Performance Evaluation of a DySER FPGA Prototype
Ranjini Mysore Nagaraju
ECE Graduate Project Report
University of Wisconsin Madison
mysorenagara@wisc.edu

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 report evaluates the fundamental principles of DySER acceleration. The key finding is that 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 6X.

1. Introduction
Accelerators are designed to improve performance and energy consumption of baseline processors. 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. These principles ultimately decide the accelerator effectiveness. While early stage results from simulation and modeling provide good estimates, performance prototyping on a physical implementation uncovers the fundamental sources of improvements and bottlenecks. This report 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

Prior works have presented DySER’s architecture and early stage results [3], [2], ISA design and proof-of-concept integration into OpenSPARC [1], compiler [4] and scheduler [5]. This report, uses a performance-capable FPGA-based prototype, meaningful workloads and undertakes an end-to-end evaluation of the SPARC-DySER system. Like any full system prototype evaluation, the end goal is to elucidate the merit of the underlying principles using detailed quantitative measurements and analysis of a physical prototype. This report analyzes DySER prototype to quantify the effectiveness of dynamic configuration and its impact on performance.

The finding of this report is that the DySER hardware speeds up the program by concurrent active functional units, and is limited by the data delivery rate of the integrated processor. This report is organized as follows. Section 2 gives a background of DySER. Section 3 presents
implementation of performance instrumentation counters. Section 4 describes evaluation methodology for the quantitative measurements. Sections 5 covers the performance analysis respectively.

2. Background

2.1. DySER Architecture

DySER’s microarchitecture and compiler concepts have been reported in previous publications. This includes the architecture description [3], [2], detailed compiler description [4], scheduler [5], and a proof-of-concept integration [1]. This section first provides an overview and background, and then lists the necessary work to bring DySER prototype to design that is capable of performance evaluation. Figure 1 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 1. Further ISA extension details are in [1], [2]. 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.

![Figure 1: Overview of DySER Architecture](image)

The processor-DySER interface is shown in Figure 1 as the striped boxes between D$ and DySER (described in [2]). 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 inter- or intra invocation, which are intractable for traditional SIMD techniques [4].

The DySER approach relies on the compiler to identify and transform regions of programs that can be accelerated by DySER. The DySER compiler, implemented on top of LLVM, creates the mentioned computation component and memory access component, and represents them with
the Access Execute Program Dependence Graph (AEPDG) [4]. 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 [1], 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 2X2, or an 8X8 configuration with only 2-bit data-path. Hence, peak performance was quantified only with simple micro-benchmarks.

2.2 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;
3. It has limited resources for DySER, due to FPGA size constraints.

![Figure 2: Summary of works towards Performance Evaluation](image)

We describe novel implementations that overcome these hurdles:

1. **Retire Buffer and Stallable Design**: For simplicity, the OpenSPLySER prototype does not consider OpenSPARC T1 traps and exceptions and hence cannot deploy real workloads. Moreover, because 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 2(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 [2]. To achieve a performance-accurate design and implementation without significantly increasing design complexity, we implemented a simplified vector interface, as shown in Figure 2(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 it does not affect the benchmark’s execution path.
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 data-path. 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 as shown in Figure 2(c), 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 data-path intact, and we continue to issue dconfig instructions, even though they do not actually reconfigure DySER.

3. **Implementation of PIC**

This section describes architecture and implementation of performance instrumentation counters (PIC) designed for DySER. The PIC are designed to measure the state of functional units (wait for processor, wait for control flow or compute) and active ratio of DySER which are described later in section 5. The PIC are comprised mainly of two parts: Counters and Counter Control Unit.

**Counters**: The PIC are comprised of three different types of counters.

1. **Up counters**: Simple counters which increment at every cycle the enable is activated.
2. **Start and Stop counters**: These counters have a separate start and stop signals instead of a single enable.
3. **Histogram Counters**: These are a set of 32 counters along with customized decoding logic and enable signals for each counter. They collectively keep track of functional unit states while DySER is active.

All of the above counters are associated with respective IDs. The counter control unit generates reset, read and enable signals for all counters. The counters are connected in a scan chain and the front end of chain is connected to output port 31 of DySER which is used to read counter values. Two consecutive reads to port 31 of DySER will read a counter value along with its ID. Rest of the counter values and their IDs shift toward the front end as they are read.

**Counter Control Unit**: The counter control unit interfaces with control signals from DySER. It generates separate enable/start and stop signals, read and reset signals for all counters.

4. **Evaluation Methodology**

The RTL implementation of DySER (along with PIC) and OpenSPARC are synthesized using XST and downloaded to Xilinx 5 FPGA. All the benchmarks are run on the resulting FPGA prototype and statistics are collected from PIC. We select some representative benchmarks in the Parboil [6] suite, throughput kernels from [7], and SPECINT [8]. The Parboil and throughput kernels serves as emerging workloads, and the SPEC benchmarks help us to understand DySER’s effectiveness
on “code in the wild” or legacy code. Overall, our goal is to understand the effectiveness of DySER’s three driving principles through quantitative analysis.

5. 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 two perspectives: the sources and bottlenecks of the performance. In this section (and all remaining sections) we use hand DySERized benchmarks for evaluation.

5.1 Performance Sources & Bottlenecks

Figure 3 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 3 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. The major source of the speedup comes from the fact that DySER can extract more ILP than the 1-issue SPARC baseline processor. Also, we observed low average/maximum utilization of DySER’s functional units. In Figure 4, we make further examination of DySER and show the reason behind low utilization. We categorize the state of DySER functional units into:

1. Wait-Processor, which means the functional unit is stalling because at least one of its input data is not fetched by processor pipeline,
2. Wait- Fabric, which means the functional unit is waiting for switches to pass the input data,
3. Compute, which means it is computing.
To sum up, 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 deliver rate is considerably low, such that DySER is idling and waiting for the data from processor. Furthermore, DySER cannot pipeline iterations because of lack of 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 processor pipeline. Table 1 shows the stall statistics of SPARC-DySER pipeline. The second and third row shows the stalling ratio in execution. We list the reasons for stalling behavior below:

1. \textit{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.

2. \textit{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: 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.

5.2. DySER challenge benchmarks

SPEC In this section, we describe the SPEC benchmarks alone and discuss the performance sources and bottlenecks. First, Table 2 shows the overall speedup (slowdown) of SPEC benchmarks accelerated with DySER. Similar to the compiler analysis, the SPEC benchmarks reports negligible speedup Table 3 also gives the fact that the FU activation maximum of our SPECINT implementation is only 1-2. We characterize the reason as follows:

1. Control-flow: Because of the lack of the conditional DySER access instructions (conditional DySER load and stores) in our prototype, more instructions has to be used in the memory access component to check the validity of DySER generated values (astar and libquantum). Also, bzip2 has huge control-flow statements and hence DySER FU utilization rate is low.

2. Loop carried dependence: hmmer has loop-carried dependence such that our DySER prototype cannot unroll loops and create larger computation component.

3. Small computation component: h264ref and mcf’s frequent executed region has much more memory operations than computation.

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

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>astar</th>
<th>bzip2</th>
<th>h264ref</th>
<th>hmmer</th>
<th>libquantum</th>
<th>mcf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>1.01</td>
<td>1.00</td>
<td>0.97</td>
<td>1.11</td>
<td>0.75</td>
<td>1.02</td>
</tr>
</tbody>
</table>

Table 2: DySER Challenge Benchmarks: SPEC

<table>
<thead>
<tr>
<th>bench.</th>
<th>astar</th>
<th>bzip2</th>
<th>h264ref</th>
<th>hmmer</th>
<th>libquantum</th>
<th>mcf</th>
</tr>
</thead>
<tbody>
<tr>
<td>DySER</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>8</td>
<td>5</td>
<td>6</td>
<td>14</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3: Maximum Concurrent Active FUs: SPEC
6. Conclusion

DySER provides performance through parallel computation. The data it computes, however, is fetched from the host processor, whose performance is a major bottleneck to DySER. DySER provides negligible benefit on programs with low computation to memory ratio and irregularity. Current DySER implementation lacks conditional interface to accelerate control-heavy programs.

7. References