

Detailed Performance Evaluation of Data-Parallel Workloads on the DySER Prototype System

University of Wisconsin-Madison ECE Graduate Project

Completed By:
Zachary Marzec

Abstract

With the end of classic Denard scaling, new improvements to microarchitecture are needed in order to maintain performance while continuing to decrease power consumption. An emerging trend has put focus on hardware accelerators as a means of increasing efficiency. One such accelerator, the Dynamically Specialized Execution Resource block accelerator (DySER), utilizes a highly-flexible functional unit array to exploit dataflow techniques in order to achieve large scale throughput and energy savings. In this project, performance of DySER on the gem5 simulator as well as the OpenSPLYSER system, which includes the DySER accelerator integrated with an OpenSPARC T1 processor, will be investigated. The primary focus of this performance evaluation will be on data-parallel workloads taken from UIUC's PARBOIL benchmark suite as well as other common high throughput kernels. This document will detail benchmark details, implementation efforts, analysis of performance figures, and provide key points for future work.

Outline

This project report will be presented in the following order:

1. Introduction and Background – Details DySER history and OpenSPARC Integration
2. Benchmarks – Explains Data-Parallel nature of benchmarks and details each
3. Implementation Details – Issues with SPARC and benchmark integration
4. Performance Evaluation – Analysis of the DySER performance on simulator and VCS
5. Future Work

Any comments or questions about this work can be sent to zmarzec@gmail.com.

1 Introduction

General purpose processors are fast approaching a power limit due to the end of classical voltage scaling and decreasing capacitance [6]. While technology continues to follow Moores law and transistor density increases, efficiency and total transistor use is set to decrease. Studies have shown that a 80W power budget on a modest sized die with recent technology will only be able to switch less than 7% of the die at full frequency, leading us to an age of “Dark Silicon” [4]. Thus, many have been turning to hardware accelerators and other specialized hardware for efficiency improvements [1, 4, 8]. These accelerators are not a new concept; a long running example are GPU processors that are commonly found in almost all desktop computing solutions for high performance video. These extra components provide a way to keep on track with Moores law by increasing power efficiency and further improving processor performance. One accelerator of keen interest is the Dynamically Specialized Execution Resource block accelerator (DySER) which will be the focus for this report.

My primary contributions to this project are as follows:

- Converted PARBOIL and SEE throughput benchmarks to SPARC and debugged issues
- Detailed the schedules for the benchmarks as well as provided a basis for a heterogeneous DySER
- Performed detailed analysis of benchmarks and determined areas for possible improvement
- Provided future work and suggestions for improvements to the current DySER Toolchain

2 Background

This section describes the details required to understand the DySER hardware accelerator, our DySER+OpenSPARC prototype, the enhancements made, and other DySER specific lingo. Much of the DySER project was a collaborative effort over the course of many years. While I was involved with many aspects of the project, the initial formulation of DySER and its implementation into OpenSPARC are not my primary work. To avoid disrupting the flow of this document, these details are all combined and presented in this background section.

2.1 The DySER Accelerator

Our interest in DySER comes from a few key areas. First, it is a general purpose accelerator based off the principals of dataflow: the method where many operations are “chained” together in order to flow data without the need for frequent write backs to a register file. DySER utilizes programmable functional units connected by switches on a mesh-interconnect in order to follow dataflow techniques to reap the benefits of highly data parallel sections of code. It has shown to increase code speedup on average by 2.1x while decreasing energy by up to 70% across a variety of workloads [6]. DySER also has been shown to be energy efficient. In traditional superscalar processors, the write back stage consumes a great deal of power; use of DySER reduces this through long computation chains with no intermediary register write backs and fewer instruction fetches.

An example of how a code can be generated into a DySER can be shown in Figure 1. Suppose that the given function “foo” was to be mapped. DySER would be a perfect candidate to run this region of code. A data flow graph could be generated and the DySER could be configured accordingly. The functional units pass data between the switches which are latched for performance. In order for data to travel between functional units, a minimum 2 cycles penalty is

paid (one to travel from the functional unit to the switch, one to travel from the switch to the next functional unit). Paths can be long and are minimized in configurations to increase IPC and utilization. Shown in the figure is a 2x2 DySER; a block 2 functional units high and wide. DySER can be any size but a square configuration is generally used.

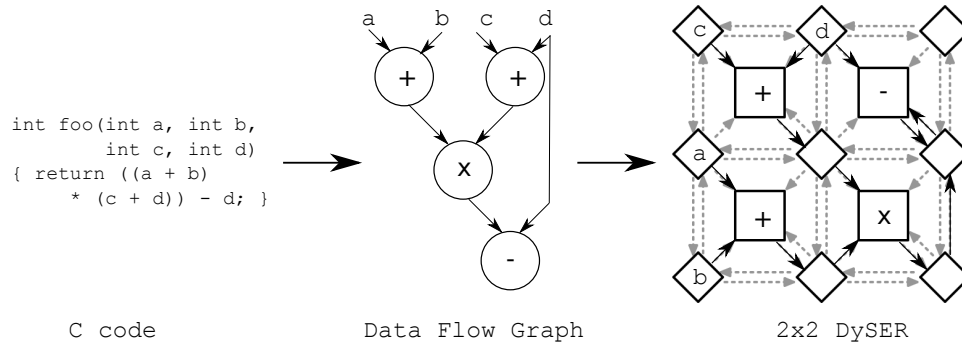


Figure 1: An example C-code being translated and mapped to a 2x2 DySER block.

Another key feature of DySER is its non-invasive nature; it does not disturb already existing pipeline functionality. As shown in Figure 2, DySER “sits” below the execute stage of a pipeline. DySER is a computation engine that relies on the main processor to handle loading and storing values for computation. Thus, if an instruction does not need to be executed DySER, it can bypass it altogether with no additional changes. DySER is a standalone unit requiring very few changes to the ISA. It is also quite small, taking up an area comparable to a 64K block of SRAM [6].

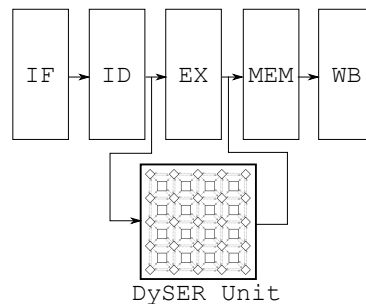


Figure 2: A typical pipeline setup with a DySER block.

In short, DySER is lightweight, has shown performance gains, and can provide for lower power across several workloads.

2.2 The DySER Prototype

The DySER model was taken from simulator to prototype as shown in [1]. DySER integration into Sun’s OpenSPARC T1 processor was chosen due to its open source nature and commercial availability. Following DySER’s lightweight nature, minimal changes were required for hardware integration. Only 5 instructions were added to the SPARC ISA to accommodate initial configuration, sending, and receiving data from DySER. In order to accomplish this, an implement-dependent instruction within the ISA was utilized. Only 514 lines of Verilog code were added or changed in order for basic DySER functionality. Notable inclusions to the RTL are the IFU (275 lines changed to reverse engineer opcodes for DySER use), the LSU (23 lines changed for memory control), and the EXU (216 lines changed to add DySER model and 18 addition flip-flops). It is important to note that the MMU was not changed.

To verify the DySER integration, a set of SPARC regression tests were utilized to ensure that the original pipeline kept its full functionality. Once completed, a special set of DySER tests

were performed via unit-level testing of the DySER module. After DySER functionality was tested and confirmed, the OpenSPlyER prototype was mapped to a Vertex 5 FPGA board and was able to run Ubuntu 7.10 Linux. This operating system was a SPARC version and, while DySER was not used to enhance the operating system, it's inclusion did not hinder functionality. Testing showed that DySER did not perform as well as the simulator expected due to various issues with OpenSPARC, the DySER compiler, and LLVM control flow issues [1]. A compiler for DySER exists and is an integral part of the entire DySER project. However, hardware changes and performance analysis are the primary aspects of this project and all compiler issues are considered out of scope.

2.3 DySER Terminology and Example

In order to better understand the terminology presented throughout this project, a small example will be presented on how to transform standard C code into old DySER prototype code (with single loads) as well as the new prototype parallel loads.

For this example, we look at a simple C code used for a cumulative sum:

```

for(int k = 0; k < num; k++) {
    sum += AMatrix[k];
}

```

In order to begin utilizing DySER, we must create a schedule. In this schedule, the DySER compiler determines how to take the region on interest (the cumulative sum in our case done over 'num' times) and map it to the available amount of resources. For our example, we are utilizing a 4x4 DySER. Once this has been accomplished, we can then send data to each of the switches (represented by diamonds, i.e. 1-8, 18, 24, 28, 29, 24, 38, 39, 48). Once the data has flown through each of the functional units (represented by squares, 0/1 notate the order of data going to the functional unit, 0 first, 1 second), completed data can be received (represented by left/right triangle on the switch, i.e. port 30 in our case).

An example of a compiler generated cumulative sum schedule can be found below:

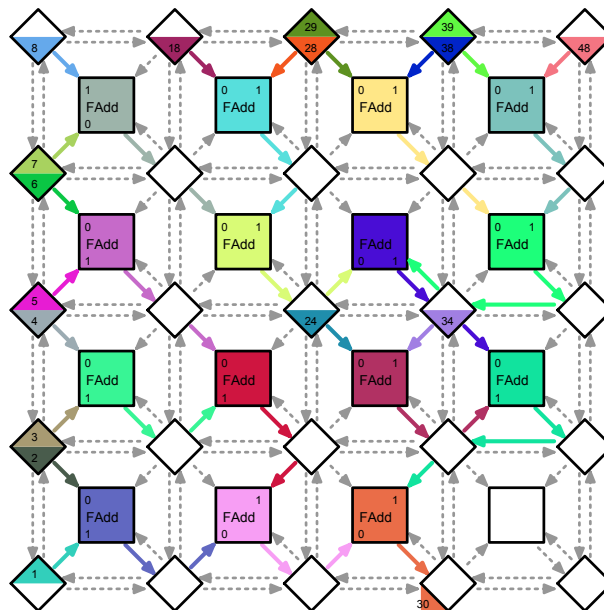


Figure 3: A 16-input cumulative sum schedule.

Once we have the schedule created, we are now able to send data to DySER. Below is styled code generated by the compiler in order to use DySER:

```

int temp;
dyInit();
for(k = 0; k < num; k+=16) {
    DyLOAD(AMatrix[k+0], 1);
    DyLOAD(AMatrix[k+1], 2);
    DyLOAD(AMatrix[k+2], 3);
    ...
    DyLOAD(AMatrix[k+14], 38);
    DyLOAD(AMatrix[k+14], 39);
    DyLOAD(AMatrix[k+15], 48);

    DyRECV(30, temp);
    sum += temp;
}

```

In order to better understand this code, provided is a list of the DySER instructions needed to communicate between the processor and DySER:

Instruction	Description
dyInit()	Initialize DySER with the specified schedule.
DySEND(reg, port)	Send single datum from the register file to a specified port in DySER.
DyLOAD(mem, port)	Send single datum from memory to a specified port in DySER.
DyRECV(port, reg)	Receive single datum from a DySER port and write it to the register file.
DySTORE(port, mem)	Receive single datum from a DySER port and write it to memory.

To continue to improve DySER performance, vectorization can be done by loading multiple values into DySER at once. Recall that DyLOAD only loads one datum at a time. Below is a table of vectorized instructions used:

Instruction	Description
DyLOADPS(mem, port)	Send 4 bytes of continuous data from memory to 4 DySER ports.
DyLOADPS1(mem, port)	Send 1 datum from memory to 4 DySER ports.
DyLOADPD(mem, port)	Send 8 bytes of continuous data from memory to 8 DySER ports.
DyLOADPD1(mem, port)	Send 1 datum from memory to 8 DySER ports.
DyLOADPQ(mem, port)	Send 16 bytes of continuous data from memory to 16 DySER ports.
DyLOADPQ1(mem, port)	Send 1 datum from memory to 16 DySER ports.

In order to use vectorization, we will need to include information in the configuration to set “wide” ports to the number of ports. Using this information, we are able to further improve our source code to the following:

```

int temp;
dyInit();
for(k = 0; k < num; k+=16) {
    DyLOADPD(AMatrix[k+0], 0);
    DyLOADPD(AMatrix[k+8], 1);

    DyRECV(30, temp);
    sum += temp;
}

// Note:
// Wide port 0 maps to DySER ports: 1 2 3 4 5 6 7 8
// Wide port 1 maps to DySER ports: 18 24 28 29 34 38 39 49

```

These wide ports are generated by the compiler in order to take advantage of the current resources. For the purposes of this document, we will assume that the largest load we can do at a given time is a DyLOADPD, that is, 8 values at a time.

2.4 DySER Extensions

The original DySER prototype was extended in order to provide higher performance. One major factor for high performance in DySER is ensuring that the unit is “well fed.” To keep ILP high, DySER must continually work on data in order to mitigate load latency. The prototype as described in [1] only supported single loads. That is, one piece of data was loaded at a time. In order to maximize throughput, loading of multiple pieces of independent data is required. For this project, we have implemented the necessary logic to simulate loading up to 8 values to 8 different DySER ports at once. To accomplish this, DySER completes a load at the initial memory location and then sends the same value to the 8 corresponding ports. In essence, all DyLOADPD DySER instructions are transformed into DyLOADPD1. While this is not a true parallel load, this project focuses on performance evaluation and this method is appropriate. To accommodate this increase in data, the input/output FIFOs for DySER required modification. Since the original prototype had only 4 deep FIFOs, new hardware was designed so that FIFOs could be any depth. In the interest of synthesized area, the depth of the FIFOs were set to 16 spots.

Changes to the VCS framework, the simulation environment for the DySER+OpenSPARC RTL, were also added. For faster validation, a checksum system was created. Each benchmark has a certain portion of their completed output matrix summed as a checksum. This value is then checked in the RTL version and, if it is within a certain range, it calls a “pass” trap; otherwise it calls “fail”. This is easy to view success/fail in these logs and corresponding register values can be viewed for debugging.

The FPGA model was also extended in order to integrated these changes. In [1], a 2x2 32-bit DySER (4 total functional units) was synthesized and utilized for performance purposes. However, through further testing, a 2x2 was not ideal and a 4x4 (16 functional unit) DySER should be utilized. Without compromising the bit width, this proved to be impossible given the amount of resources available on the FPGA. Therefore, the concept of “hardDySER” was introduced in this project. In order to save space, a DySER’s configuration is frozen and cannot be changed during kernel execution. Any switches that are not used are eliminated along with any unused paths. This allows for 4x4 and even 5x5 (20 functional unit) DySERs to fit on the FPGA. This change does not affect performance as all critical sections that are used are still implemented; no alterations to timing or code are made to enhance the performance of hardDySER as it acts like a normal DySER without reconfigurability. A new hardDySER is created for each benchmark. Since DySER cannot be reconfigured once a kernel has started, some benchmarks have been omitted.

3 Implementation Modifications

Previous work with the DySER simulator utilized benchmarks using the x86 architecture. A translation of these benchmarks to the OpenSPARC architecture was required for both the scalar case and the DySER+OpenSPARC model. This is one of the my key contributions. Completing this was fairly straightforward; some benchmarks worked with no changes whatsoever. They were tested against their x86 counterparts for full functionality and correctness. However, there were a few implementation points worth highlighting.

The original benchmarks utilized SSE/x86 specific instructions in order to optimize loading/storing large amounts of data from DySER. The OpenSPARC implementation used did not accommodate these instructions and thus had to be removed. This caused a performance degradation but is to be expected with this architecture.

The endianness of the architectures also needed to be addressed. This was only required for spots where files needed to be read since they were stored in little-endian notation (x86). This was accomplished with the following routine which was called for each byte within the given data stream:

```
uint32_t swap_byte32(uint32_t x)
{
    return (uint32_t)((uint32_t)(x) & 0xff) << 24 |
            ((uint32_t)(x) & 0xff00) << 8 | ((uint32_t)(x) & 0xff0000) >> 8 |
            ((uint32_t)(x) & 0xff000000) >> 24);
}
```

This call was completed outside of the measured kernel so no performance was lost because of this function call.

Finally, to further improve performance, software pipelining, a technique that attempts to hide latency, was also used on all of the benchmarks. To understand software pipelining in the context of DySER, we software pipeline the cumulative sum example:

```
int temp;
dyInit();

DyLOADPD(AMatrix[k+0], 0);
DyLOADPD(AMatrix[k+8], 1);

for(k = 16; k < num; k+=16) {
    DyLOADPD(AMatrix[k+0], 0);
    DyLOADPD(AMatrix[k+8], 1);

    DyRECV(30, temp);
    sum += temp;
}

DyRECV(30, temp);
sum += temp;
```

In this example, two invocations are sent in the beginning of the kernel. As we complete work, we receive the previous invocation sent in. This attempts to eliminate the latency between the last load and the first receive. In general, it is useful but causes some minor slowdown in some benchmarks due to the overhead of slow receiving.

4 Benchmarks

This section details the benchmarks used in the evaluation of DySER. In order to showcase the DySER architecture, benchmarks that were high throughput, where a small code region dominates the runtime, and where computation can easily be scheduled, were chosen. Specifically, we wanted regions where 10% of the code would occupy 90% of the runtime. We found that scientific kernels best fit this criteria. In addition, since DySER is an execution engine reliant on the processor to feed it loads, benchmarks with a regular memory access pattern will perform best on DySER; that is, memory that can be easily predicted by a loop counter or some other regular pattern. While benchmarks with irregular access, such as pointer hopping or memory access via irregular calculated address, exist within some of these scientific throughput kernels.

The first set of benchmarks chosen mimic that of GPU workloads in the PARBOIL suite created by UIUC [7]. They are detailed below:

Benchmark	Description	Memory Access
fft	Fast Fourier Transform. Specialized algorithm aimed at GPUs to combine transposes and optimize memory memory performance. SIN/COS was taken out of the Kernel of interest.	Regular
kmeans	Cluster analyses that takes n values and places them into k clusters partitioning them based on nearest mean. Heavy use of squared and add units.	Regular
mm	Dense matrix multiply (matrices filled with values).	Regular
mri-q	Magnetic Resonance Imaging. Models scanner configuration for calibration used for 3D magnetic resonance image reconstruction. Note: Use of SIN/COS used within kernel of interest; SIN/COS unit unavailable in DySER. Replaced with ADD unit.	Regular
spmv	Sparse matrix multiply (special data structures to represent sparse data). Stored in an adjacency matrix.	Irregular
stencil	3-D stencil operation.	Regular
tpacf	Two Point Angular Correlation Function. Analyzes spatial distribution of observed astronomical bodies. Note: DySER code optimized to use more regular memory accesses than original version.	Irregular

We also found that other high throughput kernels were a good candidate for high performance on DySER. They are detailed below:

Benchmark	Description	Memory Access
conv	2x2 Convolution.	Regular
radar	1D Convolution.	Regular
treesearch	Binary Search tree algorithm.	Data Dependent

By analyzing these benchmarks, we hoped to confirm the assertions presented in [5]. That is, the type of impact regular, irregular, and data dependent memory accesses as well as the other factors that play a role in DySER performance.

A great deal of time and effort went into creating hand optimized DySER schedules, the configuration of functional units and switch paths. These were done for peak performance and they can be seen in the Appendix of this work. These were assuming that a DySER could have customized functional units for each benchmark. If a DySER was needed that did not have the ability to change functional units (i.e. only switch paths could be changed, we call this “heterogeneous DySER”), the current schedules reveal the functional unit mix required for this set of benchmarks. These statistics can be viewed below:

Benchmark	FMul	FAdd	FSub	+	>f	<<	>u	OR
fft	4	3	3					
kmeans	8	7	8					
mm	8	7						
mri-q	6	3						1
spmv	8	6						
stencil	2	1	2					
tpacf	3	2		7	7			
conv	8	8						
radar	8	9	4					
tresearch				8		4	4	
max	8	9	8	8	7	4	4	1

From this chart, at least 49 functional units in order to accommodate all of these benchmarks on 1 heterogeneous DySER (a 7x7 DySER would be required). If we wish to omit the OR unit used for mri-q, we could still use a 7x7 DySER to fit all of the units appropriately and get an extra functional unit as a spare. The OR in mri-q is used to replicate data as inputs that feed two functional units was not supposed under our currently DySER model.

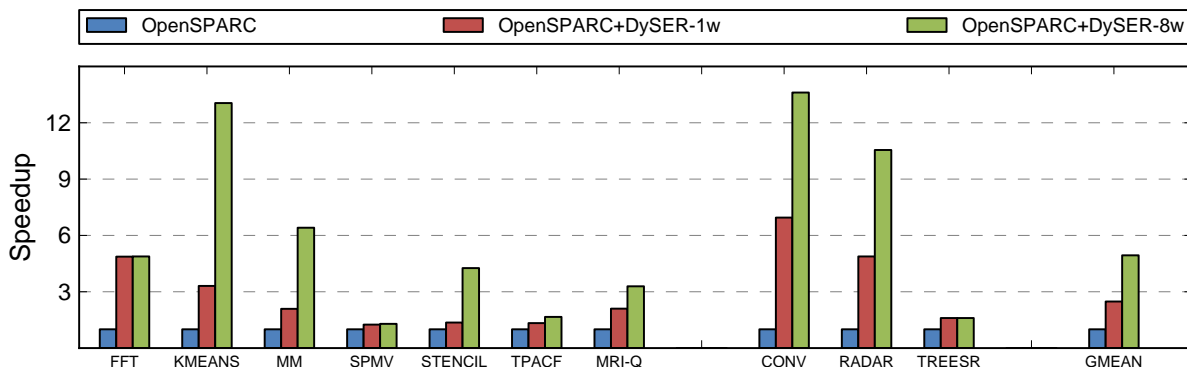
5 Performance Evaluation

This section details performance evaluation methodology and analysis for the DySER + OpenSPARC prototype.

In order to obtain performance numbers, a reasonable workload size was chosen that could be run in VCS without significant downtime. These same workloads were then run on the gem5 simulator [2] and then compared cycle by cycle. A checksum was matched in the simulator to the VCS checksumming system to verify for correctness.

5.1 RTL: Inorder versus DySER

Below is the speedup compared to the OpenSPARC baseline of DySER with no vectorization (DySER-1w) and DySER with 8-wide vectorization (DySER-8w).



A geometric mean shows us that DySER is $2.5x$ faster than the OpenSPARC baseline without DySER vectorization and $4.9x$ faster with DySER vectorization. From this graph, we are able to categorize the results into 3 categories: places of high performance, places of poor performance, and places of implementation deficiencies.

High DySER Performance

Benchmarks such as kmeans, mm, stencil, conv, and radar are able to take advantage of the abundant amounts of parallelism and maintain a high ILP. This is to be expected since there are regular memory patterns and DySER need not stall on complex address calculations.

Poor DySER Performance

There are two spots for poor DySER performance. The first is in SPMV. Sparse matrices in SPMV are stored as adjacency matrices; this requires many indirect loads. DySER is only able to complete small intermediate calculations and some control flow operations once all of the indirect loads have completed. The amount of speedup is very limited.

Another benchmark of poor performance is Treesearch. This is due to the nature of the benchmark; the treesearch algorithm is data dependent. Therefore, only a small amount of work can be done and no vectorization can be done. This explains how the 1w and 8w numbers are identical. If the algorithm was changed, vectorization can be utilized to increase DySER performance.

DySER Implementation Deficiencies

Tpacf is one benchmark that shows poor performance due our implementation. Tpacf places the first 8 computations of each pass in the load slice and does not begin to use DySER until then. Since the benchmark size is small, it is not able to take advantage of DySER enough to show a significant performance enchantment. Our gem5 results show that small benchmark sizes exhibit this behavior. If the benchmark size is increase, we expect to see performance similar to mm since our DySER version uses more regular loads than its OpenSPARC counterpart.

FFT is another instance where our implementation inhibits performance. As previously mentioned, when using vector loads, only DyLOADPD was implemented. This algorithm of fft has regions divided into loops of 2, 4, 8, and 16 values. For the 8 and 16 data value sections , we are easily able to use DyLOADPD without issue. The 2 value data section, we use 2 DyLOADs. However, for the 4 section, we should be able to use DyLOADPS to take advantage of DySER. Since this vector instruction is unimplemented, we defer this to 4 DyLOADs. Again, since the FFT benchmark size is too small, we are not able to use high amounts of the vectorized instructions to get high performance. Once the benchmark is increased in size, we expect it to get mm levels of performance.

5.2 General DySER performance

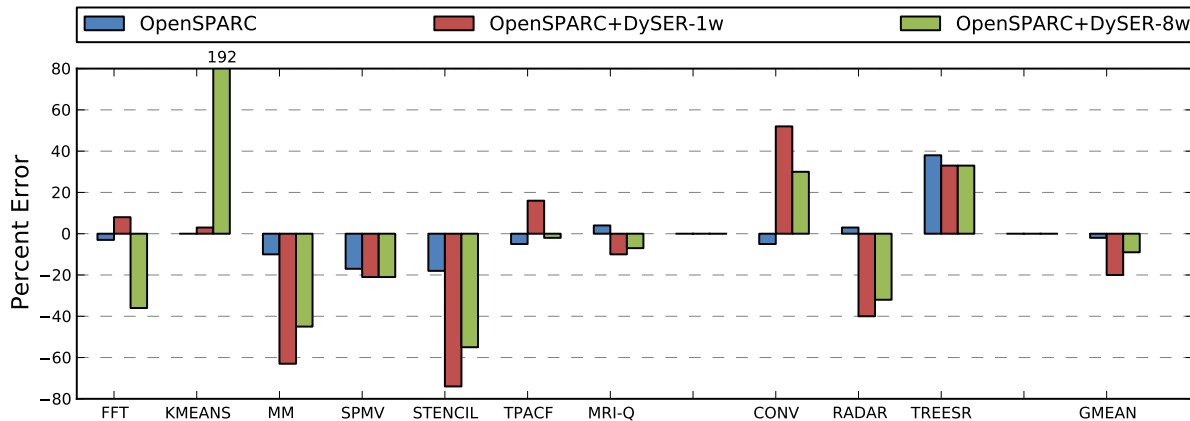
In summary, when regular memory patterns exist and there is abundant amounts of parallelism, DySER does extremely well. In cases where it performs poorly, it can exploit regions of control flow in order to give some minimal performance benefit. These findings are consistent with our simulation based estimates as seen in [5] and [6].

It is important to note that DySER does not cause performance degradation in any case compared to the OpenSPARC baseline. However, it is possible to create a DySER that causes performance slowdown. For example, if the data in DySER gets routed very inefficiently, this can cause high latency which can reduce performance. Also, if very short operations are used for DySER, the overhead of the DyLOAD and DyRECV can cause performance loss. It is the duty of the compiler to determine the best route possible and map as large a region as possible onto DySER. Therefore, these scenarios of DySER hurting baseline performance will be very rare.

A brief mention of DySER performance outside the in-order model is worthwhile. DySER reduces the number of fetch and write-back/commit instructions, yet, this is classically not a performance bottleneck. Since we defer long running loop operations to DySER, we are able to look deeper in the program code and fetch instructions at DySER completion. This, theoretically, increases our instruction window and yields a great deal of performance. DySER also takes advantage of high ILP through its many ALU's. Other accelerators methods of vectorization are unable to keep up their ILP high compared to DySER due to significant overheads.

5.3 Simulator: Inorder versus DySER

Simulators have been known to be prone to large error deviations compared to their target architecture [3]. Therefore, it is important to study our current simulator implementation and understand how much error exists and where that error originates. Binaries were run in VCS and on gem5 for all of the same benchmarks and were compared by cycles. The percent error graph is shown below:



A geometric mean of -2% for the baseline, -20% for unvectorized DySER, and -9% for vectorized DySER, were observed. These errors were calculated by taking the ratio of simulation cycles to RTL cycles and then subtracted from 1. A negative percentage shows the simulator underreporting the true number of cycles (i.e. the simulator cycle count is lower than it actually is). Conversely, simulator overreporting occurs when a positive error percentage is obtained (i.e. performance better in RTL version, positive error is ideally better).

5.4 Sources of error

Our first observation for error was the binaries run in each test case. It was found that, while the same compiler was used in both testing environments, the RTL binaries were different, sometimes drastically so. To resolve this, the assembly files for the RTL were generated and a new binary was compiled from them. All of the scalar runs in the graph have had this technique applied. For instance, radar originally had a 30% overreporting of performance. MM and Stencil also showed large improvements due to this. However, this did not resolve all large error discrepancies for the baseline.

The scalar version of Treesearch showed a large overreporting error. Treesearch’s main kernel is heavily dominated by integer operations; this was the only benchmark to exhibit this property. In the gem5 simulator, the minimum latency for any instruction is 2 cycles (1 cycle to fetch, 1 cycle to execute). This is an issue when integer operations in RTL execute in 1 cycle. One loop iteration in Treesearch for the RTL model took 9 cycles versus 16 for gem5. Expanding this error over 16,000+ iterations, the discrepancy amounts to about 40%. Once the other RTL modeling inconsistencies are factored in, this yields the error shown in the graph.

Large areas for underreporting in the scalar version come from modeling inconsistencies with the OpenSPARC model. Traps and other memory related accesses (TLB misses/MMU accesses) are not modeled in gem5 which contributes to the underreporting. Traps latencies for the benchmarks tested were quite large (in the thousands of cycles) and happened more frequently due to large input sizes. Certain functional units are also improperly modeled. The floating point multiply, for instance, exists in the FPU which sits off chip; this has latency to arbitrate for the unit, send data on the bus, pipeline where possible, and then send data back. The gem5 simulator keeps each access as constant and does not mimic OpenSPARC behavior.

The DySER discrepancies are also presented. A first step to ensure accurate analysis will be to match the gem5 binaries to the RTL binaries as done previously. This is part of ongoing work and this analysis will not be detailed in this report.

6 Future Work

While a large effort has been put into DySER project thus far, there is a great deal of work to complete in order to enhance the DySER prototype as well as continue performance testing. The method of hardDySER must be deprecated as reconfigurability is a key feature of DySER for FPGA testing. Work is ongoing and a redesigned switch shows a 4x4 32-bit DySER can be mapped to our current FPGA. We have acquired a Virtex 7 which has considerably more resources and we hope to integrate even larger DySER blocks. Once completed, performance testing will be done and we anticipate the performance numbers to be consistent with this report.

DySER stalling should also be added to the prototype in order to further increase performance. Currently, nops must be added by hand between the last send and the first receive of a DySER-ized program. Not only is this tedious (a certain amount must be added), if too many are added, performance can easily be lost (studies show that an extra 16 nops can cause a 8% performance loss). Downtime to nop tuning can be large and this technique must be applied for each new data set (larger data sets may require more nops to account for cache/TLB misses, etc). An automated way for DySER to control this is necessary to minimize this downtime.

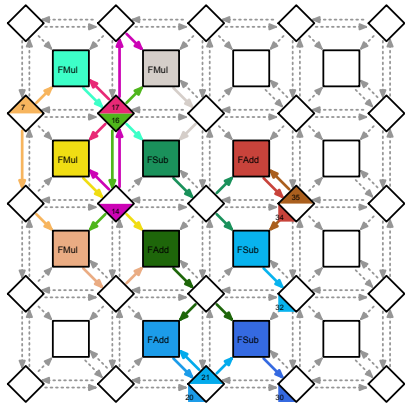
Finally, a larger array of benchmarks should be analyzed. While this primarily focuses on places where DySER succeeds, a look into order areas, such as database workloads, can show how DySER needs to improve to handle a wide variety of workloads. Memory indirection is a place for performance loss in DySER but perhaps new techniques can be studied in order to make DySER a more generalized solution across workloads of all varieties.

References

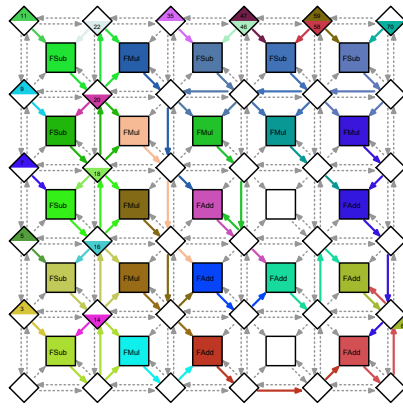
- [1] Jesse Benson, Ryan Cofell, Chris Frericks, Chen-Han Ho, Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. Design integration and implementation of the dyser hardware accelerator into opensparc. In *Proceedings of 18th International Conference on High Performance Computer Architecture "HPCA"*, 2012.
- [2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39:1–7, August 2011.
- [3] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context*, SSR '01, pages 266–277, New York, NY, USA, 2001. ACM.
- [4] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th International Symposium on Computer Architecture*, June 2011.
- [5] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying functionality and parallelism specialization for energy efficient computing. *IEEE Micro*, 33(5), 2012.
- [6] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of 17th International Conference on High Performance Computer Architecture*, 2011.
- [7] J.A. Stratton, C. Rodrigues, I.J. Sung, N. Obeid, L.W. Chang, N. Anssari, G.D. Liu, and W.W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.
- [8] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 205–218, New York, NY, USA, 2010. ACM.

Appendix

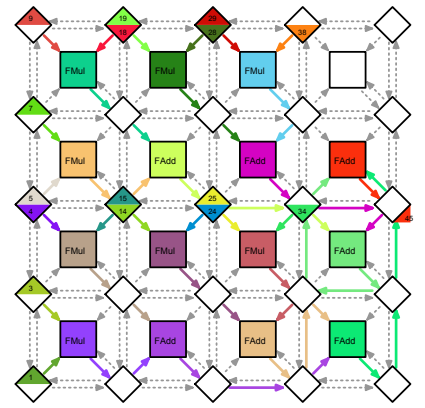
PARBOIL DySER Configurations



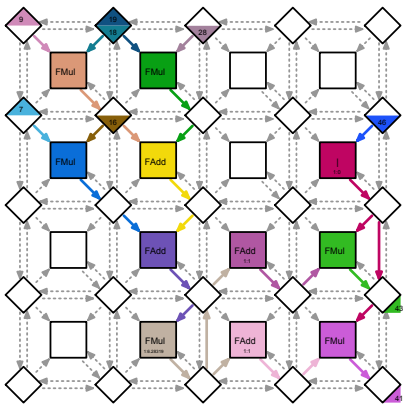
(a) fft



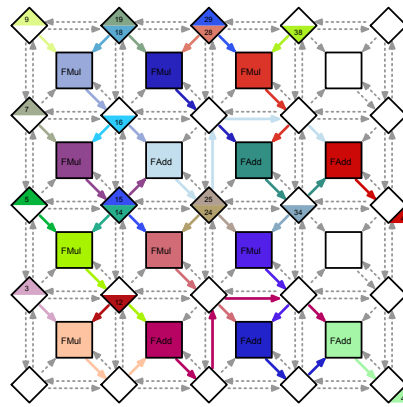
(b) kmeans



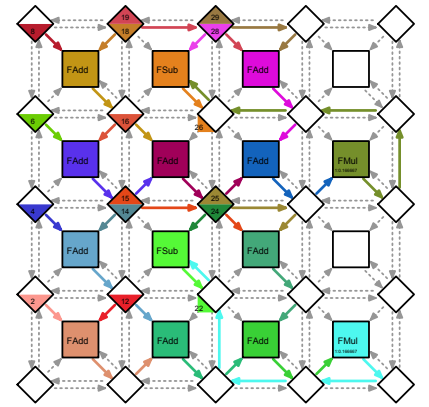
(c) mm



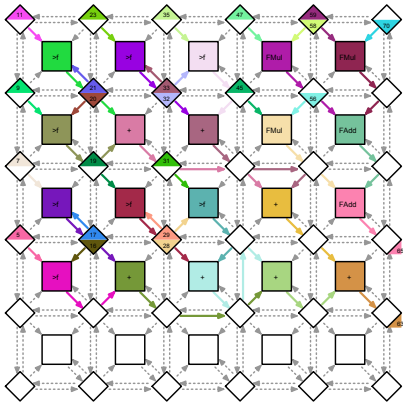
(d) mri-q



(e) spmv

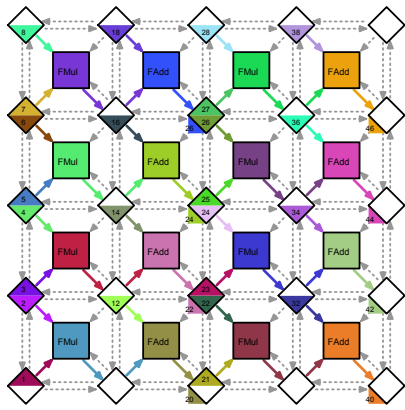


(f) stencil

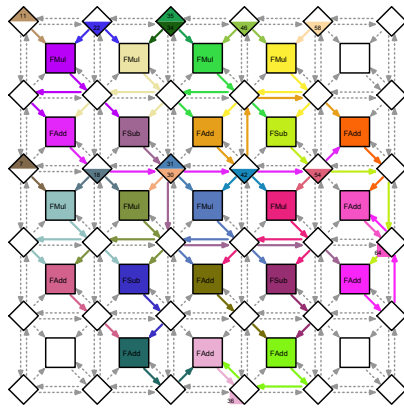


(g) tpcf

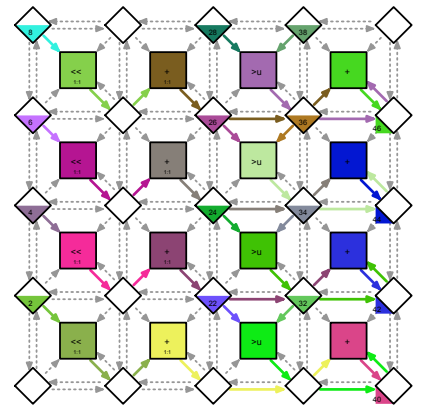
Other Kernel DySER Configurations



(a) conv



(b) radar



(c) treearch