

[51] "Parboil benchmark suite, http://impact.crhc.illinois.edu/parboil.php."

[52] Y. Park, J. J. K. Park, H. Park, and S. Mahlke, "Libra: Tailoring SIMD Execution using Heterogeneous Hardware and Dynamic Configurability," in *MICRO '12*, 2012.

[53] M. Pharr and W. R. Mark, "ispc: A SPMD Compiler for High-Performance CPU Programming," in *InPar 2012*, 2012.

[54] A. Putnam, S. Swanson, K. Michelson, M. Mercaldi, A. Petersen, A. Schwerin, M. Oskin, and S. J. Eggers, "The Microarchitecture of a Pipelined WaveScalar Processor: An RTL-based study," Tech. Rep. TR-2005-11-02, 2005.

[55] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 24–35, Jun. 2013. [Online]. Available: http://doi.acm.org/10.1145/2508148.2485925

[56] K. Sankaralingam, S. W. Keckler, W. R. Mark, and D. Burger, "Universal mechanisms for data-parallel architectures," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 303–. [Online]. Available: http://dl.acm.org/citation.cfm?id=956417.956548

[57] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?" in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 440–451. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337210

[58] Satish et al., "Can traditional programming bridge the ninja performance gap for parallel computing applications?" in *ISCA '12*.

[59] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," in *ACM SIGGRAPH 2008*, pp. 18:1–18:15.

[60] H. Singh, M.-H. Lee, G. Lu, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho, "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 465–481, May 2000. [Online]. Available: http://dx.doi.org/10.1109/12.859540

[61] J. E. Smith, G. Faanes, and R. Sugumar, "Vector instruction set support for conditional operations," in *ISCA '00*, 2000.

[62] *SPEC CPU2000*. Standard Performance Evaluation Corporation, 2000.

[63] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "WaveScalar," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 291–. [Online]. Available: http://dl.acm.org/citation.cfm?id=956417.956546

[64] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, Mar. 2002. [Online]. Available: http://dx.doi.org/10.1109/MM.2002.997877

[65] Taylor et al., "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," in *ISCA '04*.

[66] S. W. K. Tjiang and J. L. Hennessy, "Sharlit - a tool for building optimizers," in *PLDI '92*.

[67] D. Voitsechov and Y. Etsion, "Single-Graph Multiple Flows: Energy Efficient Design Alternative for GPGPUs," in *Proceedings of the 41st annual international symposium on Computer architecture*, ser. ISCA '14, June 2014.

[68] D. L. Whitfield and M. L. Soffa, "An approach for exploring code improving transformations," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 6, pp. 1053–1084, Nov. 1997.

[69] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross, "Navigating big data with high-throughput, energy-efficient data partitioning," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 249–260, Jun. 2013. [Online]. Available: http://doi.acm.org/10.1145/2508148.2485944

[70] 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 *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 255–268. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541961

[71] L. Wu, C. Weaver, and T. Austin, "CryptoManiac: A Fast Flexible Architecture for Secure Communication," *SIGARCH Comput. Archit. News*, vol. 29, no. 2, pp. 110–119, May 2001. [Online]. Available: http://doi.acm.org/10.1145/384285.379256

[72] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *ISCA '00*.

[73] ——, "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proceedings of the 27th annual international symposium on Computer architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 225–235. [Online]. Available: http://doi.acm.org/10.1145/339647.339687