*A version of this report will appear as "DySER: Unifying Functionality and Parallelism Specialization for Energy Efficient Computing" in IEEE Micro September/October 2012.*

# Mechanisms for Parallelism Specialization for the DySER Architecture

Venkatraman Govindaraju Chen-Han Ho Tony Nowatzki
Karthikeyan Sankaralingam
Department of Computer Sciences
University of Wisconsin-Madison
{venkatra,chen-han,tjn,karu}@cs.wisc.edu

**Abstract**

Specialization is a promising direction for improving processor energy efficiency. With *functionality specialization*, hardware is designed for application-specific units of computation. With *parallelism specialization*, hardware is designed to exploit abundant data-level parallelism. The hardware for these specialization approaches have similarities including many functional units and the elimination of per-instruction overheads. Even so, previous architectures have focused on only one form of specialization. Our goal is to develop mechanisms that unify these two approaches into a single architecture. We develop the DySER architecture to support both, by **Dy**namically **S**pecializing **E**xecution **R**esources to match program regions. By dynamically specializing frequently executing regions, and applying a set of judiciously chosen parallelism mechanisms—namely region growing, vectorized communication, and region virtualization—we show DySER provides efficient functionality and parallelism specialization. It outperforms an OOO-CPU, SSE-acceleration, and GPU-acceleration by up to $4.1\times$, $4.7\times$ and $4\times$ respectively, while consuming 9%, 86%, and 8% less energy. Our full-system FPGA prototype of DySER integrated into OpenSPARC demonstrates an implementation is practical.

## 1 Introduction

Future processors must improve microarchitectural efficiency to overcome slowing transistor energy efficiency and sustain performance growth. Specialization and accelerators are promising directions. One of the most mainstream specialization techniques is to specialize architectures for *data-level parallelism*. Examples include vector processors, short-vector instructions like SSE/AVX, and GPUs. *Functionality specialization* is another technique, wherein custom hardware is targeted at application functionality. Examples include Garp [3], Chimaera [15], CCA [4], PipeRench [5], Tartan [11], Phoenix [13], Conservation-Cores [14], and BERET [7].

Thus far, specialization architectures have targeted only parallelism or functionality specialization. In fact, the functionality specialization architectures are typically not evaluated on data-parallel workloads and vice versa. The reason for this distinction is that the fundamental approaches behind these strategies are

conflicting. Parallelism specialization utilizes homogeneous hardware resources with a wide/independent interconnect, while functionality specialization uses task-specific hardware resources with task-specific routing. Furthermore, parallelism specialization's homogeneous resources are simple to virtualize to support mapping arbitrarily large computations, while functionality specialization's heterogeneity means arbitrary computations face resource mapping problems.

Nevertheless, we observe that architectures like SSE and GPU are taking incremental steps towards the unification of functionality specialization with their parallelism specialization, a point we will revisit in the conclusion. The driving force is that the combination of specialization types can provide further energy and performance benefits.

DySER can be viewed as the natural progression of the trend towards unification, culminating in both functionality and parallelism exploited by **Dy**namically **S**pecializing **E**xecution **R**esources. The enabling mechanism is the configurable lightweight switching network which connects a set of heterogeneous functional units and allows customization. In DySER, parallelism is exploited by creating logical lanes of independent computation in this substrate, and functionality is exploited by creating specific datapaths for each particular computation.

Practically speaking, DySER is integrated into the execution stage of a general purpose processor, which acts as a load/store engine to feed the DySER computation substrate. To achieve functionality specialization, a compiler synthesizes datapaths between functional units specific to an application's phase. To achieve parallelism specialization, we use a judicious mix of vectorization techniques and novel hardware mechanisms. High performance is enabled by providing a dense compute fabric with low-latency integration and energy efficiency is attained through eliminating per-instruction overheads by converting code-regions into dynamically formed compound functional units. These gains can be provided without significantly disrupting either the general purpose architecture into which it is integrated, or the software development environment.

We have designed and implemented the DySER architecture and its compiler, ported applications to it and implemented an FPGA prototype. Employing functionality specialization, DySER outperforms a dual issue out-of-order(OOO) processor by $1.1\times$ to $4.1\times$, simultaneously reducing energy by 9%. Employing parallelism specialization, DySER outperforms SIMD: it is $1.3\times$ to $4.7\times$ faster than SSE, consuming 86% less energy. It outperforms GPUs with mean speedup of $1.4\times$, consuming 8% less energy.

In this article, we present the architecture and hardware/software mechanisms for functionality spe-

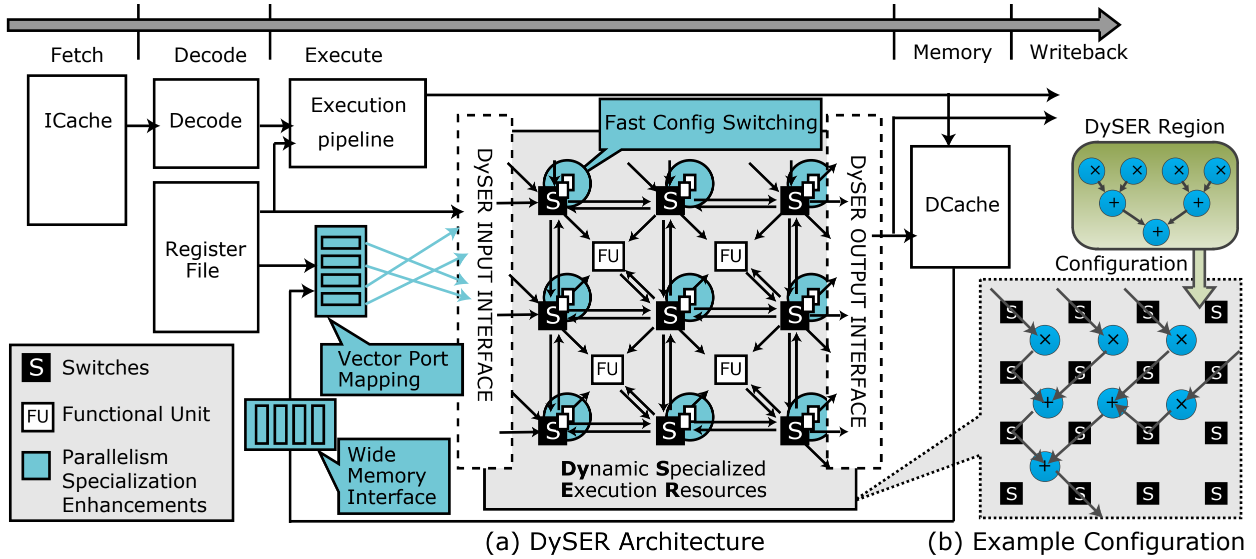



Figure 1: DySER architecture,integration to a processor pipeline and execution model

cialization and for parallelism specialization. We also report on quantitative evaluation and feasibility by describing our full-system FPGA prototype that integrates DySER into OpenSPARC. We conclude with comments on DySER's practical usage.

## 2 DySER and Functionality Specialization

The main insight in designing DySER is that programs execute in phases and only a small number of such phases or regions contribute to most of the program's execution time. Specializing such frequently execut-

ing regions can eliminate overheads and provide energy efficiency. However, the cost of having specialized hardware for all such possible regions is prohibitive. Instead, DySER dynamically creates specialized datapaths for only frequently executed regions. We also leverage the processor's memory system, utilizing its cache, prefetch, memory disambiguation and memory-dependence prediction mechanisms, thus overcoming load-store serialization bottlenecks in "irregular" code.

**Architecture:** DySER achieves functionality specialization by employing a heterogeneous array of functional units connected with simple switches as shown in Figure 1(a). A functional unit is connected to four neighboring switches which deliver its inputs and consume its output. It can be configured to get its inputs from any of its neighboring switches. Once all of its inputs have arrived, it performs the configured operation and delivers the output to the switch. Switches form a circuit-switched network and can create hardware datapaths, as shown by the example configuration in Figure 1(b). Once configured, which takes about 64 cycles, DySER computes very efficiently because it eliminates per instruction overheads such as decode, commit and unnecessary register reads and writes.

To allow pipelining inside DySER, we implement a simple credit-based flow control using a forward signal "valid" and a backward signal "credit". Functional units perform the operation when all its inputs are "valid", and data is forwarded only when the "credit" signal is asserted. Functional units and switches send credits only when they are able to accept new data. This network and data-flow execution model create a pipelined functionality specialization engine.

Figure 1(a) shows how DySER is integrated to a processor. The processor pipeline communicates to DySER through a set of named input and output ports which correspond to FIFOs that deliver data to the switches. We extend the ISA with five instructions that configure DySER, send/receive register data, and send/receive memory values.

**Execution Model:** Figures 1(c)-(f) compare the conceptual execution model of a dual issue out-of-order processor to that of DySER. The processor of Figure 1(d) executes up to two operations at a time, and is shown performing two iterations of the loop from 1(c).

The DySER version in Figure 1(e), begins by first configuring DySER for a region's datapath before it is encountered. For every instance of the region, the processor either sends register values or loads data directly to DySER. All of the sends and loads for *one* instance is referred to as an "invocation". As the

data reaches DySER, it is routed to functional units through switches according to the configuration, and execution occurs in data-flow fashion, producing results for the processor. Similar to the CPU, DySER can be speculatively invoked with the next instance of the computation, pipelining both instances together, as shown in Figure 1(f). In this example, DySER executes 5 less cycles than the processor.

**Compiler Role:** DySER relies on a compiler to create its configurations and insert instructions in the program to communicate with the processor. Our compiler does the following: i) identifies regions using profiling or static analysis, ii) partitions them into a *memory subregion*, which includes loads,stores and address calculations; and a computation subregion, which has all other instructions; iii) generates DySER configurations for *computation subregions*, and iv) inserts communication instructions into the memory subregion.

**Workload Characterization:** Considering the PARSEC and SPECINT benchmarks, we observe there are many candidate regions to specialize: 9 to 906 for PARSEC, and 46 to 10018 for SPECINT. However, about 10% contribute to 90% of the execution time. These regions are 51 to 264 instructions in length. For a 64-unit heterogeneous DySER (details in Section 4), 60% to 100% of these regions can be specialized.

## 3 DySER and Parallelism Specialization

The large number of functional units in DySER provide a great opportunity for supporting data-parallel execution. However, for DySER as described previously, parallelism specialization presents several challenges. We develop parallelism specialization mechanisms to overcome these challenges by analyzing DySER performance on data-parallel (DLP) applications. We consider hand-optimized workloads from Intel's research lab and Parboil [12]. Table 1 describes the workloads we consider and their characteristics.

### 3.1 Challenges for DySER on Data Parallel Workloads

The computation subregions of the applications considered, while providing parallelism, are ill-suited for DySER due to their "size" and "shape". Figure 2 illustrates four types of computation subregions, and column 4 in Table 1 shows the type for each benchmark.

| 1. Bench | 2. Description | 3. GPU/SIMD Performance | 4. Region Type | 5. DLP Techniques Used | 6. DySER Analysis |
|---|---|---|---|---|---|
| **SIMD Benchmarks** | | | | | |
| NBDY | Nbody Simulation | Large kernel with regular access pattern. | Superfluous | SCX, FCS, VEC-Intra | DySER throughput limited by long latency FUs (div, sqrt). |
| VR | Volume rendering | Nested loop with lots of control-flow. | Insufficient | UNR, SCX | Control flow divergence hinders SSE. DySER limited by irregular mem. |
| TSRCH | Tree Search | Irregular data accesses prevent SSE vectorization. | Insufficient | UNR, SCX, SUB | Scalar loads feed region, SSE loses because of irregular mem. access. |
| MRG | Sorting | Small kernel with unpredictable data dependent control-flow. | Superfluous | SUB, VEC-Intra | Emulates 4x4 merge network; larger region than SSE. Control flow limits perf. |
| RDR | Complex conv. | Small kernel with regular access pattern. | Insufficient | UNR,SUB,STR, VEC-Hybrid | Emulates 4 wide complex multiplier, wins with fewer instructions. |
| CONV | Image convolution | Regular comp and data accesses. No control-flow divergence. | Insufficient | UNR,SUB,STR, VEC-Intra | Emulates 8 wide SIMD, wins with larger region and memory regularity. |
| **GPU Benchmarks** | | | | | |
| GPU is Faster | | | | | |
| MRI-Q | Mag. Res. Imaging | Heavy use of Sin/Cos. Use of constant memory for less global mem B/W. | Proportional | STR, VEC-Inter | Single computation lane; DySER loses b/c of non-pipelined sin/cos FUs. |
| SPMV | Spare Matrix Vector Mult. | Indirect loads are software pipelined. Uses constant&texture mem. | Insufficient | UNR, STR | DySER loses because of irregular mem. access, no vectorization possible. |
| CTCP | 3D Grid & Point Calc. | Significant use of transcendentals. Overlaps CPU/GPU execution. | Superfluous | FCS, STR, VEC-Hybrid | Multilane pattern executes sqrt ops in parallel, lose b/c of throughput of sqrt. |
| DySER & GPU have Similar Performance | | | | | |
| MM | Dense Matrix Mult. | Standard algorithm. Shared memory & sync to reduce global memory B/W. | Insufficient | UNR, VEC-Intra | Similar perf. b/c of regular mem. access, high comp/mem. |
| STNCL | 3D Matrix Jacobi | Small comp/mem ratio. Shared memory & sync reduce global memory b/w. | Proportional | STR, VEC-Inter | 2-lane Stencil. Similar performance, limited by low comp/mem ratio. |
| SAD | Sum-of-abs. diff. | Extremely High Comp/Mem Ratio. Good Memory Locality. | Proportional | UNR,FCS,STR, VEC-Hybrid | Multilaned abs() with sum reduction. Similar perf b/c regular mem. |
| LBM | Fluid Dynamics | Extremely large computation region. Large-region ctrl-flow divergence. | Superfluous | FCS, VEC-Intra | Many reductions. Divergence hurts GPU, scattering mem. hurts DySER. |
| TPACF | Angular Correlation | Irregular memory access due to histograming. Causes branch divergence. | Superfluous | FCS, STR, VEC-Hybrid | Performs histogram with reductions. Similar b/c GPU parallelizes hist. |
| DySER is Faster | | | | | |
| KMNS | Kmeans clustering | Uses texture as cache. Regular memory access. | Insufficient | UNR, VEC-Intra | Large reduction kernel. Perf similar b/c mem. access regularity. |
| NNW | Neural Networks | Some transcendentals. Strided memory access. | Insufficient | UNR,STR, VEC-Hybrid | VEC-Inter and VEC-Intra on different arrays. Poor warp occupancy for GPU. |
| FFT | Fast Fourier Transform | Regular Memory Access. Heavy use of Sin/Cos. | Proportional | UNR, VEC-Hybrid | Single lane region. GPU implementation doesn't cache re-used sin/cos ops. |
| NDL | Dyn. Programming | GPU diagonal iteration inhibits memory coalescing. Shared mem and sync. | Insufficient | UNR, VEC-Intra | Single lane of computation. GPU suffers from excess sync & poor coalescing. |

Table 1: Benchmark Characterization

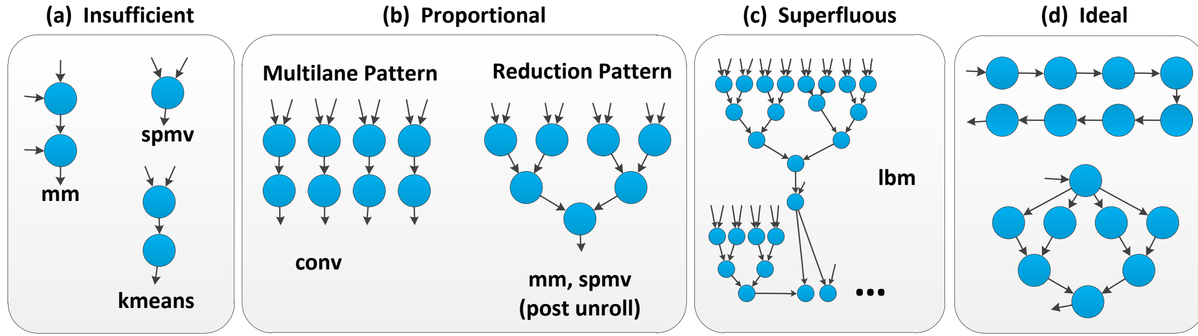



Figure 2: Types of computation subregions with example benchmarks for each type.

**Insufficient Regions:** Figure 2(a) shows computation subregions that are small in relation to DySER's resources, limiting the potential speedup and utilization.

**Proportional Regions:** Figure 2(b) shows computation subregions that are appropriately sized for DySER. These generally come in the form of the multilane and reduction patterns. Even though the potential for these regions is high, these patterns have a high communication/computation ratio, which limits speedups, since the computation subregion cannot be fed fast enough for high utilization.

**Superfluous Regions:** Figure 2(c) shows a computation subregion which is very large. While we can configure DySER separately for different sections of the computation subregion, DySER's functional units will be inactive during reconfiguration. Since reconfiguration must be performed on every invocation, overall utilization is low.

**Ideal Regions:** Figure 2(d) depicts some best-case scenarios of computation subregions, distinguished by small numbers of inputs and outputs with numerous computations. Assuming pipelinable invocations, these patterns are ideal because they have very little communication overhead. However, these are rare in most workloads, so we develop techniques to transform regions into this type.

## 3.2 Mechanisms for Parallelism Specialization

The "Transformation Flow" in Figure 3 shows our overall strategy for transforming an arbitrary computation subregion to act like an ideal region. Though performed manually for the results in this paper, these transformations are designed to be implementable in a compiler. Only the TPACF and MERGE benchmarks require additional algorithmic changes for effective parallelism specialization. Column 5 in Table 1 shows the transformations for each benchmark.

### 3.2.1 Region Growing

Regions that are too small to attain high utilization must be expanded, which can be achieved by transforming the loops. Specifically, we apply loop unrolling (UNR) until an appropriately sized computation subregion is formed, as shown in Figure 3(a). If the loops are independent, we create a multilane pattern. With a single loop carried dependence, if possible we create a reduction pattern. When profitable, we alternatively employ scalar expansion (SCX) 3(b), which enables loop parallelization by providing temporary storage for dependent variables. Scalar expansion allows us to break some reduction patterns into multilane patterns, which can be beneficial depending on the use of the region's outputs.

### 3.2.2 Vectorizing DySER

Vectorized DySER instructions can load and store only contiguous words. In order to vectorize send and load instructions efficiently, we must provide mechanisms to handle arbitrary relationships between contiguous memory and the interface to the regions. We explain several communication patterns with examples, and describe the mechanisms which make vectorization possible.

**Intra-invocation Communication (VEC-Intra):** Figure 3(d) shows the computation subregion from the convolution (CONV) benchmark. Each contiguous memory word is mapped to a different input port of DySER and used by a single invocation. Intra-invocation (Intra-VEC) communication converts DySER into a vector unit.

**Inter-invocation Communication (VEC-Inter):** Figure 3(e) shows the computation subregion from the stencil(STNCL) benchmark. Each contiguous memory word is mapped to the same port since subsequent

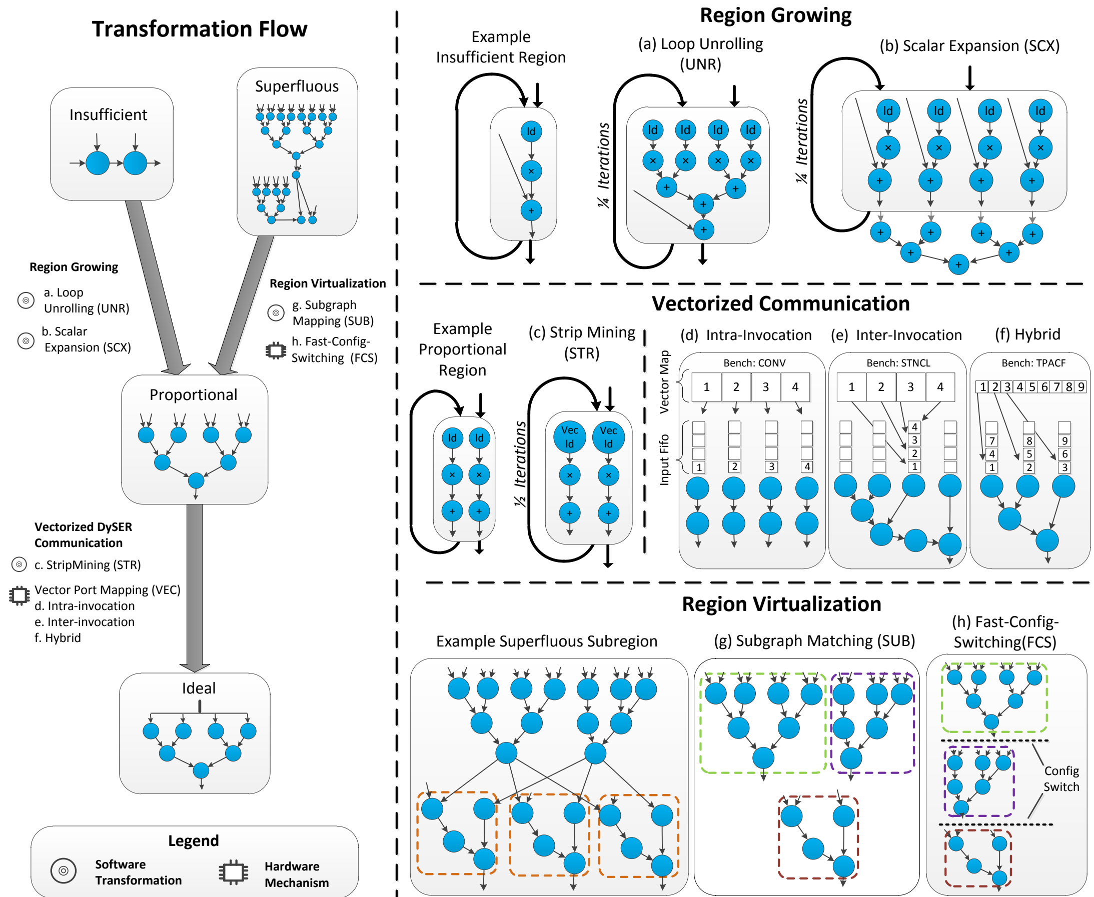



Figure 3: Overview of Hardware and Software Transformations for Parallelism Specialization

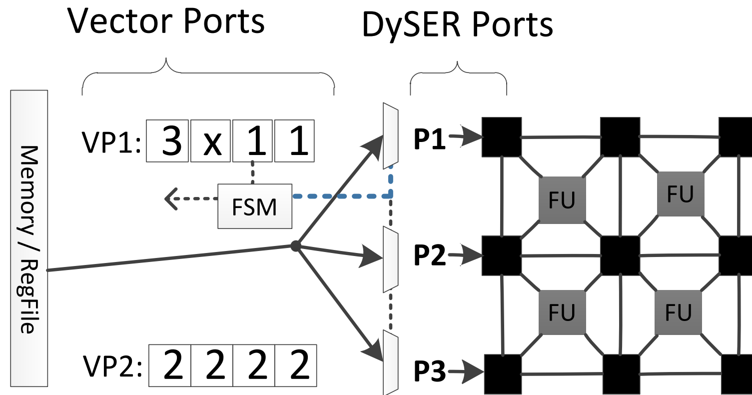



Figure 4: Implementation of Vector Ports in DySER

invocations use contiguous memory addresses, thus allowing multiple invocations to be explicitly pipelined.

**Hybrid Communication (VEC-Hybrid):** Figure 3(f) shows a computation subregion from the TPACF benchmark. Neither inter-invocation nor intra-invocation is sufficient to perform a vector load more than 3 words wide. Our strategy is to use a hybrid, where each word triplet is sent to the same invocation, and subsequent triplets are pipelined to subsequent invocations. This example is 3 "wide" and 3 "deep".

**Stripmining (STR):** Employing these communication patterns requires a transformation called stripmining, as shown in Figure 3(c). Both stripmining and loop unrolling reduce the loop trip count, but doing both is usually possible because the loops we considered have high bounds.

**Implementation:** To implement vectorized communication in hardware, we first require a wide memory interface similar to that of SSE, and a number of named vector ports. Additionally, we need a mechanism to map a vector of input/output values to or from DySER's internal input/output ports. The information that conveys this correspondence is termed the "vector map". The configuration is augmented with bits which specify a vector map for each vector port. When a vectorized DySER instruction accesses a vector port, a finite state machine (FSM) in DySER's I/O interface coordinates the transfer of data between the incomming values from the register file or memory, and DySER's internal ports. Figure 4 shows our implementation of

vectorized communication. The FSM in DySER's I/O interface uses the vector map bits to generate control signals which select the appropriate DySER port. In each subsequent cycle, the next value in the vector map is utilized. In the example shown, vector port 1 routes the first and second memory values to DySER port 1, the third memory value is ignored because it is masked off, and finally the fourth memory value is sent to DySER port 3. Note that a similar vector mapping FSM is required on the output interface as well. In this implementation, it takes N cycles to map an N element vector. Though faster implementations are possible, and can have an impact on performance, their description and evaluation is ongoing work.

#### 3.2.3 Region Virtualization

Similarly to insufficient regions, overly large regions must be "resized" to fit inside DySER to achieve high utilization. Compared to instruction-level acceleration, DySER's dynamic customization introduces resource limitation challenges, which we overcome by employing two primary techniques.

**Subgraph Matching (SUB):** First, we attempt to reduce the computational region by identifying similar computational structures, which we call Subgraph Matching as shown in Figure 3(g). The transformation is essentially to cut dataflow edges from a common subgraph, and combine all common subgraphs together. These cut edges will be reconnected through the memory subregion.

**Fast-Config-Switching (FCS):** If Subgraph Matching cannot reduce the computation subregion sufficiently, we employ a further technique to reduce the configuration penalty. Figure 3(h) shows how a superfluous computation subregion can be cut into components of appropriate size and mapped to DySER by using multiple configurations.

We enable fast-config-switching through two hardware mechanisms. First, we augment every DySER tile (FU+switch) with ability to store multiple configurations. Second, we developed a configuration switch protocol for DySER that relies on each tile being either in an active or off state.

We added to the network a 1-bit free signal, which is sent from the eight neighbors of a tile. We add one additional instruction that sends reset signals through the *old* configuration forcing every tile that has finished computation into the off state, triggering them into sending free signals to neighbors. The set signals that follow the reset signals then change any off-state tile into the new configuration. Each set signal propagates to all neighbors in the *new* configuration after receiving their free signals.

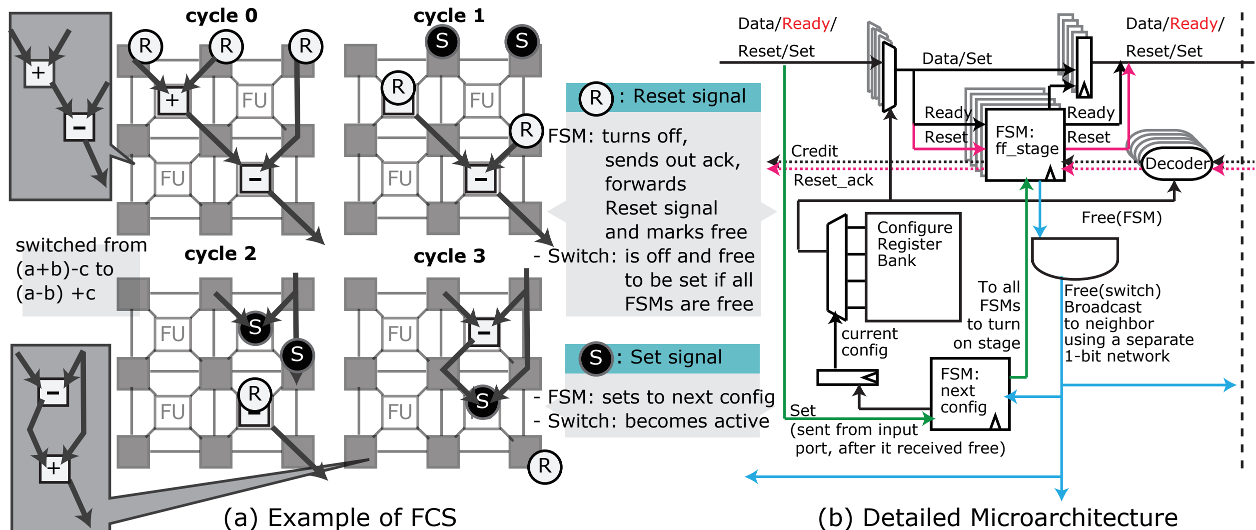



Figure 5: Fast Config Switching Example and Implementation

This protocol explicitly reuses the dataflow in the two regions to synchronize the set and reset signals without any additional networks or compiler requirements. As soon as an entire invocation has been sent, reset and set signals can also be sent. Figure 5(a) shows an example of the set and reset signals performing the configuration switching. Figure 5(b) shows the microarchitecture implementation, which requires configuration registers and finite state machines, and outlines the protocol.

Using our RTL implementation, the evaluation of the energy consumed by fast-config-switching shows it is about 2 picojoules compared to the 120 picojoules for instruction-fetch and decode of a 4-wide OOO processor.

# 4 Evaluation

Our evaluation focuses on three issues: i) functionality specialization effectiveness; ii) parallelism specialization effectiveness; and iii) implementation and integration feasibility. We evaluate DySER on these issues with simulation, RTL implementation, and a full-system FPGA implementation.

## 4.1 Evaluation Methodology

For our performance and energy evaluation, we use a simulation-based approach. We consider a dual-issue OOO processor as our baseline. This baseline processor has 64KB L1-D$, 32KB-L1-I$, and a tournament

| Unit | Count | Latency | Area ($\mu m^2$) | Description |
|---|---|---|---|---|
| INT-ADD | 16 | 1 | 2482 | OpenSPARC |
| INT-MUL | 12 | 5 | 16401 | OpenSPARC |
| FP-ADD | 16 | 4 | 14533 | OpenSPARC |
| FP-MUL | 12 | 7 | 24297 | OpenSPARC |
| FP-DIV | 4 | 12 | 16932 | Taylor-series based [9] |
| FP-SQRT | 4 | 12 | 16932 | Taylor-series based [9] |
| Switch | 81 | 1 | 8009 | |

Table 2: Details of function units used in 64-FU DySER. Unified Div/Sqrt implementation.

branch predictor with 4K BTB entries. We consider an SSE implementation in X86 and a GPU as our reference data-parallel accelerators to compare to DySER. Specifically, we consider a 1-SM/8-wide GPU since its area and functional unit mix matches one DySER block integrated with a 2-wide OOO processor. In all cases, we consider applications tuned for each architecture. We integrate DySER into a dual issue out-of-order processor that is identical to our baseline.

We evaluate a DySER that has 64 heterogeneous functional units, as shown in Table 2 (RTL implementation details in Section 4.4). We use the Gem5 simulator [1], Gem5+SSE, GPGPU-Sim [2], and Gem5 extended for DySER to evaluate the various platforms respectively. We augmented our Gem5 infrastructure with McPAT [10] based power models. We developed our own GPU power model extending the device-specific model developed by Hong and Kim [8] to allow parametrization. Its error range is $\leq 20\%$.

**Benchmarks:** For functionality specialization, we consider the PARSEC and SPECINT benchmark suites. We use code optimized with GCC -O3, and used profiling to identify regions. We chose distinct benchmarks for evaluating parallelism specialization because we wanted to manually compile and optimize each benchmark, which was intractable for PARSEC and SPECINT. A manual approach is appropriate because this is an evaluation our architectural mechanisms independent of effects from potential compiler transformations. The benchmarks described in Section 3 were good choices because they were simple enough to manually optimize and were well suited for their respective accelerators. For the DySER versions of these benchmarks, we implemented or obtained scalar C++ code, applied the transformations described in Section 3, and compiled with our GCC based DySER toolchain.

### 4.2 Functionality Specialization Results

Figures 6(a),(c) show the performance and energy improvements from DySER integration when compared to the baseline dual issue OOO processor and performing only functionality specialization. We consistently see improvements across the benchmarks with harmonic speedup of 39% and 9% reduction in energy. We achieve performance improvements in irregular workloads by forming large regions and specializing such regions with control-flow support inside DySER. With some benchmarks (eg. freqmine, gobmk), there is little performance gain because of the many insufficient regions which were not amenable to our transformations. Govindaraju *et al.* presents further analysis of these benchmarks and results [6].

### 4.3 Parallelism Specialization Results

Column 6 in Table 1 summarizes our results when performing parallelism specialization using DySER. For each benchmark it shows how DySER employs the transformations we described. Note that these benchmarks primarily benefit from parallelism specialization, but can also implicitly benefit from functionality specialization by using the DySER hardware to represent computations. Figure 6(b),(d) show the performance and energy improvements of DySER, SIMD, and GPU acceleration compared to the baseline. We provide detailed analysis below.

**DySER vs SIMD:** DySER performs significantly better for all of the SIMD benchmarks. For highly regular workloads such as CONV and RDR, DySER emulates a wider SIMD unit than SSE units and accelerates them using vectorized loads. For irregular workloads such as volume rendering (VR) and TreeSearch (TSRCH), we find independent computations and accelerate them using DySER's pipeline parallelism. However, we cannot vectorize the code beneficially because of data dependent control flow and irregular memory accesses. The benchmarks Merge and NBody have superfluous computation subregions, so we use Region Virtualization.

*DySER provides a harmonic mean speedup of* $3.2\times$ *over our baseline, with a range of* $1.5\times$ *to* $15\times$*, and energy reduction of 60%, with a range of 33% to 94%.*

*It is* $1.3\times$ *to* $4.7\times$ *faster than SSE and has similar energy efficiency.*

**DySER vs GPU:** We see various distinct types of behavior with GPU workloads. Table 1 presents details under three categories in its last column, while some highlights follow.

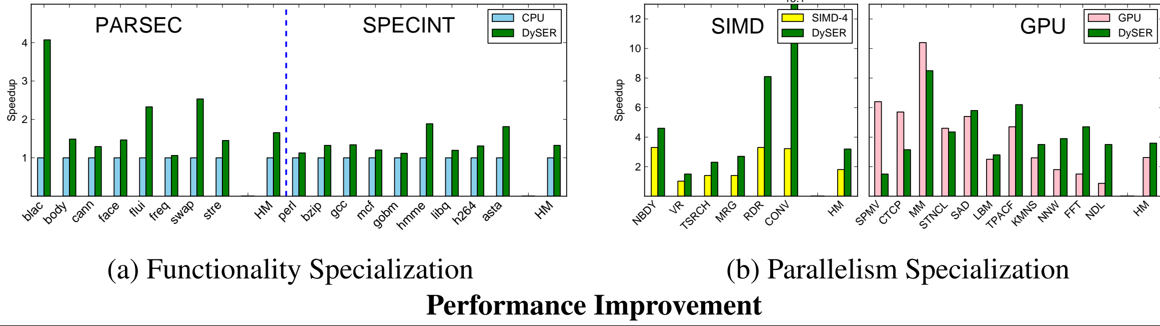
- CUTCP requires long latency functional units which are not pipelined in DySER, but are pipelined in the GPU, causing the GPU to outperform DySER.
- Benchmarks like SAD, STNCL, and MM perform similarly in both architectures because they can all exploit highly regular data access efficiently. For TPACF, the GPU and DySER end up with similar performance, but take different approaches for efficient histogram calculation. DySER can parallelize each index calculation, while the GPU uses threads to calculate multiple indices simultaneously.
- The DySER implementation of FFT caches the reused transcendental operations performed on the load slice, which turns out to be highly beneficial, and outperforms the GPU approach.

*DySER provides a harmonic mean speedup of* $3.6\times$ *over our baseline, with a range of* $1.5\times$ *to* $8.5\times$*, and energy reduction of 64%, with a range of 34% to 81%.*
*It is up to* $4\times$ *faster than GPU and 64% more energy efficient.*

**Summary:** Overall, we observe DySER can be trivially configured to exactly imitate SIMD and can surpass SIMD's performance. DySER is competitive with GPU performance and its mechanisms are equally flexible.

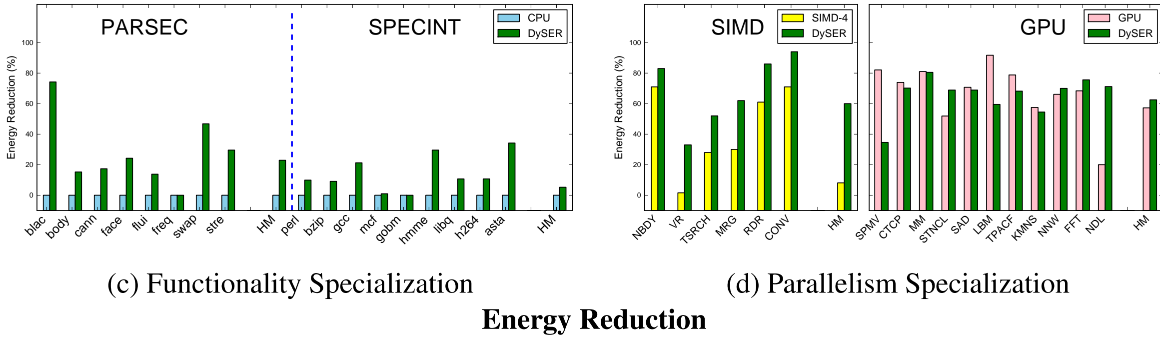
## 4.4 Feasibility of Implementing/Integrating DySER

We implemented DySER as standalone RTL for verification and to determine the feasibility in terms of design, area, and power. To attain an area estimate, we synthesized DySER using Synopsys Design Compiler with the TSMC 55nm Standard Cell library. For the area of FP-DIV and SQRT units, we scaled previous estimates [9] to 55nm. In total, the 64-unit DySER described earlier occupies an area of 1.54 $mm^2$, which is approximately the size of the Intel ATOM FP/SIMD units (from die-photos the FPC unit of Atom is 1.45 $mm^2$ in 45nm), and is about half the size of a GPU SM (from die-photos, area of one SM in NVIDIA GT200 is 2.7 $mm^2$ at 65nm), when all are scaled to 55nm. The interfaces, switches, and flip-flops contribute to 42% of DySER's area and 18% of energy. Overall we conclude DySER is area and energy efficient.




(a) Functionality Specialization (b) Parallelism Specialization

**Performance Improvement**




(c) Functionality Specialization (d) Parallelism Specialization

**Energy Reduction**

Figure 6: Performance and Energy improvements from DySER specialization (CPU = CPU w/o SIMD; SIMD-4= 4-wide SIMD; GPU = 1SM/8-Wide; DySER = 64-FU-DySER + CPU)

To demonstrate DySER can be integrated easily into conventional processors, we have integrated a prototype of DySER into the OpenSPARC processor, including SPARC ISA extensions, a compiler based on LLVM, and verified the implementation on an off-the-shelf Virtex-5 FPGA board booting unmodified Linux and running applications. Due to FPGA size limitations, we could only map a four FU DySER, which limits performance analysis. We are exploring a full-fledged 64-unit prototype.

# 5 Conclusions

We have shown how the DySER architecture unifies disparate attempts at functionality and parallelism specialization in a single architecture with a set of mechanisms. Our quantitative results show DySER is competitive or outperforms SIMD and GPU accelerators, performs well in terms of functionality specialization, and is a feasible design easily integrable with a processor. We reflect on DySER's potential practical usages.

**Programming tradeoffs:** SIMD accelerators or short-vector extensions can provide speedup, but compilers have difficulty targeting SIMD well. Programmers typically must use compiler intrinsics, which creates severe portability and maintainability problems. Although there have been successful GPGPU programming languages like CUDA, GPUs pose their own set of programming challenges. Not only must the user learn a new language, they must learn the massively multi-threaded thinking paradigm, give up on familiar sequential program debugging, and apply GPU specific optimizations. DySER programming is relatively simple, uses sequential C++ code, and uses established debugging methodologies.

**SIMD Evolution:** Even though the SSE family is SIMD, many extensions to SSE (SSE3 and later) have instructions that are not purely word parallel. For example, the instruction `HADDPD` and its variants operate on elements from the same vector. Also, there are instructions specializing the functionality like `MPSADBW`, which computes the sum of absolute differences. This exemplifies a trend towards providing functionality specialization in data parallel accelerators. SIMD evolution, by increasing width, does not provide scalable performance benefits across workloads, whereas DySER scalably adapts. Hence, we feel DySER is the natural evolution of these instructions sets.

**GPU Evolution:** Conversely, GPUs are leaning toward the CPU side by providing caches and eliminating redundant work with their scalarization approach which effectively creates a “control” core and a set of compute-cores, much like DySER’s organization. Again, we feel DySER-like integration is the direction GPUs are headed.

**Replacing SIMD or GPU:** In summary, we feel DySER is a viable candidate for replacing SIMD short vector instruction sets. With some simple extensions, DySER can be augmented to emulate existing instruction sets like SSE, thus providing backward compatibility. Clearly, DySER is not a GPU replacement, since it cannot perform graphics tasks well. It is a promising alternative for “design-constrained” environments like Tilera, ARM in servers, and Oracle’s T4 successor to target high-performance computing. In these cases, a completely new processor design like a GPU, or integration of GPU with a core, and adoption of a new software ecosystem may be prohibitively complex. In contrast, DySER’s hardware and software ecosystem are non-disruptive.

Broadly, DySER’s unifying functionality and data-parallel specialization mechanisms provide a platform

for energy efficient computing.

# References


[1] The gem5 simulator system, http://www.m5sim.org.

[2] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *ISPASS '09*.

[3] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The garp architecture and c compiler. *Computer*, 33:62–69, April 2000.

[4] 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 *ISCA '05*.

[5] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *IEEE Computer*, 33(4):70–77, April 2000.

[6] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *HPCA 2011*.

[7] S. Gupta, S. Feng, A. Ansari, S. A. Mahlke, and D. I. August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *MICRO '11*.

[8] S. Hong and H. Kim. An integrated gpu power and performance model. In *ISCA '10*.

[9] T.-J. Kwon and J. Draper. Floating-point division and square root using a taylor-series expansion algorithm. *Microelectronics Journal*, 40(11):1601 – 1605, 2009.

[10] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO 42*.

[11] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, M. Budiu, and S. C. Goldstein. Tartan: Evaluating spatial computation for whole program execution. In *ASPLOS-XII*.

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

[13] S. R. Sarangi, A. Tiwari, and J. Torrellas. Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware. In *MICRO '06: Proceedings of the 39th Annual International Symposium on Microarchitecture*, pages 26–37, 2006.

[14] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *ASPLOS '10*.

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