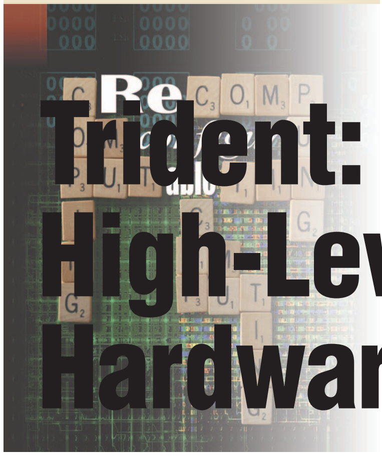


Trident: From High-Level Language to Hardware Circuitry

Justin L. Tripp, Los Alamos National Laboratory
Maya B. Gokhale, Lawrence Livermore National Laboratory
Kristopher D. Peterson, Imperial College of London

Unlocking the potential of field-programmable gate arrays requires compilers that translate algorithmic high-level language code into hardware circuits. The Trident open source compiler translates C code to a hardware circuit description, providing designers with extreme flexibility in prototyping reconfigurable supercomputers.

Reconfigurable supercomputing has shown significant promise in bioinformatics, text mining, and other data- and computation-intensive tasks involving small fixed-point integers.

In its traditional form, reconfigurable supercomputing uses field-programmable gate arrays to augment high-performance microprocessors in clusters, often involving FPGAs with millions of system gates, as well as dedicated arithmetic units and megabits of on-chip memory. More recently, approaches based on reconfigurable logic have succeeded in including floating-point tasks[1] and have realized several floating-point libraries (among them QnetiQ's Quixilica Library and the FPLibrary from the University of Lyon), computational kernels,[2,3] and applications[4] in FPGAs.

Although the kernels and applications enable a performance level much higher than that of microprocessors, they come with the high cost of having to hand-code a custom design in a hardware description language. This task is tedious and error prone since HDLs were never designed to describe algorithms, particularly those using pipelined floating-point operators.

A better alternative in debugging functionality and programmer productivity is to provide compilers that translate fixed- and floating-point algorithms in a high-level language (HLL) directly into circuit design expressed in an HDL. This process essentially synthesizes a circuit from the HLL.

Trident,[5] the recipient of a 2006 R&D 100 award for innovative technology, synthesizes circuits from an HLL. It provides an open framework for exploring algorithmic C computation on FPGAs by mapping the C program's floating-point operations to hardware floating-point modules and automatically allocating floating-point arrays to off-chip memory banks using four schedulers and a loop pipelining scheme. Users are free to select floating-point operators from a variety of standard libraries, such as FPLibrary and Quixilica, or to import their own. Adding hardware platforms is a matter of defining new interface description files and producing the code to tie the design to the description interface. Trident's open nature lets users rapidly prototype hardware from data analysis and simulation algorithms expressed in an HLL. The compiler's open source code is available on SourceForge (http://trident.sf.net).

Of the current research compilers such as the Riverside Optimizing Compiler for Configurable Computing[6] and Spark,[7] none support floating-point operations. As the "Synthesizing Circuits: A Variety of Approaches" sidebar describes, some commercial HLL-to-FPGA compilers such as Impulse C and the SRC Carte environment support floating-point operations as external libraries. However, these compilers limit the programmer to specific floating-point libraries and are mapped only to

Published by the IEEE Computer Society

Synthesizing Circuits: A Variety of Approaches

The simplest approach to circuit synthesis is to compile a subset of an existing language such as C or Java to hardware. The base language typically omits operations such as dynamic memory allocation or recursion as well as complex pointer-based data structures. Trident accepts such a subset of sequential C, extracts the available parallelism from the algorithmic description, and generates hardware circuits that execute the algorithm.

An alternative approach is to *extend a base sequential language* with constructs to manipulate bit widths, explicitly describe parallelism, and connect pieces of hardware. Celoxica's Handel C, Impulse C, and the MAP C compiler in SRC's Carte programming environment use this approach.

Another alternative is to *create a language for algorithmic description*, which is the approach that the University of Montreal's SHard[1] and the Mitrion-C dataflow language take. This alternative simplifies the compiler's work, but it can require programmers to significantly restructure algorithmic description as well as rewrite in a new syntax.

A *graphical interface* is yet another way to express an algorithm. Two tools that take this approach are Xilinx's System Generator and Starbridge's Viva. Graphical tools provide a hierarchical block diagram view that lets designers rapidly construct circuits. Such tools work best in specific application domains, such as digital signal processing.

However, without rapid prototyping through a high-level language, it is difficult to explore different algorithms and approaches. A high-level-language (HLL) compiler for FPGAs, such as Trident, frees designers to experiment with alternative hardware and software partitioning schemes and quickly determine how an algorithm will perform on a particular FPGA.

IMPULSE C

Impulse C from Impulse Accelerated Technologies (www.ImpulseC.com) is a C-based development system for coarse-grained programmable hardware targets, including mixed-processor and FPGA platforms. At the root of this technology are the Impulse C compiler and related tools and the Impulse application programmer interface (API). Impulse C can process blocks of C code, most often represented by one or a small number of C subroutines, into equivalent Verilog Hardware Description Language (VHDL) or Verilog hardware descriptions.

The Impulse compiler and optimizer enable the automated scheduling of C statements for increased parallelism and automated and semiautomated optimizations such as loop pipelining and unrolling. Interactive tools provided with the compiler let designers iteratively analyze and experiment with alternative hardware pipelining strategies.

To support mixed hardware-software targets, Impulse C's API includes C-compatible functions that let designers express system-level parallelism using a multiple-process, streaming, or shared-memory programming model. Impulse C also includes platform support packages that simplify C-to-hardware compilation for specific FPGA-based platforms. With these packages, Impulse C can automatically generate the required software-to-hardware interfaces. Figure A shows the design flow in one such package.

MITRION-C

Mitrion-C and the Mitrion virtual processor from Mitrionics (www.mitrionics.com) represent a new approach to software programmability for FPGAs. The virtual processor is a massively parallel high-performance processor for FPGAs that executes software written in the Mitrion-C programming language.

The processor's architecture follows a cluster model, placing all processing nodes within an FPGA. As Figure B shows, the Mitrion-C compiler and the processor configuration unit use the Mitrion-C source code to create processing nodes and an ad hoc network-on-a-chip.

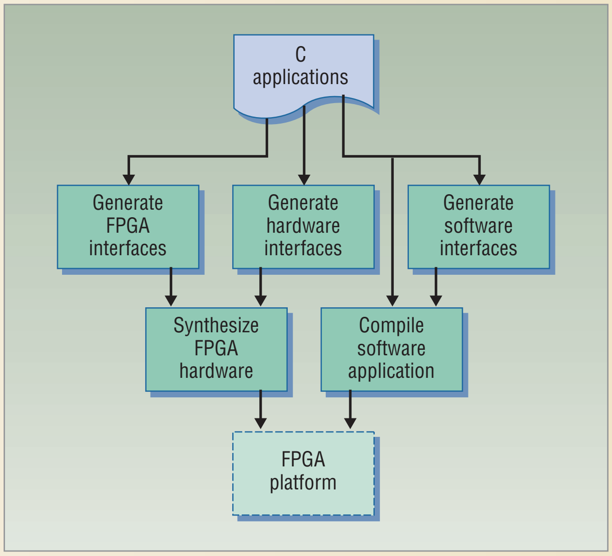

C applications
Generate FPGA interfaces
Generate hardware interfaces
Generate software interfaces
Synthesize FPGA hardware
Compile software application
FPGA platform

Figure A. Design flow in an Impulse C support package for an FPGA-based platform. Platform support packages simplify C-to-hardware compilation for specific FPGA-based platforms. Impulse C uses these packages to automatically generate the required software-to-hardware interfaces.

The network has simple point-to-point connections wherever possible and switches wherever required. Its latency of a single clock cycle is guaranteed, and network nodes are optimized to run a single instruction and communicate on every clock cycle. The result is a cluster with full fine-grained parallelism. Adapting the cluster to the program transforms the von Neumann architecture's inherently sequential problem of instruction scheduling into a parallelizable problem of data-packet switching.

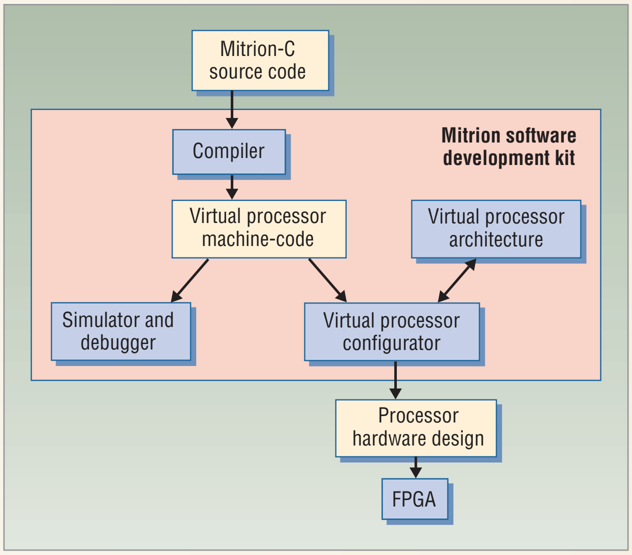

Mitrion-C source code
Mitrion software development kit
Compiler
Virtual processor machine-code
Virtual processor architecture
Simulator and debugger
Virtual processor configurator
Processor hardware design
FPGA

Figure B. Mitrion virtual processor and Mitrion-C programming language. From Mitrion-C, the compiler places all processing nodes within an FPGA. The Mitrion-C compiler and the processor configuration unit use the Mitrion-C source code to create processing nodes and an ad hoc network-on-a-chip.

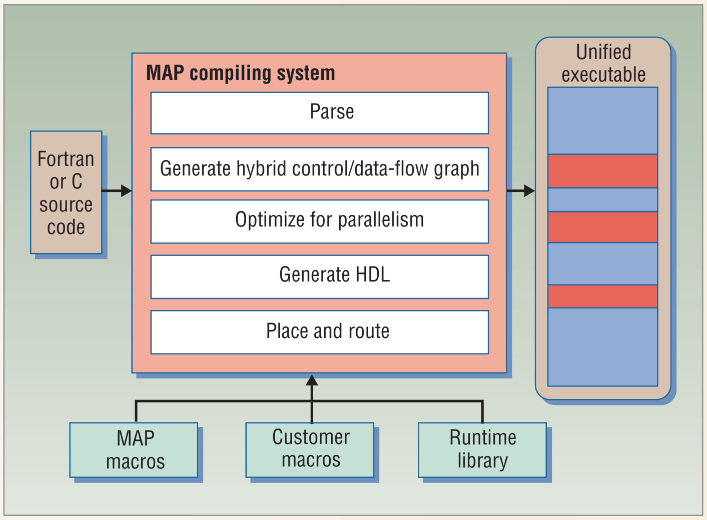


Figure C. The SRC Carte programming environment. From source code in Fortran or C, the MAP compiler generates a single Linux executable file that incorporates the microprocessor object modules, the MAP bit streams, and all the required runtime libraries.

Mitrion-C complements the processor's fine-grained parallelism by offering a fully parallel programming language. It differs from standard C in its language processing model. In standard C, programmers describe the program's order-of-execution, which does not fit well with parallel execution because it enforces a specific (sequential) execution order. Mitrion-C's processing model is based on data dependencies, which is a much better fit. A full description of a program's data dependencies is essentially a perfect description of that program's parallelism.

SRC CARTE

The Carte programming environment from SRC Computers supports a traditional program development methodology: Write code in a high-level language (C and Fortran), compile, debug via standard debugger, edit code, recompile, and so on, until correct implementation is obtained. When the application runs correctly in a microprocessor environment, it is recompiled and targeted for MAP, the direct execution logic processor.

Carte supports three compilation modes. In the *Debug* mode Carte compiles microprocessor code using a MAP emulator to verify the interaction between the CPU and MAP. In this execution mode, programmers can use standard debuggers to debug complete applications. Compilation is fast enough to allow rapid debugging.

In *Simulation* mode, Carte supports applications composed of C or Fortran and Verilog or VHDL. The compilation produces an HDL simulation executable that supports the simulation of generated logic.

Finally, in the *Hardware* compilation mode, the target is the direct execution logic that runs in MAP's FPGAs. In this mode, Carte optimizes for parallelism by pipelining loops, scheduling memory references, and supporting parallel code blocks and streams. The compilation output is a hybrid control-flow/data-flow circuit represented in an HDL which Carte then compiles into the final FPGA chip configuration bitstream.

As Figure C shows, the Carte programming environment compiles from C or Fortran to the FPGA configuration bitstream without programmer intervention and then further compiles the codes targeted to microprocessors into object modules. The final step is to create a unified executable.

RC TOOLBOX

DSPlogic's (www.dsplogic.com) Reconfigurable Computing (RC) Toolbox for the Mathworks Matlab/Simulink environment is a graphical programming environment for reconfigurable computing applications. As Figure D shows, the RC Toolbox consists of four key components.

RC Blockset allows the programming of sequential and iterative constructs directly related to those in C languages and includes four categories of blocks: program flow for sequential, parallel, and pipelined constructs; math for math functions, including floating-point types; parallel memory access for global variables and memories; and RC abstraction layer for integration with various RC platforms. Designers can use the Matlab/Simulink design environment to easily import third-party intellectual property (IP) cores as a graphical block with inputs and outputs, and hardware experts can use it to incorporate HDLs for access to low-level programming.

RC I/O consists of hardware abstraction layer libraries optimized for each RC platform.

With the *RC Debugging Toolbox,* users can validate entire applications as well as generate, collect, and visualize application data—all within the Matlab/Simulink environment.

Finally, the *RC Platform Builder* automatically generates all required logic and compiles the entire bitstream without exposing the complex FPGA implementation tools.

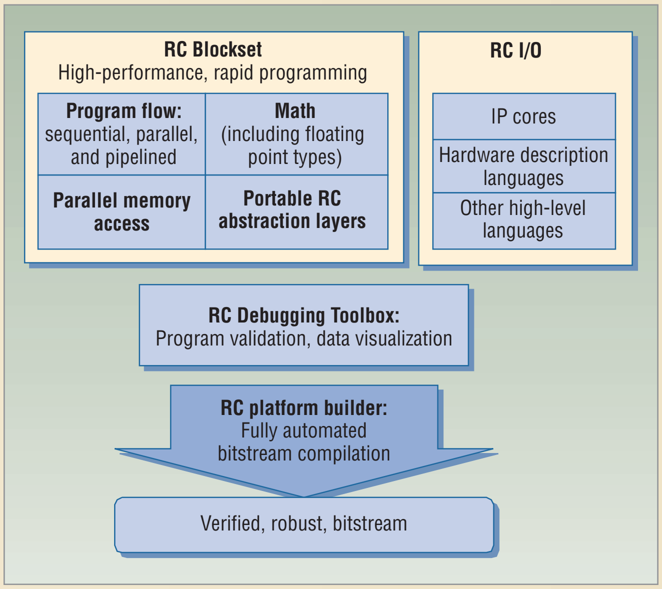

RC Blockset
High-performance, rapid programming
Program flow: sequential, parallel, and pipelined
Math (including floating point types)
Parallel memory access
Portable RC abstraction layers
RC I/O
IP cores
Hardware description languages
Other high-level languages
RC Debugging Toolbox: Program validation, data visualization
RC platform builder: Fully automated bitstream compilation
Verified, robust, bitstream

Figure D. Four components in DSPlogic's RC Toolbox.

HANDEL-C

Handel-C, part of the DK Design Suite from Celoxica (www.celoxica.com), synthesizes user code to FPGAs. As Figure E shows, users replace the algorithmic loop in the original Fortran, C, or C++ source application with a Celoxica API call to elicit the C code that is to be compiled into the FPGA. The FPGA C compiler brings in the appropriate runtime pieces to set up the interaction with the hardware environment.

Handel-C extends C with constructs for hardware design, such as parallelism and timing. It is designed around a simple timing model in which each assignment in the program takes one clock cycle to execute.

Programmers define parallel processes using extensions that instruct the compiler to create parallel hardware. The compiler translates input Handel-C code to an abstract syntax tree, which it then compiles to a high-level netlist that contains coarse function blocks. Handel-C then optimizes the high-level netlist before expanding it to a technology-specific netlist, which it then compiles to the FPGA bitstream.

The DK Design Suite also includes a GUI for integrated project management, code editing, and source-level debugging. It provides a cycle-accurate functional simulation of Handel-C designs, hardware synthesis, and a hardware and software debugging environment.

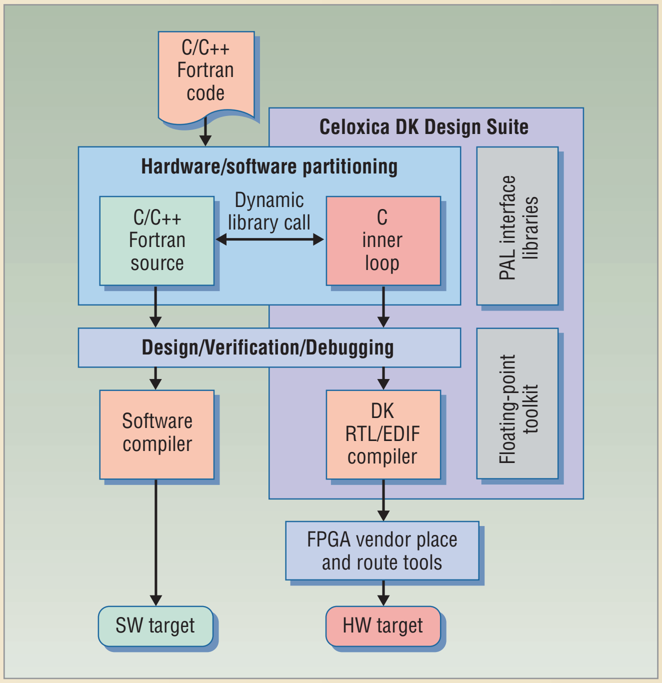


Figure E. Compilation in the DK Design Suite. Users replace the algorithmic loop in the original Fortran, C, or C++ source application with a call for the C code that is to be compiled into the FPGA. The FPGA C compiler brings in the appropriate runtime pieces to set up the interaction with the hardware environment.

Reference

1. X. Saint-Mleux, M. Feeley, and J.-P. David, "SHard: A Scheme to Hardware Compiler," *Proc. 2006 Scheme and Functional Programming Workshop*, Univ. of Chicago Press, 2006, pp. 39-49.

certain platforms. In contrast, designers can use Trident to experiment with, analyze, and optimize a variety of floating-point libraries and FPGA platforms.

COMPILATION PROCESS

To map algorithmic code to a hardware representation, Trident must combine traditional compiler analysis and transformation methods with CAD techniques. On the one hand, like compilers for traditional HLLs, it must parse the source program, perform high-level architecture-independent optimizations, and extract instruction-level parallelism. On the other, like CAD synthesis tools, it must schedule a sequence of concurrent operations and then generate circuits that control the data flow from memories to registers to operation units and back to memories. The FPGA area and routing resources—not the processor architecture—constrain the number of possible concurrent operations.

To meet the demands of its dual roles, Trident shares code from and extends SeaCucumber, a compiler developed at Brigham Young University that translates Java into FPGA circuit descriptions.[8] To SeaCucumber, Trident adds the ability to parse C input, accept floating-point operations, perform extensive operation scheduling, and generate VHDL. It also allows for additional compiler optimization and research at different abstraction levels.

To use Trident, the programmer manually partitions the program into software and hardware sections and writes C code to coordinate the data communication between the two parts. The C code to be mapped to hardware must conform to the synthesizable subset of C that Trident accepts: The code cannot contain print statements, recursion, dynamic memory allocation, function arguments or returned values, calls to functions with variable-length argument lists, or arrays without a declared size. Trident allocates arrays and variables statically during compilation and supports simple pointer references.

As Figure 1 shows, compilation with Trident has four main steps:

- *IR creation*. LLVM, a low-level virtual machine compiler infrastructure,[9] parses the C program to produce low-level, platform-independent object code, called LLVM bytecode. Trident uses the LLVM bytecode to create its intermediate representation.
- *IR transformation*. From operations in if-statements, Trident creates hyperblocks—a representation that exposes more instruction-level parallelism—and a control-flow graph that consists of hyperblock nodes. Trident uses the control-flow graph to optimize the code and map all operations into modules from hardware libraries that the user selects.
- *Array allocation and scheduling*. Trident uses one of four scheduling algorithms to schedule operations in each hyperblock.
- *Synthesis*. Trident translates the scheduled control-flow graph into a register-transfer-level HDL, using a hierarchical hardware description to preserve modularity.

The top-level circuit contains block subcircuits for each hyperblock in the control-flow graph input, as well as a global collection of registers. All block subcircuits share the register file. Each block subcircuit contains a state machine and a data path subcircuit. The state machine controls the timing of the block subcircuit's data path, and the data path subcircuit implements the logic needed to represent the data flow through the associated hyperblocks. It contains operators, predicate logic, local registers, and wires that connect all the components. Control circuits connect all the blocks to ensure properly ordered execution. All these elements combine to produce an optimized application-specific circuit.

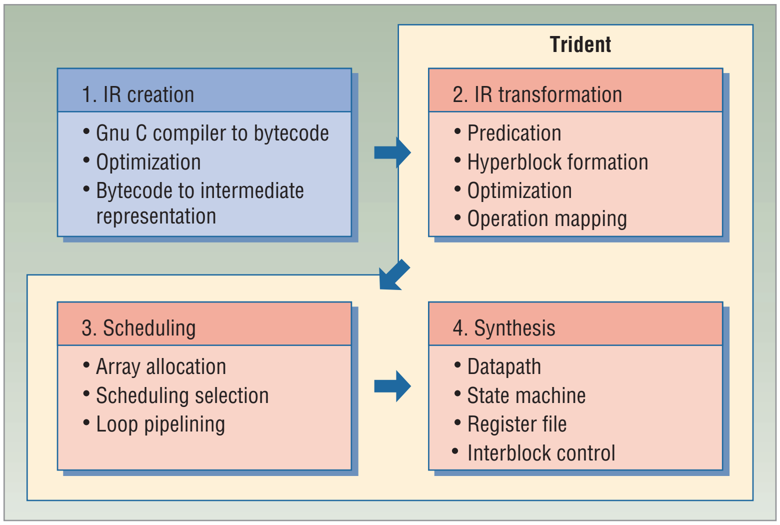


Figure 1. Compilation in Trident. A low-level virtual machine compiler parses the C program to produce platform-independent object code. Optimized LLVM object code is converted to a hardware-oriented intermediate representation, which Trident further optimizes to remove unnecessary operations. Finally, Trident maps all floating-point operations to a hardware library that the user selects.

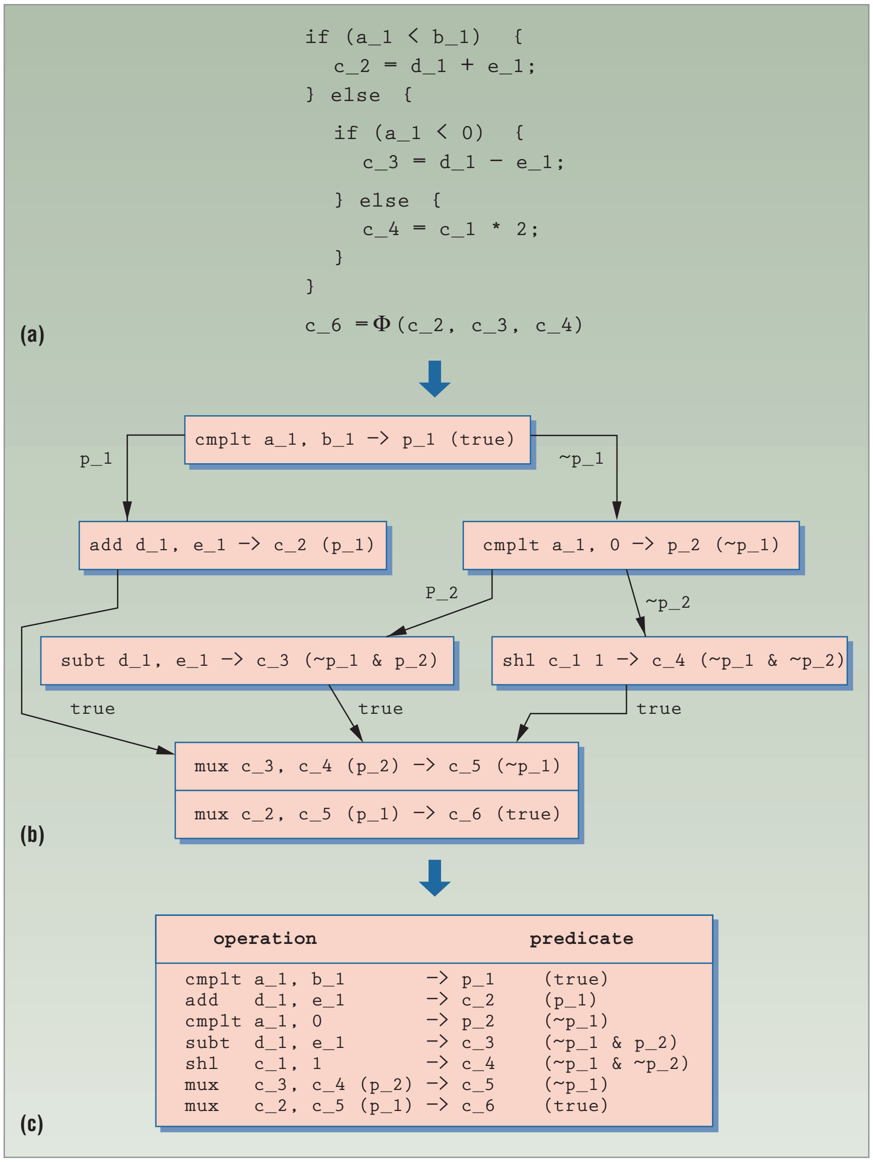
IR creation

LLVM uses the Gnu C compiler as a front end to parse C and convert it to LLVM bytecode—a low-level object code representation that uses simple instructions similar to those in reduced-instruction-set computing (RISC)—but it also offers rich, language-independent type and data flow information about operands. LLVM bytecode representation is rich enough for LLVM to perform sophisticated optimizations, yet remains lightweight enough to attach to the executable. Consequently, transformations are possible throughout the program's lifetime.

LLVM accepts C and C++ programs as input and generates architecture-independent assembly language, which Trident parses into a hardware-oriented IR. By using LLVM as a front end, Trident can focus on hardware compilation concerns and leave the parsing and baseline optimizations to LLVM.

The generated bytecode is in static single assignment form. At this point in compilation, Trident disables optimizations and library linkage. It will complete optimizations in a later step, but Trident must resolve all library function references in the bytecode. A Trident tool written in the LLVM framework then optimizes the LLVM bytecode using optimization passes that LLVM provides. These passes include but are not limited to constant propagation, small function in-lining, loop invariant hoisting, tail-call elimination, small loop unrolling, and common subexpression elimination. Calling the optimizations from the Trident-specific LLVM tool gives Trident the flexibility of adding or removing optimizations as needed.

The final Trident pass in LLVM creates a textual representation of the LLVM bytecode, which Trident reads and then uses in the hardware scheduling and allocation and synthesis phases. The textual representation includes basic blocks, loop information, control-flow graphs, and static single assignment variables—all the program information needed to build an FPGA circuit representation.


(a)
(b)
(c)
p_1
~p_1
p_2
~p_2
true
true
true
cmplt a_1, b_1 -> p_1 (true)
add d_1, e_1 -> c_2 (p_1)
cmplt a_1, 0 -> p_2 (~p_1)
subt d_1, e_1 -> c_3 (~p_1 & p_2)
shl c_1 1 -> c_4 (~p_1 & ~p_2)
mux c_3, c_4 (p_2) -> c_5 (~p_1)
mux c_2, c_5 (p_1) -> c_6 (true)

```
if (a_1 < b_1)  {
  c_2 = d_1 + e_1;
} else  {
  if (a_1 < 0)  {
    c_3 = d_1 - e_1;
  } else  {
    c_4 = c_1 * 2;
  }
}
c_6 =Φ(c_2, c_3, c_4)
```

operation	predicate
cmplt a_1, b_1 -> p_1	(true)
add d_1, e_1 -> c_2	(p_1)
cmplt a_1, 0 -> p_2	(~p_1)
subt d_1, e_1 -> c_3	(~p_1 & p_2)
shl c_1, 1 -> c_4	(~p_1 & ~p_2)
mux c_3, c_4 (p_2) -> c_5	(~p_1)
mux c_2, c_5 (p_1) -> c_6	(true)

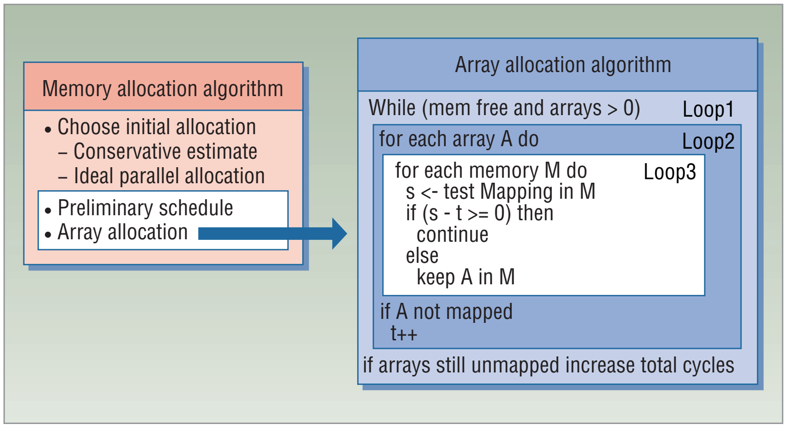
Figure 2. How Trident converts static single assignment representation to **hyperblocks**—*extended basic instruction blocks with one input path and potentially any number of output paths. The conversion process begins with LLVM static single assignment code, (a) to which Trident adds predicates to replace branches. (b) The result is the merging of operations to form a hyperblock (c) that has more instruction-level parallelism than a standard basic block.*

IR transformation

We designed the Trident IR to combine standard compiler IR with the data structures needed to generate lower level hardware. Trident's IR extends LLVM's static single assignment representation by adding predicated operations, scheduling information, resource use, and more operator types. Predicates specify the condition under which the operation should execute.

Trident creates hyperblocks using if-conversion and predication so that it can replace branches with predicated operations. A hyperblock is an extended basic instruction block with one input path but potentially any number of output paths.

As Figure 2 shows, Trident eliminates the if-statements and merges the then- and else-statements into a hyperblock until the control-flow graph contains only loop-control edges. Because the hyperblock representation exposes more instruction-level parallelism than the stan-


Memory allocation algorithm
• Choose initial allocation
– Conservative estimate
– Ideal parallel allocation
• Preliminary schedule
• Array allocation
Array allocation algorithm
Loop1
Loop2
Loop3

```
While (mem free and arrays > 0)
  for each array A do
    for each memory M do
      s <- test Mapping in M
      if (s - t >= 0) then
        continue
      else
        keep A in M
  if A not mapped
    t++
if arrays still unmapped increase total cycles
```

Figure 3. Memory allocation algorithm for mapping arrays into available memory resources. The algorithm uses an initial allocation to create a preliminary schedule. It then adjusts the array allocation (box at right). Trident uses the results of memory allocation to determine a final operation schedule.

dard basic block, the scheduling phase has additional opportunities to schedule concurrent operations.

To remove the redundant and unnecessary instructions from converting if-statements and forming hyperblocks, Trident repeats the LLVM standard optimizations (common subexpression and dead-code elimination, strength reduction, constant propagation, alias analysis, and so on). These optimizations decrease the number of operations that Trident must synthesize into hardware. To select specific floating-point hardware operators, Trident maps a generic set of operations into a particular floating-point library. Trident can map different libraries to a common set of floating-point operations, so users can easily trade off area, resources, clock speed, and latency.

Array allocation and scheduling

The Trident compiler performs all memory allocation during compilation, which means that it can schedule accesses to memory banks statically, resulting in low-latency, deterministic access to static RAMs. Trident allocates scalar variables to on-chip registers when required, but arrays often require resources that are not available on the FPGA chip. Likewise, floating-point data types (single and double precision) present a challenge to FPGA resources. Relative to integer operations, floating-point modules require significantly more logic blocks on the FPGA, and the large operand sizes complicate memory allocation and require more memory bandwidth. Finally, because floating-point operations are highly pipelined, scheduling must take care to prevent write/read data hazards.

Array allocation. In most data-intensive, streaming computation, data arrays are too large to be stored on the FPGA chip and must go to off-chip memory. This makes external memory bandwidth the primary limitation on parallelism within the circuit. Most reconfigurable computing FPGA boards contain independent parallel banks of static RAMs which act as a noncached memory subsystem with deterministic access time.

Array allocation assigns arrays to the memory banks. But allocation interacts with operation scheduling. Thus, *where* the arrays are located in memory banks determines the extent to which the memory subsystems can read and write operands concurrently and the extent to which operations that use those operands can be scheduled concurrently. The degree of concurrent operation also depends on the extent to which the operand modules are shared.

Relative to operands for small integer arrays, the 32- or 64-bit width of floating-point arrays reduces the number of operands that the memory subsystem can access concurrently. For example, if the FPGA board has four external memories, each 64 bits wide with a single read/write bus, the memory subsystem can access only four double-precision floating-point numbers in a single clock cycle.

If the computation has sufficient parallelism to consume more than the four operands from memory per clock cycle, there is no way to exploit it. When memory bandwidth is insufficient, the scheduler must order memory accesses sequentially, and the data path circuit might need additional pipeline registers, which consumes on-chip logic resources. The scheduler might also need a longer interval for introducing new operands into the pipeline, which reduces overall throughput. If the operands are 8-bit pixels, in contrast, the memory subsystem could access 32 operands concurrently from four 64-bit memories and 16 or more operations could be scheduled concurrently.

Optimizing array allocation to the memory banks requires considering several factors. The pattern of array allocation to memory influences throughput.[10,11] To maximize communication bandwidth, it makes sense to allocate arrays to different memories so that multiple independent memories can be accessed concurrently. It also makes sense to allocate arrays to memories with multiple read/write buses, since the memory subsystem would have parallel access to a single memory.

Another consideration is memory access latency—how many cycles it takes to satisfy a read or write request and if it is possible to schedule array accesses to the same memory at different times and thus not lengthen the schedule.

Finally, a possible optimization is to pack arrays in the memory data word to minimize memory accesses. If a memory is 64 bits wide, and two arrays use 32-bit operands, it is possible to pack corresponding elements of the two arrays into the same memory location and access both arrays simultaneously with one read or write operation. However, this strategy is desirable only if three conditions hold: The memory subsystem accesses the arrays with the same index, it performs the same operation (either read or write) on both arrays, and it can schedule the two operations concurrently.

As Figure 3 shows, memory allocation starts with one of two methods, which the user specifies. The first method schedules the graph with a conservative estimate of access times that uses the read and write latencies of the slowest memory for all arrays. The second method attempts an ideal allocation in which the memory subsystem accesses all arrays in parallel or in the fewest possible cycles. Both methods attempt to make the best use of memories with different access latencies.

The next step is to construct a preliminary schedule and adjust allocation to best meet the preliminary schedule constraints. If additional optimization is desired, the user can request multiple iterations of these steps for some specified period.

The array allocation algorithm is a greedy search using a cost function $c = s - t$, where s represents the increase or decrease of schedule length and t is the number of attempts made to allocate the array. The longer the schedule, the higher the value of s, and at each allocation attempt, the algorithm increments t.

The array allocation algorithm consists of three nested loops. Loop 1 repeats as long as memory is not full and unallocated arrays exist. Loop 2 iterates through every unallocated array in random order. Loop 3 iterates over the memories in random order and calculates the cost (c) of allocating the array under consideration to that memory.

The algorithm will allocate the array only if the cost is less than or equal to zero ($c \leq 0$). If it cannot find a memory for this array, the algorithm continues with the next array. After attempting to allocate every array, the algorithm tries again with any remaining unallocated arrays.

Scheduling. Once Trident has an array allocation, it invokes a user-selected scheduler to determine an execution order for the operations. Because Trident targets programmable hardware, it can schedule an arbitrary number of independent operations in parallel. In Figure 4, the scheduler defines a partial order of operations in the assignment expression $O = (A * B) + D * (C + D)$ and breaks it into four operations. Data dependence and memory access requirements partially constrain the sequence of those operations.

At this point, any of four scheduling algorithms are possible: *as soon as possible*, which schedules operations as soon as their inputs are available; *as late as possible*, which schedules operations typically just before they are used; *force directed*, which schedules operations within an execution window somewhere between the ASAP and ALAP extremes; and *iterative modulo*, for scheduling loops.

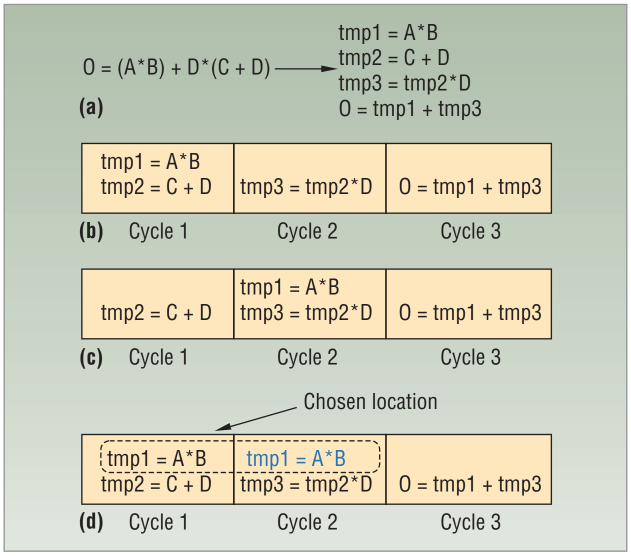


Figure 4. Scheduling in Trident. (a) Trident breaks code into individual steps, and the user selects one of Trident's four scheduling modes. (b) In the* as soon as possible *mode, Trident schedules operations as soon as their inputs are available. (c) In the* as late as possible *mode, Trident schedules operations as late as possible, as in just before they are used. (d) In the* force-directed *mode, Trident uses a system of forces to schedule operations between the ASAP and ALAP extremes.

The force-directed algorithm is useful because it spreads operations of the same type (such as adds) within the execution window. Consequently, the synthesizer can build fewer of these operator types in the hardware and share them. In Figure 4d, for example, the force-directed scheduler places the operation tmp1 = $A * B$ in the first cycle, thus reducing the number of multiplications in cycle 2.

The iterative modulo scheduling algorithm[12] schedules loops by pipelining them and beginning a new loop iteration $i + 1$ before its predecessor iteration (i) has completed. The number of clock cycles that elapse between the start of iteration i and $i + 1$ is the *initiation interval*.

Modulo scheduling uses a heuristic to find an initiation interval that does not violate dependence or array access constraints. Dependence constraints arise when an operation in iteration $i + 1$ uses data produced in iteration i. Array access constraints occur when the memory subsystem requests more read or write operations to a memory in a time slot than the memory can satisfy. When this conflict occurs, the operations must be scheduled in succeeding time slots, increasing the initiation interval. Trident's modulo scheduling algorithm also schedules reads and writes to packed arrays in the same time slot.

Given the initiation interval, the compiler generates a prologue to collect intermediate results from the initial iterations, a steady-state loop body, and an epilogue to collect results from the final iterations. The scheduling phase output is a time-annotated control-flow graph,

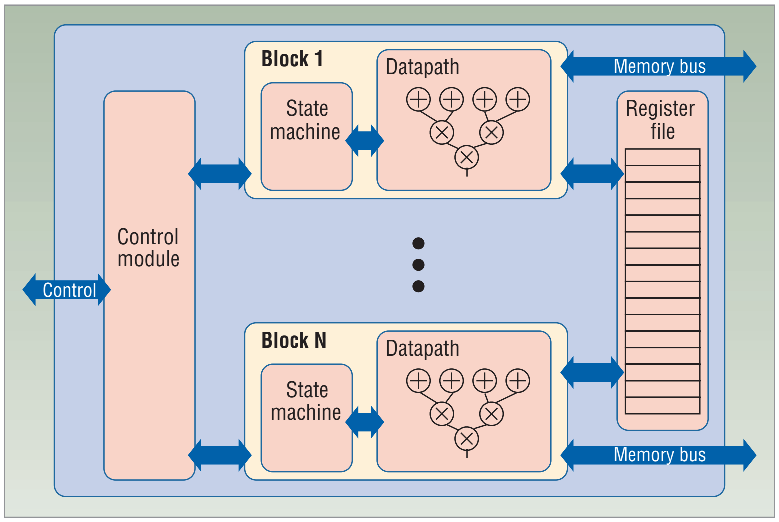


Figure 5. Abstract circuit design hierarchy. The top level contains subcircuits for each block in the control-flow graph input as well as a register set that all block subcircuits share. Each block subcircuit contains a state machine and a data path subcircuit. The state machine controls the timing of the block's data path.

Table 1. Synthesis results with Trident.

Benchmark	Clock (MHz)	Slice count	Area (%)	Blocks	States
Photon	193	11,810	50	1	112
Photon-hand	98	8,819	20	1	98
Euclid	200	6,071	25	1	71

each node of which represents a hyperblock. The time annotation describes the number of clock cycles that the scheduled hyperblock consumes. For loops, the time annotation gives the number of clock cycles that one iteration of the loop uses. The schedule associated with each hyperblock lists the operations that occur concurrently in each time step of the schedule.

Synthesis

After the scheduler schedules all operations, it passes the timing-annotated control-flow graph to the synthesizer, which creates a data path and control structure to implement the desired behavior. The synthesizer must also create any board-level circuit structures and make all necessary external connections to the synthesized circuit. It accomplishes these tasks in four major stages: library mapping, abstract design generation, board-level synthesis, and output generation.

Library mapping. The synthesizer begins by building each operation in the control-flow graph as a circuit element—an element with data path operations whose operators are either native or library. *Native* operators are suitable for integer and Boolean operations. External data files define *library* operators, which include required input and output ports, required external library declarations, and the input and output mappings. Mappings can be to internal ports or constants or can be left open.

Although existing libraries provide some floating-point operations, most are incomplete and do not include all possible floating-point operations; a notable omission is casts from various integer types to float types and vice versa. To account for the incompleteness of these libraries, the library operation mapping configuration makes it possible to select individual operations from different libraries.

Abstract design generation. This stage generates a top-level design blueprint, such as that in Figure 5, while leaving the circuit's underlying technology open until the final code generation stage. The underlying technology consists of the target HDL and hardware modules, such as the floating-point cores.

The block's data path implements the logic that represents the data flow through all the operations in the control-flow graph—operators, predicate logic, local registers, and the wires that connect all the components. If the target is a pipelined design, the circuit generator adds pipeline registers between data path operators to preserve correct data-flow behavior.

Board-level synthesis. The circuit generator must insert not only hierarchy, control, and data path elements, but also the interface to the board-level design. A board description file similar to the library operator file describes the top-level interface and the required input and output mappings. If additional low-level details of signaling protocols are needed to describe board-level interactions, designers can supply these as additional code during this phase.

Output generation. Trident's internal technology-independent circuit representation can accommodate multiple output representations and currently supports VHDL. In addition, Trident can output a file that will help users visually debug the design's structure. Back-end generation is simple enough to allow relatively straightforward additions to the list of target technologies. Each target technology's back-end generator extends the abstract circuit generator's class. Thus, the target technology's back-end generator actually generates each abstract component.

COMPILER BENCHMARKS

Table 1 shows benchmark results from Trident experiments, in which Trident targeted the Cray XD1 (Xilinx

Virtex2Pro 50) using the Xilinx ISE 6.3p3 tools with the Quixilica floating-point library. The overhead for interfacing to the Cray FPGA board is about 10 to 15 percent of the total area.

Photon is a compiler-generated inner loop from a Monte Carlo radiative heat transfer simulation. *Photon-hand* is a design for Photon that an engineer generated. The results for Photon-hand target a Virtex2Pro 100 and are just for the design pipeline; they exclude any overhead required to interface with a particular board. *Euclid* calculates the Euclidean distance between two points in 3D space.

As the table shows, Photon is almost twice as fast as Photon-hand. Factoring in the overhead of the XD1 interface logic, Photon is also competitive in area relative to Photon-hand.

Currently, partitioning the hardware and software portions of an algorithm must be completed manually. To move toward automating this process, we could use LLVM capability to do a runtime analysis since it can provide profiling information. With the profiling information, we can potentially identify the computation-intensive code portions, identify functional block reuse, and understand data movement. With this insight, we could partition the code automatically and better allocate external memory when both dynamic and static RAM are available. Also, combining the execution profile information with an FPGA execution model would deepen the understanding of overall system speedup. Our hope is that Trident's open source nature will facilitate the interaction needed to further development in these areas. ■

References

1. K. Underwood, "FPGAs vs. CPUs: Trends in Peak Floating-Point Performance," *Proc. 12th ACM Int'l Symp. Field-Programmable Gate Arrays* (FPGA 04), ACM Press, 2004, pp. 171-180.
2. G. Govindu et al., "Area and Power Performance Analysis of Floating-Point-Based Application on FPGAs," *Proc. 7th Ann. Workshop High-Performance Embedded Computing* (HPEC 03), Sept. 2003; www.ll.mit.edu/HPEC/agenda03.htm.
3. M. Gokhale et al., "Monte Carlo Radiative Heat Transfer Simulation on a Reconfigurable Computer," *Proc. Int'l Conf. Field-Programmable Logic and Applications* (FPL 04), Springer, 2004, pp. 95-104.
4. J.P. Durbano et al., "FPGA-Based Acceleration of 3D Finite-Difference Time-Domain Method," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, IEEE Press, 2004, pp. 156-163.
5. J.L. Tripp et al., "Trident: An FPGA Compiler Framework for Floating-Point Algorithms," *Proc. Int'l Conf. Field-Programmable Logic and Applications* (FPL 05), IEEE Press, 2005, pp. 317-322.
6. Z. Guo and W. Najjar, "A Compiler Intermediate Representation for Reconfigurable Fabrics," *Proc. Int'l Conf. Field-Programmable Logic and Applications* (FPL 06), IEEE Press, 2006, pp. 741-744.
7. S. Gupta, R.G. Nikil, and D. Dutt, *Spark: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*, Springer, 2005.
8. J.L. Tripp, P.A. Jackson, and B.L. Hutchings, "SeaCucumber: A Synthesizing Compiler for FPGAs," *Proc. 12th Int'l Conf. Field Programmable Logic and Applications* (FPL 02), Springer-Verlag, 2002, pp. 875-885.
9. C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," *Proc. Int'l Symp. Code Generation and Optimization* (CGO 04), IEEE CS Press, 2004, pp. 75-86.
10. M.B. Gokhale and J.M. Stone, "Automatic Allocation of Arrays to Memories," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, IEEE CS Press, 1999, pp. 63-69.
11. H. Lange and A. Koch, "Memory Access Schemes for Configurable Processors," *Proc. 12th Int'l Conf. Field-Programmable Logic and Applications* (FPL 00), Springer-Verlag, 2000, pp. 615-625.
12. B.R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," *Proc. 27th Ann. Int'l Symp. Microarchitecture*, ACM Press, 1994, pp. 63-74.

Justin L. Tripp *is a technical staff member on the Application-Specific Architectures Team in the Advanced Computing Laboratory at Los Alamos National Laboratory. His research interests include reconfigurable logic designs, synthesis, compilers, and parallel computing. Tripp received a PhD in electrical engineering from Brigham Young University. He is a member of the IEEE Computer Society. Contact him at jtripp@lanl.gov.*

Maya B. Gokhale *is a member of the staff at the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. Her research interests include reconfigurable computing with FPGAs, high-performance computing, parallel languages, and embeddable architectures. Gokhale received a PhD in computer and information sciences from the University of Pennsylvania. She is an IEEE Fellow and a member of Phi Beta Kappa. Contact her at maya@llnl.gov.*

Kristopher D. Peterson *is a PhD student in the Bioengineering Department at the Imperial College of London. His research interests are insect neuroscience and vision and robotics. He received an MS in evolutionary and adaptive systems from the University of Sussex. Contact him at krisdpeterson@gmail.com.*