Prototyping the DySER Specialization Architecture with OpenSPARC

Jesse Benson, Ryan Cofell, Chris Frericks,
Venkatraman Govindaraju, Chen-Han Ho, Zachary Marzec, Tony Nowatzki,
Karu Sankaralingam
University of Wisconsin-Madison
Contact Email: karu@cs.wisc.edu

This paper describes the prototype implementation of the DySER specialization architecture integrated into the OpenSPARC processor. The paper's description covers the hardware, compiler, and application tuning. The prototype system provides speedups up to $14\times$ *over OpenSPARC (geometric mean* $5\times$*). The architecture is more flexible than SIMD and GPU- based acceleration while supporting a more diverse set of workloads.*

Overview Future processors must improve microarchitectural efficiency in order to overcome slowing transistor energy efficiency and sustain performance growth. The DySER architecture uses dynamic specialization to provide energy efficient performance improvements by complementing conventional processors. By using a co-designed hardware-compiler approach that avoids disruptive hardware or software changes, the architecture **Dy**namically **S**pecializes **E**xecution **R**esources to match application phases and achieves both functionality specialization (like Garp, Chimaera, Conservation-Cores) and parallelism specialization (like GPUs and SIMD short-vector extensions). We describe here the DySER architecture and its execution model, design and implementation of its compiler, prototype implementation, and conclude with performance results and significance of this work.

Architecture DySER is an array of configurable functional units connected with a circuit switched network of simple switches as shown in Figure 1. A functional unit can be configured to get its inputs from any of its neighboring switches. When all its inputs arrive, it performs the operation and delivers the output to a neighboring switch. Switches can be configured to route their inputs to any of their outputs, forming a circuit switched network. With this configurable network of functional units, a specialized hardware datapath can be created for a sequence of computation. To enable pipelining and dataflow like execution, both switches and functional units implement a simple credit based flow control that ensures data is forwarded only when the credit is available. Credits are generated when a functional unit/switch can accept new data. The switches in the edge of the array are connected to FIFOs, which are exposed to the processor core as DySER's input/output ports. DySER is tightly integrated with a general purpose processor pipeline, and acts as a long latency functional unit that has a direct datapath from the register file and from memory. The processor can send/receive data or load/store data to DySER directly through ISA extensions.

Execution Model Figure 2 shows DySER's execution model. Before a program uses DySER, it configures DySER by providing the configuration for functional units and switches. Then it sends data to DySER either from registers or from memory. Once data operands arrive at DySER's input FIFOs, they follow the configured path through the switches. When the data operands reach the functional units, the functional units perform the operation in dataflow fashion. Finally, the results of the computation are delivered to the output FIFOs, from where the processor fetches the outputs and sends them to the register file or to memory using ISA extensions. Further details are here [3, 2].

Compiler Design and Implementation DySER's compilation consists of four main phases and the key mechanism we leverage is the development of a new program representation called the Access-Execute Program Dependence Graph (AEPDG) that exposes the spatial and temporal aspects of dependences to the compiler. The four phases are : i) Selecting regions from the full program Program Dependence Graph (PDG) that are candidates for mapping to the DySER hardware. ii) Formation of the basic AEPDG encapsulating those code regions. iii) AEPDG transformation and optimizations to meet the goodness characteristics for the DySER architecture. iv) Code generation of the AEPDG. Our compiler implements a set of judiciously chosen and intuitive heuristics to produce good quality code as part of the transformations and optimizations phase. These are:

- Loop Unrolling/PDG Cloning
- Strip Mining/Vector Deepening
- Subgraph Matching
- Execute-PDG Splitting
- Scheduling Execute-PDG
- Loop Unrolling/Dependence Analysis
- Traditional Loop Vectorization
- Load/Store Coalescing.

To implement our compiler, we leverage the LLVM compiler framework and its intermediate representation(IR). We have developed LLVM optimization passes that process the LLVM-IR to construct the AEPDG and apply the associated transformations. Finally, we extend the LLVM code-generator to assemble DySER instructions and configurations.

Prototype Implementation We have completed a full RTL implementation of the DySER architecture integrated into the OpenSPARC pipeline. In terms of physical design, we have synthesis based results. The DySER block occupies an area of 1.54 mm^2 using a 55nm ASIC library, and on average consumes 72 mW.

In terms of implementation complexity, our prototype shows the DySER design is practical. The final interface consisted of only 11 signals in the RTL between OpenSPARC and DySER,

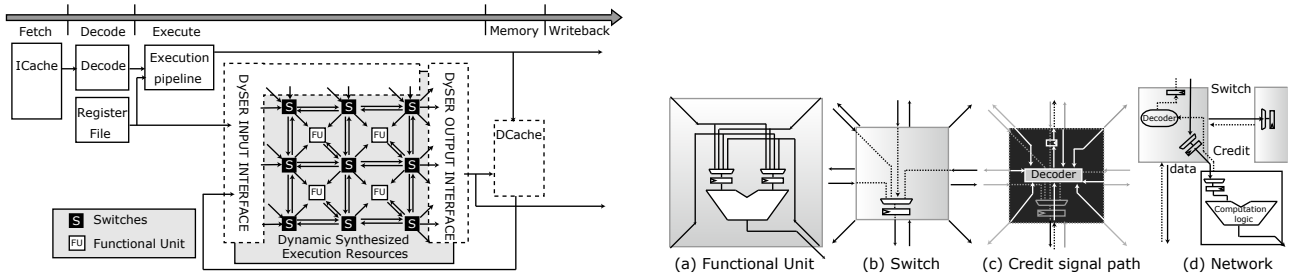


Figure 1: Processor Pipeline with DySER Datapath and DySER Elements

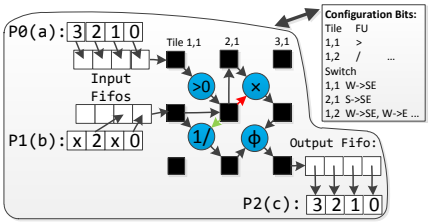
```
9

for(i=0; i<n; ++i) {
 if(a[i]>0)
   c[i] = 1/b[2i];
 else
   c[i] = b[2i]*2;
}
```

(a) Original Loop

```
9
DyConfig <Config>
for(i=0; i<n-n%4; i+=4){
  DyLd_Vec a[i:i+4]->P0;
  DyLd_Vec b[2i:2i+4]->P1;
  DyLd_Vec b[2i+4:2i+8]->P1;
  DySt_Vec P2->c[i];
}
...
```

(b) DySER-Accelerated Code



(c) DySER Configuration

Figure 2: DySER Execution Model

and a total of less than 750 lines of code modified in OpenSPARC. Further details are here [1].

We have also completed a mapping to FPGA of the full design using the Vertex-5 board. This FPGA implementation boots unmodified Ubuntu 7.10 Linux and runs C/C++ programs compiled through our toolchain. For detailed performance evaluation on our FPGA prototype, we implemented several FPGA optimizations to the architecture. These include simplifying the switches, "hardening" the configuration information and creating a FPGA bit-file specific to each application, and simplifications to the load-store interface.

Performance To measure DySER's efficiency in specialization, we have evaluated its performance on a suite of SIMD and GPU workloads to capture its functionality and parallelism specialization capability. We compare performance of DySER-accelerated implementations of these benchmarks to the sequential OpenSPARC implementation, and hand-optimized SIMD and GPU implementations. Based on measurements on our FPGA prototye implementation, compared to the OpenSPARC baseline, the DySER prototype, provides a speedup of up to 7×, with a geometric mean speedup of 3× on this diverse benchmark suite. Adding a vectorized mode to DySER provides up to 14× speedup with a geometric mean speedup of 5×. We observe that OpenSPARC's single-issue pipeline is the main bottleneck throttling the rate at which DySER is fed. When integrated with a dual-issue out-of-order processor, results from our *cycle-accurate performance simulator* show DySER continues to provide similar speedups: up to 14×, with a geometric mean speedup of 3.5×. As elaborated in [2], compared to SSE, DySER provides geometric mean 2.5× speedup, and compared to GPU execution, it provides 1.2× speedup.

Implications and Significance DySER is the culmination and generalization of trends already occurring for popular parallelism based accelerators. SSE has been augmented with both functionality-specialized and non-purely word parallel instructions. Instructions in NVIDIA Kepler GPUs are specialized for the particular region with compiler annotations indicating when to issue.

Not only is DySER a more natural evolution of specialization strategies, but it is also more practical to implement. From a software perspective, it is a more flexible compiler target than SSE, and DySER does not require a new software stack and application implementations as for the GPU. From a hardware perspective, its interface enables simple integration with a processor pipeline.

The most profound implication of DySER is that the execution model and architecture provide a practical way to implement instruction-set specialization, SIMD specialization, and domain-driven accelerators using one substrate. With its impressive speedup and corresponding energy gains, DySER significantly improves architectural energy efficiency using specialization. The novel architecture, its prototype implementation, and energy efficiency implications of the execution model provide a set of promising mechanisms.

References

[1] 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 *HPCA '12*.

[2] 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*, 32(5), 2012.

[3] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically Specialized Datapaths for Energy Efficient Computing. In *HPCA '11*.

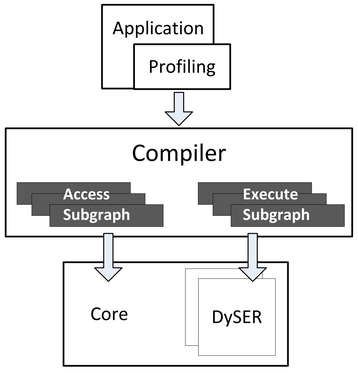
Prototyping the DySER Specialization Architecture with OpenSPARC

Jesse Benson, Ryan Cofell, Chris Frericks, Venkatraman Govindaraju,
Chen-Han Ho, Zachary Marzec, Tony Nowatzki, and Karu Sankaralingam

THE UNIVERSITY of WISCONSIN MADISON

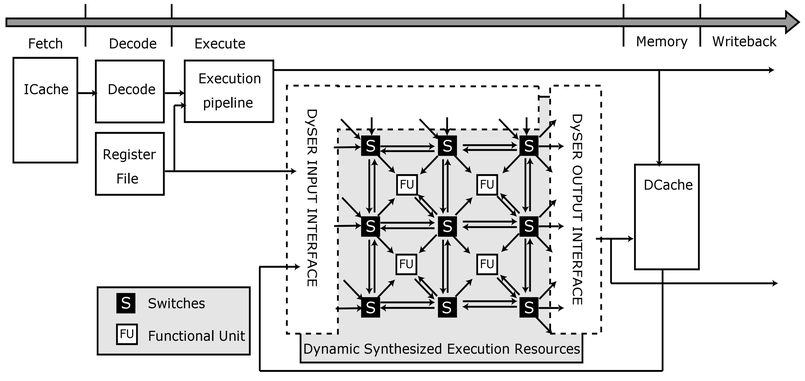
DySER Approach

- Compiler assisted dynamically specialized computation through heterogeneous array of functional units
- DySER configured once for multiple invocations



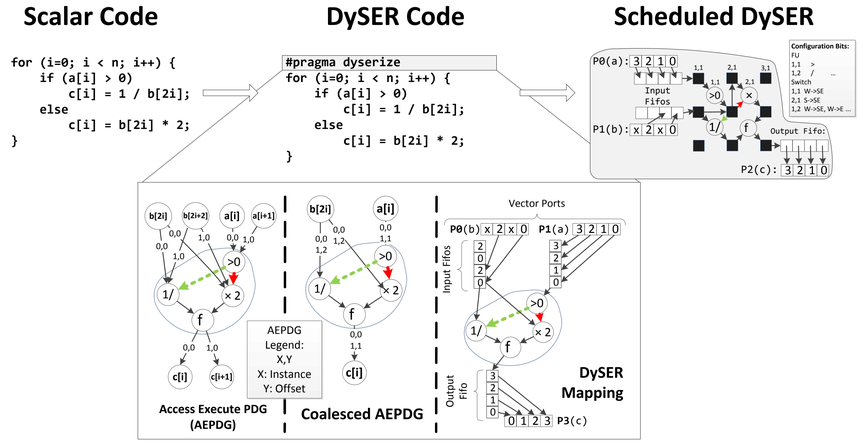
DySER Architecture

- Alongside Execute stage in processor pipeline
- Concurrently executes DySER and non-DySER code



DySER Compiler

- LLVM based compiler
- Generates specialized binaries for DySER from C/C++ source code



OpenSPARC T1 Integration

- Limited ISA extensions required
  `d_init, d_send, d_recv, d_load, d_store`
- Only **eleven** interface signals in RTL/microarch
- Few lines of changed Verilog code:

Unit	Lines Changed	Notes
IFU	275	Reserved opcodes used for DySER Instructions
LSU	23	Reverse engineered memory control
EXU	216	DySER model Verilog and 18 FF added
MMU	0	Unchanged!
Total	514	Minimal changes!

FPGA Prototype

- Utilizes Xilinx Virtex5 FPGA Board
- Fits a “hard” 4x4 DySER with fixed paths
- Boots unmodified OpenSPARC Ubuntu 7.10
- DySER is not on the critical path!

ubuntu

ASIC Synthesis @ 55nm

Area: 1.54mm^2 Power: 72mW

DySER Performance

- Throughput/high-performance workloads
- Competitive or surpasses SIMD/GPU approach

Non-vectorized (1 wide) : 2.5x speedup
Vectorized (8 wide): 4.9x speedup