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

Chen-Han Ho, Venkatraman Govindaraju, Tony Nowatzki,
Zachary Marzec, Preeti Agarwal, Chris Frericks, Ryan Cofell, Jesse Benson,
Karthikeyan Sankaralingam
Vertical Research Group
Department of Computer Sciences
University of Wisconsin-Madison.
http://research.cs.wisc.edu/vertical

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, namely: exploiting specializable regions, dynamically specializing hardware, and tight processor integration. To this end, we explore the accelerator's performance, power, and area, and consider comparisons to state-of-the-art microprocessors using energy/performance frontier analysis of both the prototype and simulated DySER-accelerated cores.

Among many positive findings, two key ones are: i) the DySER execution model and microarchitecture provides energy efficient speedups and it does not introduce overheads due to its internal microarchitecture management structures – overall, DySER's performance improvement to OpenSPARC is 6×, consuming only 200mW ; ii) on the compiler side, the DySER compiler is effective on computationally intensive regular and irregular code. However, some challenges restrict DySER's domain of effectiveness: i) Some corner cases curtail the compiler's effectiveness for arbitrarily organized code like the SPEC benchmarks; ii) It is possible that OpenSPARC's limited performance masks the challenges and bottlenecks of integration with higher performance cores.

1. Introduction

For building the next generation of processors, accelerators are becoming a primary approach for boosting performance and energy efficiency [12, 16, 29, 15, 2, 14, 10]. Accelerators 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 [14, 29, 15, 10, 18]. These principles ultimately decide the magnitude and direction of the benefit vector components. This metric 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.

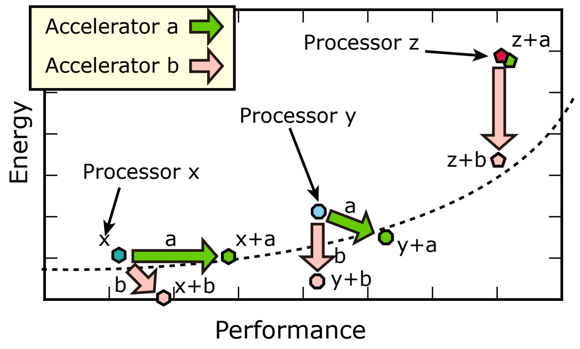
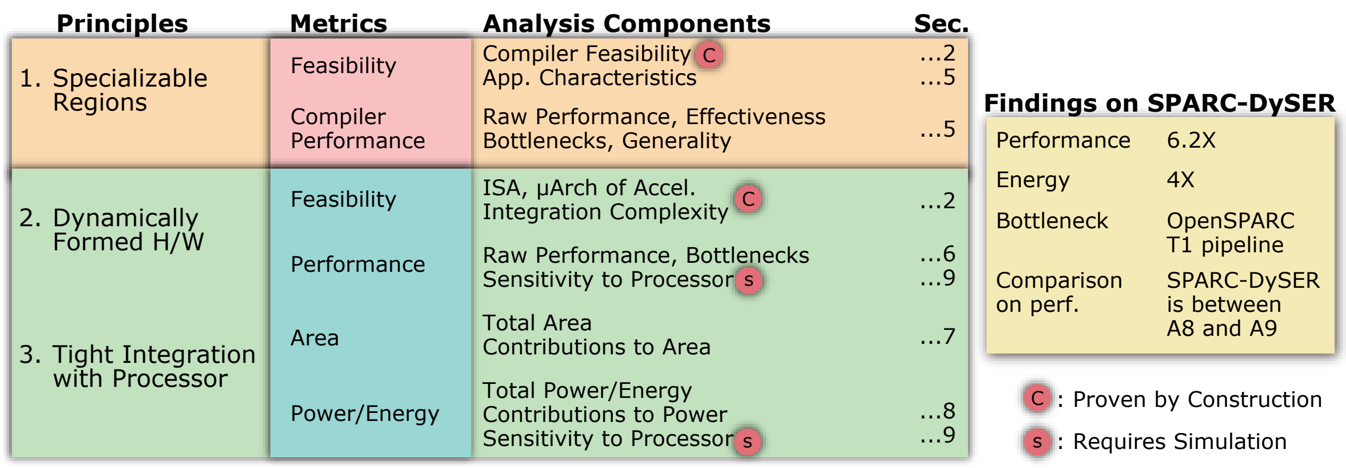


Figure 1: Energy/Performance Frontier & Accelerators

This paper undertakes such a prototype evaluation of the DySER accelerator which is based on three principles:

1. To exploit frequently executed, specializable code regions
2. To dynamically configure accelerator hardware for particular regions
3. To integrate the accelerator tightly, but non-intrusively, to a processor pipeline

Prior work has presented DySER's architecture and early stage results [12, 11], ISA design and proof-of-concept integration into OpenSPARC [6], compiler [13] and scheduler [19]. In this paper, we use a performance-capable FPGA-based prototype, its compiler, 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 that end, we have organized measurements and



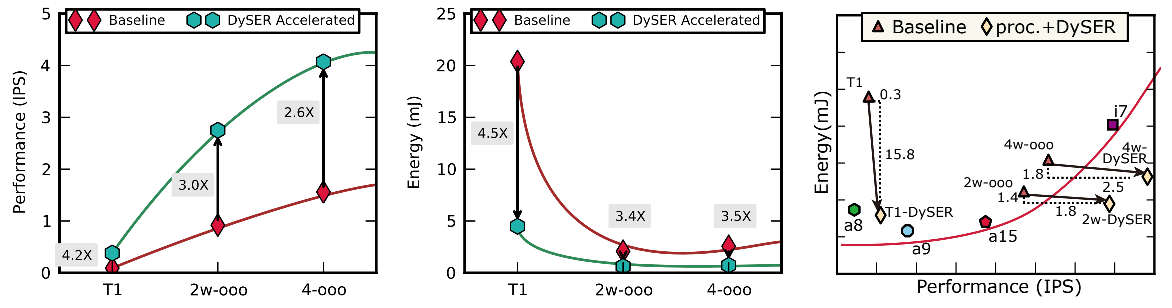
Principles	Metrics	Analysis Components	Sec.
1. Specializable Regions	Feasibility	Compiler Feasibility C	...2
		App. Characteristics	...5
	Compiler Performance	Raw Performance, Effectiveness Bottlenecks, Generality	...5
2. Dynamically Formed H/W	Feasibility	ISA, μArch of Accel. Integration Complexity C	...2
	Performance	Raw Performance, Bottlenecks	...6
		Sensitivity to Processor S	...9
3. Tight Integration with Processor	Area	Total Area Contributions to Area	...7
	Power/Energy	Total Power/Energy Contributions to Power	...8
		Sensitivity to Processor S	...9

Findings on SPARC-DySER

Performance	6.2X
Energy	4X
Bottleneck	OpenSPARC T1 pipeline
Comparison on perf.	SPARC-DySER is between A8 and A9

C : Proven by Construction
S : Requires Simulation

Figure 2: DySER Principles and Analysis Overview



(a) Performance vector (b) Energy vector (c) Energy-performance frontier

Figure 3: Executive Summary of the DySER benefit Vectors

analysis as shown in Figure 2. The three principles outlined above are evaluated with appropriate metrics. Each metric itself is broken into distinct analysis components, and for each we name the corresponding section of the paper. Analysis components are either quantitative or are proven by construction (marked with a C on the Figure).

The analysis and data is organized in parallel structure across the different metrics as detailed next. The first DySER principle is evaluated primarily with compiler analysis in terms of the feasibility of finding specializable regions and the ability of the compiler to use these regions to generate high-performance code. Feasibility is shown both by the characteristics of applications, and by construction through prior work. The compiler performance is demonstrated through the analysis components of effectiveness, bottlenecks which limit effectiveness, and generality to all types of code. The second two DySER principles, dynamically formed hardware and tight processor integration, are closely intertwined, and are therefore jointly evaluated. The feasibility of both principles has been shown constructively in prior work and reviewed in Sections 2 and 3. The metrics of performance, area, and power are each evaluated through three analysis components: i) the raw metric itself; ii) the bottlenecks which limit the metric; and iii) and the sensitivity of the metric to the baseline processor. Figure 2 includes a table summarizing key findings.

A complex aspect of the study is the sensitivity of the performance and energy metrics to the baseline processor. Since it requires simulation, we defer it to a section of its own (Section 9) to avoid intermingling simulator data with prototype measurements. We use energy frontier curves and the benefit vector concept for this analysis — we form the DySER benefit vector, and analyze the position of DySER-accelerated processors in the energy/performance frontier. This captures DySER's benefits in a nut-shell — Figure 3(a) shows the performance of DySER accelerated cores and the performance component of the benefit vector in IPS (instructions per second). Figure 3(b) shows the energy of DySER accelerated cores and the energy component of the benefit vector. Figure 3(c) shows the energy/performance frontier for all processors considered in this study. The overall finding is the following: *A DySER accelerator provides more performance and energy benefits as the baseline processor improves. However, the speedup and energy improvement ratio will degrade because a better baseline processor is more difficult to accelerate. The energy improvement ratio would be close to the speedup ratio for a simple and power efficient implementation.*

Our paper is organized as follows. Sections 2 and 3 cover the feasibility metric. Section 4 presents methodology for the quantitative measurements. Sections 5-9 cover the compiler, performance, area, power/energy, and frontier analysis respectively. Section 10 concludes with lessons learned.

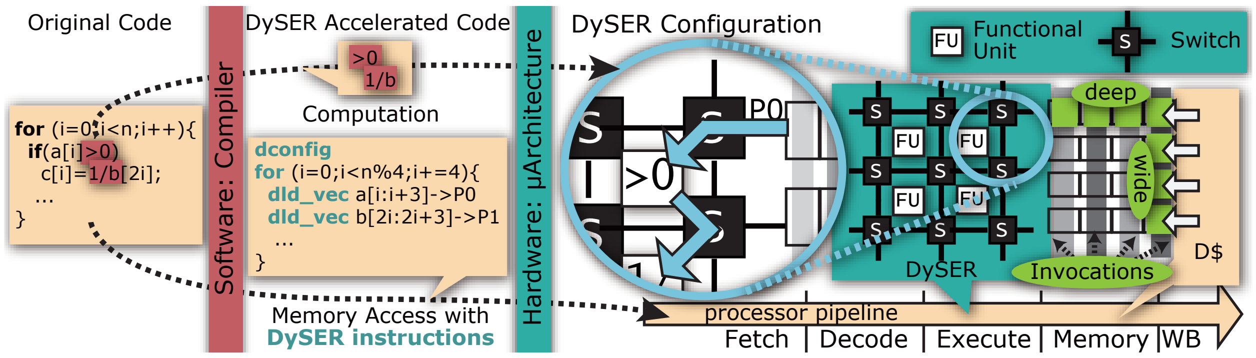


Figure 4: Overview of the DySER Architecture

2. DySER Design

DySER's microarchitecture and compiler concepts have been reported in previous publications. This includes the architecture description [12, 11], detailed compiler description [13], scheduler [19], and a proof-of-concept integration [6]. This material, which we summarize below, demonstrates the feasibility of the three DySER principles.

DySER & Execution Model DySER (Dynamically Specialized Execution Resources), offers a reconfigurable way to build efficient specialized datapaths to offload work from the energy-hungry processor pipeline.

Figure 4 shows an overview of the DySER architecture. First, the original code is processed by the DySER compiler, a process we term "DySERizing." DySERization splits the original code into two components; one is the computation component that is suitable for DySER acceleration, and the other is the memory access component, which consists of loads, stores and supporting instructions. Communication between the memory access component (on the main processor) and the computation component (on DySER) occurs with ISA extensions, through which the processor pipeline can send data to or retrieve data from the DySER hardware. One example of a DySER instruction is the DySER vector load, shown as *dld_vec* in Figure 4. Further ISA details are in [6, 11].

The computation component is shown in the blue circle of Figure 4. The configuration is mapped onto DySER through the dconfig DySER instruction, which sets up the functional units and switches prior to the region of code which uses the particular configuration. The functional units perform computations, and the switches construct a light-weight circuit-switched network for the data values.

The processor-DySER interface is shown in Figure 4 as the striped boxes between D$ and DySER (described in [11]). The interface allows a vector operation to communicate either to a single DySER port (*deep* communication), or 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. Each input data of an invocation forms a wavefront across each logical input FIFO. A deep vector operation transmits the same input of the computation across multiple DySER invocations, while a wide vector operation transmits different data elements for the same invocation. This flexibility allows DySER to vectorize loops which are intractable for traditional SIMD techniques[13].

DySER Compiler The DySER approach relies on the compiler to identify and transform regions of programs that can be accelerated by DySER. In brief, the compiler creates the mentioned computation component and memory access component, and represents them with the Access Execute Program Dependence Graph (AEPDG) [13]. The DySER compiler performs transformations on the AEPDG to vectorize the execution, and outperforms ICC for SSE and AVX by 1.8×.

DySER Proof-of-Concept 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.

3. From Prototype to Performance Evaluation

Though the original prototype, OpenSPLySER, provides support for the claim that DySER is a non-intrusive approach, it is not a feasible platform for performance evaluation because: 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 overcome these hurdles, and achieve a performance-capable prototype by: 1) replacing the integration between OpenSPARC and DySER with a stall-able design; 2) enhancing the prototype with a vector interface and optimizing the microarchitectural and physical implementation; and 3) coping with the FPGA resource limitations by removing unused functional units and switches in DySER for each benchmark. Figure 5 summarizes these three approaches, and we describe them in more detail below.

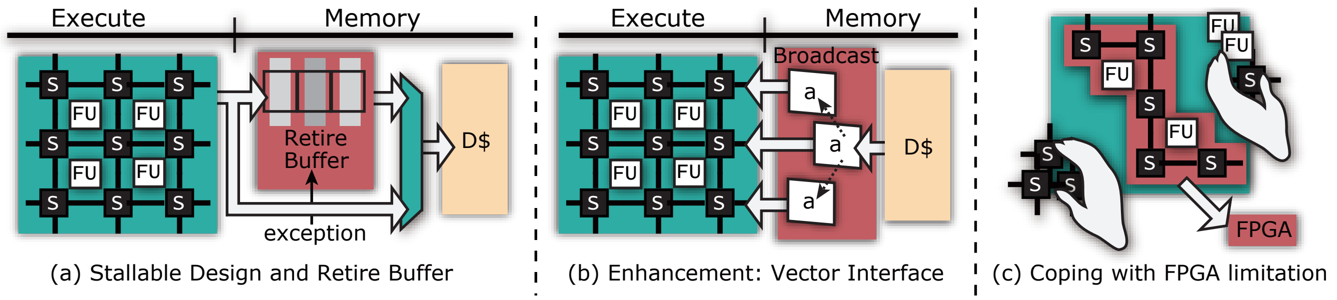

Execute
Memory
S
FU
Retire
Buffer
D$
exception
(a) Stallable Design and Retire Buffer
Execute
Memory
Broadcast
S
FU
a
D$
(b) Enhancement: Vector Interface
S
FU
FPGA
(c) Coping with FPGA limitation

Figure 5: Summary of Works towards Performance Evaluation

3.1. Retire Buffer and Stall-able Design

To simplify the implementation, the OpenSPLySER prototype does not consider OpenSPARC T1 traps and exceptions. As a result, when moving from microbenchmarks to real workloads, it sometimes corrupts the architectural state. We solve this by modifying existing rollback mechanisms to be consistent with DySER's view of the architectural state.

However, this is not a full solution for handling DySER receive instructions, as they update the interface FIFOs (part of the architectural state) before the writeback stage. This means that a re-executed receive, because of some exception at writeback, may not find the data in the interface FIFO. Therefore, we add a three-entry retire buffer at the DySER output, which is shown in Figure 5(a). The retire buffer discards DySER outputs only after all exceptions are resolved.

3.2. Enhancements for performance

A vector interface for DySER-processor communication, allowing multiple values injected per input port, dramatically improves performance [11]. Since OpenSPLySER was not designed for performance evaluation it did not include this interface. Integrating this to OpenSPARC's memory pipeline is quite complex. To achieve a performance-accurate design and implementation without significantly increasing design complexity, we implemented a simplified vector interface design, as shown in Figure 5(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.

In addition to the vector interface, other minor optimizations are made to further increase the performance and area of DySER. These include an improved switch flow control protocol implementation and FPGA physical optimization passes, such as register retiming.

3.3. Coping with FPGA limitation

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 for each configuration, and perform FPGA synthesis for each configuration. By removing the unused devices, we are able to synthesize and map all critical configurations found in our benchmarks. 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.

4. Evaluation Methodology

The previous sections demonstrated the feasibility of the three DySER principles, and we now begin the discussion of quantitative metrics. This section describes our methodology for measurements and analysis, shown by Figure 6. Beginning with annotating and DySERizing the benchmarks, we run the binaries on VCS, the FPGA, and GEM5-based cycle accurate simulator to acquire performance metrics. We also run the non-DySERized binaries on other native platforms for comparison analysis. On the hardware side, we enhance the SPARC-DySER as mentioned, using VCS to generate architectural events, and use the Synopsys Design Compiler and IC Compiler, to acquire the power, area and layout metrics. The analysis metrics, measurements and benchmarks used are summarized in Table 1.

The most meaningful workloads for this study should have sufficient computation and be representative of emerging areas. To avoid selection bias, our goal was to pick existing suites - the Parboil [21] suite and throughput kernels from [23] meet both needs. While they are complex and challenging, they are small enough that they allow detailed understanding to extract out bottlenecks and insights. Time-constraints limited us to only two throughput kernels - convolution and radar. The Parboil and throughput kernels are the primary benchmarks considered across all metrics. It is also important to understand effectiveness on "code in the wild" or legacy code. To that end, we also examined the SPEC benchmarks.

In addition to raw performance, we compare DySER's effectiveness to state-of-the-art processors, and study the sensitivity to the baseline processor for integration. We make a best effort for fair and rigorous comparisons as described below. We choose three representative "architecture types" for comparison. First, ARM's Cortex A8, A9 and A15 architectures represent low-power microprocessors. Second, Intel's x86 Ivy

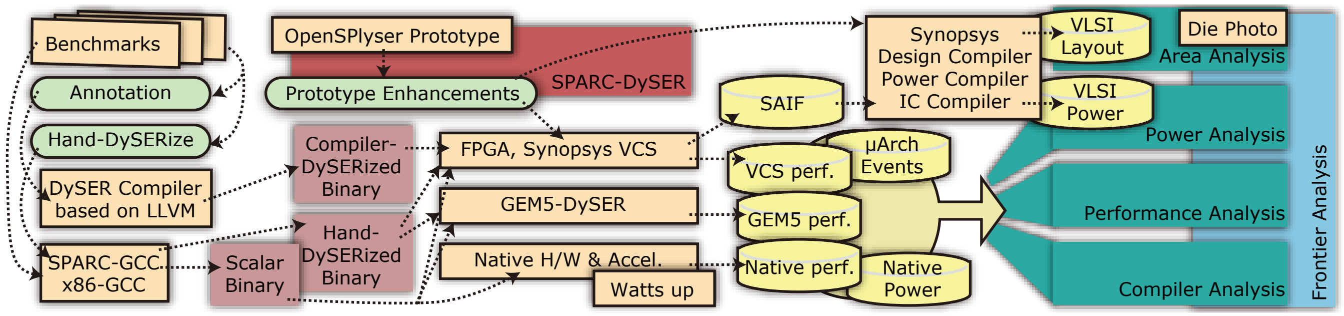


Figure 6: Overview of the DySER Performance Evaluation Flow

Benchmarks	Parboil [21], two throughput kernels, and SPECINT [24].
Metrics	Performance, energy and area.
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 1: 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 2: Summary of the Native Platforms

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 3: Characterization of Top Regions

Bridge architecture, with SSE and AVX support, represent a high-performance general purpose microprocessor with a SIMD accelerator. We use GCC to compile our scalar version, and use its auto-vectorizer to generate SIMD accelerated binaries. Last, we chose NVIDIA's SIMT architecture, as the accelerator principles are very different. We use a one SM GPU as our native GPU platform to keep the SIMD width and accelerator area in a roughly comparable to other accelerators. Detailed specification of native platforms are listed in Table 2.

Limitations We believe that the main limitation of our methodology comes from area and power measurements. SPARC-DySER's measurements are from layout and annotated power simulation. However, the native platforms' power are measured with a Watts up meter (more accurate than simulating native platforms with McPAT etc.). We did our best effort to eliminate errors here.

Organization Across the next four sections, we begin with a guiding question, present analysis, and summarize with inferences. In all results, the term *DyVec* refers to vectorized DySER code and the term *DySER* refers to unvectorized code.

5. Compiler Analysis

The DySER compiler relies on the architecture's fundamental principle that there exists code regions which are specializable. In this section, we show that the DySER compiler can find and target these regions. To this end, we first present how much computation in the frequently executed regions we can offload to DySER. Second, we present the speedup of the compiled DySER code over the scalar code. Third, we describe the compiler effectiveness quantitatively by comparing the performance of hand DySERized binary to that of compiler generated binary. We first focus on the emerging workloads and conclude this section describing generality of the current compiler implementation by compiling SPECINT for DySER.

5.1. Benchmark Characterization

Q: Do the applications have specializable regions?

Table 3 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 overhead instructions, which are required for DySER-processor communication. With vectorization, these communication instructions are amortized and on average 48% of code is off-loaded to DySER. Note that the code on the processor is now different and as we will show in Section 6, effectively both the off-loaded code and processor's code are speeded up.

Observation: The DySER compiler can extract frequently executed specializable regions.

5.2. Compiler Performance

Q: What is the performance of compiler DySERized code?

Figure 7 shows the speedup of the automatically DySERized benchmarks over the scalar version of the code. On average, compiler generated code performs 1.9× faster than the scalar version. When DySERized regions have data-level parallelism, the compiler utilizes the vector interface to DySER and re-

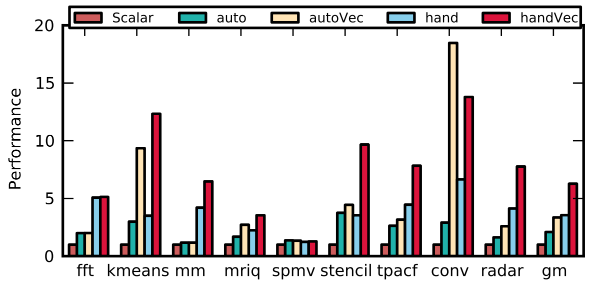


Figure 7: Compiler Performance

Benchmark	Characteristics	Auto-Vectorized?
fft	Strided memory, variable loop count	Yes
kmeans	Regular memory	Yes
mm	Regular memory after blocking	Yes†
mriq	Regular memory	Yes†
spmv	Indirect memory access	No
stencil	Regular memory	Yes†
tpacf	Regular control, irregular mem	Partial
conv	Regular memory and control	Yes
radar	Regular memory and control	Yes†

Table 4: Bench. Characteristics (†vectorized with scalar registers)

duces the number of DySER loads and DySER stores. With vectorization, the geometric mean speedup of the DySERized code is 3× over the scalar code.

Table 4 lists the characteristics of the benchmarks, which can lead to performance gain with DySER. The compiler exploits the regular memory access and regular control to generate code that can effectively utilize DySER's functional units. When the DySER compiler vectorizes the application automatically (Column 3), it uses the vector-memory interface to further improve performance. Although vectorization helps, certain optimizations are limited because of the lack of wide vector registers in the SPARC core. Where vector registers are needed, the compiler uses multiple scalar registers to emulate a vector register, which requires more instructions and increases the register pressure. A prior work compares the DySER compiler to ICC on x86 (where a vector register file is available), and shows that DySER vectorization outperforms ICC vectorization for SSE & AVX by 1.8× [13].

Observation: The DySER compiler exploits benchmark characteristics well and generates optimized binaries which are 3× faster than the scalar version.

5.3. Compiler Effectiveness

Q: How effective is compiler at generating DySER code?

We compare the speedup of compiler generated DySERized code to the hand DySERized code, as shown in Figure 7. In fft, mm, stencil and tpacf, the hand DySERized code performs significantly better than the compiler generated code because of benchmark specific optimizations. For instance, in the hand DySERized version of mm, a clever blocking strategy is used to attain high performance, whereas the compiler uses naive blocking, which does not perform well. 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 to be spilled to stack than necessary.

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

5.4. Compiler Bottlenecks

Q: Why does the DySER compiler generate suboptimal code?

First, hand-optimized code uses software pipelining to overlap execution of multiple iterations of the kernel inside DySER, but our current compiler lacks this optimization. Second, programmers may modify the algorithm to achieve high performance in the DySER accelerator. For example, in tpacf, the DySER version of the algorithm trades off some redundant computation for improved instruction level parallelism. Finally, hand optimized code schedules the DySER carefully and tries to reduce unnecessary computations in DySER that decrease the performance. For instance, it is better to schedule the reduction processing inside DySER as this does not require (unavailable) wide vector registers. However, for reduction processing, the DySER compiler greedily performs scalar replacement [11], which requires wide vector registers. This improves the utilization of DySER, but overlooks the unavailability of wide vector registers.

Observation: There is some algorithmic and implementation work to further enhance the compiler.

5.5. Compiler Generality

Q: Can we compile arbitrary applications with the current implementation of the compiler and expect speedups?

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 5 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 when large, the communication instructions inserted end up creating slowdowns.

The summary from detailed analysis of each benchmark is that these legacy codes have significantly more irregular control-flow graph shapes interacting with memory accesses 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 8 shows the shapes of control flow that are the source of problems and Table 5 assigns them to benchmarks.

Control dependent memory ops and multiple exits: The current implementation of the compiler schedules all the control instructions, for which there are dependent loads, stores or region exit branches, into the main processor pipeline along

Benchmark	Scalar	DyAccess	DyOps	CFG shape
400.perlbench	55	54	5	Small-loop, Mult-exit
401.bzip2	21	19	9	Mult-exit
429.mcf	56	61	10	Ctrl-dep-mem
445.gobmk	128	140	29	Mult-exit
456.hmmer	106	110	7	Ctrl-dep-mem
458.sjeng	8	8	0	Small-loop
462.libquantum	16	19	5	Ctrl-dep-mem
464.h264ref	9	9	0	Mult-exit
473.astar	224	224	0	Mult-exit

Table 5: SPECINT: Acceleratable Region size

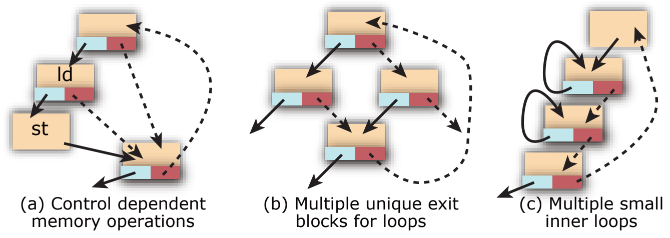

ld
st
(a) Control dependent memory operations
(b) Multiple unique exit blocks for loops
(c) Multiple small inner loops

Figure 8: Compiler on irregular benchmarks

with their backward slices. This causes problems on the two control flow graph shapes shown in Figure 8(a) and 8(b), the control-dependent memory operations and the exit branches. These two cases limits the number of DySERizable instructions in the regions. One solution is the 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.

Multiple small loops: When a region has multiple small inner loops as shown in Figure 8(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. Currently the compiler lacks the loop coalescing optimizations and does not DySERize multiple loops simultaneously because the compiler does not have loop dependence analysis that spans across multiple loops.

Observation: The DySER compiler finds acceleratable regions even on highly irregular legacy codes. More heuristics and software engineering in code-generation is required for accelerating them and producing efficient code.

6. Performance Analysis

We now describe a performance analysis which demonstrates the efficacy of the second and third principles. The ability of the microarchitecture to be dynamically specialized for a computation, combined with the tight integration of the accelerator to the processor pipeline, are the principles which define the DySER accelerator's performance. We begin our analysis with a qualitative study on the FPGA evaluation platform to understand certain FPGA performance artifacts. We then compare the performance of SPARC-DySER to state-of-the-art processors. We then describe the source of bottlenecks in performance. To eliminate compiler effects, in this section, and all remaining sections, we use hand DySERized benchmarks.

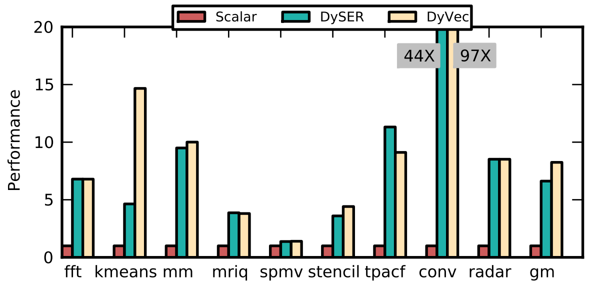


Figure 9: SPARC-DySER FPGA Performance

bench.	fft	km	mm	mriq	spmv	stencil	tpacf	conv	radar	gm
Scalar	0.08	0.08	0.05	0.06	0.09	0.07	0.16	0.06	0.07	0.08
DySER	0.37	0.26	0.18	0.12	0.11	0.24	0.39	0.38	0.29	0.24
DyVec	0.37	0.93	0.28	0.19	0.11	0.67	0.68	0.78	0.55	0.42

Table 6: SPARC-DySER Effective IPC

6.1. Overall performance

Q: What is SPARC-DySER's performance on the FPGA and in VLSI? What are the FPGA artifacts which affect performance?

FPGA Performance To understand the performance analysis, we first explain the main differences between the FPGA and VLSI implementation of the baseline processor, OpenSPARC T1. In the OpenSPARC FPGA evaluation platform, certain modules are emulated in software that run on the Xilinx MicroBlaze microprocessor [8]. The software emulated modules include the floating point unit and L2 cache, meaning all floating point instructions and level-1 I and D cache misses incur additional overhead. The software emulation of these instructions takes much longer than the corresponding hardware execution, and moreover, the emulation latency is not fixed.

The performance of the SPARC-DySER architecture on the FPGA is shown in Figure 9, normalized to the scalar version of each benchmark. Overall, the geometric mean speedup is 6.2× and 8.2× for *DySER* and *DyVec* on the Virtex-5 FPGA.

FPGA vs. VLSI Performance The FPGA speedup is significantly higher than the VCS results, previously shown in Figure 7. For conv, the speedup is even higher because the DySERized floating point instructions can eschew issues with the MicroBlaze. For the *DyVec* results, fewer instructions and data cache misses further increase the performance gap.

Since some of the FPGA performance gain is due to artifacts in the evaluation platform, the remainder of the paper uses VLSI-based VCS results. Here we show the VLSI-based performance in a different metric, IPC, in Table 6. The instructions per cycle reflects the effective issue width of the SPARC-DySER architecture. For the scalar version, the OpenSPARC pipeline can only achieve a mean IPC close to 0.1. This can be explained by the fact that the OpenSPARC T1 is in-order, and has no mechanism for hiding L1 access and miss latencies.

The IPC of *DySER* and *DyVec* is the number of scalar instructions divided by the execution cycles of *DySER* and *DyVec*, respectively. Compared to the scalar baseline, SPARC-DySER can increase the average IPC to 0.42, which is more than 5× better. In the best case, kmeans, the SPARC-DySER

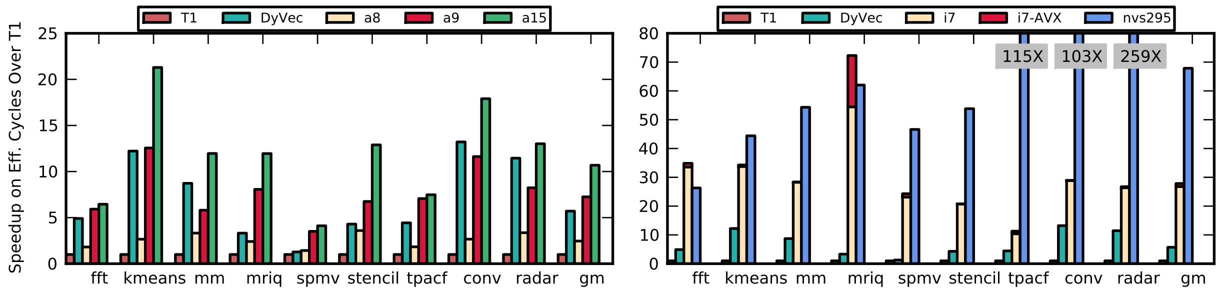


Figure 10: Architecture Comparison: Performance in cycles

can effectively issue close to one SPARC instruction per cycle.

Observation: The DySER accelerated performance is 8.2× and 6.2× better than the scalar version, for the FPGA and VLSI respectively. SPARC-DySER has an average IPC of 0.42.

6.2. Performance Comparison

Q: How does SPARC-DySER's performance compare to the state-of-the-art? Can a simple OpenSPARC core be accelerated to match high-performance x86/ARM processors?

Figure 10 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, 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.

T1	DySER	DyVec	A8	A9	A15	i7	SSE	AVX
0.08	0.24	0.42	0.21	0.63	0.86	2.47	2.47	2.55

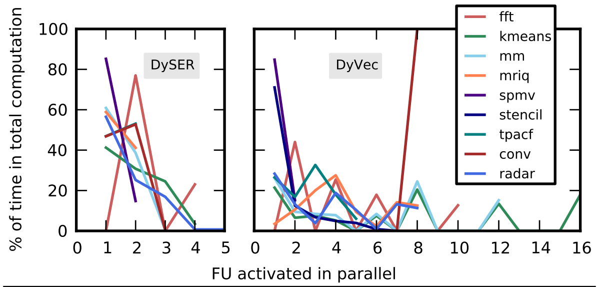
Table 7: Architecture Comparison: IPC

Table 7 shows the performance with IPC, which helps us understand the instruction-level parallelism that a processor can exploit. Note that this table shows the effective IPC(all architectures uses the scalar instruction counts of the program). Overall, *DyVec* has 2× better IPC than A8, but 1.5× worse IPC than A9 using the harmonic mean over all benchmarks. Moreover, high-performance processors can always achieve an IPC above 1, and on average have IPCs more than 2, which is never attained by SPARC-DySER. Also, observe that while DySER shows significant speedup over its baseline, SSE and AVX achieve very little speedup.

Observation: SPARC-DySER has the performance between the ARM Cortex-A8 and Cortex-A9, but is behind the state-of-the-art x86 microprocessors, with and without accelerators.

6.3. Performance Sources & Bottlenecks

Q: What are the sources of speedup and what are the performance bottlenecks of SPARC-DySER?



Version	fft	km	mm	mriq	spmv	stencil	tpacf	conv	radar
DySER	4	5	3	3	2	1	3	3	5
DyVec	10	16	12	9	2	8	6	8	8
Total	10	23	15	11	14	14	20	22	16

Maximum Concurrent Active FUs during Execution

Figure 11: Distribution of Concurrent Active FUs

The primary source of performance improvement comes from DySER being able to concurrently execute many operations as shown in Figure 11. The x-axis is the event that N number of functional units are computing in parallel, and the y-axis is the total percentage of such a event during the computation. The results shows that around 2 functional units are activated in parallel while using DySER and with the vector interface, typically 3 functional units.

The table in Figure 11 shows the best case of parallel FU activation during execution and the total number of FUs in the configuration. This result shows how far we are behind the ideal case, wherein we can fetch data continuously and keep all DySER functional units computing. Overall, we are only able to achieve max functional unit activity in fft within a small percentage of total computation. This leads us to the next stage of the analysis: While we have shown that DySER can provide significant parallel execution, the performance may be limited in practice by a number of issues, which we address below.

Potential Bottleneck: DySER Active Ratio Figure 12 shows the percentage of time that DySER is active, which means that there is data in the DySER fabric. On average, DySER is active for 40% of the total execution time. Vectorization reduces both the time that DySER is active (because it is fed faster), and the total execution time. Depending on

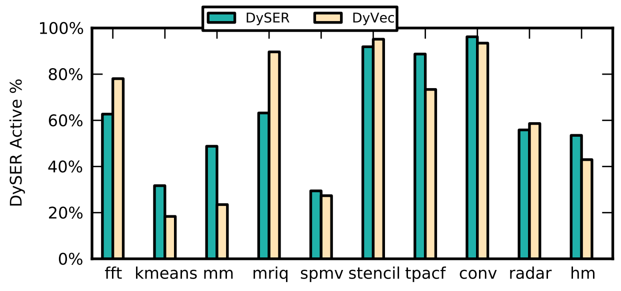


Figure 12: DySER Active Ratio during Execution

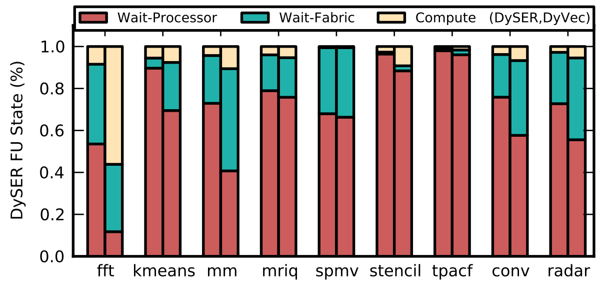


Figure 13: DySER internal utilization while active

which is reduced more, vectorization will either increase or decrease the active ratio. Benchmarks can show a higher active ratio for two reasons: First, the use of software pipelining (in mriq, stencil, radar) with vectorization, where we overlap sends and receives of different invocations, will prolong DySER activation. Second, the lack of DySERized vector store can mean that address calculation can delay the receiving of data, causing DySER to be active longer (in fft).

Potential Bottleneck: DySER Internal Utilization Figure 13 describes DySER's internal utilization by showing the breakdown of DySER functional unit states. We categorize the state of DySER functional unit into: i) Wait-Processor, 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. The first bar shows unvectorized DySER code, and the second bar shows vectorized DySER code. From the results, we observed that while the processor is a major bottleneck, functional units are more often waiting on data in the flow-control network than performing computation. This is because, in the current implementation, the latency of one functional unit is relatively small compared to the delay through the network from input to the functional unit itself. The degree of this effect depends on the routing in the configuration, and we achieve a best case utilization of 50% for fft. To summarize, while the OpenSPARC processor is the bottleneck in fetching data, the switch fabric, flow-control protocol, and the schedule of fetching data could also be improved.

Potential Bottleneck: Stalls caused by DySER The last potential bottleneck comes from the interaction between DySER and processor pipeline. Table 8 shows the stall statistics of SPARC-DySER pipeline. The second and third row

benchmarks	fft	km	mm	mriq	spmv	stencil	tpacf	conv	radar
Stall %	0.16	0.00	0.07	0.19	0.00	0.37	0.05	0.10	0.12
Stall w/ Vec	0.08	0.06	0.14	0.00	0.00	0.15	0.10	0.35	0.00

Table 8: Percentage of stalls attributable to DySER

shows the stalling ratio in execution. We list the reasons for stalling behavior below:

- 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. fft, mriq, stencil, radar belong to this category.
- 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. kmeans, mm, tpacf, conv belong to this category.

Overall, the first type of stalling behavior shows that the stalls can be reduced by pipelining the invocations, and the second type of stalling behavior shows that improving the interface will improve the performance, but may also increase the processor pipeline stall time.

Observation: DySER speedup comes from concurrently activating more FUs. The main bottleneck is the processor pipeline, which cannot feed data fast enough. The switch fabric and DySER induced stalls are secondary bottlenecks.

7. Area Analysis

In this section, we compare the area of SPARC-DySER to commercial processors. The area, like the performance, is dictated by the second and third DySER principles. The dynamic specialization principle defines the area of DySER itself, and the tight integration principle allows the DySER accelerator to use the main processor's area as a memory access engine. We elaborate below.

Q: Is SPARC-DySER area-efficient? How large is SPARC-DySER compared to state-of-the-art commercial processors?

Figure 14 shows the hierarchy view of the SPARC-DySER layout with the Synopsys 32nm generic library. The SPARC-DySER core occupies 7.56mm^2 in 32 nm, where the total cell area (the standard cell area without wiring) is 4.5 mm^2. Most of area is occupied by L1 data and instruction cache, and DySER occupies around 10% of total area. The internal breakdown is 70% functional units and switch fabric, and 15% each for input and output interface.

We show the comparison of core areas in Figure 15. 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

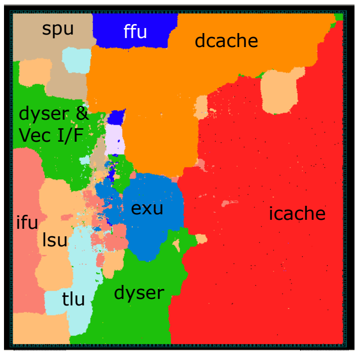


Figure 14: SPARC-DySER Layout

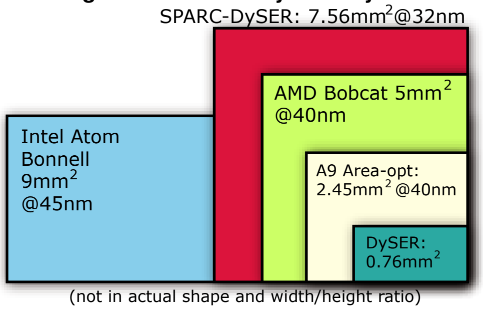

SPARC-DySER: 7.56mm²@32nm
AMD Bobcat 5mm² @40nm
Intel Atom Bonnell 9mm² @45nm
A9 Area-opt: 2.45mm² @40nm
DySER: 0.76mm²
(not in actual shape and width/height ratio)

Figure 15: Architecture Comparison: Area

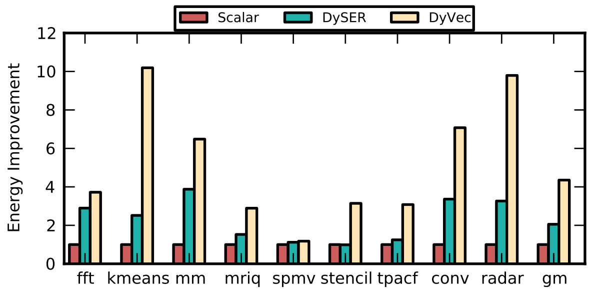


Figure 16: Energy Improvement

modules are inefficient. For DySER itself, the simple vector interface can also be improved for lower area.

Observation: A SPARC-DySER core occupies 7.56mm^2 in 32nm, and DySER itself occupies around 10% of the total area. SPARC-DySER and DySER itself can be further improved by physical-design optimizations in size.

8. Power and Energy Analysis

The power and energy analysis, performed here, also demonstrates the impact of the second and third DySER principles. Dynamic specialization allows the execution of frequent regions to be highly efficient, and tight accelerator integration requires the processor pipeline to perform high-energy operations. We elucidate on the tradeoffs below.

8.1. Overall Power and Energy

Q: What is the overall power/energy improvement?

Figure 16 shows the normalized energy of SPARC-DySER over 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 17 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.

Observation: The energy consumption of SPARC-DySER is 4× better compared to OpenSPARC baseline, but requires 25%-50% more power.

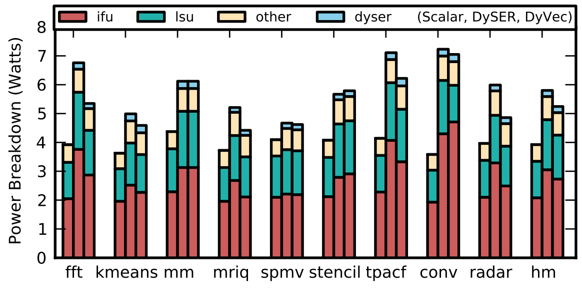


Figure 17: SPARC-DySER Power Breakdown

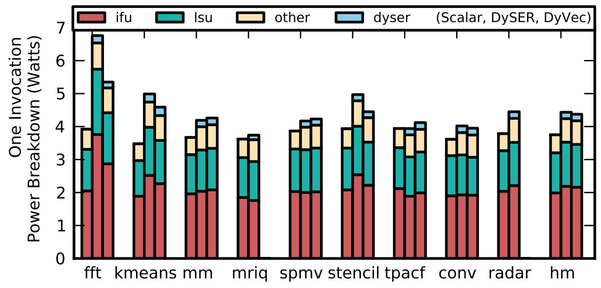


Figure 18: SPARC-DySER Power Breakdown in 1 Invocation

8.2. Energy/Power Sources & Bottlenecks

Q:What are the sources/bottlenecks for energy improvement?

Figure 17 also shows the breakdown of power sources, 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 9 (the real instructions issued per cycle, in contrast to effective IPC based on scalar instructions). If we are issuing more instructions 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*.

To eliminate the power effect of non-DySERized instructions in each program region, we also present the invocation power breakdown in Figure 18. Each invocation "lasts" between its first dsend and final drecv. Here, we observed less

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 9: SPARC-DySER Actual IPC

ifu power increase between baseline and SPARC-DySER, as expected (the power consumption of other instructions is not inflated, because of the restricted measurement window). The benchmark fft is an outlier because there are many necessary address calculation instructions. Overall, SPARC-DySER only introduced 6% more power within one invocation.

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

9. Frontier Analysis

Using the analysis of SPARC-DySER's performance, energy, and area, we summarize the impact of DySER's fundamental principles with performance/energy frontier analysis [4].

9.1. Exploring the Energy/Performance Frontier

Q: What is the position of SPARC-DySER in the energy/performance frontier? How does the integration of DySER move the position of OpenSPARC T1?

To compare and visualize the energy/performance impact of DySER integration, we show the energy/performance frontier in Figure 19(a). 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). 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. In all, the energy of DyVEC is above both A8 and A9 even though the performance of DyVEC is in-between. If we look at the projected points *DyVEC-opt*, we can observe that SPARC-DySER can achieve slightly lower energy than A9, with lower performance.

Figure 19(b) gives us the architectural view of the energy/performance tradeoff. In this figure, we i) scale the frequency of each processor to the same point, ii) find out the corresponding power at each such frequency using the

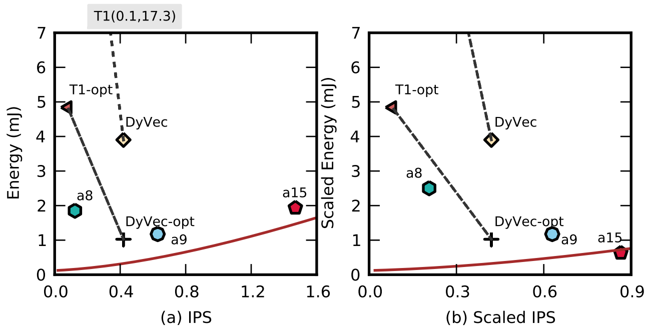


Figure 19: Energy/Performance Frontier

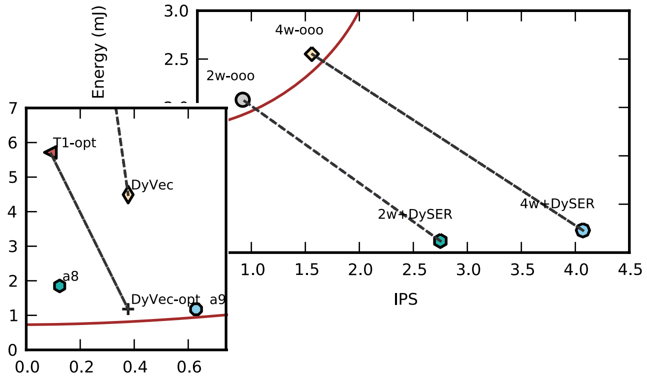


Figure 20: Energy/Performance Frontier with OoO Proc.

DVFS trend line in [5], and iii) further scale the power to the same technology node. As a result, all architectures are now compared at same frequency and technology. From the graph, we observe that ARM processors have increasingly better performance/energy behavior between generations. DySER moves OpenSPARC toward a similar direction, providing both energy and performance benefits.

Observation: DySER integration is one approach that can increase energy efficiency and performance, and could be considered when developing the next generation of an architecture. If the physical implementation of OpenSPARC is improved, SPARC-DySER has the potential to provide similar energy consumption and performance to the ARM Cortex-A9.

9.2. Sensitivity to Baseline Processor

Q: What is the performance of DySER accelerated cores with various baseline processors?

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 20. 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 than the 2-wide, because DySER is more effective when it can be fed data faster.

Here we revisit the performance and energy component of

Work	Quantitative results	Demonstrated insights
DySER	*Early-stage*: 2.1 × AVG on workloads [12] *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 [9]	Dataflow efficiency
RAW	○ up to 10× on ILP workloads ○ up to 100× on stream workloads [26]	Tiled architecture
Wave Scalar	○ 0.8 to 1.6 AIPC on SPEC ○ 10 to 120 AIPC with mutli-threading [22]	Dataflow efficiency
Imagine	IPC from 17 to 40, GFLOPS from 1.3 to 7.3 [3]	Streaming

Table 10: Summary of the Performance Evaluation Works

the DySER benefit vector in Figure 3(a)(b) from the introduction. In addition to the magnitude shown on axes, the speedup and the energy reduction factor (in ratios) is annotated with gray boxes. As the baseline processor performance improves, although the magnitude of the performance increased and the energy decreased are larger with better baseline processor, the speedup and energy reduction ratio decreases because the baseline processor's performance is better.

Observation: The performance component of the DySER benefit vector increases in magnitude, with better baseline processor performance.

10. Lessons Learned

The findings of some other prototype evaluations are summarized in Table 10. 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 represented by Parboil. 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 eliminate the potential benefit of these features.

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. Although done by design and simulation, we could not demonstrate by construction that DySER can be non-intrusively integrated into high performance processors. 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-art designs. Among what is available, OpenRISC [20] and Fabscalar [7] 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 [27] and the Gospel [28] systems provided ideas along these lines. More recently Rhodium and works inspired by it [17, 25] discuss declarative specifications and creating optimizations from code examples — approaches to reduce implementation time. Such frameworks, in a readily usable form, in a production compiler like LLVM or GCC, would be immensely useful for future prototyping works.

In this work, we observed that the most limiting component of 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. If a specialized memory access engine can be built, when integrated with a high-performance processor or a power-critical platform, we can turn off most of the processor core and use only a portion of hardware for DySER tasks. In all, we think specialization is a promising solution to break the energy/performance frontier.

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] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, "Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis," in *ISCA '10*.

[5] M. Baron, "The single-chip cloud computer," *Microprocessor Report*, April 2010.

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

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

[8] "OpenSPARC T1. [Online]. Available http://www.oracle.com/technetwork/systems/opensparc/opensparc-t1-page-1444609.html."

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

[10] 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.

[11] 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.

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

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

[14] 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*.

[15] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997.

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

[17] S. Lerner, T. Millstein, E. Rice, and C. Chambers, "Automated soundness proofs for dataflow analyses and transformations via local rules," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '05.

[18] 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.

[19] 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*.

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

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

[22] 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.

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

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

[25] R. Tate, M. Stepp, and S. Lerner, "Generating compiler optimizations from proofs," in *POPL '10*.

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

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

[28] 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.

[29] 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*.