Implementing the Scale Vector-Thread Processor

RONNY KRASHINSKY, CHRISTOPHER BATTEN, and KRSTE ASANOVIĆ
Massachusetts Institute of Technology

The Scale vector-thread processor is a complexity-effective solution for embedded computing which flexibly supports both vector and highly multithreaded processing. The 7.1-million transistor chip has 16 decoupled execution clusters, vector load and store units, and a nonblocking 32KB cache. An automated and iterative design and verification flow enabled a performance-, power-, and area-efficient implementation with two person-years of development effort. Scale has a core area of 16.6 mm^2 in 180 nm technology, and it consumes 400 mW–1.1 W while running at 260 MHz.

Categories and Subject Descriptors: C.5.3 [**Computer System Implementation**]: Microcomputers—*Microprocessors*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*VLSI (very large scale integration)*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*Single-instruction-stream, multiple-data-stream processors; multiple-instruction-stream, multiple-data-stream processors; array and vector processors*

General Terms: Design, Verification

Additional Key Words and Phrases: Vector processors, multithreaded processors, vector-thread processors, iterative VLSI design flow, hybrid C++/Verilog simulation, procedural datapath preplacement

ACM Reference Format:

Krashinsky, R., Batten, C., and Asanović, K. 2008. Implementing the scale vector-thread processor. ACM Trans. Des. Autom. Electron. Syst. 13, 3, Article 41 (July 2008), 24 pages, DOI = 10.1145/1367045.1367050 http://doi.acm.org/10.1145/1367045.1367050

1. INTRODUCTION

As embedded computing applications become more sophisticated, there is a growing demand for high-performance, low-power information processing. Full

This work was funded in part by DARPA PAC/C award F30602-00-2-0562, NSF CAREER award CCR-0093354, and NSF graduate fellowship, donations from Infineon Corporation, and an equipment donation from Intel Corporation.

Authors' addresses: R. Krashinsky; email: ronny@alum.mit.edu; C. Batten, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology; email: cbatten@mit.edu; K. Asanović, Department of Electrical Engineering and Computer Science, University of California, Berkeley; email: krste@eecs.berkeley.edu.

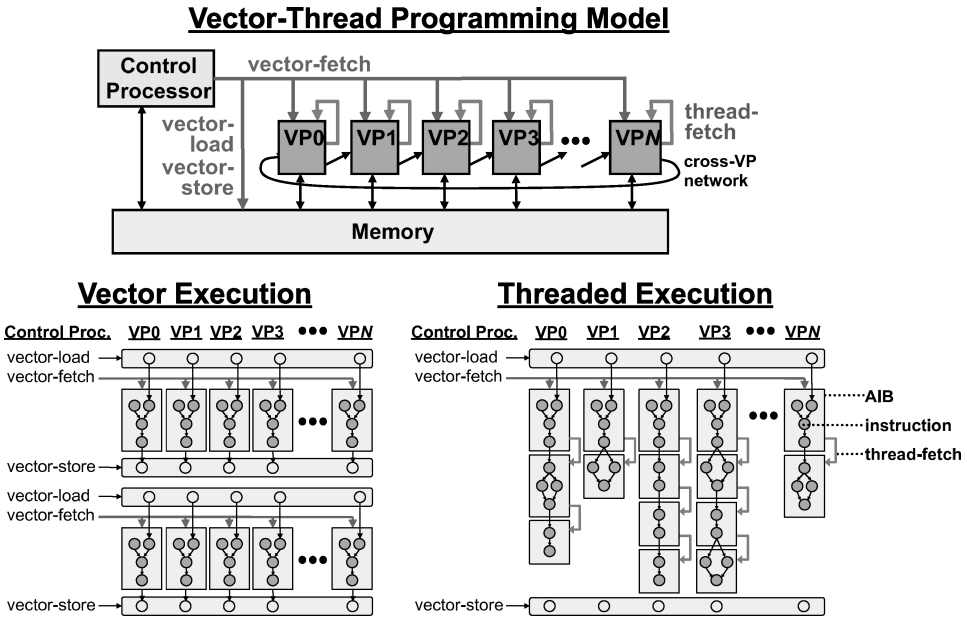


Fig. 1. Vector-thread programming model.

systems often require a heterogeneous mix of general-purpose processors, programmable domain-specific processors, and hardwired application-specific chips. The resulting devices are costly, difficult to program, and inefficient when the application workload causes load imbalance across the cores. We propose the Scale vector-thread (VT) processor as an "all-purpose" programmable solution to the embedded computing challenge.

Figure 1 illustrates the VT programming model. A control processor uses *vector-fetch* commands to broadcast atomic instruction blocks (AIBs) to a vector of virtual processors (VPs), or each VP can use a *thread-fetch* to request its next AIB. A VP thread persists as long as each of its AIBs executes a fetch instruction, and fetches may be predicated to provide conditional branching. Vector loads and stores efficiently move blocks of data in and out of VP registers, while VP loads and stores provide indexed accesses. VT allows software to succinctly expose loop-level parallelism, using vector commands when the control flow is uniform across loop iterations, or threads when the iterations have independent control flow. A cross-VP ring network also enables support for loops with cross-iteration dependencies. By interleaving vector and threaded control at a fine granularity, VT can parallelize more codes than a traditional vector architecture, and can amortize overheads more efficiently than a traditional multithreaded architecture.

The Scale VT processor exploits the parallelism and locality exposed by the VT architectural paradigm to provide high performance with low power and small area. In our prior work we presented detailed descriptions of the Scale architecture [Krashinsky et al. 2004; Batten et al. 2004] and demonstrated through simulation that Scale provides competitive performance across

a wide range of applications. When executing a vectorized IEEE 802.11a wireless transmitter, Scale sustains 9.7 compute operations per cycle. For ADPCM speech decoding, a nonvectorizable kernel with cross-iteration loop dependencies, Scale exploits the available parallelism between loop iterations using decoupled lane execution and vector-loads to achieve 6.5 operations per cycle. For pointer-chasing code like Internet routing-table lookups, Scale uses fine-grained multithreaded execution to achieve 6.1 operations per cycle, while still using efficient vector-loads to feed the threads. These results represent hand-optimized code, but a parallelizing vector-thread compiler has also been developed [Hampton and Asanovic 2008].

Although we had already conducted many simulation studies to evaluate VT, we decided to implement Scale in hardware. Our goals in building the Scale chip are to prove the plausibility of the vector-thread architecture and to demonstrate the performance, power, and area efficiency of the design. Measuring power and area in particular led us to build an ASIC-style chip rather than emulating the design using an FPGA.

The Scale VT architecture is complexity effective, but it is still a relatively complicated processor to build. It includes a scalar control processor; a four-lane vector-thread unit with 16 decoupled execution clusters together with instruction fetch, load/store, and command-management units; a vector-memory access unit with support for unit-stride, strided, and segment loads and stores; and a four-port, nonblocking, 32-way set-associative, 32 KB cache. Scale has 7.14-million transistors, around the same number as the Intel Pentium II, the IBM/Motorola PowerPC 750, or the MIPS R14000. However, as part of a university research project, we could not devote industry-level design effort to the chip-building task. In this article, we describe the implementation and verification flow that enabled us to build an area- and power-efficient prototype with only two person-years of effort.

2. CHIP ARCHITECTURE

Figure 2 shows a block-diagram overview of the Scale VT architecture that we set out to implement. The control processor (CP) is a single-issue pipelined RISC core that is custom designed to interface with the vector-thread unit (VTU). The VTU includes a parallel array of four lanes, each with four execution clusters (C0–C3), a store-data cluster (SD), and a command management unit (CMU). Software is mapped to the VTU using a virtual processor abstraction of a lane. The physical registers in each lane are partitioned among the VPs, and the number of VPs (i.e., the vector length) depends on the software-configured number of registers per VP. With 32 registers per cluster and four lanes, Scale can support up to 128 VP threads.

The virtual processors mapped onto the VTU execute atomic instruction blocks, and the lanes each contain a small 32-entry cluster-partitioned AIB cache. A lane's command-management unit manages the cache and orchestrates refills with the shared AIB-fill unit. Once an AIB is cached, the CMU sends compact execute directives (EDs) to the clusters in the lane. An ED instructs a cluster to execute an AIB for either a particular VP or for all the VPs mapped to the lane, and the cluster issues instructions in the same way

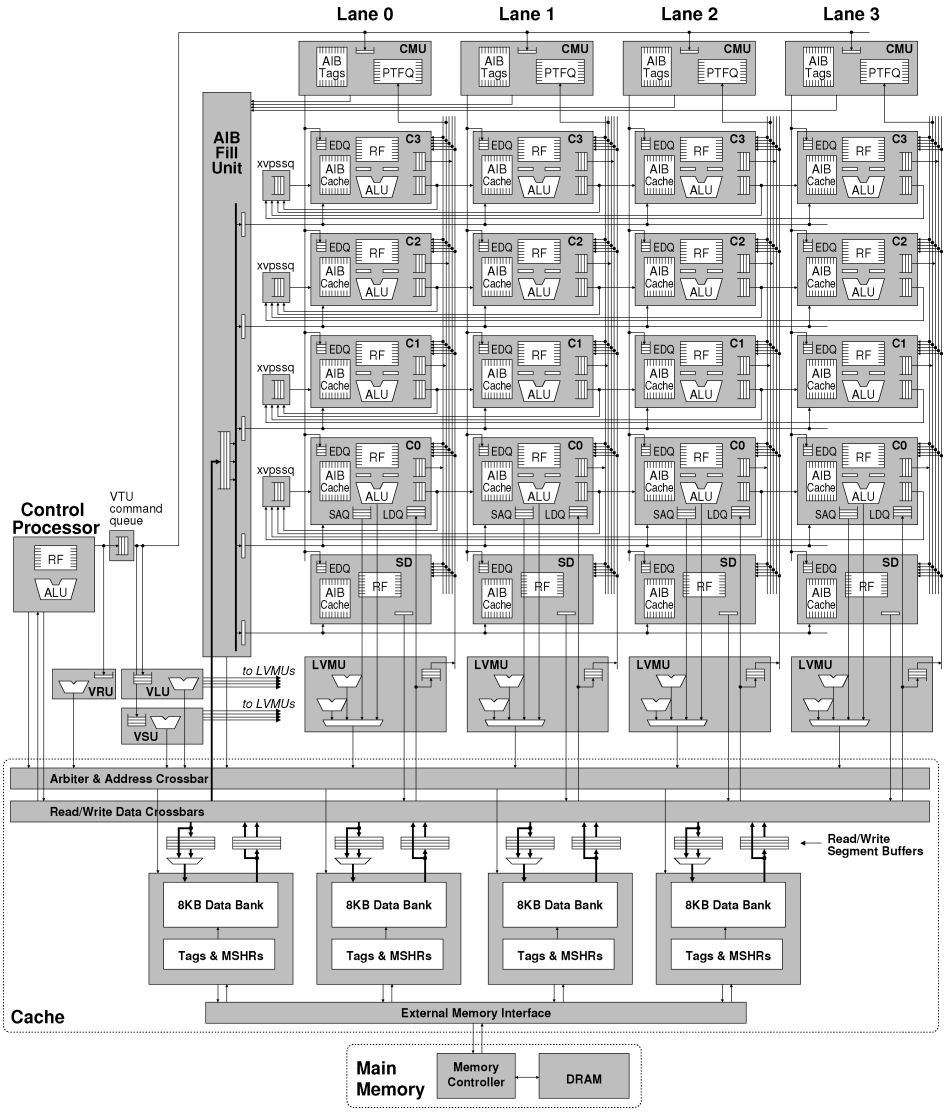


Fig. 2. Scale microarchitecture overview.

of whether regardless, it is executing a vector- or thread-fetched AIB. The clusters in a lane issue instructions independently, and each contains decoupled transport and writeback units to send and receive data to/from other clusters. Some clusters are specialized: C0 can execute loads and stores, C1 can execute thread-fetches, and C3 can execute 16 $\times$ 16-bit multiplies and iterative 32-bit multiplies and divides.

In addition to fetch commands, the control processor can also issue unit-stride and segment-strided vector-memory commands to the VTU. Segments group together several contiguous memory elements for each VP, and segment buffers allow multiple elements from a single cache access to be sent to or from

a lane over several cycles. Scale's decoupled vector-load unit (VLU) and vector-store unit (VSU) process vector-memory commands from the control processor and manage the interface between VTU lanes and the memory system. They each include an address generator for unit-stride accesses, and per-lane address generators for segment-strided accesses. The per-lane portions of the vector-load and -store units are organized as lane-vector memory units (LVMUs). The vector-refill unit (VRU) preprocesses vector-load commands to bring data into the cache [Batten et al. 2004].

The Scale memory system includes four 8 KB cache banks which can each read or write a 128-bit word every cycle. An arbiter chooses between 11 requesters at each cycle, selecting up to four simultaneous accesses to the independent banks. Read and write crossbars transfer data between requesters and data banks. In the basic testing mode for the chip, there are no cache misses, and load and store addresses directly index the RAM.

When caching is enabled, a tag search for an address determines whether the line is present, and, if so, the correct RAM index. The tags in each bank are divided into eight 32-entry CAM subbanks, making the cache 32-way set-associative. The miss-status-handling registers (MSHRs) track outstanding cache misses to enable nonblocking operation. Each of the 32 MSHR entries includes the primary miss address and destination information, and a queue of replay entries to track up to 4 secondary misses to the in-flight cache line. The external memory-interface block arbitrates between cache-bank misses and sends cache-line refill and writeback requests over the chip I/O pins.

Finally, Scale includes a host interface block (HTIF) which uses a low-bandwidth asynchronous protocol to communicate with an external host over the chip I/O pins. This block allows the host to read and write memory and to interact with the Scale control processor. We use this interface to download and run programs and to service operating-system calls (e.g., file I/O).

3. CHIP IMPLEMENTATION

The Scale design contains about 1.4 million gates, a number around 60 times greater than that of a simple RISC processor. To implement Scale with limited manpower, we leverage Artisan standard cells and RAM blocks in an ASIC-style flow targeting TSMC's 180 nm 6 metal-layer process technology (CL018G). We use Synopsys Design Compiler for synthesis and Cadence SoC Encounter for place-and-route.

In establishing a tool flow to build the Scale chip, we strove to make the process as automated as possible. A common design approach is to freeze blocks at RTL level and then push them through to lower levels of implementation using hand-tweaked optimizations along the way. Instead, our automated flow enables an iterative chip-building approach more similar to software compilation. After a change in the top-level RTL, we simply run a new iteration of synthesis and place-and-route to produce an updated full chip layout, even including power and ground routing. Despite this level of automation, the tool flow still allows us to preplace datapath cells in regular arrays, to incorporate optimized memories and custom circuit blocks, and to easily and flexibly provide a custom floorplan for the major blocks in the design.

With our iterative design flow, we repeatedly build the chip and optimize the source RTL or floorplan based on the results. We decided to use a single-pass flat tool flow in order to avoid the complexities involved in creating and instantiating hierarchical design blocks. As the design grew, the end-to-end chip-build time approached two days. We are satisfied with this methodology, but we would use a hierarchical design flow to avoid prohibitive runtimes for a design larger than Scale.

3.1 RTL Development

Our early research studies on Scale included the development of a detailed microarchitecture-level simulator written in C++. This simulator is flexible and parameterizable, allowing us to easily evaluate the performance impact of many design decisions. The C++ simulator is also modular with a design hierarchy that we carried over to our hardware implementation.

We use a hybrid C++/Verilog simulation approach for the Scale RTL. After implementing the RTL for a block of the design, we use Tenison VTOC to translate the Verilog into a C++ module with input and output ports and a clock-tick evaluation method. We then wrap this module with the necessary glue logic to connect it to the C++ microarchitectural simulator. Using this methodology we are able to avoid constructing custom Verilog test harnesses to drive each block as we develop the RTL. Instead, we leverage our existing set of test programs as well as our software infrastructure for easily compiling and running directed test programs. This design approach allowed us to progressively expand the RTL code base from the starting point of a single cluster, to a single lane, to four lanes; and then to add the AIB-fill unit, the vector-memory unit, the control processor, and the memory system.We did not reach the milestone of having an all-RTL model capable of running test programs until about fourteen months into the hardware implementation effort, five months before tapeout. The hybrid C++/Verilog approach was crucial in allowing us to incrementally test and debug the RTL throughout the development.

3.2 Datapath Preplacement

Manually organizing cells in bit-sliced datapaths improves timing, area, and routing efficiency compared to automated placement [Chinnery and Keutzer 2002]. However, to maintain an iterative design flow, a manual approach must still easily accommodate changes in the RTL or chip floorplan. We used a C++-based procedural datapath tiler which manipulates standard cells and creates design databases using the OpenAccess libraries. The tool allows us to write code to instantiate and place standard cells in a virtual grid and create nets to connect them together. After constructing a datapath, we export a Verilog netlist together with a DEF file with relative placement information.

We incorporate datapath preplacement into our CAD tool flow by separating out the datapath modules in the source RTL. For example, the cluster datapaths for Scale include the ALU, shifter, and many 32-bit muxes and latches. We then write tiler code to construct these datapaths and generate cell netlists. During synthesis we provide these netlists in place of the source RTL for the datapath

```
wire [31:0] xw_sd_pn;
wire [31:0] w_sd_np;

dpMux2 #(32)
  xw_sd_mux
    ( .s    (xw_sdsel_pn),
      .i0   (xw_reg_pn),
      .i1   (xw_sdc_pn),
      .o    (xw_sd_pn));

dpLatch_h_en #(32)
  w_sd_hl
    ( .clk  (clk),
      .d_n  (xw_sd_pn),
      .en_p (x_sden_np),
      .q_np (w_sd_np));
```

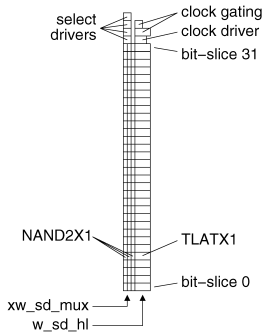
(a) Verilog RTL

```
Tiler tl;
tl.addNet  ( "xw_sd_pn", 32 );
tl.addNet  ( "w_sd_np", 32 );

dpMux2 ( tl, Drive::X1, 32,
    "xw_sd_mux",
    "xw_sdsel_pn",
    "xw_reg_pn",
    "xw_sdc_pn",
    "xw_sd_pn");

dpLatch_h_en ( tl, Drive::X1, 32,
    "w_sd_hl",
    "clk",
    "xw_sd_pn",
    "x_sden_np",
    "w_sd_np");
```

(b) C++ preplacement code



(c) datapath cells

Fig. 3. Datapath preplacement code example.

modules, and we flag the preplaced cells as `dont_touch`. In this way, Design Compiler can correctly optimize logic which interfaces with the datapath blocks. During the floorplanning step before place-and-route, we use scripts to flexibly position each datapath wherever we want on the chip. These scripts process the relative placement information in the datapath DEF files, combining these into a unified DEF file with absolute placement locations. We again use `dont_touch` to prevent Encounter from modifying the datapath cells during placement and optimization.

As a simple example of the ease with which we can create preplaced datapath arrays, Figure 3(a) shows a small snippet of Verilog RTL from Scale which connects a 32-bit mux with a 32-bit latch. Figure 3(b) shows the corresponding C++ code which creates the preplaced datapath diagrammed in Figure 3(c). The placement code is simple and very similar to the RTL; the only extra information is the output drive strength of each component. The supporting component-builder libraries (`dpMux2` and `dpLatch_h_en`) each add a column of cells to the virtual grid in the tiler (`tl`). By default, the components are placed from left to right. In this example, the `dpMux2` builder creates each two-input multiplexer using three NAND gates. The component builders also add the necessary clock-gating and driver cells on top of the datapath, and the code automatically sets the size of these based on the bit-width of the datapath.

We used our datapath preplacement infrastructure to create parameterizable builders for components like muxes, latches, queues, adders, and shifters. It is relatively straightforward to assemble these components into datapaths, and easy to modify the datapaths as necessary. Figure 4 highlights the preplaced cells in a plot of a Scale cluster. In the end, we preplaced 230-thousand cells, 58% of all standard cells in the Scale chip.

3.3 Memory Arrays

Wherever possible, we use single-port and two-port (read/write) RAMs created by the Artisan memory generators. These optimized memories provide high-density layout and routing with six or eight transistors per bit-cell. To

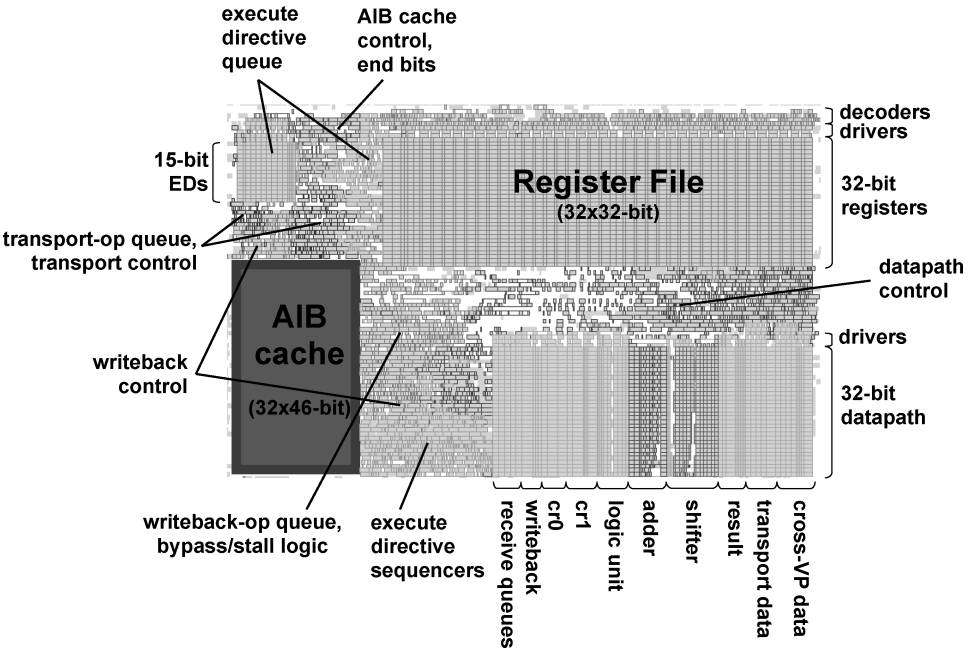


Fig. 4. Plot of basic cluster implementation (cluster 2). The register file and datapath use preplaced cells, and the AIB cache is a generated RAM block.

incorporate the RAM blocks into our CAD tool flow, we use behavioral RTL models for simulation. These are replaced by the appropriate RAM data files during synthesis and place-and-route.

Artisan does not provide memory generators suitable for Scale's register files and CAM arrays. We considered using custom circuit design for these, but decided to reduce design effort and risk by building these components out of Artisan standard cells. The two-read-port two-write-port register file bit-cell is constructed by attaching a mux cell to the input of a latch cell with two tri-state output ports. We found that the read ports could effectively drive 8 similar bit-cells. Since our register file has 32 entries, we made the read ports hierarchical by adding a second level of tri-state drivers. For the CAM bit-cell we use a latch cell combined with an XOR gate for the match. The OR reduction for the match comparison is implemented using a two-input NAND/NOR tree which is compacted into a narrow column next to the multibit CAM entry. For the cache-tag CAMs, which have a read port in addition to the match port, we add a tri-state buffer to the bit-cell and use hierarchical bit-lines similar to those in the register file.

We pay an area penalty for building the register file and CAM arrays out of standard cells. We estimate that our constructed bit-cells are 3–4 times larger than a custom bit-cell. However, since these arrays occupy about 17% of the core chip area, the overall area impact is limited to around 12%. We found that these arrays are generally fast enough to be off of the critical path. For example, a register-file read only takes around 1.4 ns (16 fan-out-of-four gate

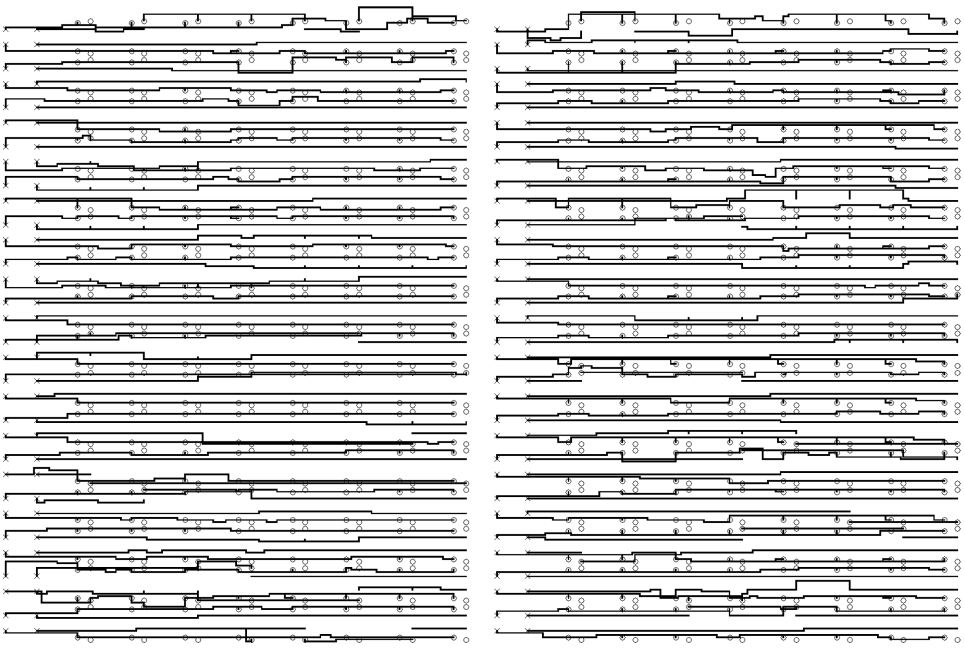


Fig. 5. Routing of read bit-lines in a register file. The section shown covers 32 bits in the vertical direction and 16 registers in the horizontal. Each bit-line connects to a group of 8 registers. Top-level hierarchical bit-lines are not shown.

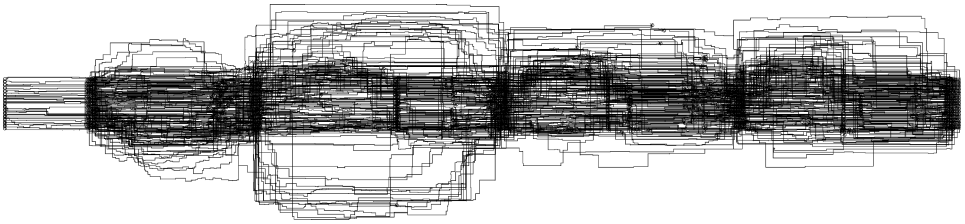


Fig. 6. Routing of cluster transport data-busses in a lane. Each cluster broadcasts data on a 32-bit bus that connects to every other cluster in the lane. These busses are shown in Figure 2 (in the vertical direction).

delays), including the decoders. The power consumption should also be competitive with custom design, especially since our arrays use fully static logic instead of precharged dynamic bit-lines and matchlines.

3.4 Routing

We use Encounter to do all routing automatically, including the preplaced datapaths. This avoids the additional effort of routing by-hand, and we have found that the tool does a reasonable job after the datapath arrays have been preplaced. Figure 5 shows local routing of read bit-lines in a register file, and Figure 6 shows global routing of the cluster transport data-busses in a

lane. Local routes are relatively direct, while global routes can be somewhat meandering due to congestion.

3.5 Clocking

Scale uses a traditional two-phase clocking scheme which supports both latches and flip-flops. The datapaths primarily use latches to enable time-borrowing across clock phases, as eliminating hard clock-edge boundaries makes timing optimization easier. Also, the preplaced register file, CAM, and RAM arrays use latches to reduce area. A standard-cell latch optimized for memory arrays with a tri-state output is around 41% smaller than a flip-flop. The control logic for Scale primarily uses flip-flops, as this makes the logic easier to design and reason about.

The Scale design employs aggressive clock-gating to reduce power consumption. The architecture contains many decoupled units, and typically several are idle at any single point in time. Control signals for operation liveness and stall conditions are used to determine which pipeline registers must be clocked at each cycle. In many cases, these enable signals can become critical paths, especially when gating the clock for low latches and negative-edge triggered flip-flops and RAMs. We make the enable condition conservative where necessary in order to avoid slowing down the clock frequency. We considered adding global clock-gating for entire clusters in Scale, but abandoned this idea, due to the critical timing and complexity of ensuring logical correctness of the enable signal.

We incorporate clock-gating logic into the preplaced datapath components. Typically, an enable signal is latched and used to gate the clock before it is buffered and broadcast to each cell of a multibit flip-flop or latch. This reduces the clock load by a factor approximately equal to the width of the datapath, for example, 32× for a 32-bit latch. We implement similar gating within each entry in the register file and CAM arrays, but we also add a second level of gating which only broadcasts the clock to all entries when a write operation is live.

For clock-gating within the synthesized control logic, we use Design Compiler's automatic `insert_clock_gating` command. To enable this gating, we simply make the register updates conditional, based on the enable condition in the source Verilog RTL. Design Compiler aggregates common enable signals and implements the clock-gating in a similar manner to our datapath blocks. Out of 19,800 flip-flops in the synthesized control logic, 93% are gated, with an average of about 11 flip-flops per gate. This gating achieves a 6.5 times reduction in clock load for the synthesized logic when the chip is idle.

In addition to gating the clock, we use data gating to eliminate unnecessary toggling in combinational logic and on long buses. For example, where a single datapath latch sends data to the shifter, adder, and logic unit, we add combinational AND gates to the bus so as to only enable the active unit at each cycle.

3.6 Clock Distribution

The clock on the Scale chip comes from a custom-designed voltage-controlled oscillator that we have used successfully in previous chips. The VCO frequency

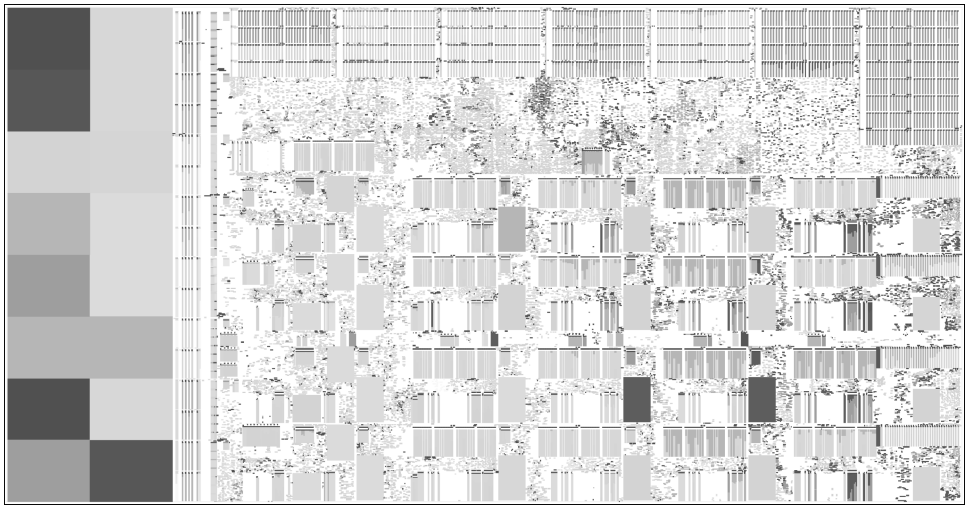


Fig. 7. Chip clock-skew. All of the clocked cells are highlighted.

is controlled by the voltage setting on an analog input pad, and has its own power pad for improved isolation. We can also optionally select an external clock input, and divide the chosen root clock by any integer from 1–32. There are a total of around 94,000 flip-flops and latches in the design. An 11-stage clock tree is built automatically by Encounter using 688 buffers (not counting those buffers in the preplaced datapaths and memory arrays). The delay is around 1.5–1.9 ns, the maximum trigger-edge skew is 233 ps, and the maximum transition time at a flip-flop or latch is 359 ps. We use Encounter to detect and fix any hold-time violations.

Figure 7 highlights all of the clocked cells in Scale and shows their relative clock-skew. A regular cell layout in the preplaced datapaths and memory arrays makes it easier for the clock-tree generator to route the clock while minimizing skew. Those few cells with high skew have an intentionally delayed clock edge to allow time-borrowing across pipeline stages.

3.7 Power Distribution

Standard-cell rows in Scale have a height of nine metal-3 or metal-5 tracks. The cells get power and ground from metal-1 strips that cover two tracks, including the spacing between tracks. The generated RAM blocks have power and ground rings on either metal-3 (for two-port memories) or metal-4 (for single-port SRAMs).

A power distribution network should supply the necessary current to transistors while limiting the voltage drop over resistive wires. We experimented with some power-integrity analysis tools while designing Scale, but mainly we relied on conservative analysis and an overdesigned power distribution network to avoid problems. We tried running fat power strips on the upper metal-layers of the chip, but these create wide horizontal routing blockages where vias connect

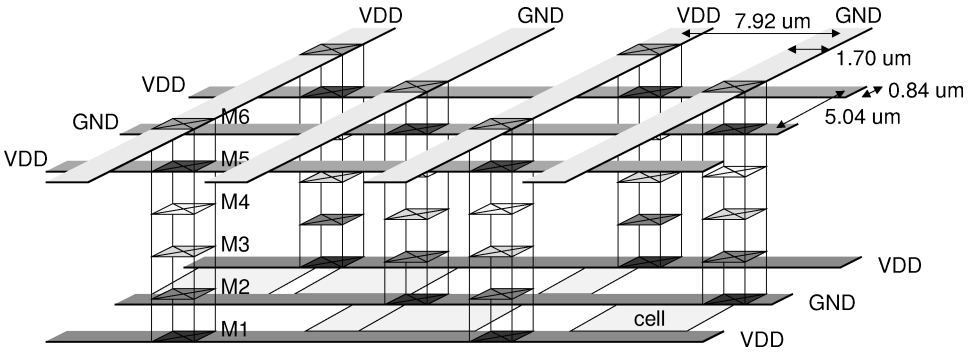


Fig. 8. Chip power grid. A small section of two standard-cell rows is shown.

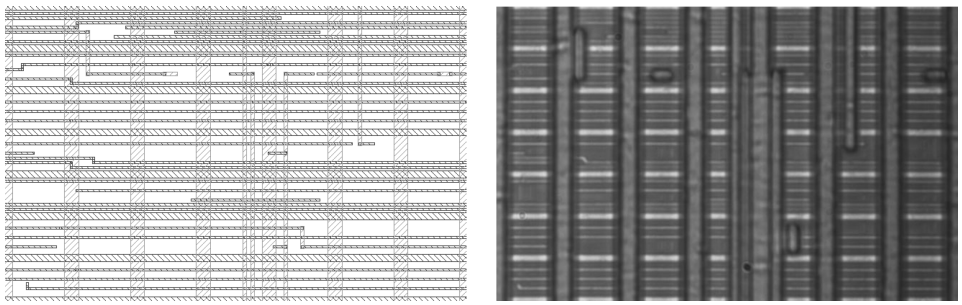


Fig. 9. Plot and corresponding photomicrograph showing chip power grid and routing. Metal layers 5 (horizontal) and 6 (vertical) are visible, and the height is seven standard-cell rows (about 35 μm).

down to the standard-cell rows. These wide blockages cause routing problems, especially when falling within the preplaced datapath arrays.

We settled on a fine-grained power distribution grid over the entire Scale chip; a diagram is shown in Figure 8. In the horizontal direction we route alternating power and ground strips on metal-5, directly over the metal-1 strips in the standard-cell rows. These use two routing tracks, leaving seven metal-3 and metal-5 tracks unobstructed for signal routing. In the vertical direction, we route alternating power and ground strips on metal-6. These cover three metal-2 and metal-4 tracks, leaving nine unobstructed for signal routing. Figure 9 shows a plot and photomicrograph of the power grid, including some data signals that are routed on metal layers 5 and 6. Overall, the power distribution uses 21% of the available metal-6 area and 17% of metal-5. We estimate that this tight power grid can provide more than enough current for Scale, and we have found the automatic router to deal well with the small blockages.

Another power-supply concern is the response time of the network to fast changes in current (di/dt). At frequencies on the order of a few 100 MHz, the main concern is that the chip has enough on-chip decoupling capacitance to supply the necessary charge after a clock edge. Much of the decoupling capacitance comes from filler cells which are inserted into otherwise unused space on the chip. Around 20% of standard-cell area on the chip is unused, and filler cells with decoupling capacitance are able to fit into 80% of this unused area, a

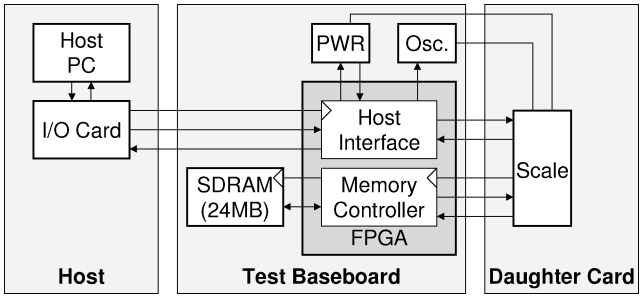


Fig. 10. Block diagram of Scale test infrastructure.

total of 1.99 mm^2. Scale uses a maximum of around 2.4 nC (nano-coulombs) per cycle, and we estimate that the total on-chip charge storage is around 21.6 nC. This means that even with no response from the external power pads, a single cycle can only cause the on-chip voltage to drop by around 11%. We deemed this an acceptable margin, and we rely on the external power pads to replenish the supply before the next cycle.

3.8 System Interface

In order to reduce design effort, we decided to reuse an existing test platform to evaluate the Scale chip. This allowed us to avoid building a custom test-board, but as a consequence Scale's memory system is limited by the capabilities of surrounding components. Although low memory bandwidth can limit application performance, we can still fully evaluate the Scale chip by running test programs that do not stress the external memory system.

The chip test infrastructure includes three primary components: a host computer, a general test baseboard, and a daughter card with a socket for the Scale chip (see Figure 10). To plug into this socket, Scale uses a 121-pin ceramic package (PGA-121M). The test baseboard includes a host interface and a memory controller implemented on a Xilinx FPGA, as well as 24MB of SDRAM, adjustable power supplies with current measurement, and a tunable oscillator. Using this test setup, the host computer is able to download and run programs on Scale while monitoring the power consumption at various voltages and frequencies.

The Scale chip has 32-bit input and output ports to connect to an off-chip memory controller. This interface runs synchronously with the Scale clock, at a configurable divided frequency. Along with the data, Scale sends a clock for the memory controller, and Scale's clock generator can digitally tune the relative frequency and phase alignments between the internal chip clock, external memory-controller clock, and data bus. We designed the input and output ports to operate in 8-, 16-, or 32-bit modes, depending on the capabilities of the surrounding system. The existing test infrastructure limits the memory interface to 16-bits in each direction.

With its nonblocking cache, the Scale chip itself has the ability to drive a high-bandwidth memory system. However, the SDRAM in the test infrastructure is

only 12-bits wide and the frequency of the memory controller is limited by the outdated FPGA. The few bits of memory bandwidth per Scale clock cycle will certainly limit performance for applications with working sets that do not fit in Scale's 32 KB cache. However, for prototyping purposes we can emulate systems with higher memory bandwidths. To accomplish this, we use Scale's configurable clock generator to run the processor at a lower frequency so that the memory system becomes relatively faster. The chip also supports a special DDR mode in which data is transferred on both edges of the Scale clock. This allows us to emulate a memory bandwidth of up to 4 bytes in and out per Scale cycle.

For testing purposes, the Scale chip supports an *on-chip RAM mode* in which cache misses are disabled, and the memory system is limited to the 32 KB of RAM on chip. In this mode, the cache arbiter and datapaths are still operational, but all accesses are treated as hits. As the tapeout deadline approached, it became apparent that the frequency of the design was limited by some long paths through the cache tags and cache-miss logic, which we did not have time to optimize. Since the primary goal for the chip is to demonstrate the performance potential of the vector-thread unit, and given the memory-system limitations imposed by the testing infrastructure, we decided to optimize timing for the on-chip RAM mode of operation. Unfortunately, this meant that the tools essentially ignore some of the cache paths during the timing optimization and, as a result, the frequency when the cache is enabled is not as fast as it could be.

4. CHIP VERIFICATION

We ensure correct operation of the Scale chip using a three-step process: First, we establish that the source RTL model is correct. Second, we prove that the chip netlist is equivalent to the source RTL. Third, we verify that the chip layout matches the netlist.

4.1 RTL Verification

Our overall strategy for verifying the Scale RTL model is to compare its behavior to the high-level Scale ISA simulator. We use the full chip VTOC-generated RTL model, and the C++ harness downloads programs over the chip I/O pins through the host interface block. After the program runs to completion, the output is read from memory using this same interface and then compared to reference outputs from the Scale ISA simulator. In addition to a suite of custom-directed tests and the set of application benchmarks for Scale, we developed VTorture, a random-test-program generator. The challenge in generating random tests is to create legal programs that also stress different corner cases in the design. VTorture randomly generates relatively simple instruction sequences of various types, and then randomly interleaves these sequences to construct complex, yet correct, programs. By tuning parameters which control the breakdown of instruction-sequence types, we can stress different aspects of Scale. Although our full chip RTL simulator runs at a modest rate on the order of 100 cycles per second, we used a compute farm to simulate over a billion cycles of the chip running test programs.

In addition to running test programs on the full RTL model, we also developed a test harness to drive the Scale cache on its own. We use both directed tests and randomly generated load and store requests to trigger the many possible corner cases in the cache design.

The VTOC-based RTL simulator models the design using two-state logic. This is sufficient for most testing; however, we must be careful not to rely on uninitialized state after the chip comes out of reset. To complete the verification, we use Synopsys VCS as a four-state simulator. We did not construct a new test harness to drive VCS simulations. Instead, we use the VTOC simulations to generate value-change-dump (VCD) output which tracks the values on chip I/O pins at every cycle. Then we drive a VCS simulation with these inputs, and verify that the outputs are correct at every cycle. Any use of uninitialized state will eventually propagate to `X`'s on the output pins, and in practice we did not find it very difficult to track down the source.

Unfortunately, the semantics of the Verilog language allow unsafe logic in which uninitialized `X` values can propagate to valid 0/1 values [Turpin 2003]. In particular, `X`'s are interpreted as 0's in conditional `if` or `case` statements. We dealt with this by writing the RTL to carefully avoid these situations when possible, and by inserting assertions (error messages) to ensure that important signals are not `X`'s. For additional verification that the design does not rely on uninitialized state, we run two-state VTOC simulations with the state initialized with all 0's, all 1's, and random values.

4.2 Netlist Verification

The preplaced datapaths on Scale are constructed by-hand, and therefore error prone. We had great success using formal verification to compare the netlists from preplaced datapaths to corresponding datapath modules in the source RTL. The verification is done in seconds using Synopsys Formality.

As a double-check on the tools, we also use formal verification to prove equivalence of the source RTL to both the postsynthesis netlist and final netlist after place-and-route. These Formality runs take many hours and use many gigabytes of memory, but remove the need for gate-level simulation.

4.3 Layout Verification

The final verification step is to check the GDSII layout for the chip. We convert the final chip netlist from Verilog to a SPICE format and use Mentor Graphics Calibre to run a layout-versus-schematic (LVS) comparison. This check ensures that the transistors on the chip match the netlist description. LVS can catch errors such as routing problems which incorrectly short nets together. We also use Calibre to run design-rule checks (DRC) and to extract a SPICE netlist from the layout.

We use Synopsys Nanosim to simulate the extracted netlist as a final verification that the chip correctly operates. This is the only step which actually tests the generated Artisan RAM blocks, so it serves as an important check on our high-level Verilog models for these blocks. The Nanosim setup uses VCD traces to drive inputs and to check outputs at every cycle (as in VCS simulation,

Table I. Scale Chip Statistics

process technology	TSMC 180 nm, 6 metal layers
transistors	7.14M
gates (equivalent metric as reported by Encounter)	1.41M
standard cells	397K
flip-flops and latches	94K
RAM bits	300K
core area	16.61 mm^2
chip area	23.14 mm^2
frequency at 1.8 V (on-chip RAM mode)	260 MHz
frequency at 1.8 V (caching-enabled mode)	180 MHz
power at 1.8 V, 260 MHz	0.4–1.1 W
design time	19 months
design effort	24 person-months

the VCD is generated by simulations of the source RTL). To get the I/O-timing correct, these simulations actually drive the core chip module, which does not include the clock generator. We crafted separate Nanosim simulations to drive the chip I/O pins and verify the clock-generator module. We also use Nanosim simulations to estimate power consumption.

5. CHIP RESULTS AND MEASUREMENTS

We began RTL development for Scale in January of 2005, and taped-out the chip on October 15, 2006. Subtracting 2 months during which we implemented a separate test chip, the design time was 19 months, and we spent about 24 person-months of effort. We received chips back from fabrication on February 8, 2007, and were able to run test programs the same day. The off-chip memory system was first operational on March 29, 2007. The first silicon has proven fully functional with no known bugs after extensive lab testing. Figure 11 shows a die photo of the fabricated chip and Table I summarizes the design statistics.

5.1 Area

Scale's core area is 16.61 mm^2 in 180 nm technology. If implemented in 45 nm technology, Scale would only use around 1 square millimeter of silicon area, allowing hundreds of cores to fit in a single chip. Even with the control overheads of vector-threading and its nonblocking cache, Scale's 16 execution clusters provide relatively dense computation. The core area is around the same size as a single tile in the 16-tile RAW microprocessor [Taylor et al. 2003]. In 130 nm technology, around 40 Scale cores could fit within the area of the two-core TRIPS processor [Sankaralingam et al. 2006]. In 90 nm technology, Scale would be around 28% the size of a single 14.8 mm^2 SPE from the 8-SPE cell processor [Flachs et al. 2006]. These comparisons are not one-to-one, since the various chips provide different compute capabilities (e.g., floating-point) and have different amounts of on-chip RAM. Nonetheless, Scale's compute density is at least competitive with these other many-operation-per-cycle chips.

We initially estimated that the Scale core would be around 10 mm^2 [Krashinsky et al. 2004]. This estimate assumed custom layout for cache tags

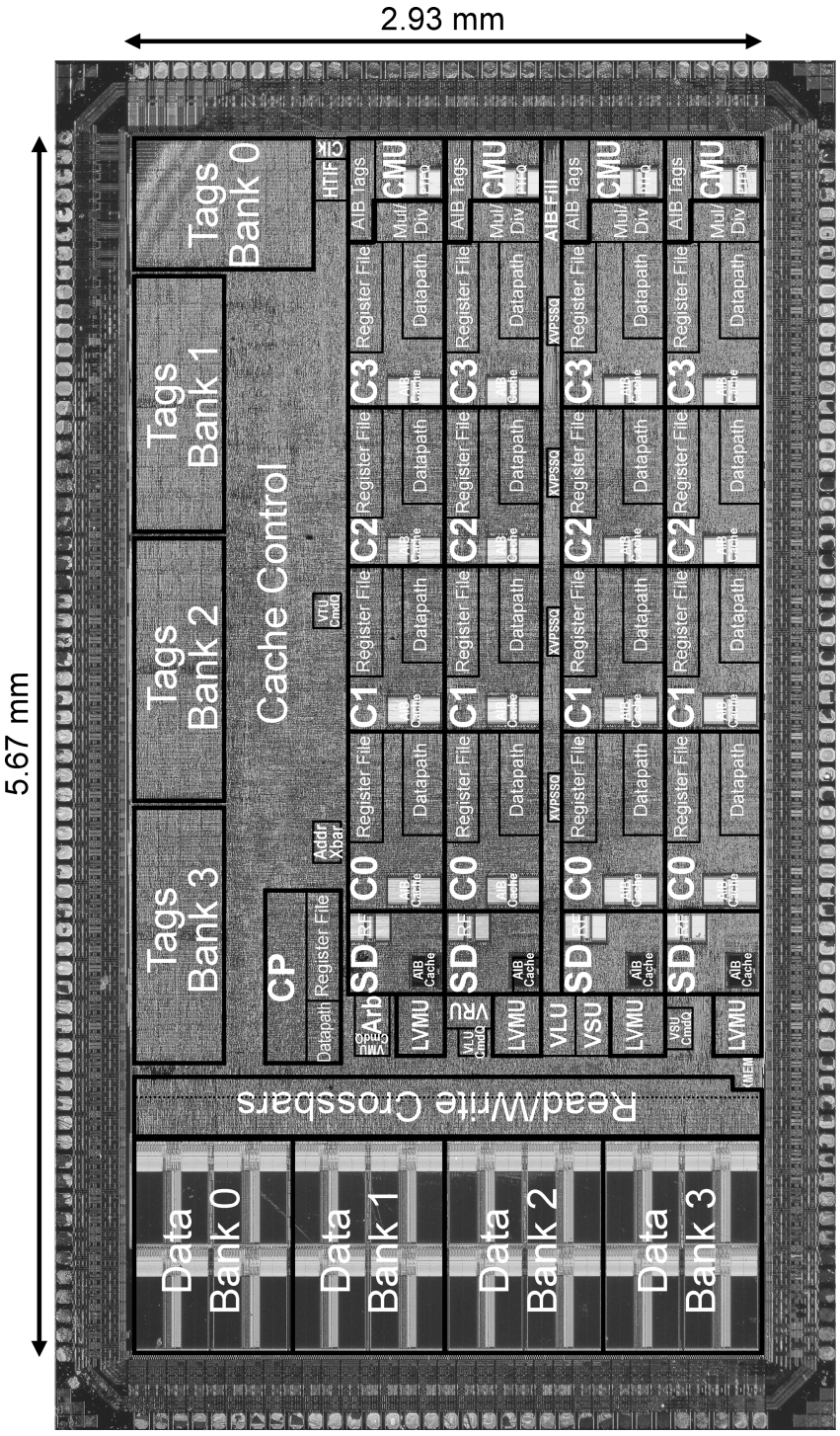


Fig. 11. Scale die photo.

Table II. Scale Area Breakdown

	Gates (thousands)	Gates percentage			Cells (thousands)	RAM bits
Scale (total)	1407.0	100.0			397.0	300032
Control Processor	28.8	2.0	100.0		10.9	
Register File	9.8		34.0		3.0	
Datapath	4.7		16.3		1.8	
Mult/Div	2.5		8.7		1.2	
VLMAX	1.4		4.9		0.8	
Control Logic	10.3		35.8		4.1	
Vector-Thread Unit	678.2	48.2	100.0		211.4	33792
VTU command queues	11.2		1.7		4.0	
VLU	3.1		0.5		1.7	
VSU	3.0		0.4		1.5	
VRU	3.0		0.4		1.4	
AIB Fill Unit	12.8		1.9		5.3	
XVPSSQs	5.6		0.8		2.0	
Lane (4×)	159.6		23.5	100.0	48.8	8448
CMU	13.4			8.4	4.2	896
C0	31.8			19.9	10.0	1472
C1	28.5			17.9	8.5	1472
C2	28.1			17.6	8.4	1472
C3	35.6			22.3	11.4	1472
SD	13.5			8.5	2.8	1664
LVMU	6.6			4.1	2.7	
Memory System	688.7	48.9	100.0		171.2	266240
Data Arrays	287.6		41.8			266240
Tag Arrays	177.1		25.7		83.1	
Tag Logic	61.6		8.9		33.0	
MSHRs	43.5		6.3		15.2	
Arbiter	1.7		0.2		0.9	
Address Crossbar	1.1		0.2		0.6	
Read Crossbar	50.7		7.4		13.0	
Write Crossbar	27.1		3.9		11.7	
Other	38.3		5.6		13.7	
Host Interface	3.1	0.2			1.1	
Memory Interface	2.1	0.1			0.8	

The "Gates" metric is an approximation of the number of circuit gates in each block as reported by the Cadence Encounter tool. Different subbreakdowns are shown in each column of the "Gates percentage" section. The "Cells" metric reports the actual number of standard cells, and the "RAM bits" metric reports the bit count for the generated RAM blocks.

and register files, whereas the actual Scale chip builds these structures out of standard cells. The estimate was also based on a datapath-cell library which was more optimized for area. Furthermore, the initial study was made before any RTL was written for the Scale chip, and the microarchitecture has evolved considerably since then. For example, we added the store-data cluster and refined many details in the nonblocking cache design.

Tables II and III show the area breakdown for the Scale chip and for an individual cluster, respectively. The area is split roughly evenly between the

Table III. Cluster Area Breakdown

	Gates	Gates percentage	Cells
Cluster 2 (total)	28134	100.0	8418
AIB Cache RAM	4237	15.1	
AIB Cache Control	376	1.3	129
Execute Directive Queue	908	3.2	351
Execute Directive Sequencers	1221	4.3	526
Writeback-op Queue, Bypass/Stall	1231	4.4	410
Writeback Control	582	2.1	227
Register File	10004	35.6	3013
Datapath	6694	23.8	2605
Datapath Control	2075	7.4	828
Transport Control	472	1.7	188

The data is for cluster 2, a basic cluster with no special operations. Columns are as in Table II.

vector-thread unit and the memory system, with the control processor only occupying around 2% of the total. As somewhat technology-independent reference points, the control processor uses a gate-equivalent area of around 3.3 KB of SRAM, and the vector-thread unit uses the equivalent of around 76.6 KB of SRAM. The actual area proportions on the chip are somewhat larger than this, since standard-cell regions have lower area utilization than SRAM blocks.

Support for vector-thread commands does not add undue complexity to the control processor. The largest area overhead is the logic to compute the vector length (`vlmax`), which takes up around 5% of the CP area. Even including the VT overhead, the baseline architectural components of the control processor (i.e., the 32-entry register file, the adder, logic unit, shifter datapath components, and the iterative multiply/divide unit) occupy around half of its total area.

The vector-thread unit area is dominated by the lanes. Together, the VTU command queues, vector-load unit, vector-store unit, vector-refill unit, and AIB-fill unit use less than 5% of VTU area. Within a lane, the compute clusters occupy 79% of the area, and the command-management unit, lane-vector memory unit, and store-data cluster occupy the remainder. The area of a basic cluster (c2) is dominated by the register file (36%), datapath (24%), and AIB cache (15%). The execute-directive queue uses 3% of area, and the synthesized control logic makes up the remaining 22%.

It is also useful to consider a functional breakdown of the VTU area. Register files in the VTU take up 26% of its area, and the arithmetic units, including cluster adders, logic units, shifters, and multiply/divide units, take up 9%. Together, these baseline compute and storage resources comprise 35% of the VTU area. In a vector-thread processor with floating-point units or other special compute operations, this ratio would be even higher. The vector memory-access units (the LVMUs, and the VLU, VSU, VRU, and their associated command queues) occupy 6% of the total, and the memory-access resources in cluster 0 and the store-data cluster add another 3%. The AIB-caching resources, including the AIB-cache RAMs, tags, control, and the AIB-fill unit, comprise 19% of the total. The vector- and thread-command control overhead (the VTU command queue, execute-directive queues and sequencers, the pending-thread-fetch queue, and

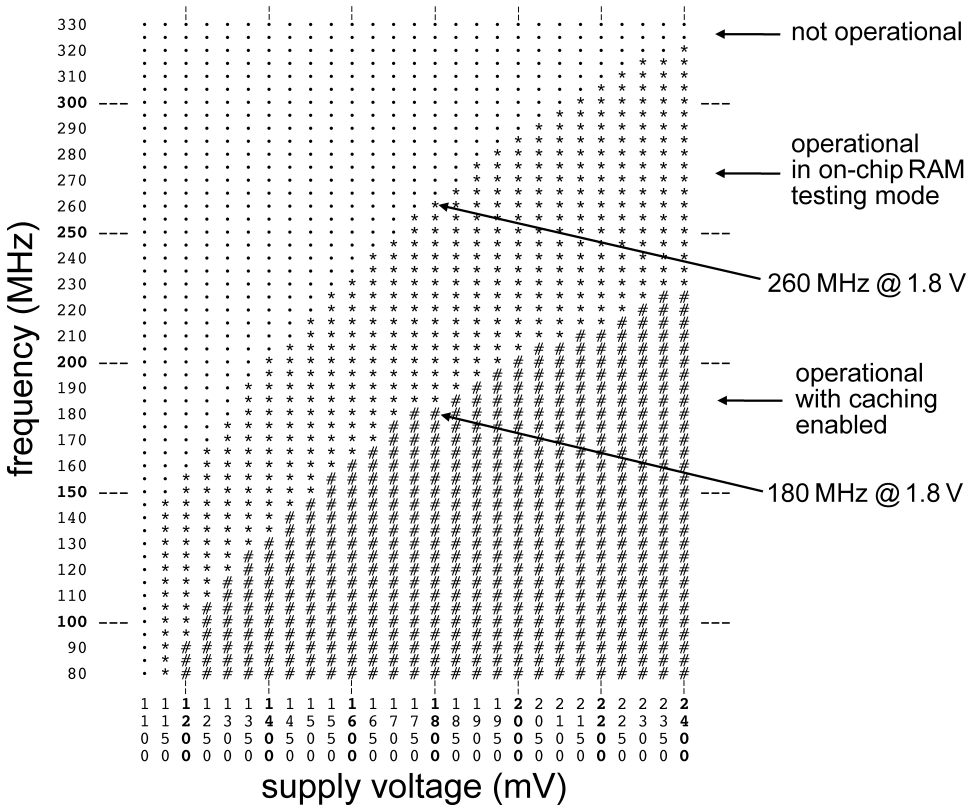


Fig. 12. Shmoo plot of frequency versus supply voltage.

other logic in the CMUs) together take up 11% of VTU area. The writeback-decoupling resources in the VTU, including datapath latches and bypass/stall logic for the writeback-op queue, take up 8% of the area, and the transport-decoupling resources take up 4%. The cross-VP queues in the lanes and the cross-VP start/stop queues together occupy 3%. The remaining 11% of VTU area includes the cluster-chain registers, predicate registers, and other pipeline latches, muxes, and datapath-control logic.

5.2 Frequency

Figure 12 shows a shmoo plot of frequency versus supply voltage for the Scale chip. In the on-chip RAM testing mode, the chip operates at a maximum frequency of 260 MHz at a nominal supply voltage of 1.8 V. The voltage can be scaled down to 1.15 V, at which point the chip runs at 145 MHz. The frequency continues to improve well beyond 1.8 V, and reaches 350 MHz at 2.9 V (not shown in the shmoo plot). However, we have not evaluated the chip lifetime when operating above the nominal supply voltage. Of 13 working chips (out of 15), the frequency variation was generally less than 2%. We were surprised to find that using an undivided external clock input (which may have an uneven duty cycle) only reduced the maximum frequency by around 3% compared

to using the VCO. When caching is enabled, Scale's maximum frequency is 180 MHz at 1.8 V.

In the on-chip RAM operating mode, Scale's cycle time is around 43 FO4 (fan-out-of-four gate delays) based on a 90 ps FO4 metric. This is relatively competitive for an ASIC design flow, as clock periods for ASIC processors are typically 40–65 FO4 [Chinnery and Keutzer 2002]. Our initial studies evaluated Scale running at 400 MHz [Krashinsky et al. 2004], but this estimate assumed a more custom design flow. We did previously build a test chip which ran at 450 MHz (25 FO4) using the same design flow and process technology [Batten et al. 2007]. However, the test chip was a simple RISC core that was much easier to optimize than the entire Scale design.

The timing optimization passes in the synthesis, and place-and-route tools work to balance all paths around the same maximum latency; that is, paths are not made any faster than need be. As a result, there are generally a very large number of "critical" timing paths. Furthermore, different paths appear critical at different stages of the timing optimization, and affect the final timing even if they become less critical at the end. Throughout the Scale implementation effort, we continuously improved the design timing by restructuring logic to eliminate critical paths. A list of some of the critical paths is available in Krashinsky [2007].

5.3 Power

The power consumption data in this section was recorded by measuring steady-state average-current draw during operation. Power measurements are for the core of the chip and do not include input and output pads. We measure the idle-power draw when the clock is running but Scale is not fetching or executing any instructions. To estimate typical control-processor power consumption, we use a simple test program which repeatedly calculates the absolute values of integers in an input array and stores the results to an output array. To estimate typical power consumption for a program that uses the VTU, we use an ADPCM speech decoder benchmark (`adpcm_dec`) with a dataset that fits in the cache. This test program executes around 6.5 VTU compute operations per cycle.

Figure 13 shows how the power consumption scales with supply voltage with the chip running at 100 MHz. At 1.8V, the idle power is 112 mW, the power with the control processor running is 140 mW, and the power with the VTU also active is 249 mW. The figure also shows the power for Scale operating in the on-chip RAM testing mode. At 1.8 V the difference is only 3%, and the chip uses 241 mW. The power scaling is roughly linear, but the slope increases slightly above the nominal supply voltage.

Figure 14 shows how energy scales with frequency when voltage scaling is used. To calculate the average energy per cycle, we multiply average power consumption by clock period. From the baseline operating point of 1.8 V and 260 MHz, the energy can be reduced by 57% (from 2.3 nJ per cycle to 1.0 nJ) if the frequency is scaled down by 42% to 150 MHz and the voltage is reduced to 1.2 V. The frequency can also be increased by 23% to 320 MHz, but the supply

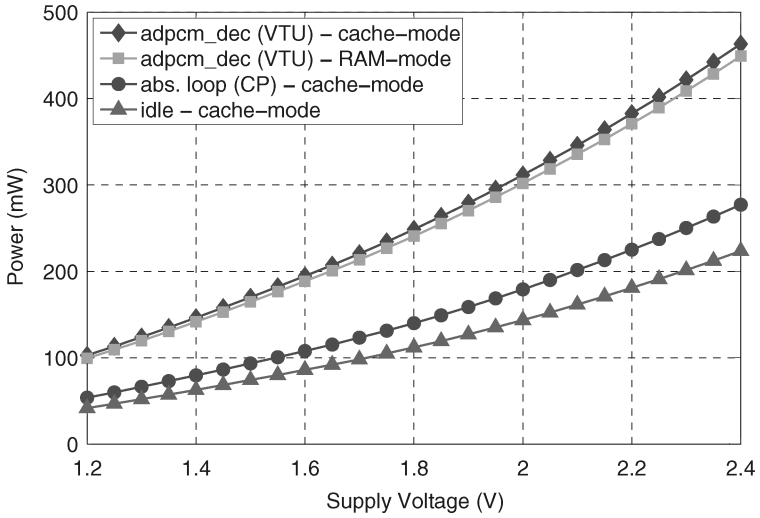


Fig. 13. Chip power consumption versus supply voltage at 100 MHz. Results are shown for idle operation (clock only), the control processor absolute-value test program, and the VTU `adpcm_dec` test program.

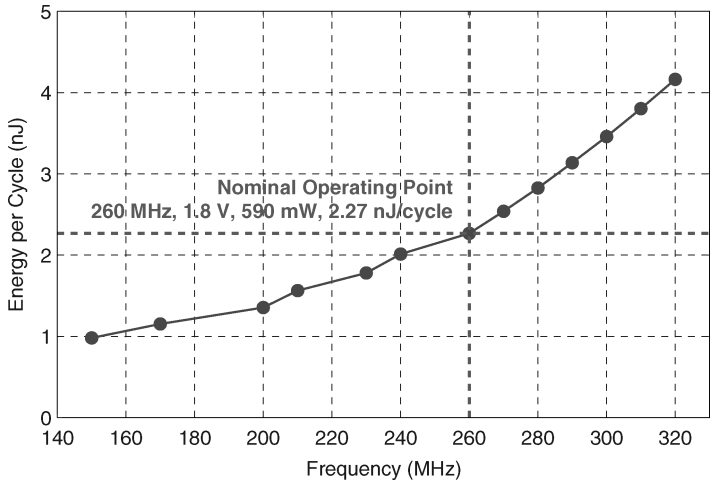


Fig. 14. Chip energy versus frequency while running the `adpcm_dec` test program in the on-chip RAM operating mode. Each data point is measured at maximum operating frequency as the supply voltage varies from 1.2 V (150 MHz) to 2.4 V (320 MHz) in 0.1 V increments.

voltage must be raised to 2.4 V, which increases the energy consumption by 83% (to 4.2 nJ).

5.4 Application Benchmarks

Figure 15 shows performance and power results for a variety of application benchmarks running on the Scale chip. The data measurements were made

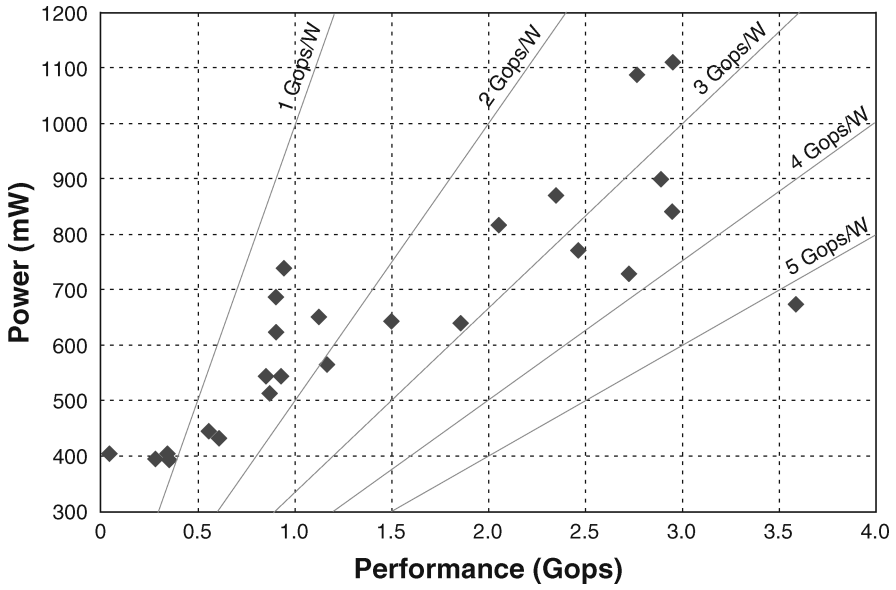


Fig. 15. Power versus performance for various application benchmarks. Each data point represents the average results for one benchmark.

with caching enabled, then scaled to 260 MHz. Details about the benchmarks and methodology are available in Krashinsky [2007].

At 260 MHz, Scale consumes 400 mW–1.1 W, achieving up to 5.3 Gops/W (billions of operations per-second per-watt) for 32-bit integer operations. The variation in power consumption corresponds well with the operations-per-cycle utilization rates for the different benchmarks, a testament to the effectiveness of clock-gating in the Scale chip. The benchmark results also demonstrate that in addition to achieving large speedups, Scale typically consumes significantly less energy than does a simple RISC processor. In comparison, other designs often improve performance only at the cost of large increases in energy consumption.

6. CONCLUSION

Our tool and verification flow helped us to build a performance-, power-, and area-efficient chip with limited resources. With a hybrid C++/Verilog simulation approach, we were able to reuse test infrastructure and incrementally transition from a C++ microarchitectural model to an RTL implementation. A highly automated methodology allowed us to easily accommodate an evolving design. Indeed, we only added the nonblocking cache-miss logic to the chip during the final two months before tapeout. Our iterative approach allowed us to continuously optimize the design, and to add and verify new features up until the last few days before tapeout. Even though we used a standard-cell-based ASIC-style design flow, procedural datapath preplacement allowed us to optimize timing and area. We reduced power consumption by using extensive clock-gating in both the preplaced datapaths and synthesized control logic.

The prototype Scale VT processor demonstrates that vector-threading can be implemented with compact and energy-efficient hardware structures. The Scale chip provides relatively high compute density with its 16 execution clusters and core area of 16.6 mm^2 in 180 nm technology. Its clock frequency of 260 MHz is competitive with other ASIC processors, and at this frequency the power consumption of the core is typically less than 1 W. A modern 45 nm technology could fit around 100 Scale cores per square centimeter of silicon area, making VT a plausible, performance-efficient, and flexible architecture for future processor arrays.

ACKNOWLEDGMENTS

The authors acknowledge and thank Albert Ma for designing the VCO and providing extensive help with CAD tools, Mark Hampton for implementing VTorture and the Scale compilation tools, Jaime Quinonez for the baseline datapath tiler implementation, Asif Khan for implementing the DRAM controller on the test baseboard, Jared Casper for work on an initial cache design and documentation, and Jeffrey Cohen for initial work on VTorture.

REFERENCES

BATTEN, C., KRASHINSKY, R., AND ASANOVIC, K. 2007. Scale control processor test-chip. Tech. Rep. MIT-CSAIL-TR-2007-003, CSAIL Technical Reports, Massachusetts Institute of Technology.

BATTEN, C., KRASHINSKY, R., GERDING, S., AND ASANOVIC, K. 2004. Cache refill/access decoupling for vector machines. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 331–342.

CHINNERY, D. AND KEUTZER, K. 2002. *Closing the Gap between ASIC and Custom: Tools and Techniques for High-Performance ASIC Design*. Kluwer Academic.

FLACHS, B., ASANO, S., DHONG, S. H., HOFSTEE, H. P., GERVAIS, G., KIM, R., LE, T., LIU, P., LEENSTRA, J., LIBERTY, J., MICHAEL, B., OH, H.-J., MUELLER, S. M., TAKAHASHI, O., HATAKEYAMA, A., WATANABE, Y., YANO, N., BROKENSHIRE, D. A., PEYRAVIAN, M., TO, V., AND IWATA, E. 2006. The microarchitecture of the synergistic processor for a cell processor. *IEEE J. Solid-State Circ. 41,* 1 (Jan.), 63–70.

HAMPTON, M. AND ASANOVIC, K. 2008. Compiling for vector-thread architectures. In *Proceedings of the 6th International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society.

KRASHINSKY, R. 2007. Vector-Thread architecture and implementation. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA.

KRASHINSKY, R., BATTEN, C., HAMPTON, M., GERDING, S., PHARRIS, B., CASPER, J., AND ASANOVIC, K. 2004. The vector-thread architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 52.

SANKARALINGAM, K., NAGARAJAN, R., GRATZ, P., DESIKAN, R., GULATI, D., HANSON, H., KIM, C., LIU, H., RANGANATHAN, N., SETHUMADHAVAN, S., SHARIF, S., SHIVAKUMAR, P., YODER, W., MCDONALD, R., KECKLER, S., AND BURGER, D. 2006. The distributed microarchitecture of the TRIPS prototype processor. In *MICRO-39*.

TAYLOR, M. B., KIM, J., MILLER, J., WENTZLAFF, D., GHODRAT, F., GREENWALD, B., HOFFMAN, H., JOHNSON, P., LEE, W., SARAF, A., SHNIDMAN, N., STRUMPEN, V., AMARASINGHE, S., AND AGARWAL, A. 2003. A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network. In *Proceedings of the International Solid State Circuits Conference (ISSCC)*.

TURPIN, M. 2003. The dangers of living with an x (bugs hidden in your Verilog). In *Synopsys Users Group Meeting*.

Received September 2007; revised March 2008; accepted March 2008