Design, Implementation, and Verification of DySER for Dynamic Specialization in General Purpose Processors

Jesse Benson
University of Wisconsin – Madison
jmbenson2@wisc.edu

Abstract

The Dynamically Synthesized Execution (DySE) model is an execution model to improve the energy efficiency and performance of general purpose programmable processors through dynamic specialization. The purpose of this project was to develop a prototype implementation of a DySE Resource (DySER) to demonstrate the viability of the design. A scalable DySER block was developed and integrated into the OpenSPARC T1 pipeline. The modified processor was synthesized onto a Virtex-5 FPGA and can execute application binaries created with our co-designed compiler.

This paper provides an overview of the DySE Resource (DySER) design, hardware and compiler implementations, verification, and design analysis. Performance analysis shows speed-up can be achieved for programs that execute in phases or that contain ample computation reuse.

1. Introduction

The Dynamically Synthesized Execution (DySE) model is an execution model to improve the energy efficiency and performance of general purpose processors [1,2]. The main insight behind the model is as follows. Applications often execute in phases, and these phases can generally be identified at compile time. A heterogeneous array of computational units and interconnection can be configured to execute a portion of the phase's datapath. A co-designed compiler constructs application path-trees to identify phases. The compiler slices path-trees and maps them onto DySE Resource (DySER) blocks [3]. The programmer continues to write programs in high-level languages such as C++, requiring no extra programmer burden.

My contributions to the research include the implementation of DySER in Verilog, verification, analysis, and contributions to the compiler. The current design is scalable, and a script allows automatic generation of any desired DySER block size. A simulator for DySER was created for inclusion in the GEMS framework for performance analysis and ongoing research. The co-designed LLVM compiler was extended to target this particular DySER design. A block containing 16 functional units was integrated into the OpenSPARC T1 pipeline [5] and successfully synthesized onto a Virtex-5 board as a prototype. A suite of micro-benchmarks demonstrates the supported functionality and identifies workloads that

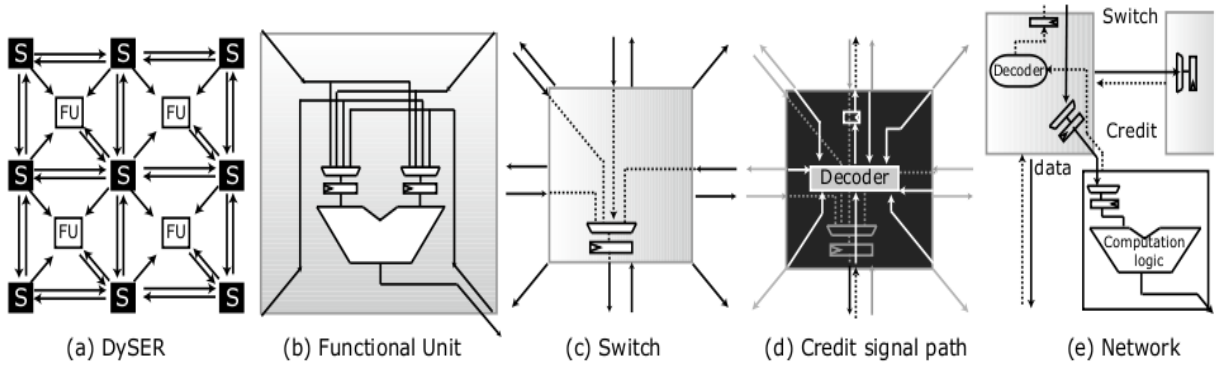

Figure 1: Major components of a Dynamically Synthesized Execution Resource.

perform well and workloads that perform poorly on a DySER-enhanced processor.

The remainder of this paper is organized as follows. Section 2 describes the DySER implementation and section 3 describes the integration with the OpenSPARC processor and FPGA synthesis. Section 4 discusses trade-offs in the design and section 5 describes the compiler. Section 6 discusses verification and section 7 contains performance analysis. Section 8 concludes.

2. DySER Implementation

This research demonstrates one possible instantiation of the proposed Dynamically Synthesized Execution model. This section provides an overview of the modules that make up a DySE Resource (DySER) block. The core of a DySER is made up with an array of functional units interconnected with switches (Figure 1a). An input and output bridge provide the interface between the processor and a DySER block.

2.1 Functional Unit

The functional unit provides the computation in a DySER block (Figure 1b). The mix of computation provided by the functional units can vary throughout a block. Based on configuration (configuration registers not shown in the diagram), the functional unit will take one or two operands from the four input directions. The right operand can optionally be a constant embedded during configuration. The result of the computation is always sent to the south-east switch.

To provide control flow in a DySER block, the functional units have the ability to optionally predicate the result. This is done by using a third input as the predicate flag, again using one of the four input directions. Predication is discussed more in Section 4.

2.2 Switch

The switch is the main module for moving data within a DySER block and to functional units (Figure 1c). A switch takes inputs from the four orthogonal "neighbor" switches, and from the north-west functional unit. It provides outputs to the four orthogonal switches and four neighboring functional units, with edge switches having correspondingly fewer inputs and outputs (note Figure 1a). Based on configuration (not shown), each of the eight outputs will either be off or configured to

Module	Area
SPARC ALU	1.000
Functional unit (with SPARC ALU)	1.944
Switch (no Phi-function)	2.281
Switch (Phi-function)	2.525

Table 1: Approximate synthesis areas for DySER components (normalized to the SPARC ALU).

forward data from one of the five input directions.

In the compiler literature, there is a notion of a Phi-function, which is related to control flow and diverging execution paths. Switch outputs support being configured as a Phi-function. This allows it to select between two values based on those values' predication.

2.3 Processor Interface

A DySER block must first be configured before it can be used. Each switch and functional unit requires configuration. A special instruction (*dyser_init* - see section 3) is used to send configuration bits to the block. A sequence of *dyser_init* instructions send configuration bits through a statically determined path through the existing switch network until all switches and functional units are configured. The number of *dyser_init* instructions required to configure a block is proportional to the number of switches and functional units.

Once configured, data is sent from the register file or memory to a DySER input port (corresponding to one of the edge switches). Data flows through the DySER block along the now configured paths. Switches and functional units use credit-based flow control to manage the flow of data in the network (Figure 1d,e). The results can be retrieved by reading from a DySER output port.

2.4 Module Synthesis

The modules were synthesized to compare approximate areas for the main components in a DySER block. Synthesis was performed using the Synopsis tool chain with a 55 nm standard cell library (Table 1). The area breakdown is normalized to the area of a SPARC ALU, and shows where effort should be focused to make the most effective improvements. Ideally, the majority of the DySER block area would be consumed by computational components with minimal routing. A DySER block is made up of approximately an equal number of functional units and switches. A switch's primary function is to route data between functional units, so its area should be minimized. From Table 1, a switch actually consumes the most area, making it an ideal target for improvements. A reasonable goal would be to get the "overhead" reduced below the area of the components providing computation.

3. Integration with OpenSPARC

This section provides an overview of modifications to the OpenSPARC processor pipeline [4] and FPGA synthesis. The OpenSPARC modifications were primarily accomplished by Chris Frericks and the FPGA synthesis done by Ryan Cofell. A DySER block

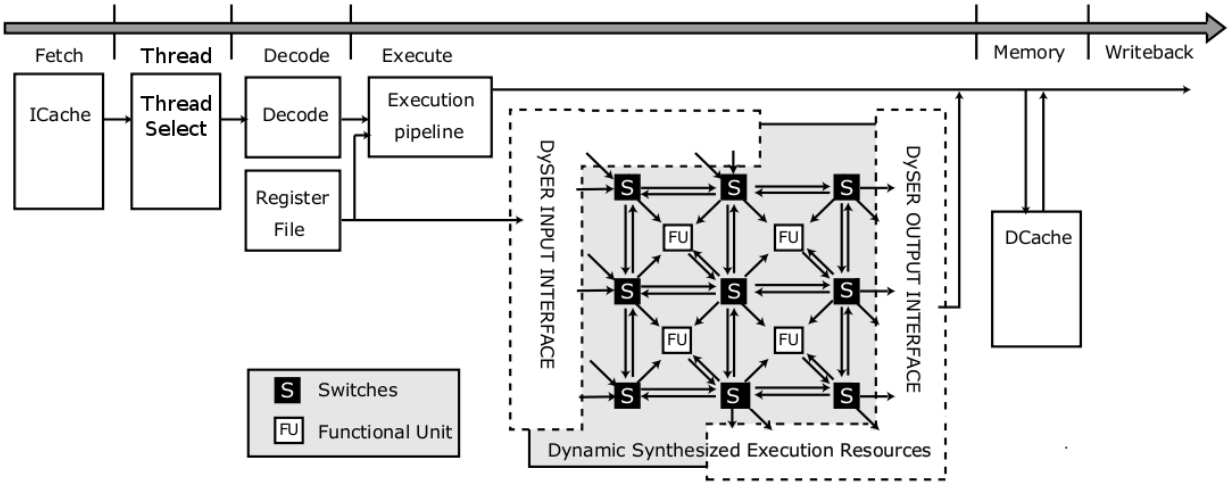

Figure 2: The OpenSPARC T1 pipeline with a DySER block integrated in the execute pipeline stage.

Instruction	Description
dyser_init [config data]	The DySER block is placed in config mode, and the config data is shifted into the block.
dyser_send RS1=>DI1 *dyser_send* RS1=>DI1, RS2=>DI2	Reads from the register file and sends the data to a DySER block. 1 or 2 source registers (RS1, RS2) are sent to the specified DySER input ports (DI1, DI2).
dyser_recv DO=>RD	Writes output data from DySER to the register file.
dyser_load [RS]=>DI	Reads from memory using the address in register RS and sends the result to DySER input port DI.
dyser_store DO=>[RS]	Writes data from DySER output port DO to memory using the address in register RS.
dyser_commit	Signals DySER to write all ready data back to general-purpose registers and/or memory. This facilitates out-of-order execution.

Table 2: Stylized SPARC instruction set extensions to support DySER operations.

was integrated into the OpenSPARC pipeline logically in the execution stage (Figure 2).

3.1 Instruction Set Extensions

The SPARC instruction set was extended with six instructions to facilitate processor and compiler interaction with a DySER block. The instructions allow the compiler to configure the DySER block, send data from memory or the register file to the DySER block, and send data from the DySER block to memory or the register file. The instructions are detailed in Table 2.

3.2 FPGA Prototype

FPGA prototyping of the DySER-enhanced processor was one of the original goals of the project. The prototype shows it's feasible to integrate our "accelerator" into a commercial processor. The integrated design and FPGA prototype allows us to run a full operating system and full binaries. A lite version of Ubuntu 7.10 is being used. The application binaries are created using the co-designed compiler (discussed further in Section 5). As a consequence, full benchmarks can be executed with larger data sets on the FPGA prototype than can be simulated.

We have successfully synthesized a DySER block containing 16 functional units arranged in a 4x4 tiled array onto a Virtex-5 board. The modified OpenSPARC processor operates at a 50 MHz clock frequency. The prototype is also being used to verify the GEMS DySER simulator for functional correctness and cycle accuracy. Once the cycle accuracy of the simulator is known for the smaller DySER blocks that fit on the FPGA, larger blocks or multiple blocks can be simulated with an empirical confidence.

4. Design Decisions and Trade-offs

This project involved a number of design decisions and trade-offs. The project involved three large concurrent tasks: DySER development, OpenSPARC modifications, and the co-designed compiler. All three portions informed design decisions among each other. This section discusses several of these decisions.

The co-designed compiler informed several changes to the DySER hardware implementation. Analysis from the compiler determined it's common to have constants across invocations of a computation slice. Under normal operation, all values must be sent to the DySER block using *dyser_send* instructions. For values that stay constant across DySER invocations, this represents a significant waste. The hardware was extended to support stored constants as an input to any functional unit. To support this, the constant values are embedded in the configuration bits. This makes configuration time longer (a one time occurrence), but reduces the number of *dyser_send* operations.

The first design iteration did not support control flow execution. Supporting simple control flow cases would allow more path-trees to be scheduled onto a DySER block, which is attractive. To do this, functional units were extended to have predicated results, and switches have embedded Phi-function support to choose between two predicated results. This provides a simple control flow mechanism.

Configuration management and interaction with the operating system were design concerns. One main concern was handling interrupts, system calls, or any other event that leave the DySER block in a partially configured state. The chosen design was to make configuration idempotent and restrict the operating system from using a DySER block. The configuration bits are embedded in the *dyser_init* instructions. This is a simple but wasteful design, since each 32-bit instruction embeds only 21 configuration bits. However, it was chosen because it naturally benefits from existing cache techniques, and page faults/cache misses do not cause any problems. If configuration is interrupted, resuming at the paused instruction (which most processors do by default) or restarting from the beginning of the configuration instructions will both result in correct execution.

The OpenSPARC modifications were done to be minimally disruptive. An implementation dependent instruction was used for four of the DySER instructions. The DySER block is logically placed parallel to the SPARC ALU in the execute phase. This choice enabled the instructions to act like any other R-type instruction except they use the DySER block instead of the normal execution pipeline. The two memory instructions used the existing SPARC load and

store instructions by mapping DySER as an alternative memory space (the normal memory hierarchy is alternate space 0). This choice allowed most of the existing decode and control signals to be re-used.

The main objective with FPGA synthesis was to create a working prototype. It showed the feasibility of integrating a DySER block into a commercial processor. We were limited by area to integrating a 16 functional unit array onto the Virtex-5 board, which was smaller than desired. However, as a proof-of-concept and to validate the simulator, this is sufficient. It allowed all functionality to be tested and reasonable performance on several benchmarks. In future work, the project will investigate cross synthesis on multiple FPGAs to allow mapping of larger (e.g. 8x8) or multiple DySER blocks.

5. DySER Software Compiler

A critical component of the DySE model is the accompanying compiler. This section provides a brief overview of the compiler. The DySER compiler is an extension of LLVM performed mostly by Venkatraman Govindaraju and Tony Nowatzki. My contributions to the compiler work were to provide the details of the DySER block to the compiler, generate the configuration information for a scheduled computation slice, and create a DySER simulator to verify the compiler's code.

The compiler creates a program dependency graph (PDG), analyzes a sample execution of the program, slices the PDG, and schedules slices onto a DySER block. The PDG determines all possible execution paths in a program (Figure 3a). The code is then instrumented and executed with representative inputs to analyze the frequency that different execution paths are taken. The compiler uses heuristics to find suitable candidate sections of the PDG to map onto a DySER block (Figure 3b). Each suitable PDG slice is scheduled onto the available DySER block's resources (Figure 3c). Finally, the supporting code is generated to configure, send data to, and receive data from the DySER block.

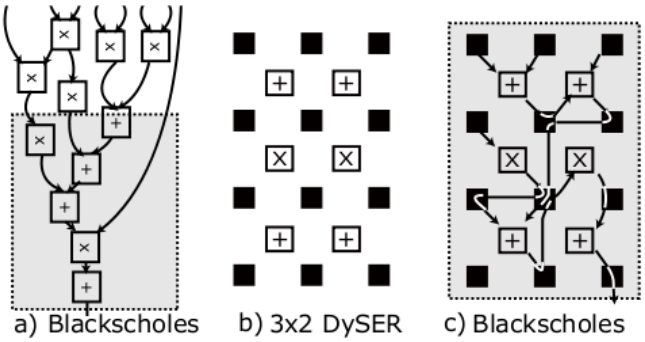

Figure 3: Scheduling a phase of Black-Scholes onto a DySER block with a 3x2 array of functional units.

5.1 DySER Configuration

A model of a DySER block is provided to the compiler to schedule slices of the PDG onto. Currently, the model needs to exactly match what is provided in the hardware on which the program will be executed. Once a PDG slice is scheduled onto the block, there is a one to one mapping to what the configuration bits need to be. An ordered list of *dyser_init* instructions are generated, which is the code to configure the DySER block to perform that computation slice.

5.2 DySER Simulator

A simulator for DySER was created to allow verification of the compiler generated code. The simulator in written in C++ to closely match

how execution happens in hardware. The switches, functional units, input ports, output ports, and flow of data in DySER are all modeled. Instructions (*dyser_init*, *dyser_send*, *dyser_recv*, etc.) can be executed on the simulator in sequence, as they would in hardware. The results can be compared to a reference output, such as the results from the program execution used for compiler path-tree analysis.

In future work, the simulator's cycle accuracy will be improved so that performance heuristics can be incorporated into the compiler to help inform when scheduling a portion of code will be beneficial. The simulator can also be extended with a power model or other metrics.

6. Verification

Verification was performed across all parts of the project. A suite of verilog testbenches were used to verify the DySER implementation and the C++ simulator. A provided regression suite was used to verify the modified OpenSPARC processor maintained ISA compatibility. The DySER simulator was used to verify the compiler. A suite of self-checking micro-benchmarks were developed to run on the FPGA prototype to verify correct functionality of the DySER block.

6.1 DySER

Verilog testbenches were created to validate each individual DySER component, and were combined into a regression suite. The top level testbench tested the interface between the processor and DySER. The C++ DySER simulator was designed to look very similar to the DySER verilog testbench. This allowed the verilog simulation results to be used to verify the C++ simulator.

6.2 OpenSPLySER

OpenSPARC is released with a regression test suite which verifies compatibility with the SPARC ISA. As the pipeline was modified, the regressions ensured the processor still performed normal SPARC instructions correctly, which significantly eased verification. We added a several tests to the suite to ensure functionally correct DySER instruction operation when a DySER block is present.

6.3 Compiler

The compiler output can be validated using the DySER simulator. The compiler includes a step to execute the original source code with a representative input set. The results of this run can be compared with the simulation of the final source code to verify valid generated code.

6.4 FPGA Prototype

Once all the components are verified in simulation, the synthesized FPGA prototype needs to be verified. To do this, a suite of self-checking micro-benchmarks were compiled to binaries targeting Ubuntu running on the modified OpenSPARC processor. This regression suite can verify the DySER block is functionally correct and meeting timing.

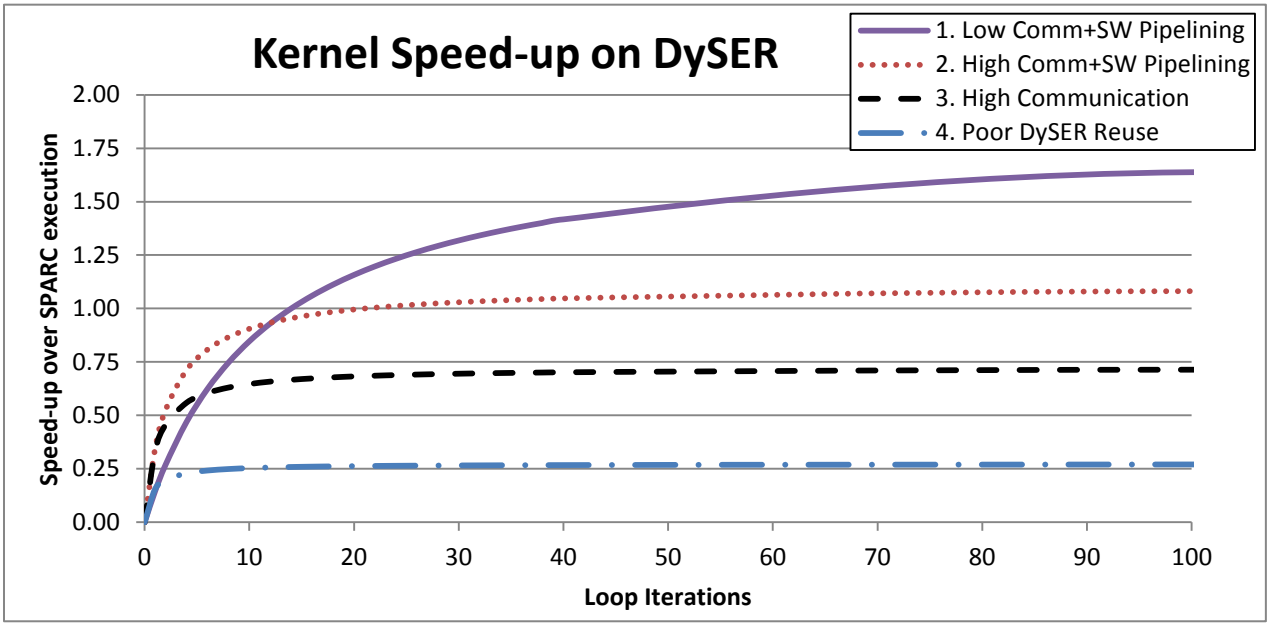

Figure 4: Speed-up for four sample micro-kernels on a 2x2 DySER block versus executing the same computation on the baseline SPARC processor.

7. Performance Analysis

The purported benefits of the DySE model is improved performance and/or increased energy efficiency. Some initial performance analysis has been done on the modified OpenSPARC processor in both simulation and on the FPGA prototype. A GEMS simulation framework is being developed to simulate a DySER block with cycle accuracy close to the hardware implementation. This enables a faster simulation than gate-level simulation, and allows larger DySER blocks than can fit on an FPGA while maintaining reasonable cycle accuracy.

7.1 Micro-benchmark Performance

A number of micro-benchmarks have been developed to demonstrate general application characteristics that perform well and that perform poorly on a DySER-enhanced processor (Figure 4). The graph shows the kernel speed-up on a 2x2 DySER block versus executing the same computation on the SPARC processor.

The sample kernels demonstrate a few key points about DySER execution. A DySER block must be configured before use, which is shown by the ramp-up time to achieve speed-up in the graph. The configuration time is quickly amortized as the iteration count increases. Kernel 1 is characterized by low communication between the processor and DySER (few *dyser_send* and *dyser_recv* instructions) and has been properly software pipelined. It is an ideal kernel for this 2x2 DySER block and achieves an asymptotic speed-up of 1.8. Kernels 2 and 3 contain the same computation which is characterized by high communication, except kernel 2 has been properly software pipelined and thus achieves higher speed-up. Kernel 4 reconfigures the DySER block often, which demonstrates a poor reuse of the DySER block.

7.2 Ideal Kernel Characteristics

The following are general characteristics of that a micro-benchmark can contain and their associated performance effects when executing on a DySER block:

Perform Well:

1. Application executes in phases.
2. Computationally intensive code regions.
3. Frequent execution of code regions.
4. High reuse of variables in computations.

Perform Poorly:

1. Poorly identifiable phases.
2. Minimal re-execution of code regions.
3. Sporadic execution of code regions.
4. Little or no variable reuse in computations.

Applications that execute in phases or have computationally intensive regions that are invoked frequently (not necessarily back to back) tend to get the most benefit. This is because the DySER block will have high utilization of functional units, and the configuration time will be amortized over the numerous invocations. A computational region that contains a high degree of variable reuse or constant values will have a strongly positive effect on performance because the variable will be sent to the DySER block once and used in numerous functional units.

Applications with no identifiable execution phases, or that execution several code regions sporadically rather than one region repeatedly will suffer in performance. Executing a code region requires the DySER block to be configured for that region. If that region is not executed enough times before the next code region needs to be executed, the configuration time for each region will eliminate most performance gains. Similarly, sending data to and reading data from a DySER block requires *dyser_send* and *dyser_recv* instructions. A computation slice with little variable reuse will require more send and receives, mitigating some of the performance gains.

The theoretical maximum speed-up for this DySER implementation is given by the number of functional units divided by the time to send and receive all the inputs and outputs. This can be understood by the following. A configured DySER block looks and acts like a long latency function with multiple inputs and outputs. A properly scheduled DySER block can be utilizing all functional units on each cycle. Each invocation of the DySER block requires at least one *dyser_send* and at least one *dyser_recv* instruction. Since the OpenSPARC T1 processor is a single-issue, in-order processor, this requires at least 2 cycles so the theoretical maximum speed-up on it is the number of functional units divided by two.

8. Conclusion

The Dynamically Synthesized Execution (DySE) model allows dynamic specialization in general purpose processors to achieve performance improvement or energy efficiency. The co-designed compiler analyzes and schedules code onto the DySE Resource (DySER) block, letting the programmer continue to write in high-level languages without extra burden.

The primary focus of this research was to create a proof of concept implementation of the proposed DySE model. The design described in this paper shows one possible instantiation of the DySE model. A scalable DySE Resource

block was integrated into the pipeline of a simple, in-order OpenSPARC T1 processor. The design was successfully synthesized onto a Virtex-5 FPGA as a prototype. The board runs a lite Ubuntu 7.10 operating system and can execute application binaries created using the co-designed compiler.

Future work and directions exist for this design. Ideally, the majority of the area would be consumed by computational blocks. Anlysis shows there is still significant area overheads due to data routing and configuration flexibility. The performance analysis shows the types of workloads and behaviors that result in good and poor performance on a DySER-enhanced processor. This analysis can be incorporated in the compiler to improve the heuristics. This DySER design can be incorporated into both superscalar and out-of-order processors.

Acknowledgements

Many thanks to the Vertical Research Group including my advisor, Karthikeyan Sankaralingam; Chris Frericks and Ryan Cofell for the OpenSPARC and FPGA work; Tony Nowatzki and Venkatraman Govindaraju for compiler work; and Chen-han Ho for the initial DySER design.

References

[1] K. Jeong, A. Kahng, A power-constrained MPU roadmap for the International Technology Roadmap for Semiconductors (ITRS), *SoC Design Conference (ISOCC), 2009 International*, pp. 49-52, Nov 2009.

[2] V. Govindaraju, C. Ho, and K. Sankaralingam. Dynamically Specialized Datapaths for Energy Efficient Computing. In *Proceedings of 17th International Conference on High Performance Computer Architecture*, February 2011.

[3] C. Ho. Dynamically Synthesized Execution Resources (DySER) Design Specification. Technical Report.

[4] C. Frericks, J. Benson, and R. Cofell. OpenSPLySER: The Integrated OpenSPARC and DySER Design. University of Wisconsin – Madison Computer Science Technical Report.

[5] OpenSPARC T1 Microarchitecture Specification. Sun Microsystems, Inc. 2006.

[6] D. Patterson and J. Hennessy. Computer Organization and Design. Morgan Kaufman Publisher 4th Edition, 2008.