DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing

Venkatraman Govindaraju
Chen-Han Ho
Tony Nowatzki
University of Wisconsin—Madison

Jatin Chhugani
Nadathur Satish
Intel

Karthikeyan Sankaralingam
University of Wisconsin—Madison

Changkyu Kim
Intel

THE DYSER (DYNAMICALLY SPECIALIZING EXECUTION RESOURCES) ARCHITECTURE SUPPORTS BOTH FUNCTIONALITY SPECIALIZATION AND PARALLELISM SPECIALIZATION. BY DYNAMICALLY SPECIALIZING FREQUENTLY EXECUTING REGIONS AND APPLYING PARALLELISM MECHANISMS, DYSER PROVIDES EFFICIENT FUNCTIONALITY AND PARALLELISM SPECIALIZATION. IT OUTPERFORMS AN OUT-OF-ORDER CPU, STREAMING SIMD EXTENSIONS (SSE) ACCELERATION, AND GPU ACCELERATION WHILE CONSUMING LESS ENERGY. THE FULL-SYSTEM FIELD-PROGRAMMABLE GATE ARRAY (FPGA) PROTOTYPE OF DYSER INTEGRATED INTO OPENSPARC DEMONSTRATES A PRACTICAL IMPLEMENTATION.

Future processors must improve microarchitectural efficiency to overcome slowing transistor energy efficiency improvements and sustain performance growth. Specialization and accelerators are promising directions. A mainstream specialization technique is to specialize architectures for *data-level parallelism* (DLP)—for example, vector processors, short-vector instructions such as Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX), and GPUs. *Functionality specialization* is another technique, wherein custom hardware is targeted at application functionality. Examples include Garp,[1] Chimaera,[2] Configurable Compute Accelerator (CCA),[3] PipeRench,[4] Tartan,[5] Phoenix,[6] Conservation Cores,[7] and Beret (Bundled Execution of Recurring Traces).[8]

Thus far, specialization architectures have targeted only parallelism or functionality specialization, but not both. In fact, the functionality specialization architectures typically aren't evaluated on data-parallel workloads, and vice versa. The reason for this distinction is that the fundamental approaches behind these strategies are conflicting. Parallelism specialization uses homogeneous hardware resources with a wide and independent interconnect, whereas functionality specialization uses task-specific hardware resources with task-specific routing. Furthermore, parallelism specialization's homogeneous resources are simple to virtualize, to support mapping arbitrarily large computations, whereas functionality specialization's heterogeneity means arbitrary computations face resource-mapping problems. Nevertheless, architectures such

Published by the IEEE Computer Society

as short-vector extensions (for example, SSE) and GPUs are taking incremental steps toward unifying functionality specialization with their parallelism specialization approach. The driving force is that the combination of specialization types can provide further energy and performance benefits.

We can view the DySER (Dynamically Specializing Execution Resources) architecture as the natural progression of this trend toward unification, culminating in both functionality and parallelism specialized in a single architecture. The enabling mechanism is a configurable lightweight switching network that connects a set of heterogeneous functional units and allows customization. DySER exploits parallelism by creating logical lanes of independent computation in this substrate and exploits functionality by creating specific data paths for each particular computation.

Practically speaking, DySER is integrated into a general-purpose processor's execution stage, which acts as a load/store engine to feed the DySER computation substrate. To achieve functionality specialization, a compiler synthesizes data paths among functional units, specific to an application's phase. To achieve parallelism specialization, we use a judicious mix of vectorization techniques and novel hardware mechanisms. We enable high performance by providing a dense computational fabric with low-latency integration to a processor, and we attain energy efficiency through the elimination of per-instruction overheads by converting code regions into dynamically formed compound functional units. These gains are possible without significant disruption to either the general-purpose processor architecture into which DySER is integrated or the software development environment.

We designed and implemented the DySER architecture and its compiler, ported applications to it, and implemented a field-programmable gate array (FPGA) prototype. Employing functionality specialization, DySER outperforms a dual-issue out-of-order (OOO) processor by 1.1× to 4.1×, while simultaneously reducing energy by 9 percent. Employing parallelism specialization, DySER outperforms single instruction, multiple data (SIMD): it is 1.3× to 4.7× faster than SSE, consuming 86 percent less energy. It also outperforms GPUs, with a geometric-mean speedup of 1.4×, while consuming 8 percent less energy.

DySER and functionality specialization

The main insight in designing DySER is that programs execute in phases, and only a few of these phases or regions contribute to most of the program's execution time. Specializing such frequently executing regions can eliminate overheads and provide energy efficiency. However, the cost of having specialized hardware for all such possible regions is prohibitive. Instead, DySER dynamically creates specialized data paths for only frequently executed regions. We also leverage the processor's memory system, using its cache, prefetch, memory disambiguation, and memory-dependence prediction mechanisms, thus overcoming load/store serialization bottlenecks in irregular code.

Architecture

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

To allow pipelining inside DySER, we implement a simple credit-based flow control using a forward signal (*valid*) and a backward signal (*credit*). Functional units perform the operation when all its inputs are valid, and data is forwarded only when the credit signal is asserted. Functional units and switches send credits only when they can accept new data. This network and data-flow

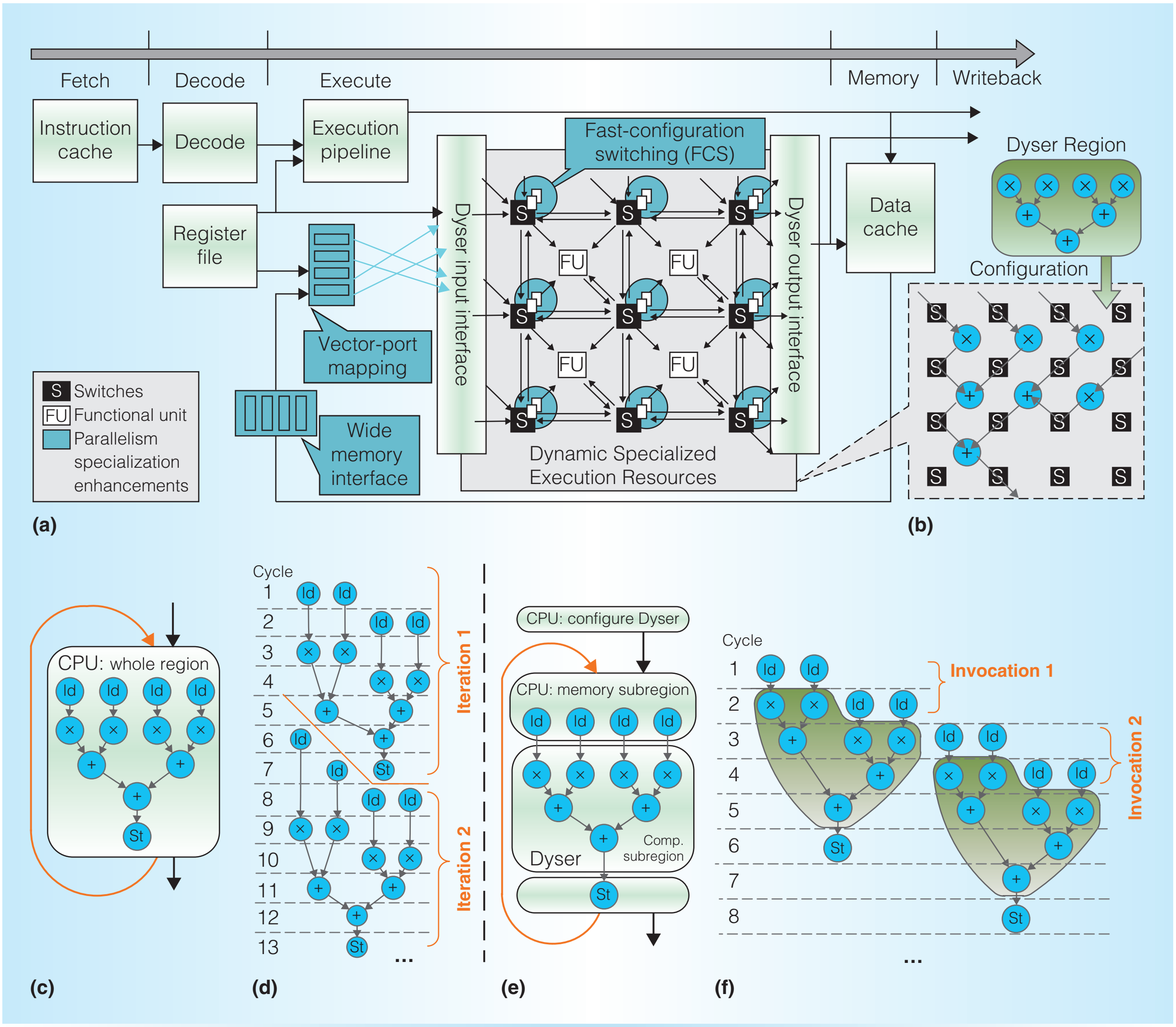

Figure 1. Overview of the DySER (Dynamically Specializing Execution Resources) architecture integrated into a processor pipeline highlighting the data-parallel extension. The bottom half of the figure contrasts execution in a conventional processor with execution on a processor with DySER. DySER architecture (a), example configuration (b), static program control-flow graph (CFG) (c), dual-issue CPU execution (d), and pipelined DySER execution (f).

execution model create a pipelined functionality specialization engine.

Figure 1a shows how DySER is integrated into a processor. The processor pipeline communicates to DySER through a set of named I/O ports corresponding to first-in, first-out (FIFO) buffers that deliver data to the switches. We extend the instruction set architecture (ISA) with five instructions that configure DySER, send/receive register data, and send/receive memory values.

Execution model

Figures 1c through 1f compare the conceptual execution model of a dual-issue OOO processor to that of DySER. The processor in Figure 1d executes up to two operations at a time, and is shown performing two iterations of the loop from Figure 1c.

The DySER version in Figure 1e begins by first configuring DySER for a region's data path before the region is encountered. For every instance of the region, the processor either sends register values or loads data directly to DySER. We refer to all of the sends and loads for one instance as an *invocation.* As the data reaches DySER, it is routed to functional units through switches according to the configuration, and execution occurs in data-flow fashion, producing results for the processor. Similar to the CPU, DySER can be speculatively invoked with the next instance of the computation, pipelining both instances together, as Figure 1f shows. In this example, DySER executes five fewer cycles than the processor.

Compiler role

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

Workload characterization

Across the benchmarks in the Parsec and SPECint benchmark suites, there are many candidate regions to specialize—ranging in size from nine to 906 for Parsec and 46 to 10,018 for SPECint. However, about 10 percent of the regions contribute to 90 percent of the execution within each benchmark. These regions are 51 to 264 instructions in length. With a 64-unit heterogeneous DySER block, we can specialize 60 to 100 percent of these regions.

DySER and parallelism specialization

DySER's many functional units provide an opportunity for supporting data-parallel execution. However, for DySER, parallelism specialization presents several challenges. We developed parallelism specialization mechanisms to overcome these challenges by analyzing DySER's performance on data-parallel applications. We considered hand-optimized workloads from Intel's research lab and Parboil (http://impact.crhc.illinois.edu/parboil.php). Table 1 describes the workloads we considered and their characteristics.

Challenges for DySER on data-parallel workloads

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

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

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

Superfluous regions. Figure 2c shows a very large computation subregion. Although we can configure DySER separately for different sections of the computation subregion, DySER's functional units will be inactive during reconfiguration. Because we must perform reconfiguration on every invocation, overall utilization of DySER is low.

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

Mechanisms for parallelism specialization

"Transformation flow" in Figure 3 shows our overall strategy for transforming an arbitrary computational subregion to act like an ideal region. Although we performed them

Table 1. Benchmark characterization.

Benchmark	Description	GPU and SIMD performance analysis	Region type
NBDY	N-body simulation	Large kernel with regular access pattern	Superfluous
VR	Volume rendering	Nested loop with lots of control-flow	Insufficient
TSRCH	Tree search	Irregular data accesses prevent SSE vectorization	Insufficient
MRG	Sorting	Small kernel with unpredictable data-dependent control flow	Superfluous
RDR	Complex convolution	Small kernel with regular access pattern	Insufficient
CONV	Image convolution	Regular computation and data accesses; no control-flow divergence.	Insufficient
MRI-Q	Magnetic resonance imaging	Heavy use of sine and cosine; use of constant memory for less global memory bandwidth.	Proportional
SPMV	Sparse matrix-vector multiplication	Indirect loads are software pipelined; uses constant and texture memory.	Insufficient
CTCP	3D grid and point calculation	Significant use of transcendentals; overlaps CPU and GPU execution.	Superfluous
MM	Dense matrix multiplication	Standard algorithm; shared memory and synchronization to reduce global memory bandwidth.	Insufficient
STNCL	3D Jacobi stencil operation	Small compute-to-memory ratio; shared memory and synchronization reduce global memory bandwidth.	Proportional
SAD	Sum of absolute differences	Extremely high compute-to-memory ratio; good memory locality.	Proportional
LBM	Fluid dynamics	Extremely large computation region; large-region control-flow divergence.	Superfluous
TPACF	Angular correlation	Irregular memory access due to histograming; causes branch divergence.	Superfluous
KMNS	K-Means clustering	Uses texture as cache; regular memory access.	Insufficient
NNW	Neural networks	Some transcendentals; strided memory access.	Insufficient
FFT	Fast Fourier transform	Regular memory access; heavy use of sine and cosine.	Proportional
NDL	Dynamic programming	GPU diagonal iteration inhibits memory coalescing; shared memory and synchronization.	Insufficient

* abs(): an operation that computes the absolute value; FCS: fast-configuration switching; SCX: scalar expansion; SIMD: single instruction, multiple data; SSE: Streaming SIMD Extensions; STR: strip mining; SUB: subgraph matching; UNR: loop unrolling; VEC-Hybrid: hybrid communication; VEC-Inter: interinvocation communication; VEC-Intra: intra-invocation communication.

manually for this article, we designed these transformations to be implementable in a compiler. Only the TPACF and Merge benchmarks require additional algorithmic changes for effective parallelism specialization. Column 5 in Table 1 shows the transformations for each benchmark.

Region growing. If regions are too small to attain high utilization, we must expand them by transforming the loops. Specifically, we apply loop unrolling (UNR) until an appropriately sized computation subregion is formed, as Figure 3a shows. If the loops are independent, we create a multilane pattern.

Data-level parallelism techniques used	DySER analysis	Comments
SCX, FCS, VEC-Intra	DySER throughput limited by long latency functional units: division (div) and square root (sqrt).	SIMD benchmark
UNR, SCX	Control-flow divergence hinders SSE; DySER limited by irregular memory.	SIMD benchmark
UNR, SCX, SUB	Scalar loads feed region; SSE loses because of irregular memory access.	SIMD benchmark
SUB, VEC-Intra	Emulates 4 × 4 merge network; larger region than SSE. Control flow limits performance.	SIMD benchmark
UNR, SUB, STR, VEC-Hybrid	Emulates four-wide complex multiplier, wins with fewer instructions.	SIMD benchmark
UNR, SUB, STR, VEC-Intra	DySER emulates eight-wide SIMD; DySER wins with larger region and memory regularity.	SIMD benchmark
STR, VEC-Inter	Single computation lane; DySER loses because of non-pipelined sine and cosine functional units.	GPU benchmark; GPU is faster.
UNR, STR	DySER loses because of irregular memory access; no vectorization is possible.	GPU benchmark; GPU is faster.
FCS, STR, VEC-Hybrid	Multilane pattern executes sqrt operations in parallel; DySER loses because of the sqrt operation's throughput.	GPU benchmark; GPU is faster.
UNR, VEC-Intra	Similar performance due to regular memory access and high compute-to-memory ratio.	GPU benchmark; DySER and GPU have similar performance.
STR, VEC-Inter	Two-lane stencil; similar performance, limited by low compute-to-memory ratio.	GPU benchmark; DySER and GPU have similar performance.
UNR, FCS, STR, VEC-Hybrid	Multilaned abs() with sum reduction; similar performance due to regular memory.	GPU benchmark; DySER and GPU have similar performance.
FCS, VEC-Intra	Many reductions; divergence hurts GPU; scattering memory hurts DySER.	GPU benchmark; DySER and GPU have similar performance.
FCS, STR, VEC-Hybrid	Performs histogram with reductions; similar because GPU parallelizes history.	GPU benchmark; DySER and GPU have similar performance.
UNR, VEC-Intra	Large reduction kernel; performance is similar due to memory access regularity.	GPU benchmark; DySER is faster.
UNR, STR, VEC-Hybrid	VEC-Inter and VEC-Intra on different arrays; poor warp occupancy for GPU.	GPU benchmark; DySER is faster.
UNR, VEC-Hybrid	Single lane region. GPU implementation doesn't cache reused sine and cosine operations.	GPU benchmark; DySER is faster.
UNR, VEC-Intra	Single lane of computation; GPU suffers from excess synchronization and poor coalescing.	GPU benchmark; DySER is faster.

With a single loop-carried dependence, we create a reduction pattern, if possible. When profitable, we alternatively employ scalar expansion (SCX) (Figure 3b), which enables loop parallelization by providing temporary storage for dependent variables. Scalar expansion lets us break some reduction patterns into multilane patterns, which can be beneficial depending on the use of the region's outputs.

Vectorizing DySER. Vectorized DySER instructions can load and store only contiguous words. In order to vectorize send and

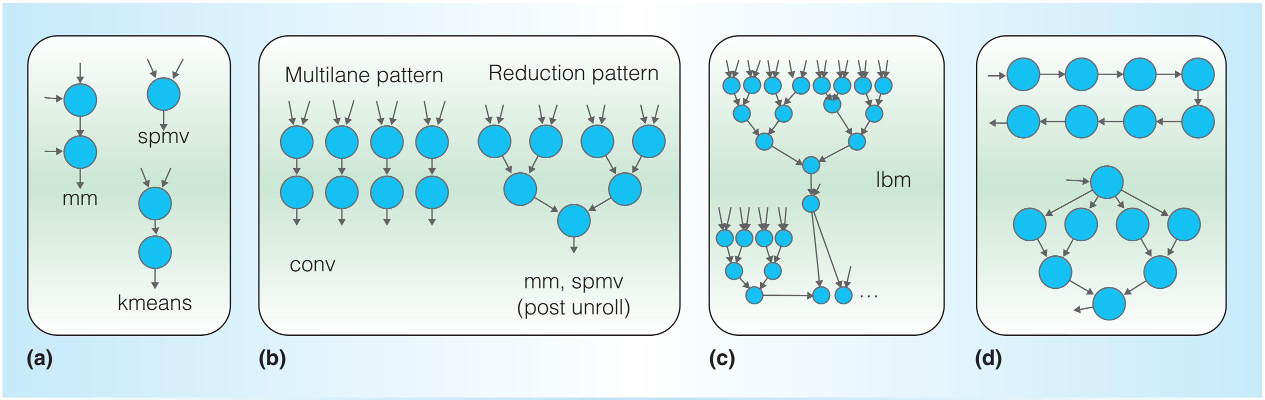

Figure 2. Types of computation subregions with example benchmarks for each type: insufficient (a), proportional (b), superfluous (c), and ideal (d). Through a set of transformations, we convert all region types to resemble the ideal region type.

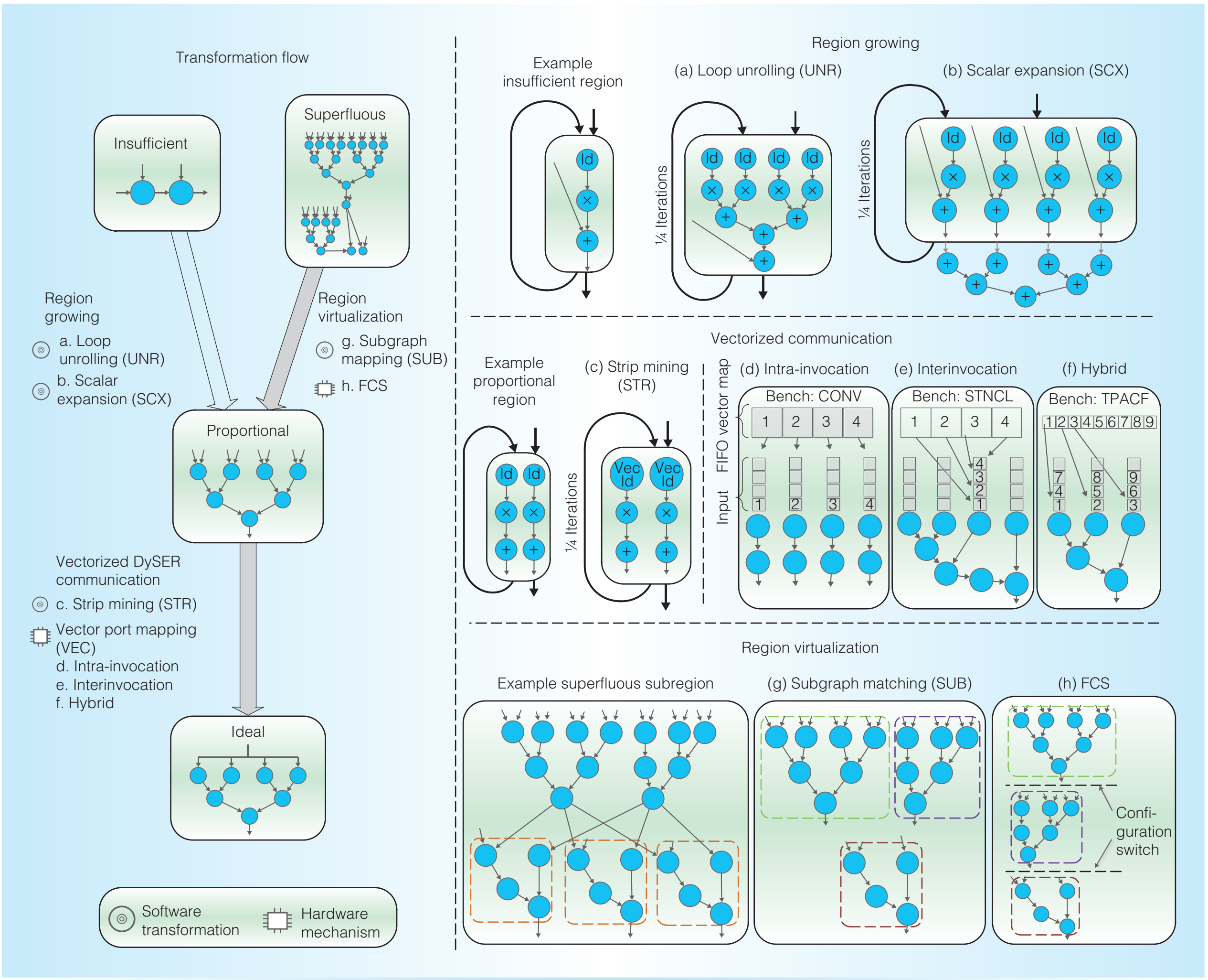

Figure 3. Overview of hardware and software transformations for parallelism specialization. The transformation flow indicates the strategy for transforming an arbitrary computational subregion into an ideal region.

load instructions efficiently, we must provide mechanisms to handle arbitrary relationships between contiguous memory and the interface to the regions. We explain several communication patterns with examples, and describe the mechanisms that make vectorization possible:

- *Intra-invocation communication (VEC-Intra).* Figure 3d shows the computation subregion from the convolution (CONV) benchmark. Each contiguous memory word is mapped to a different input port of DySER and used by a single invocation. Intra-invocation communication converts DySER into a vector unit.
- *Interinvocation communication (VEC-Inter).* Figure 3e shows the computation subregion from the stencil (STNCL) benchmark. Each contiguous memory word is mapped to the same port because subsequent invocations use contiguous memory addresses, letting multiple invocations be explicitly pipelined.
- *Hybrid communication (VEC-Hybrid).* Figure 3f shows a computation subregion from the TPACF benchmark. Neither interinvocation nor intra-invocation is sufficient to perform a vector load more than three words wide. Our strategy is to use a hybrid, where each word triplet is sent to the same invocation, and subsequent triplets are pipelined to subsequent invocations. This example is three words wide and three words deep.

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

To implement vectorized communication in hardware, we augment the configuration with additional bits that specify a vector map for each port. This mapping effectively creates logical vector ports in DySER. When a vectorized DySER instruction accesses these ports, an additional state machine in DySER's I/O interface coordinates the transfer of data. Also, vector loads/stores require a wide memory interface similar to that used by SSE.

Region virtualization. Similar to the way we handle insufficient regions, we must resize overly large regions to fit inside DySER in order to achieve high utilization. Compared to instruction-level acceleration, DySER's dynamic customization introduces resource limitation challenges, which we overcome by employing two primary techniques:

- *Subgraph matching (SUB).* First, we attempt to reduce the computational region by identifying similar computational structures, which we call subgraph matching, as shown in Figure 3g. The transformation is essentially to cut dataflow edges from a common subgraph, and combine all common subgraphs together. These cut edges will be reconnected through the memory subregion.
- *Fast-configuration switching (FCS).* If subgraph matching can't reduce the computation subregion sufficiently, we employ a further technique to reduce the configuration penalty. Figure 3h shows how a superfluous computation subregion can be cut into appropriately sized components and mapped to DySER using multiple configurations.

We enable FCS through two hardware mechanisms. First, we augment every DySER tile (functional unit and switch) with the capability to store multiple configurations. Second, we have developed a configuration switch protocol for DySER that relies on each tile being in either an active or off state. We add to the network a 1-bit free signal, which is sent from a tile's eight neighbors. We add one additional instruction that sends reset signals through the old configuration, forcing every tile that has finished computation into the off state, triggering them into sending free signals to neighbors. The set signals that follow the reset signals then change any off-state tile into the new configuration. Each set signal propagates to all neighbors

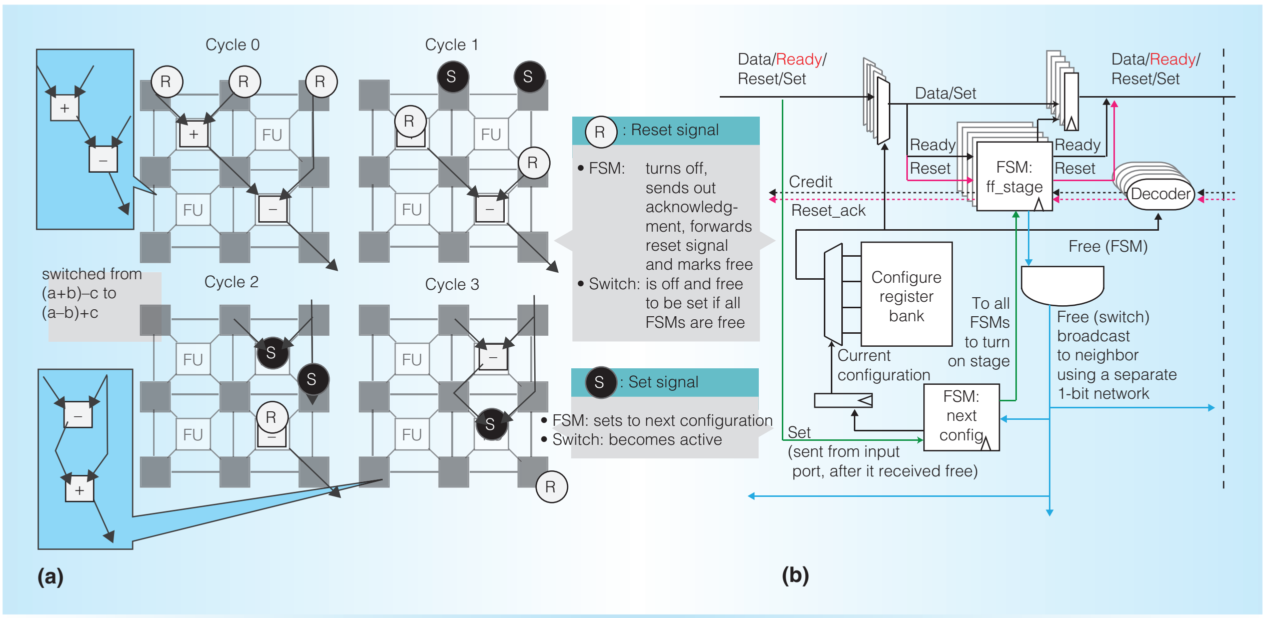
Cycle 0
Cycle 1
Cycle 2
Cycle 3
FU
R
S
switched from (a+b)–c to (a–b)+c
R : Reset signal
• FSM: turns off, sends out acknowledgment, forwards reset signal and marks free
• Switch: is off and free to be set if all FSMs are free
S : Set signal
• FSM: sets to next configuration
• Switch: becomes active
(a)
Data/Ready/ Reset/Set
Data/Set
Ready
Reset
FSM: ff_stage
Decoder
Credit
Reset_ack
Free (FSM)
Configure register bank
To all FSMs to turn on stage
Current configuration
Free (switch) broadcast to neighbor using a separate 1-bit network
FSM: next config
Set (sent from input port, after it received free)
(b)

Figure 4. Fast-configuration switching (FCS) example and implementation. The set and reset signals perform configuration switching (a). The detailed microarchitecture, which requires configuration registers and finite-state machines (b).

in the new configuration after receiving their free signals.

This protocol explicitly reuses the dataflow in the two regions to synchronize the set and reset signals without any additional networks or compiler requirements. As soon as an entire invocation has been sent, the program can execute the instruction that sends reset and set signals to DySER. Figure 4a shows an example of the set and reset signals performing the configuration switching. Figure 4b shows the microarchitecture implementation, which requires configuration registers and finite-state machines, and outlines the protocol.

Using our register-transfer level (RTL) implementation, the energy consumed by FCS is about 2 picojoules compared to 120 picojoules for instruction-fetch and decode in a four-wide OOO processor.

Evaluation

Our evaluation focused on three issues: functionality specialization effectiveness, parallelism specialization effectiveness, and implementation and integration feasibility. We evaluated DySER in terms of these issues with simulation, RTL implementation, and a full-system FPGA implementation.

Evaluation methodology

For our performance and energy evaluation, we used a simulation-based approach. We considered a dual-issue OOO processor as our baseline. This baseline processor had a 64-Kbyte Level-1 (L1) data cache, a 32-Kbyte L1 instruction cache, and a tournament branch predictor with 4,096 branch target buffer (BTB) entries. We considered an SSE implementation in X86 and a GPU as our reference data-parallel accelerators to compare to DySER. Specifically, we considered a GPU that has eight processing elements in an SM (streaming multiprocessor) because its area and functional-unit mix matched one DySER block integrated with a two-wide OOO processor. In all cases, we considered applications tuned for each architecture. We integrated DySER into a dual-issue OOO processor that was identical to our baseline.

We evaluated a DySER architecture that had 64 heterogeneous functional units (see Table 2). We used the Gem5 simulator (http://www.m5sim.org), Gem5 + SSE, GPGPU-Sim,[9] and Gem5 extended for DySER to evaluate the various platforms, respectively. We augmented our Gem5 infrastructure with McPAT-based power models.[10]

Table 2. Details of functional units used in a 64-functional-unit DySER. We use a unified division (div) and square root (sqrt) implementation.

Unit	Count	Latency	Area (μm^2)	Description
INT-ADD	16	1	2,482	OpenSparc
INT-MUL	12	5	16,401	OpenSparc
FP-ADD	16	4	14,533	OpenSparc
FP-MUL	12	7	24,297	OpenSparc
FP-DIV	4	12	16,932	Taylor-Series based[11]
FP-SQRT	4	12	16,932	Taylor-series based[11]
Switch	81	1	8,009	N/A

* FP-ADD: floating-point add; FP-DIV: floating-point divide; FP-MUL: floating-point multiply; FP-SQRT: floating-point square root; INT-ADD: integer add; INT-MUL: integer multiply.

We developed our own GPU power model, extending the device-specific model developed by Hong and Kim[12] to allow parameterization. Its error range was $\leq$20 percent.

For functionality specialization, we considered the Parsec and SPECint benchmark suites. We used code optimized with GCC (GNU Compiler Collection) -O3, and we used profiling to identify regions. We chose distinct benchmarks for evaluating parallelism specialization, because we wanted to manually compile and optimize each benchmark, which was intractable for Parsec and SPECint. A manual approach is appropriate, because we were evaluating our architectural mechanisms independently of effects from potential compiler transformations. The benchmarks we described earlier were good choices, because they were simple enough to manually optimize and were well-suited for their respective accelerators. For the DySER versions of these benchmarks, we implemented or obtained scalar C++ code, manually applied the transformations described earlier, and compiled the benchmarks with our GCC-based DySER toolchain.

Functionality specialization results

Figures 5a and 5c show the performance and energy improvements from the DySER integration when compared to the baseline dual-issue OOO processor and performing only functionality specialization. We consistently saw improvements across the benchmarks, with harmonic speedup of 39 percent and energy reduction of 9 percent. We achieved performance improvements in irregular workloads by forming large regions and specializing such regions with control-flow support inside DySER. With some benchmarks (such as freqmine and gobmk), we saw little performance gain, because of the many insufficient regions that weren't amenable to our transformations. Govindaraju et al. present further analysis of these benchmarks and results.[13]

Parallelism specialization results

The last two columns in Table 1 summarize our results when parallelism specialization using DySER was performed. For each benchmark, the table shows how DySER employs the transformations we described. These benchmarks primarily benefit from parallelism specialization, but they can also implicitly benefit from functionality specialization by using the DySER hardware to represent computations. Figures 5b and 5d show the performance and energy improvements of DySER, SIMD, and GPU acceleration compared to the baseline.

DySER versus SIMD. DySER performs significantly better for all of the SIMD benchmarks. For highly regular workloads such as CONV and RDR, DySER emulates a wider SIMD unit than SSE units and accelerates them using vectorized loads. For irregular workloads such as volume rendering (VR) and tree search (TSRCH), we found independent computations across iterations

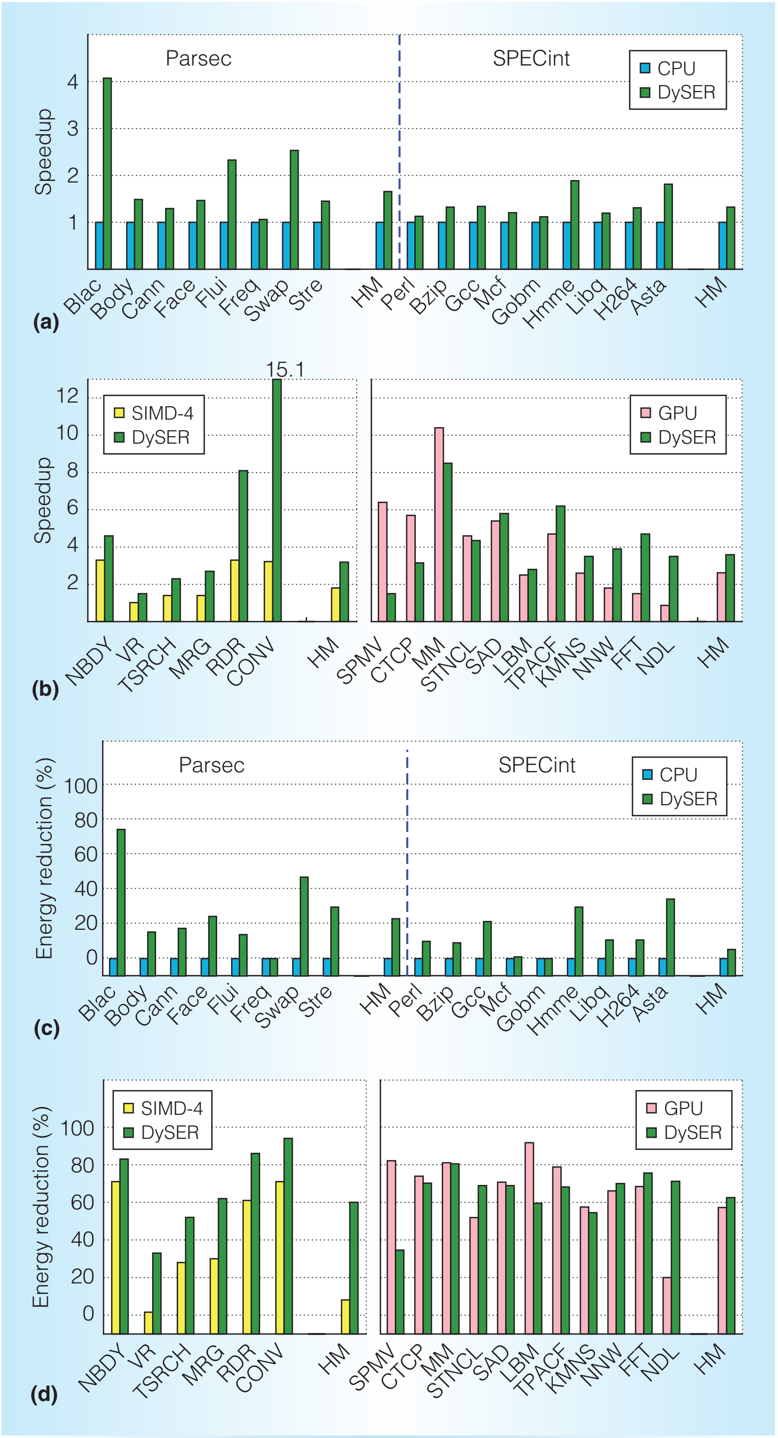

Figure 5. Performance and energy improvements from DySER specialization: functionality specialization performance improvement (a), parallelism specialization performance improvement (b), functionality specialization energy reduction (c), and parallelism specialization energy reduction (d). (CPU: CPU without SIMD; DySER: 64-functional-unit DySER + CPU; GPU: one streaming multiprocessor (SM) with eight processing elements; SIMD-4: four-wide SIMD + CPU).

and accelerated them using DySER's pipeline parallelism. However, because of data-dependent control flow and irregular memory accesses, we cannot vectorize the code beneficially. The Merge and N-body benchmarks have superfluous computation subregions, so we used region virtualization.

DySER provides a harmonic mean speedup of 3.2× over our baseline, with a range of 1.5× to 15×. Furthermore, it reduces energy consumption by 60 percent, with a range of 33 to 94 percent. It is 1.3× to 4.7× faster than SSE and has similar energy efficiency.

DySER versus GPU. We see various distinct types of behavior with GPU workloads. The sixth column of Table 1 presents details under three categories. Some highlights are as follows:

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

DySER provides a harmonic mean speedup of 3.6× over our baseline, with a range of 1.5× to 8.5×. It reduces energy consumption 64 percent, with a range of 34 percent to 81 percent. It is up to 4× faster than the GPU and 64 percent more energy efficient.

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

Feasibility of implementing and integrating DySER

We implemented DySER as a stand-alone RTL for verification and to determine its feasibility in terms of design, area, and power. To attain an area estimate, we synthesized DySER using Synopsys Design Compiler with the Taiwan Semiconductor Manufacturing Company (TSMC) 55-nm Standard Cell library. For the area of the FP-DIV and SQRT units, we scaled previous estimates[11] to 55 nm. In total, the 64-unit DySER we described earlier occupies an area of 1.54 mm^2. When all are scaled to 55 nm, this is approximately the size of the Intel Atom floating-point and SIMD units (from die photos, Atom's FPC [floating-point cluster] unit is 1.45 mm^2 in 45 nm), and is about half the size of a GPU SM (from die photos, the area of one SM in Nvidia's GT200 is 2.7 mm^2 at 65 nm). The interfaces, switches, and flip-flops contribute to 42 percent of DySER's area and 18 percent of its energy. Overall, DySER is area and energy efficient.

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

The DySER architecture unifies disparate attempts at functionality and parallelism specialization in a single architecture with a set of mechanisms. DySER's unifying functionality and data-parallel specialization mechanisms provide a platform for energy-efficient computing. Our quantitative results show that DySER is competitive or outperforms SIMD and GPU accelerators, performs well in terms of functionality specialization, and is a feasible design that is easy to integrate with a processor.

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

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

Conversely, GPUs are leaning toward the CPU side by providing caches and eliminating redundant work with their scalarization approach, which effectively creates a control core and a set of computing cores, much like DySER's organization. Again, DySER-like integration is the direction in which GPUs appear to be headed.

Hence, DySER is a viable candidate for replacing SIMD short-vector instruction sets. With some simple extensions, we can augment DySER to emulate existing instruction sets such as SSE, thus providing backward compatibility. Clearly, DySER is not a GPU replacement, because it can't perform graphics tasks well. However, it's a promising alternative for design-constrained environments such as Tilera, ARM in servers, and

Oracle's T4 successor targeting high-performance computing. In these cases, a completely new processor design like a GPU, or the integration of a GPU with a core, along with the adoption of a new software ecosystem, could be prohibitively complex. In contrast, DySER's hardware and software ecosystem are nondisruptive. MICRO

References

1. T.J. Callahan, J.R. Hauser, and J. Wawrzynek, ''The GARP Architecture and C Compiler,'' *Computer,* Apr. 2000, pp. 62-69.
2. Z. Ye et al., ''Chimaera: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit,'' *Proc. 27th Int'l Symp. Computer Architecture* (ISCA 00), IEEE CS, 2000, pp. 225-235.
3. N. Clark et al., ''An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors,'' *Proc. 32nd Ann. Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS, 2005, pp. 272-283.
4. S.C. Goldstein et al., ''PipeRench: A Reconfigurable Architecture and Compiler,'' *Computer,* Apr. 2000, pp. 70-77.
5. M. Mishra et al., ''Tartan: Evaluating Spatial Computation for Whole Program Execution,'' *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 06), ACM, 2006, pp. 163-174.
6. S.R. Sarangi, A. Tiwari, and J. Torrellas, ''Phoenix: Detecting and Recovering from Permanent Processor Design Bugs with Programmable Hardware,'' *Proc. 39th Ann. Int'l Symp. Microarchitecture,* IEEE CS, 2006, pp. 26-37.
7. G. Venkatesh et al., ''Conservation Cores: Reducing the Energy of Mature Computations,'' *Proc. 15th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 10), ACM, 2010, pp. 205-218.
8. S. Gupta et al., ''Bundled Execution of Recurring Traces for Energy-Efficient General Purpose Processing,'' *Proc. 44th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* ACM, 2011, pp. 12-23.
9. A. Bakhoda et al., ''Analyzing CUDA Workloads Using a Detailed GPU Simulator,'' *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 09), IEEE CS, 2009, pp. 163-174.
10. S. Li et al., ''McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Many-Core Architectures,'' *Proc. 42nd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* ACM, 2009, pp. 469-480.
11. T.-J. Kwon and J. Draper, ''Floating-Point Division and Square Root Using a Taylor-Series Expansion Algorithm,'' *Microelectronics J.,* vol. 40, no. 11, 2009, pp. 1601-1605.
12. S. Hong and H. Kim, ''An Integrated GPU Power and Performance Model,'' *Proc. 37th Ann. Int'l Symp. Computer Architecture* (ISCA 10), ACM, 2010, pp. 280-289.
13. V. Govindaraju, C.-H. Ho, and K. Sankaralingam, ''Dynamically Specialized Datapaths for Energy Efficient Computing,'' *Proc. IEEE 17th Int'l Symp. High Performance Computer Architecture* (HPCA 11), IEEE CS, 2011, pp. 503-514.

Venkatraman Govindaraju is a PhD student in the Department of Computer Sciences at the University of Wisconsin—Madison. His research interests include energy-efficient computer architecture and compiler techniques for hardware specialization. Govindaraju has an MS in computer science from the University of Wisconsin—Madison. He is a student member of IEEE and the ACM.

Chen-Han Ho is a PhD student in the Department of Computer Sciences at the University of Wisconsin—Madison. His research interests include computer architecture, register-transfer level (RTL) design, and field-programmable gate array (FPGA) prototyping. Ho has a BS in electrical engineering from National Taiwan University. He is a student member of IEEE.

Tony Nowatzki is a PhD student in the Department of Computer Sciences at the University of Wisconsin—Madison. His research interests include architectural specialization, dynamic optimization, and parallelizing compilers. Nowatzki has a BS in computer science and computer engineering from the University of Minnesota. He is a student member of IEEE and the ACM.

Jatin Chhugani is a researcher in the Parallel Computing Lab at Intel. His research interests include developing parallel algorithms for modern architectures. Chhugani has a PhD in computer science from Johns Hopkins University.

Nadathur Satish is a research scientist in the Parallel Computing Lab at Intel. His research interests include next-generation parallel applications and architectures. Satish has a PhD in electrical engineering and computer sciences from the University of California, Berkeley.

Karthikeyan Sankaralingam is an assistant professor in the Departments of Computer Sciences and Electrical and Computer Engineering at the University of Wisconsin–Madison, where he also leads the Vertical Research Group. His research interests include microarchitecture, architecture, and very large-scale integration (VLSI). Sankaralingam has a PhD in computer science from the University of Texas at Austin. He is a senior member of IEEE.

Changkyu Kim is a research scientist at Intel. His research interests include high-performance computing, memory systems, and databases on modern parallel architectures. Kim has a PhD in computer science from the University of Texas at Austin.

Direct questions and comments about this article to Venkatraman Govindaraju, University of Wisconsin–Madison, Department of Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685; venkatra@cs.wisc.edu.