Dynamically Specialized Datapaths for Energy Efficient Computing

Venkatraman Govindaraju   Chen-Han Ho   Karthikeyan Sankaralingam
Vertical Research Group
University of Wisconsin-Madison
{venkatra,chen-han,karu}@cs.wisc.edu

Abstract

Due to limits in technology scaling, energy efficiency of logic devices is decreasing in successive generations. To provide continued performance improvements without increasing power, regardless of the sequential or parallel nature of the application, microarchitectural energy efficiency must improve. We propose Dynamically Specialized Datapaths to improve the energy efficiency of general purpose programmable processors. The key insights of this work are the following. First, applications execute in phases and these phases can be determined by creating a path-tree of basic-blocks rooted at the inner-most loop. Second, specialized datapaths corresponding to these path-trees, which we refer to as DySER blocks, can be constructed by interconnecting a set of heterogeneous computation units with a circuit-switched network. These blocks can be easily integrated with a processor pipeline.

A synthesized RTL implementation using an industry 55nm technology library shows a 64-functional-unit DySER block occupies approximately the same area as a 64 KB single-ported SRAM and can execute at 2 GHz. We extend the GCC compiler to identify path-trees and code-mapping to DySER and evaluate the PARSEC, SPEC and Parboil benchmarks suites. Our results show that in most cases two DySER blocks can achieve the same performance (within 5%) as having a specialized hardware module for each path-tree. A 64-FU DySER block can cover 12% to 100% of the dynamically executed instruction stream. When integrated with a dual-issue out-of-order processor, two DySER blocks provide geometric mean speedup of 2.1X (1.15X to 10X), and geometric mean energy reduction of 40% (up to 70%), and 60% energy reduction if no performance improvement is required.

1 Introduction

Materials and device-driven technology challenges are ushering an era of non-classical scaling. While the number of devices is expected to double every generation, the power efficiency of devices is growing slowly. The main reason behind this trend is that classical voltage scaling has effectively ended and capacitance of transistors is reducing slowly from one generation to another [1]. While the number of transistors increases sixteen-fold from now through 2020, capacitance only reduces by 3X. Using a conservative model of 5% reduction in voltage per generation and no reduction in gate capacitance in future technology nodes, power increases by *3.4X* in 10 years. Architectural techniques are required to improve performance while being energy efficient as highlighted by others as well [18, 16].

While there is consensus that hardware specialization can energy-efficiently improve performance, the programmability and implementation challenges are daunting. This paper explores hardware specialization using a co-designed hardware-compiler approach that avoids disruptive hardware or software changes. The idea is to dynamically specialize hardware to match application phases. We call the execution model, **Dy**namically **S**pecialized **E**xecution (DySE). The compiler slices applications into phases and maps them to the hardware substrate – Dynamically Specialized Execution Resource (DySER) blocks.

A DySER block is integrated like a functional unit into a processor's pipeline. It is a *heterogeneous array* of computation units interconnected with a *circuit-switched* mesh network, without any storage or other overhead resources. The key insight is to leave only the computation units on the commonly executed hardware path, eliminating per-instruction overheads such as fetch, decode, and register access. Circuit-switching in the network provides energy efficiency. By providing support for network flow-control, we can pipeline multiple DySER invocations and tolerate loads/stores cache misses. As part of the compiler, we develop a novel path-profiling technique that develops trees of paths ranging hundreds of instructions that capture the most commonly executed code. These trees are then mapped to DySER blocks creating specialized datapaths at run-time.

Thus, DySER integrates a very general purpose and flexible accelerator into a processor pipeline and with its co-designed compiler, the same hardware can target "any" application and diverse domains through dynamic specialization. This specialization can help improve energy efficiency. Judiciously exploiting technology and architecture capability, DySER overcomes the complexity, domain-specialization, program-scope and language restrictions, and scalability restrictions of previous efforts.

We have implemented the DySER module in Verilog and synthesized it on a 55nm technology library and we have built a compiler-pass in GCC to identify path-trees and create mappings. Our results show: i) A 64-functional-unit DySER datapath occupies the same area as a 64KB single-ported SRAM and can cover 12% to 100% of applications' dynamically executed instruction stream. ii) In most cases two DySER blocks can achieve the same performance (within 5%) as having a specialized block for each path-tree. iii) When coupled to a single-issue processor, two DySER blocks provide geometric mean speedup of 2.1X (1.1X to 10X), and geometric mean energy reduction of 40% (up to 70%) and geometric-mean energy-delay product reduction of 2.7X. iv) With a dual-issue and 4-issue out-of-order (OOO) machine we see similar performance improvements.

The remainder of this paper is organized as follows.

Section 2 provides rationale and overview. Section 3 describes the architecture, Section 4 describes the compiler's role and Section 5 presents results. Section 6 discusses related work and Section 7 concludes.

2 A Case for Dynamically Specialized Datapaths

Figure 1 shows a spectrum of hardware specialization increasing in granularity. On the left is implicit microarchitecture specialization. An example is macro-op fusion where the microarchitecture fuses sequences of instructions to amortize per-instruction overheads like decoding, register reads, renaming etc.. In the middle is instruction-set specialization which is visible to the compiler. Examples include encryption accelerators in Niagara [13], media extensions like SSE [34], and GPU instruction sets. Their key problems are that they do not generalize outside the specific domain and can be hard to program.

Dynamically Specialized Execution: DySE uses insights from domain-driven accelerators to *dynamically* specialize datapaths to capture application phases. Our motivation came from building application-specific accelerators for a few PARSEC benchmarks [4], which we sketch in Figure 2a-b. This exercise and VLSI area-analysis led to the following insights: i) A heterogeneous array of functional units can cover many applications. Figure 2c shows such an array and Figure 2d,e show how different applications can map to same array. ii) Datapath wires must be statically-routed to minimize overheads. iii) VLSI area densities allow tens to hundreds of functional units.

In the DySE execution model, the application is abstracted as a sequence of ultra-wide "instructions" each representing an application phase. These phases communicate with each other through memory. The hardware for executing these instructions are dynamically specialized execution resources (DySER). The high-level design is shown in the right extreme of Figure 1. The wide instructions encode the physical routes on the substrate and inputs are injected from the processor's datapath. The model hinges on the assumption that only a few such "wide instructions" are *active* during a phase of an application. Thus, setting up the static routes once amortizes the execution of the DySER unit over many invocations. The potential benefits are:

- **Energy efficiency:** Datapath specialization removes overheads in programmable processors.
- **Area efficiency and Programmability:** Dynamically specializing these datapaths instead of designing accelerators for each phase provides area efficiency and programmability.
- **Design complexity:** The implementation lends itself to easy integration with the processor pipeline.
- **Flexible execution model:** The execution-model of ultra-wide instructions with dynamically specialized

hardware implementation unifies different specialization techniques like SIMD-execution, instruction-specialization, and loop-accelerators with potentially little efficiency loss.

We discuss below the key challenges in realizing the potential benefits of the approach.

Can phases be converted into ultra-wide instructions? First, while it is accepted that applications execute in phases and loops, for the DySE approach to work, we must be able to determine these phases. Second, applications must re-execute such an "instruction" many times to amortize the cost of dynamically synthesizing a datapath. We develop a novel path-tree based representation of programs (Section 4) and show that applications spend hundreds to thousands of cycles in two to three path-trees. Thus, path-trees serve as a good candidate for ultra-wide instructions. Table 2 shows quantitative data for a diverse application suite.

Isn't hardware and functional-unit mix highly application dependent? Since the path-trees are themselves large, a law of large numbers phenomenon showed that instruction mix across applications was similar. We observed a distinct floating-point and integer based distribution, but little divergence with these domains. Thus, it is possible to build a common-case array of heterogeneous units.

Aren't data-flow routes data-dependent? Data-flow becomes data-dependent if buffers are used to share a functional unit between many operations. If instead, we take the radical approach of providing a single computational unit for each primitive operation in our DySE instruction, routing values is no longer data-dependent thus providing an opportunity for circuit-switched static routing. VLSI area constraints support a case for such a decision. For example, an on-chip router's area with its buffering and control-logic can exceed the size of a 32-bit adder and multiported register-files can be much larger than a 32-bit adder. Thus, if the execution model supports it, a circuit-switched network is more efficient.

What about load-stores and control flow? Related work on such specialization has found irregular memory accesses to be a problem. They sidestep the problem by restricting their domains to where memory accesses are regular [8], or restrict the program scope [8, 39, 17], or by enforcing specialized languages [15, 14]. The resulting architectures are unscalable and/or highly domain-specialized.

We exploit a simple insight, which some may feel is counter-intuitive – use a general purpose processor. Driven by sophisticated advances in memory disambiguation [35], prefetching [19], and streaming [24], general-purpose processors with short physical paths to hardware managed caches provide effective low-latency access to memory.
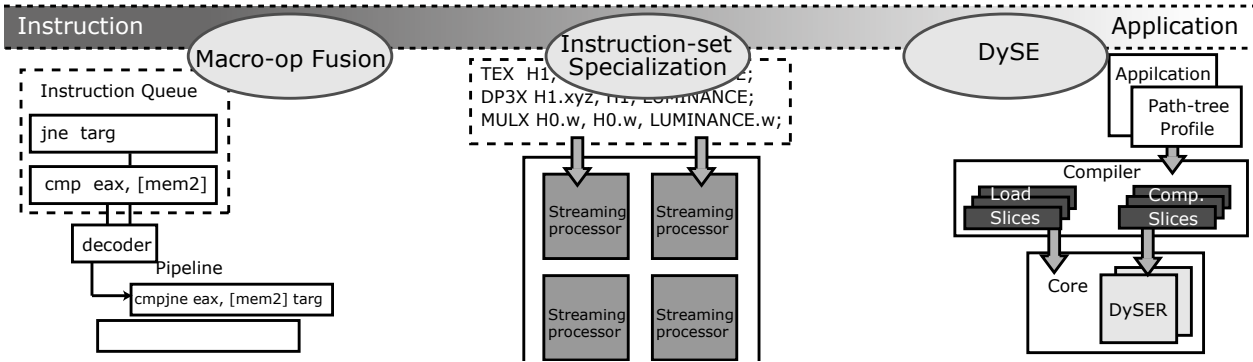
Figure 1. Specialization Spectrum

a) Blackscholes   b) Streamcluster   c) 3x2 DySER   d) Blackscholes   e) Streamcluster
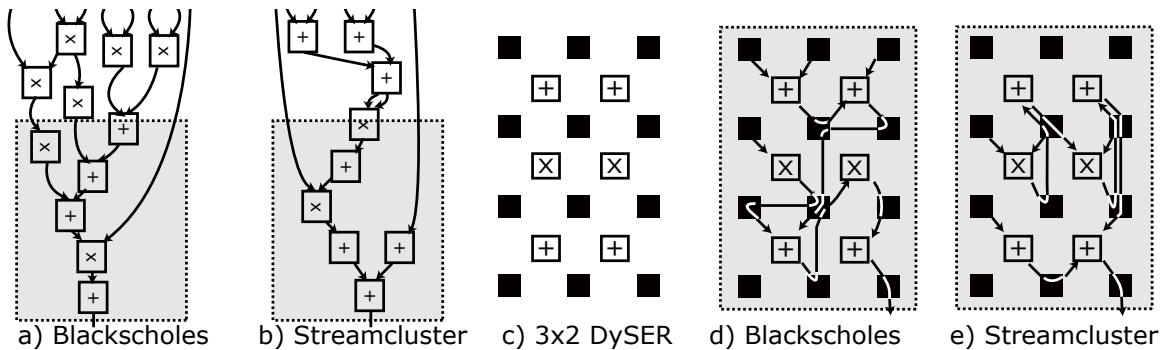
Figure 2. Application-specific datapaths and generalization

Quantitatively, the PARSEC benchmarks typically have L1 data-cache miss-rates less than 2% [3] and the SPECCPU benchmarks typically have 30 data-cache misses per thousand instructions [20]. *Hence, our solution is to utilize general-purpose processors as a load/store engine to feed a specialized datapath.* This provides sufficient support to explore practical computation specialization without disrupting the hardware/software stack.

In the DySE execution model, a program is explicitly partitioned by the compiler into a load back-slice which includes all computation of memory addresses and a computation slice which consumes values, performs computation, and sends store values/addresses back to the processor. This insight provides the generality and freedom to investigate large application phases for specialization and a hardware block simple enough to integrate with processors like a functional unit. A place-holder instruction in the load back-slice ensures load-store ordering and allows the processor's memory disambiguation optimizations to proceed unhindered. Control-flow is implemented with a hardware selection node exposed to the compiler.

3 Architecture

Execution Model and Overview: Dynamically specialized datapaths are meant to be integrated as functional units into a pipelined processor as shown in Figure 3.

The compiler and processor view the DySER block as a block of computational units that consume inputs (memory words and named registers) and produce outputs (memory word/address pairs and register value/name pairs). Figure 3 show a logical FIFO-based processor interface.

Execution with DySER datapaths proceeds as follows. When the program reaches a region that can be executed on the DySER array, the hardware configures the DySER block and the main processor starts injecting register values and memory values into the FIFOs. After configuration, the DySER block pops values from the FIFOs and execution proceeds in data-flow fashion with values routed between functional units through a *circuit-switched* network. Outputs are delivered to the output interface and written back to the processor registers or memory. While this approach adds configuration overheads, it works because: i) the processor executes multiple instances of an "instruction" before a new configuration is required and ii) large program regions can be converted into "wide instructions" (quantitative measurements in Section 5).

To create these "instructions" that represent large program regions, our compiler first creates path-trees (see Section 4) and for each path-tree it creates a separate load back-slice and computation slice. The load back-slice executes on the processor and the computation slice becomes

(a) DySER (b) Functional Unit (c) Switch (d) Credit signal path (e) Network
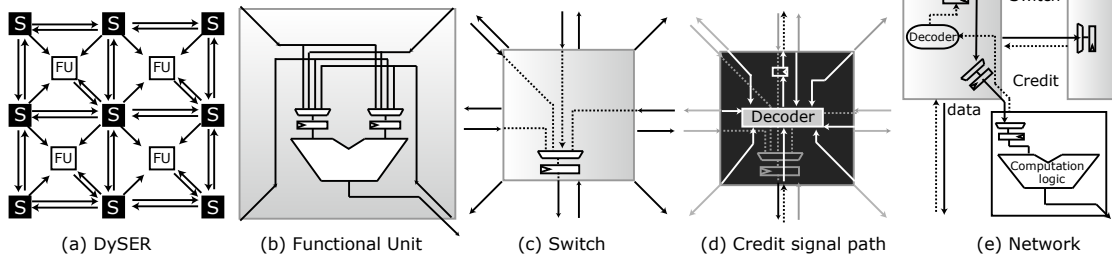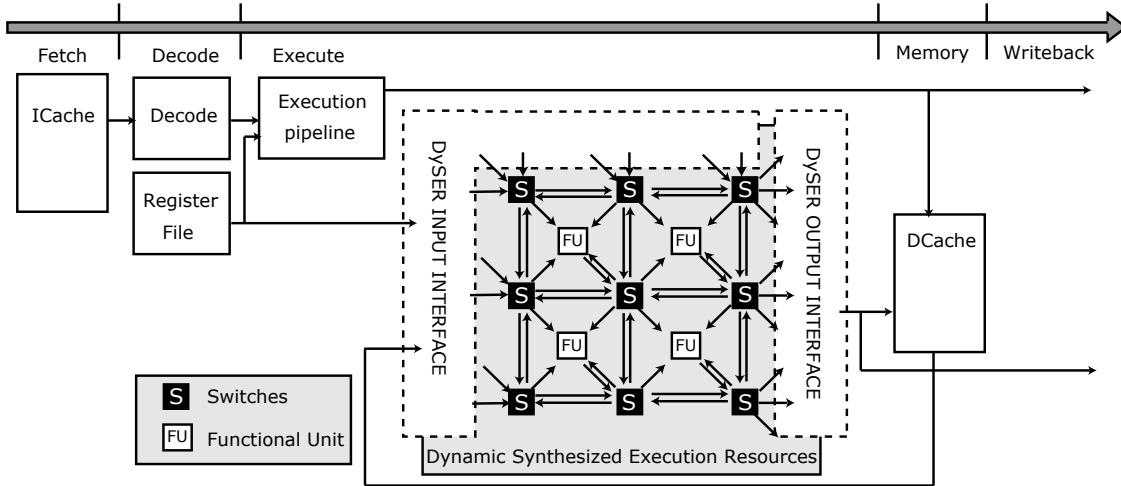
Figure 3. Processor Pipeline with DySER Datapath and DySER Elements

a stateless piece of code ideally suited for hardware specialization. Table 1 shows a code snippet, its stylized assembly code, load back-slice and computation slice. To interface with DySER, we introduce these ISA extensions: i) `dyser_init`: an instruction to configure the DySER unit, ii) `dyser_send`: an instruction to send register values to the DySER unit, iii) `dyser_load`: an instruction that reads from the memory (caches) and sends the value into the DySER unit, iv) `dyser_store`: a place-holder instruction used by the main processor for tracking load-store ordering and v) `dyser_commit`: an instruction to commit output values. We now describe the DySER array, network, processor interface and pipeline integration.

3.1 DySER Block

The DySER array consists of *functional units* (FU) and *switches*. The functional units form the basic computation fabric as shown in Figure 3a. Each FU is connected to four neighboring switches from where it gets input values and injects outputs. Each FU also includes a configuration register that specifies which function to perform. Figure 3b shows the details of one FU. For example, an integer-ALU functional unit can perform addition, subtraction, and a few logical operations. Each FU also includes one data register and one status register for each input switch. The status registers indicate whether values in the data registers are valid or not. The data registers match the word-size of the

machine. The simplest DySER implementation is an array with homogeneous functional units, with each unit capable of primitive operations like addition, multiplication, and logic operations. However, this would take too much area. Instead, we use a heterogeneous array determined by instruction mix analysis.

The switches in the DySER array allows datapaths to be dynamically specialized (black squares in Figure 3a). They form a *circuit-switched network* that creates explicit hardware paths from inputs to the functional units, between functional units, and from functional units to outputs. Figure 3c shows the basic switch with the dotted lines representing the possible connections of one port to all possible output ports. *This forms the crux of DySER's capability to dynamically specialize computation units.* Along with data and status registers, each switch includes a configuration register which specifies the input to output port mappings. Switches in DySER have 8 outputs to 8 directions, 4 inputs from neighbor switches, and 1 input from functional units.

The basic execution inside a DySER block is data-flow driven by values arriving at a functional unit. When the valid bits for both left and right operands are set, the functional unit *consumes* this input, and a fixed number of cycles later produces output writing into the data and status register of the switch. Figure 5d shows the computation slice mapped on a DySER array.

Table 1. DySER code : load-slice and computation-slice

(a) C-code

```
void
quantum_toffoli(int cntrl1,
                int cntrl2,
                int trgt,
                quantum_reg
                      *reg)
{
 int i, size = reg->size;
 for (i = 0; i < size; ++i)
  if (reg->node[i].state
      & (1<<cntrl1)) {
    if (reg->node[i].state
        & (1<<cntrl2)) {
      reg->node[i].state ^=
          (1<<trgt);
    }
  }
}
```

(b) Assembly

```
entry:
  LD    reg->size => R1
  CMP   R1, 0
  BLE   exit
bb1:
  MOV   cntrl1  => R2
  MOV   cntrl2  => R3
  MOV   1 => R4
  SLL   R4, trgt => R4
  LD    reg->node  => R5
bb2:
  LD    [R5+offset(state)] => R6
  SRL   R6, R2  => R7
  ANDCC R7, 1   => R0
  BE    bb5
bb3:
  SRL   R6, R3  => R7
  ANDCC R7, 1   => R0
  BE    bb5
bb4:
  XOR   R6, R4, R7
  ST    R7, [R5+offset(state)]
bb5:
  ADD   R5, sizeof(node), R5
  ADDCC R1, -1, R1
  BNE   bb2
exit:
```

(c) Load-slice

```
entry:
  LD    reg->size => R1
  CMP   R1, 0
  BLE   exit
bb1:
  MOV   control1  => R2
  MOV   control2  => R3
  MOV   1 => R4
  SLL   R4, target => R4
  LD    reg->node => R5
  DYSER_INIT [COMPSLICE]
  ; Send inputs to DySER
  DYSER_SEND R2 => DI1
  DYSER_SEND R3 => DI2
  DYSER_SEND R4 => DI3
bb2:
  DYSER_LOAD [R5+offset(state)] => DM0
  DYSER_STORE:DO2 DO1, [R5+offset(state)]
  DYSER_COMMIT
  ADD   R5, sizeof(node), R5
  ADDCC R1, -1, R1
  BNE   bb2
exit:
```

(d) Computation-slice

```
COMPSLICE:
  SRL   DM0, DI1 => T1
  ANDCC T1, 1    => P1
  SRL:P1 DM0, DI2 => T2
  ANDCC T2, 1    => P2
  XOR:P2 DM0, DI3 => DO1
  AND   P1, P2   => DO2
```
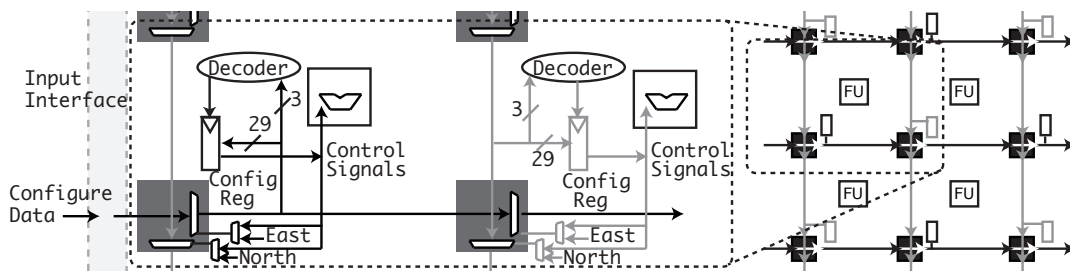

Figure 4. Configuration path: switch and functional unit's configuration registers combined.

A DySER array is configured by writing into configuration registers at each functional unit and switch. We use a novel way of reusing the data network to also transmit configuration information as shown in Figure 4. Every switch includes a small 3-bit decoder and a path from the switch's inputs to its configuration register. The data in a network message is interpreted as a 3-bit target and 29-bit payload data when in configuration mode. The switch uses the decoder to check if the message is meant for the current node (by examining the target field) and if so the value is written into the configuration registers. In addition, all configuration messages are forwarded to the next switch. To set up multiple configuration "streams", all switches forward the input from east port to west port and input from north port to south port. With this design, DySER blocks are configured using the datapath wires without any dedicated configuration wires. Compared to repeatedly fetching and executing, DySER blocks are configured once and re-used many times.

Multiple DySER Blocks and Pipelining: The execution model allows multiple DySER blocks where each block is configured differently. With multiple DySER blocks, we can predict the next block and configure it before its inputs are produced by the processor. The large granularity allows easy predictability and we observed more than 99% prediction accuracy with a 1K-bit two-level history table.

Like pipelining long-latency functional units, DySER datapaths can be pipelined with multiple invocations executing simultaneously. The following execution semantics of the DySER instruction extensions allow pipelining. dyser_init which configures a new phase, waits and stalls until all prior invocations are complete. dyser_send, dyser_load, and dyser_store are all either data-dependent on values or resources in the processor. dyser_commit starts a new invocation and allow the pipelining of multiple invocations. Since the DySER block receives inputs asynchronously at arbitrary times from the FIFO interfaces, some flow-control is required in the network to prevent values from a new invocation clobbering values from a previous invocation. We implement a credit-based flow-control simplified for our statically-switched network, by adding one backward signal called the credit signal (Figure 3d,e). Physically, the credit-signal is routed in the opposite direction of the data signal. Any stage (FU or switch) needs one credit to send data, and after sending the data it sends a credit signal to its predecessor. If a stage is processing or delayed waiting for data, the valid bit is cleared and credit is not passed to the previous stage.

3.2 Processor Interface

All the inputs to a DySER block are fed through a logical FIFO, which delivers register inputs and memory val-

ues. Each entry specifies a switch and port which effectively decide where the value will be delivered in the array, because DySER uses circuit-switched routing. Outputs follow a similar procedure. Each port in the output switches corresponds to one possible DySER output. Since for each output port, the DySER produces outputs in order, no FIFOs are required on the output side. When values arrive at the output ports, an output interface writes them to the register file or memory. A detailed description is in [12].

Integration with pipeline: DySER can be easily integrated into conventional in-order and out-of-order pipelines and with architectures like Intel Larrabee [31] or Copernicus [11] as an accelerator. Integration with an in-order pipeline is simple and the DySER block interfaces with the instruction fetch stage for obtaining the configuration bits, the register file stage and the memory stage of the pipeline. A state machine must be added to the instruction cache to read configurations bits for a block and send them to the input interface. DySER integration with an OOO-pipeline requires more careful design. The processor views DySER as a functional unit but the input ports are exposed to the issue logic to ensure two `dyser_send`'s to a port are not executed out-of-order. Since loads can cause cache misses, when a `dyser_load` executes in the processor, the corresponding input port is marked busy in the input buffers. When the data arrives from the cache, the input port is marked ready, which prevents a subsequent `dyser_load`'s value from entering the DySER block earlier.

Squashes, Page Faults, Context-switches, and Debugging: When a branch is mispredicted in the load back-slice, the values computed in DySER must be squashed. This is implemented by first marking the status of all input buffer entries to be invalid-but-ready. This ensures that all inputs are available for DySER to compute and produce outputs. Second, we abort the outputs by changing commit entries in the commit status queue to "abort" and hence the outputs for the misspeculated invocation will be ignored. Finally, we restart the DySER using the values in the input buffer and injecting new, correct values.

Since page faults can only be raised by the load back-slice, almost no changes are required to the processors' existing mechanisms. The processor services the page-fault and resumes execution from the memory instruction that caused the fault. We assume that the OS routine to handle page-faults does not use the DySER block. To handle context-switches, the processor waits until all DySER invocations are complete before allowing the operating system to swap in a new process. These techniques describe the mechanisms but detailed exploration and implementation of such code in a real operating system is future work.

From a program developer perspective of debugging, our current software environment generates processor code for the computation slice as well. We anticipate the programmer will debug this version using conventional debuggers as part of the software development process and DySER execution is turned on as a performance optimization. Debugging the DySER instructions, single-stepping DySER code (which can help in further performance optimizations of DySER code) is likely necessary, but only rarely, and mechanisms to support this are future work.

3.3 Implementation and Physical design

We have implemented and verified the DySER array in Verilog and synthesized using an 55nm standard cell library with Synopsys Design Compiler. Our results show DySER blocks are simple and area/energy efficient.

For the floating-point and integer execution units, we synthesized the implementations available from OpenSparc T1 for a 2 GHz clock frequency. The input and output interfaces and switches were implemented with our own Verilog. The Table below shows the area of the different units.

FU	**area**(μm^2)	**FU**	**area**(μm^2)
int alu	2481	fp add/sub	14533
multiply	16401	fp mul	24297
div	5278	fp divide	5924

We used application profiling to arrive at an instruction mix for the functional units (details in Section 5). Based on this analysis, 60% integer ALU, 10% integer multiply, 30% floating point units provided the best mix. A 64-functional-unit DySER block with the aforementioned mix, has an area of 0.92 mm^2. It occupies the same area as a 64KB SRAM estimated from CACTI [38]. Comparing to area of other structures in modern processors, one DySER block is less area than the Atom L1 data cache (including its data and tag array): area estimated as 1.00 mm^2 from CACTI and 1.32 mm^2 from the die photo scaled to 55nm. Synopsys Power Compiler estimates DySER's power as 1.9 Watts at 2.0 GHz based on its default activity factors assumptions for datapath. The simple design and quantitative results show that the DySER architecture is practical.

4 The Compiler's Role

To support the execution model, we developed a program representation called a path-tree that represents application phases. The compiler slices path-trees to make them amenable for DySER datapaths. Detailed compiler design, formalism of the path-tree and optimizations are future work. This section describes our current framework.

Path-Trees: A *path* is an acyclic execution trace of basic blocks. A *path-tree* is a rooted tree with basic blocks as its vertices and represents a collection of paths starting at the root of the path-tree. Based on profiling for each path in a path-tree, the most frequently executed paths are mapped
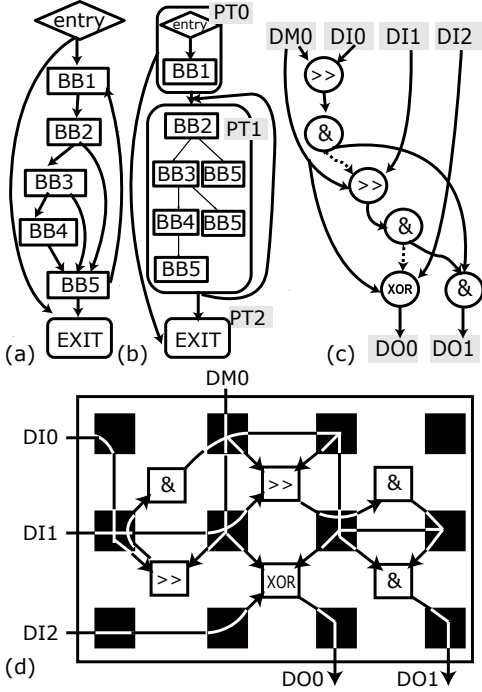
Figure 5. Compiling for DySER

to DySER . A control-flow graph and path-tree is shown in Figure 5a,b.

Slicing the Path-Tree A path-tree itself cannot be mapped to a DySER block because it includes memory instructions, while DySER provides only computation resources. The path-tree is sliced into a load back-slice – a sequence of instructions in the path-tree that affect any load in the path-tree – and a computation slice. The computation slice of a path-tree is all instructions except the load back-slice. ϕ-nodes in the static single assignment form (SSA) [10] of the computation slice represent control-flow. Simply mapping these ϕ-nodes to a functional unit allows control-flow in DySER. An example computation slice is shown in Figure 5c which corresponds to the code-snippet from Table 1. Others have also used slicing for improving hardware efficiency [40, 21].

Load/Store Ordering: In our execution model, the main processor executes loads as part of load back-slice. If the DySER block sends values directly to the memory system, load/store ordering may be violated, which we solve with the dyser_store instruction. It executes as part of the load back-slice and is inserted immediately after the nearest load in the original code. The dyser_store instruction specifies a DySER output port as an input operand, corresponding to the value/address for that store. It logically fetches the address and/or value from DySER output ports and sends the value and address to the load-store queue or write-buffer. Since dyser_store executes as part of the

load back-slice in the main processor, the memory dependency between loads and stores can be resolved using the processor's memory disambiguation mechanisms. This decoupling is a key simplification that allows DySER to leverage existing memory disambiguation mechanisms and remain beneficial in many application domains.

DySER Mapping: Mapping of the computation slice to DySER occurs in three steps. First, the compiler sets up the communication between the load back-slice and computation slice by inserting dyser_send, dyser_load, and dyser_store instructions in the load back-slice . The destination targets for these instructions are named DySER ports. Second, the compiler maps each instruction in the computation slice to a node in the DySER datapath. Third, it configures the switches to create physical paths corresponding to data-flow edges. These two steps that are done by the scheduler are implemented with a greedy heuristic that considers the nodes in topological sort order. Since the DySER network is circuit-switched, we must map data-flow edges to hardware paths making the scheduling problem fundamentally different from tiled architectures like TRIPS [5], WaveScalar [36], and RAW [37]. Figures 5d shows DySER mapping for the code-snippet.

Implementation We have developed extensions to the GCC toolchain for evaluation of the DySER architecture. Our framework operates on the SPARC backend and does path-profiling and DySER mapping. The total configuration information is 327 bytes for one configuration of a 64-FU block, with 10-bits per FU and 19-bits per switch.

We conclude with two observations on path-trees: *1) The number of path-trees in many applications is small enough to create specialized units for each path-tree. 2) Applications remain in a few path-trees for many invocations before entering a different path tree.*

5 Evaluation

Benchmarks: We evaluate applications from the SPEC CPU2006 [33], Parboil [30], and the PARSEC [4] benchmark suite to cover traditional workloads, GPU workloads, and emerging workloads respectively[1]. We consider several benchmark suites to demonstrate the architecture's performance across a diverse suite.

Modeling and Simulation: We modified the Multifacet GEMS [27] OPAL cycle-accurate simulator to support DySER datapaths. Functional unit delays, path delays (one cycle per hop), and the input/output interface are modeled for DySER blocks. We include a 128-entry 2-bit predictor with 4-bits history to predict the next path-tree to hide configuration delays. We model a configuration delay of 64

---

[1]Some of the applications in the PARSEC and SPEC suites do not work with our compiler passes yet and those are not reported (Fortran-code, library-issues, and input-file endianness problems).

Col-1	Col-2	Col-3	Col-4	Col-5	Col-6	Col-7	Col-8	Col-9	Col-10	Col-11
Suite	Benchmarks	#PathTree(90%)	#Paths	#Static Ins	Top-5 paths # Ins	Cvg.%	CS %	Opt.CS %	64 FU %	256 FU %
Parboil	cp	6 (1)	4	187	187	100	78	98	49	100
	pns	7 (1)	3	131	131	100	75	96	82	100
	rpes	12 (1)	62	516	270	83	54	75	28	82
	sad	23 (2)	34	400	266	89	49	55	12	60
PARSEC	blackscholes	9 (3)	2	51	51	100	57	78	100	100
	bodytrack	322 (9)	5	264	255	100	61	75	42	83
	canneal	89 (12)	5	73	71	100	38	38	98	100
	facesim	906 (22)	3	124	123	99	53	65	75	93
	fluidanimate	33 (2)	14	149	123	94	51	63	35	100
	freqmine	151 (31)	6	62	54	99	21	21	95	100
	streamcluster	61 (1)	6	108	62	100	22	43	100	100
	swaptions	36 (6)	6	87	72	99	57	63	97	100
SPEC INT	perlbench	1729 (250)	22	246	80	89	13	13	74	88
	bzip2	252 (33)	8	235	164	98	47	53	72	89
	gcc	10018 (1048)	15	99	58	94	39	41	88	94
	mcf	46 (10)	10	110	67	99	55	59	96	96
	hmmer	113 (3)	39	123	95	90	55	66	31	97
	h264ref	650 (29)	3	149	144	99	39	54	81	84
	astar	132 (6)	1003	139	82	60	55	62	53	57
SPEC FP	namd	236 (28)	7	509	345	100	63	79	31	54
	soplex	731 (105)	5	80	68	99	27	30	91	96
	lbm	18 (1)	4	393	393	100	76	91	21	75
	sphinx3	496 (15)	5	98	97	99	53	63	96	97

Cvg%: Top 5 path runtime coverage, CS:computation slice , 64 FU (256 FU) %: percent inst. schedulable in 8x8 tiles (16x16 tiles)

Table 2. Application Characterization

cycles, but it is often hidden by double-buffering. Binaries that have been modified by our GCC-toolchain are used by the simulator. If a path-tree deviates from the mapped paths, we ignore the outputs from the DySER block and simulate the load-slice to completion. We then branch to the original code and execute on the main processor. For all applications, we simulate multiple sample points of path-trees that account for 90% of the application. Each sample point, fast-forwards to the start of a different path tree and simulates 100 million instructions. We simulate a clock frequency of 2GHz since this is the fastest clock speed for our functional units. For workload analysis, we use an LLVM [23] based tool since its bitcode is easier to work with.

We used the Wattch-based power model in GEMS. We extended it for DySER and refined it using the ITRS 2008 projections and CACTI [38] area and power models. We assume both clock-gating and power-gating is implemented for DySER. To model this, we removed the leakage power consumption of unused units in simulation. Power gating [26] can be implemented for DySER by turning off the unused functional units and switches. Moreover, the wake-up cost is less because each DySER block is executed many times before changing configuration.

We study DySER datapaths integrated with simple single-issue processors, dual-issue OOO processors, and aggressive 4-issue OOO processors. The DySER blocks are pipelined with support for eight concurrent invocations. Such a block has the same area as a 64KB single-ported SRAM.

5.1 Characterization

Coverage: Table 2 shows the characteristics of the different applications. Column three through seven are measure-ments for the entire application's execution using LLVM-based profiling. In most cases, a very small number of path-trees contribute to 90% of the application's dynamically executed instructions. For example, for bzip2, 33 specialized accelerators can cover 90% of the application. The fourth column shows the average number of paths within these path trees and the fifth column shows the total number of static instructions in these trees. As shown, numerous instructions are required for 100% coverage within a tree, making the design of a datapath intractable. The sixth column (Top-5 paths # Ins) shows the number of static instructions in the five most frequent paths. The seventh column (Cvg %), shows the application coverage that can be obtained when only the five most frequent paths are considered in each path tree. For most GPU-like (Parboil) and emerging applications (PARSEC), close to 100% of the application can be covered by a few path trees and for the more *irregular* SpecINT applications, 60% to 99% are covered.

The eighth column shows the percentage of the path-tree which is the computation slice. The ninth column shows the expanded computation slice with a peep-hole optimization to coalesce loads with consecutive address into a single load. This is a safe conservative analysis requiring static disambiguation. For most applications, the computation slice accounts for 59% to 95% of the execution time.

Phase Behavior: The dynamic path tree trace can provide the working set of path-trees in a time-window. We want to determine the smallest number of DySER blocks that can capture these sets of trees. In terms of cache terminology, we have an N-entry fully associative cache of blocks, each entry in the trace is an access and we want to determine the number of accesses between misses. Table 3 shows the number of available blocks and the average number of

DySER Blocks	cp	pns	rpes	sad	bs	bt	canneal	facesim	fa	fm	sc	swaptions
1	1.3M	87K	16M	27	5	45	12	12	169	1	30K	20
2	266M	95K	160K	37	204	111	14	14	337	1	380K	546
4	266M	338M	160K	7,835	8M	245	16	17	337	1	380K	546

DySER Blocks	perlbench	bzip2	gcc	mcf	hmmer	h264ref	astar	namd	soplex	lbm	sphinx3
1	5	9	4	20	32	2	8	18	12	163K	20
2	6	16	5	32	53	4	23	445	22	1M	26
4	6	29	6	490	3,265	8	354	821	32	5M	29

bs: blacksholes, bt: bodytrack, fa: fluidanimate, fm: freqmine. M = million; K = 1000s

Table 3. Number of tree invocations captured by N DySER blocks.

dynamic tree invocations before a miss. With two blocks, the number of invocations between misses varies from 4 to 266 million, corresponding to 400 to several million cycles. Hence, two DySER blocks are sufficient to capture significant amount of phase behavior.

Mapping: The last two columns of Table 2 show quantitative mapping data to make the case for the DySER approach. It shows the percentage of the program that has been mapped to a 64-functional-unit DySER block and a 256-FU DySER block. On average, 70% of the program can be mapped to a 64-FU DySER block. With further compiler work, applications with large path-trees can be partitioned.

5.2 Quantitative Evaluation

Performance: Figure 6a shows performance improvement with 1, 2, and infinite number of DySER blocks integrated with a single-issue in-order processor. In all cases, the load back-slice becomes the bottleneck. *Geometric mean performance improves by 2.1X with a range of 1.1X to 9X*. In cases where the performance improvement is low (like freqmine, perlbench), it is because the computation slice contributes 10% to 25% the program. The SpecINT programs also benefit and show 1.1X to 2.2X speedups.

Figure 6b shows performance improvement with DySER blocks integrated with a dual-issue OOO processor. *We see consistent improvements for DySER + dual issue OOO processor, with geometric mean of 2.2X*. With the OOO processor, effectively we have a better memory system engine which can feed the DySER array better. For both processors, two DySER blocks come close to the performance of infinite blocks. For facesim and gcc, the working set of path-trees is large and the number of DySER blocks is the bottleneck.

The source of improvements: A 4-wide OOO processor with 2 DySER blocks, essentially provides a better load-store engine and we saw similar improvements, but still bottlenecked by the load-slice. As an extreme, we integrated DySER blocks with a 4-issue OOO processor with a perfect cache. We now observed that DySER blocks become the bottleneck as the load-slice typically completes first. As shown in Figure 7, we still see performance improvements resulting from DySER's additional computation resources.

Less of the configuration delays are hidden and geometric mean speedups are 2.8X. We also simulated a hypothetical machine with 128 each of every functional unit with single-cycle full bypass which allows complete specialization of all path-trees. DySER is a practical realization of this machine. *The 4-issue OOO + 2 DySER blocks perform within 26% to 95% of per-phase dedicated datapaths, and on average 68%, showing that the DySER implementation provides an efficient dynamically specialized datapath.*

Energy: Figure 6c (first two stacks) shows energy reduction provided by DySER. Geometric mean energy reduction is 40% and slightly more for the in-order case. Energy-delay improvements ranged from 1.5X to 12X with an average reduction of 2.7X. Some applications like gobmk show worse energy consumption because of too much divergence in the path trees. For these cases, the DySER blocks shouldn't be used.

In energy-constrained mobile environments the DySE approach can trade-off performance improvement to exploit DVFS and provide energy efficiency. We use the detailed DVFS data from the Intel SCC [2] processor for these estimates. Figure 6c (3rd and 4th stack) shows the improvement in energy when frequency (and correspondingly voltage) is reduced to equal the performance of the baseline. We see 5% to 90% reductions with a geometric mean of 60%.

Sensitivity Studies: So far, we have considered one implementation of the functional units. Only longer latency or lower frequency functional units may be feasible for a real chip implementation. To understand this sensitivity, we ran experiments simulating DySER at $\frac{1}{2}$, $\frac{1}{4}th$, and $\frac{1}{8}th$ frequency of the processor. We continue seeing performance improvements. For the 2-issue OOO processor, with DySER at half the frequency, speedups are 0.99X to 9.9X (GM 1.84X), at $\frac{1}{4}th$ frequency, speedups are 0.63X to 9.52X (GM 1.65X), at $\frac{1}{8}th$ frequency, speedups are 0.38X to 8.72X (GM 1.41X, only 16 of 26 show speedup). This shows that pipelining with multiple invocations effectively hides the functional unit latencies.

6 Related work

The closest work to DySER is the Burroughs Scientific Processor (BSP) [22]. BSP uses arithmetic elements imple-
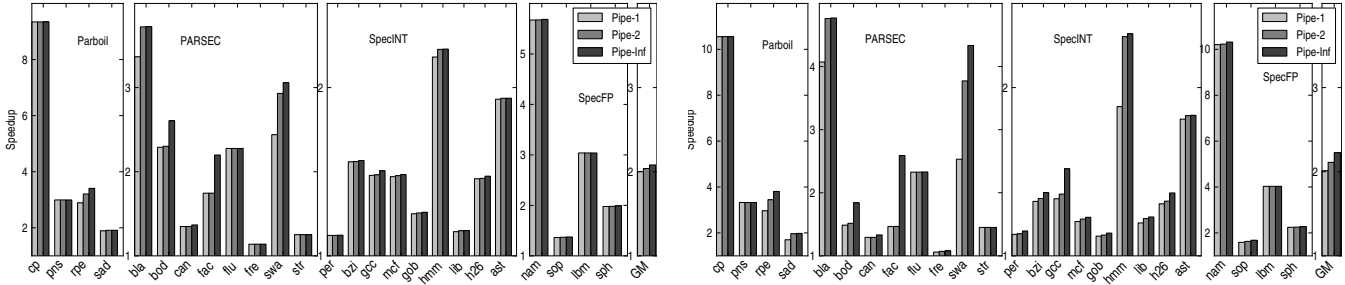
(a) DySER + 1-issue in-order


(b) DySER + 2-issue OOO.

Benchmark key: PARBOIL (cp: cp, pns: pns, rpe: rpes, sad: sad), PARSEC (bla: blackscholes, bod: bodytrack, can: canneal, fac: facesim, flu: fluidanimate, fre: freqmine, swa: swaptions, str: streamcluster), SPECINT (per: 400.perlbench, bzi: 401.bzip2, gcc: 403.gcc, mcf: 429.mcf, gob: 445.gobmk, hmm: 456.hmmer, lib: 462.libquantum, 464: 464.h264ref, ast: 473.astar), SPECFP (nam: 444.namd, sop: 450.soplex, lbm: 470.lbm, sph: 482.sphinx3)

Performance improvement from DySER
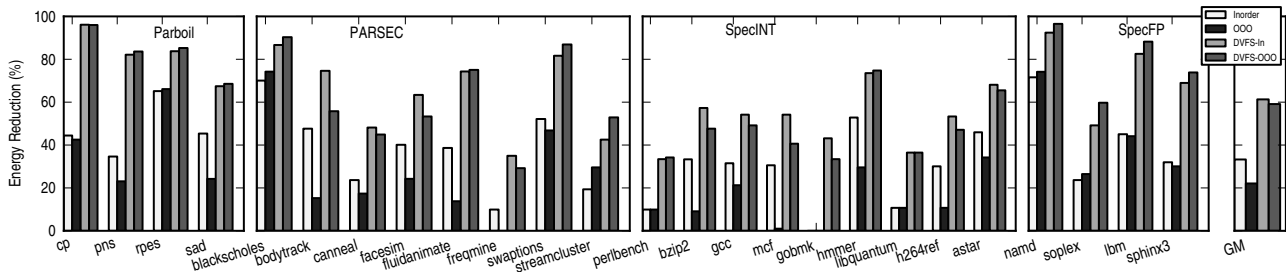
(c) Energy reduction (%) compared to baseline processor using 2 DySER blocks

Figure 6. Performance Improvement and Energy Reduction.
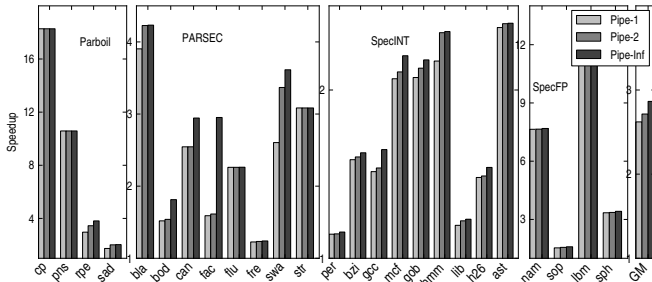

Figure 7. DySER with OOO + perfect cache

mented with pipelining to accelerate vectorized FORTRAN code. Evolution of three important insights from BSP lead to the DySER architecture. First, to achieve the generality, both BSP and DySER utilize compiler support to generate mappings to execute on the execution component. DySER further expands the flexibility and efficiency by introducing a circuit-switch network in the execution array. This improvement needs several new supporting designs such as the configuration path and flow-control. Second, both BSP and DySER identify the critical role of intermediate value storage. The arithmetic elements in the BSP have dedicated register files which are not part of the architecture state. Unlike this "centralized" design, DySER provides distributed storage in its network using pipeline registers. Third, to gen-

erate useful code BSP exploits vector forms, while DySER uses a co-designed compiler than can generate regions of code. A final difference is in the implementation. While the BSP spends much effort on building a fast storage system (register, I/O, special memory), DySER uses a conventional core for efficient data management to achieve same goal.

From the recent literature, the CCA and VEAL architectures [8, 28, 6, 9, 7] are related. The key differences are: VEAL is limited to inner-most loops that must be modulo-schedulable, CCA has limited branching support[2], they do not support a diverse application domain like DySE, and have memory access limitations (CCA does not allow a code-region to span across load/stores). These implementations have limited scalability, supporting only a small number of functional units. VEAL exploits the loop's modulo-schedulability for a novel design which is limited to a small number of functional units (2 INT, 2 FP and one compound unit), while CCA uses a feed-forward cross-bar network connecting consecutive rows which can limit scalability and thus generality to many domains. *The circuit-switched interconnection network and pipelining are the profoundly powerful features in DySER.*

---

[2]Simple branches are moved out of CCA and cleanup-code executes if wrong-path taken. Merge points (PHI-functions) and branching that is dependent on CCA-computed values are not supported.

	DySER	CCA/VEAL [8,26]	Tiled [5,27,34,35]	Specialized [12,13]
Software				
Scope	Complete	Inner-loop	Complete	Complete
Generality	General-purpose	Domain-specific†	General-purpose	Domain-specific
Hardware				
Scalability	Yes	Limited	Yes	Yes
Overall Complexity	Low	Medium	High	High
Integration	In-core	In-Core	Dedicated	Dedicated
Comp. Eff.	High	High	Medium	High
Area	Small	Small	Large	Large
Mechanisms				
ISA	Extension	Co-desg	New	New
Compute elements	FU & switch	FU & Complex-FU	Cores, RF, buffers	Simple cores
Network	Circuit Switch	Circuit Switch	Packet Switch	Packet Switch
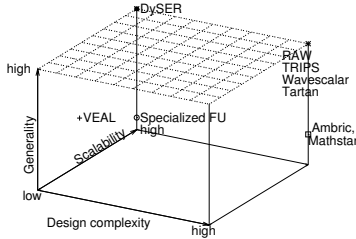
Table 4. Design space and related work

DySER blocks are similar to tiled architectures like RAW [37], Wavescalar [36], and TRIPS [5]. DySER is a pure computation fabric and thus has no buffering, instruction storage, or data-memory storage in the tiles. Second, DySER implements circuit-switched static routing of values, thus making the network far more energy efficient than the dynamically arbitrated networks. *While these architectures distribute design complexity to different tiles, DySER blocks do not add complexity. The slicing of applications provides several simplifications and DySER blocks need only perform computation.* The Tartan architecture and compiler memory analysis explores spatial computing at fine granularity with entire applications laid out on-chip [29]. The DySE execution model is far less disruptive and achieves efficiency by dynamically reconfiguring the DySER datapaths as the application changes phase. *Compared to these architectures, DySER achieves similar or superior performance without massive ISA changes and at significantly smaller area and design complexity (compared to these full-chip solutions, DySER's area is the same as a 64KB SRAM). It lacks their intellectual purity, since we relegate load/store processing to the processor - which we argue (and show) is a judicious decision.*

Other reconfigurable architectures include the following: Garp uses an FPGA-like substrate for tight-inner loops but suffers when loop iterations are small [17]. Chimaera maps instruction sequences to a reconfigurable sub-

strate [39]. Ambric [14] and Mathstar [15] explore domain-specific specialization. Table 4 classifies the related work in terms of software capability, hardware constraints, and their basic mechanisms. In terms of generality, scalability, and overall complexity[3], *DySER achieves all three with a co-designed compiler and simplified computational as shown in the design space tradeoffs plotted in the Table.*

The path-tree construction is similar to hyper-blocks [25], but our representation is rooted at the inner-most block and captures all paths. Our slicing approach is similar to early efforts of decoupled access/execute machines [32] but we map one of the slices to hardware.

7 Future Work and Conclusions

This paper introduced the DySE execution model of dynamically synthesizing datapaths, presented the DySER architecture for hardware specialization, and evaluated it. The basic hardware organization of DySER is a circuit-switched tiled array of heterogeneous computation units which are dynamically specialized by a compiler that uses profiling to map application phases to these blocks. DySER effectively adds very general purpose accelerators into a processor pipeline and with its co-designed compiler, the same hardware can target diverse application domains. Results from a diverse application suite consisting of the SPEC, PARSEC, and Parboil benchmark suites show impressive results. When integrated with a single-issue processor, two DySER blocks, which each occupies approximately the area of a 64KB SRAM, show 1.1X to 10X performance improvements with up to 70% reduction in energy. Furthermore, the approach scalably provides energy-efficient performance as the baseline processor improves.

Further work is required to understand the effectiveness of the DySE execution model and the DySER architecture. First, a detailed compiler design and implementation that builds path-trees in the compiler's intermediate representation (instead of modifying generated code as is currently done) can help generate larger path trees and is required for a practical implementation. Further optimizations to consider the impact of loop unrolling, function inlining and techniques to reduce control-flow graph disruptions as a result of slicing, and scheduler optimizations should help. While the ISA impact should be small, further investigation on other ISAs like x86 ISA which has few general purpose registers is required. Finally, dynamically profiling and compilation is an avenue of promising future work.

The execution model with the DySER block provides a practical way to implement instruction-set specialization, SIMD specialization, and domain-driven accelerators using one substrate. This work focuses on the architecture and

---

[3]We are referring to how easily these systems can co-exist with conventional processors and how disruptive a change they require in hardware and software development

microarchitecture of DySER. Further work is required to compare how general the execution model and DySER's dynamic specialization really is.

Finally, we are investigating a detailed prototype implementation of the DySER block integrated with a real processor. This implementation can answer questions on design complexity, area, and power trade-offs, and can quantify the gap between DySER and a truly specialized design. The execution model can be implemented with other architectures including application-specific accelerators where computation-slices are compiled to silicon and an FPGA substrate. Direct compilation to silicon suffers from design-time freezing of the accelerator. The area density and potential frequency from an FPGA substrate will be less than DySER. However, the FPGA can provide capability for synthesizing compound computational blocks and probably better diversity. This FPGA approach requires further advancements in the synthesis of the computation slices and introduces scope for novel solutions in that space. Detailed analysis is required for a definitive answer.

In general, architectural techniques like DySER can address the slowing of device-level energy improvements by energy efficiently improving performance.

Acknowledgments

References

[1] Semiconductor Industry Association (SIA), Process Integration, Devices, and Structures, International Roadmap for Semiconductors, 2009 edition. .

[2] M. Baron. The single-chip cloud computer. *Microprocessor Report*, April 2010.

[3] M. Bhadauria, V. M. Weaver, and S. A. McKee. Understanding PARSEC performance on contemporary CMPs. In *IISWC, 2009*, pages 98–107, Austin, TX.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT '08*.

[5] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[6] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi. The reconfigurable streaming vector processor (rsvptm). In *MICRO 36*, page 141, 2003.

[7] 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*, pages 272–283, 2005.

[8] N. Clark, A. Hormati, and S. Mahlke. Veal: Virtualized execution accelerator for loops. In *ISCA '08*, pages 389–400, 2008.

[9] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *MICRO 37*, pages 30–40, 2004.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, Oct 1991.

[11] V. Govindaraju, P. Djeu, K. Sankaralingam, M. Vernon, and W. R. Mark. Toward a Multicore Architecture for Real-time Ray-tracing. In *MICRO '08*, pages 176–187.

[12] V. Govindaraju, C. Ho, and K. Sankaralingam. Design and evaluation of dynamically specialized datapaths with the dyser architecture. Technical report, The University of Wisconsin-Madison, Department of Computer Sciences.

[13] G. Grohoski. Niagara-2: A highly threaded server-on-a-chip. In *18th Hot Chips Symposium*, 2006.

[14] T. R. Halfhill. Ambric'S New Parallel Processor - Globally Asynchronous Architecture Eases Parallel Programming. *Microprocessor Report*, October 2006.

[15] T. R. Halfill. MathStar Challenges FPGAs. *Microprocessor Report*, 20(7):29–35, July 2006.

[16] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, pages 37–47, 2010.

[17] J. R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 16–18, April 1997.

[18] M. Hempstead, G.-Y. Wei, and D. Brooks. Navigo: An early-stage model to study power-constrained architectures and specialization. In *Workshop on Modeling, Benchmarking, and Simulation*, 2009.

[19] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *MICRO 39*, pages 397–408.

[20] A. Kejariwal, A. V. Veidenbaum, A. Nicolau, X. Tian, M. Girkar, H. Saito, and U. Banerjee. Comparative architectural characterization of SPEC CPU2000 and CPU2006 benchmarks on the intel core2 duo processor. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 132–141.

[21] D. Kim and D. Yeung. Design and evaluation of compiler algorithms for pre-execution. *SIGPLAN Not.*, 37(10):159–170, 2002.

[22] D. J. Kuck and R. A. Stokes. The burroughs scientific processor (bsp). *IEEE Trans. Comput.*, 31:363–376, May 1982.

[23] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–88.

[24] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. In *ISCA '07*, pages 358–368.

[25] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *ISCA '92*, pages 45–54.

[26] H. Makino, Y. Tujihashi, K. Nii, C. Morishima, and Y. Hayakawa. An auto-backgate-controlled MT-CMOS circuit. In *Proceedings Symposium on VLSI Circuits*, pages 42–43, 1998.

[27] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, , and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News (CAN)*, 2005.

[28] B. Mathew and A. Davis. A loop accelerator for low power embedded vliw processors. In *CODES+ISSS '04*, pages 6–11.

[29] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu. Tartan: evaluating spatial computation for whole program execution. In *ASPLOS-XII*, pages 163–174.

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

[31] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008*, pages 18:1–18:15.

[32] J. E. Smith. Decoupled access/execute computer architectures. In *ISCA '82*, pages 112–119, 1982.

[33] SPEC CPU2006, Standard Performance Evaluation Corporation.

[34] Intel streaming simd extensions 4 (sse4), http://www.intel.com/technology/architecture-silicon/sse4-instructions/index.htm.

[35] S. Subramaniam and G. H. Loh. Fire-and-forget: Load/store scheduling with no store queue at all. In *MICRO 39*, pages 273–284.

[36] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *ISCA '03*, pages 291–302.

[37] M. B. Taylor et al. The RAW Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs, IEEE Micro, 22(2):25-35, March, 2002.

[38] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20, HP Labs.

[39] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *ISCA '00*, pages 225–235.

[40] C. B. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. In *ISCA '00*, pages 172–181.