Attacking the Semantic Gap Between Application Programming Languages and Configurable Hardware

Greg Snider, Barry Shackleford, and Richard J. Carter
Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304 U.S.A.
greg_snider@hp.com

ABSTRACT

It is difficult to exploit the massive, fine-grained parallelism of configurable hardware with a conventional application programming language such as C, Pascal or Java. The difficulty arises from the mismatch between the synchronous, concurrent processing capability of the hardware and the expressiveness of the language—the so-called "semantic gap." We attack this problem by using a programming model matched to the hardware's capabilities that can be implemented in any (unmodified) object-oriented language, and building a corresponding compiler. The result is application code that can be developed, compiled, debugged and executed on a personal computer using conventional tools (such as Visual C++ or Visual Cafe), and then recompiled without modification to the configurable hardware target. A straightforward C++ implementation of the Serpent encryption algorithm compiled with our compiler onto a Virtex XCV1000 FPGA yielded an implementation that was smaller (3200 vs. 4502 CLBs) and faster (77 MHz vs. 38 MHz) than an independent VHDL implementation with the same degree of pipelining. A tuned version of the source yielded an implementation that ran at 95 MHz.

1. INTRODUCTION

Most application programming languages adhere to a simple, sequential programming model. Given their heritage, this is not surprising; since early processors were incapable of much more than simple, sequential execution, there was little motivation to include language features (such as concurrency) that could not be effectively exploited by a sequential processor.

As processor designs have become denser and more complex, the quest for ever-increasing performance has put pressure on compilers to take advantage of the parallelism offered by the hardware. RISC processors often expose part of their pipeline, allowing the compiler to take advantage of overlapped execution of instructions by filling "delay slots" following branches. VLIW processors offer multiple, pipelined function units, rotating registers and other mechanisms that assume compilers capable of extracting fine-grained, instruction-level parallelism. This has inspired increasingly sophisticated compiler algorithms in dependence analysis, scheduling, software pipelining, predication, speculative execution, and so on.

Hardware complexity has progressed to the point where it is now possible to design processors with thousands of function units, and advances in molecular computing suggest the possibility of constructing processors with millions [24]. How can this massive parallelism be exploited? There are two general approaches:

1. *Improve algorithms for extracting parallelism from "dusty deck" sequential code.* Fine-grained (instruction level) parallelism is insufficient in most applications to keep large numbers of function units busy. Most medium-grained parallelization research has focussed on parallelization of regular structures, such as nested loops. These can exploit the hardware parallelism, but often impose constraints on the style in which the code is written (e.g. affine array references) [11]; although such code can often (but not always) be rewritten to meet the constraints, it is then no longer dusty deck, but rather uses a new programming model. Less work appears to have been done in the automatic extraction of irregular, medium-grained (procedure level) parallelism. The dusty deck approach has the obvious advantages of maintaining a familiar programming model and allowing the acceleration of legacy code, but it is debatable whether it will ever be able to fully exploit the potential of configurable hardware.

2. *Use a parallel programming model.* Parallel programming models have been successfully used in systolic arrays and cellular automata. Compilers for such architectures are easy to build, but the programming models are quite restrictive, limiting their domains of applicability, and the languages often unfamiliar, which hinders acceptance. Parallel models have also been used for decades in logic design, which probably explains the current interest in hardware design languages, such as VHDL, for programming configurable systems. Hardware design languages, though, have the disadvantages of being very low-level, unfamiliar to application programmers, and difficult to integrate with other application code. The parallel programming model has the obvious challenge of user acceptance.

1.1 The problem

We wish to bring the power of configurable hardware to application programmers (rather than hardware designers), sparing them the low-level details of the hardware, and yet providing performance comparable to an implementation written in a hardware design language.

1.2 Overview of our approach

We are exploring the use of a simple, explicitly parallel programming model combined with a conventional application language to exploit the parallelism of configurable hardware architectures. We require the user to specify the medium-grained parallelism explicitly, but rely upon the compiler to extract the fine-grained, instruction-level parallelism.

Our programming model, which we call "Machines," is really just an adaptation to conventional application languages of the concurrent state machine model used by hardware designers for decades. However, it is somewhat more abstract, in that Machines may be nested, and micro-architectural issues (micro-pipelining, micro-sequencing, clocking, and serial vs. parallel arithmetic) are handled by the compiler. It allows for the creation of large numbers of synchronous, medium-grained "processes" with relatively unconstrained communication paths (although the user will pay the penalty for a lack of discipline here). Pipelines, systolic arrays, cellular automata, "butterfly" computations (e.g. FFT) and many other structures, including irregular computations, may be easily expressed in this model. We believe the model exploits the fine-grained, synchronous, high-bandwidth communication of FPGAs and other configurable architectures, presents a simple model of communication and synchronization, and also allows the construction of efficient compilers without heroic algorithmic efforts. The downside is that it requires the user to extract the medium-grained parallelism of the algorithm directly—no dusty decks here.

Our implementation language can be any object-oriented language without extensions or modifications (although it is easier to implement in an object-oriented language with operator overloading and parametric classes, like C++, than a language like Java). Using, and preserving the semantics of, a conventional language has several advantages:

- *Familiarity*—a programmer familiar with the language can read and understand such programs once the programming model is understood.
- *Development*—standard tools and development environments (such as Visual C++ or Visual Cafe), may be used to design, debug and execute the parallel application on a conventional sequential processor before doing a final compile and download of the (debugged) application to hardware.
- *Execution speed*—since the source language can be compiled directly to machine code, it can execute more quickly during development than hardware design languages, such as Verilog or VHDL, that must be simulated.

We have been unable to implement the model in procedural languages such as C or Pascal without either perturbing the semantics of the language, introducing awkwardness in the expression of certain algorithms, or creating potential state inconsistency problems that would be difficult for the compiler to detect.

2. RELATED WORK

Languages for programming configurable hardware fall into two broad categories: hardware design languages, and conventional application languages with modifications and/or extensions.

Hardware design languages offer the advantage of expressiveness—they are a close fit to the hardware—but can be complex and low-level. The older languages (Verilog and VHDL) cannot be compiled to native machine code and must be simulated for debugging, slowing down the development cycle. Newer hardware languages which are attempting to resolve these problems include System C, Spec C and Superlog [22].

Application language solutions fall into several different camps, depending on the programming model they use and how they affect the grammar and semantics of the original language. Systems that extend a conventional language with non-standard parallel constructs (which means they cannot be compiled by a standard compiler for the base language) include Handel-C [18], HardwareC [20], RL [21], Transmogrifier C [13], Spyder C++ [14], Data Parallel C [8], Picasso [19] and SA-C [9]. Unconventional application languages created with an explicit parallel programming model include CAM [25], TAO [23] and an unnamed language by Wirth [7]. Systems mapping unmodified languages onto hardware include BRASS [2, 12], RAW [1, 3, 5], Ocapi [6], Forge [15, 16], Nenya [10] and the work by Babb [4]. Systems which expose the parallelism of the underlying hardware (often through libraries) within the framework of a conventional language include RaPiD [30], PipeRench [31], PAM-Blox [32], JHDL [33] and the C Level libraries [17].

3. PROGRAMMING MODEL

3.1 Objectives/philosophy

Our goal is to compile applications onto configurable hardware, not raw silicon. We assume that the target has large numbers of synchronous function units and that there is high bandwidth communication between function units. These assumptions are, of course, technology dependent, and would not serve as a useful model for, say, independent processors communicating over a network.

In creating the model, we have adopted the tactics of:

- medium-grained parallelism specified by user;
- fine-grained parallelism extracted by the complier;
- user spared from micro-architectural and design issues.

3.2 Example 1: Counter

To introduce the model, here's a program (written in Java) that implements a counter:

```
class Counter extends Machine {
   int count = 0; // state variable
   void step() { count++; }
}
```

A Machine is an *active* entity that is the smallest independent unit of execution, analogous to a thread in an application language or a process in VHDL. Since Counter extends the Machine class, it, too, is a Machine. The count data member here holds the state of the counter (initialized to 0). The step() method is the core of the Machine: it gets called automatically once per "cycle" by the runtime environment to increment and update the stored count. When instantiated, this Machine will behave as though the following code were executed:

```
Counter counter = new Counter();
while (true)
   counter.step();
```

Note that this implicit active behavior differentiates a Machine from an Object, even though the implementation is in an object-oriented language.

3.3 Example 2: Filter Pipeline

Machines may have inputs and outputs. The following code implements an IIR low-pass filter that accepts an input data stream, filters it, and emits the filtered version:

```
class LowPassFilter extends Machine {
   int value = 0; // state variable
   protected int inputData; // "buffered" input.

   void input(int in) { inputData = in; }
   void step() { value = value / 2 + inputData / 2; }
   int output() { return value; }
}
```

This illustrates how inputs and outputs are declared and implemented. The protected keyword is overloaded to declare a variable used to buffer inputs to the machine[1]; the input() method does nothing more than take one or more data inputs to the Machine and store them in protected data members for later use by step(). The step() method uses the current state (value) and input (inputData) to compute and update the state of the machine. The output() method allows an output data value to be extracted from the Machine in a controlled way.

A Machine is also a data type and may be declared as member data just like a primitive data type. This allows us to create and connect multiple Machines to form arbitrary topologies of cooperating, concurrent processors. The following code shows how multiple LowPassFilter Machines may be composed to form a pipeline of ten low-pass filters:

```
class FilterPipeline extends Machine {
   LowPassFilter[] stages = new LowPassFilter[10];

   void input(int in) {
      stages[0].input(in);
      for (int i = 1; i < 10; i++)
         stages[i].input(stages[i-1].output());
   }
   void step() {}
   int output() { return stages[9].output(); }
}
```

In this example, the ten low-pass filters are declared as data members in the enclosing FilterPipeline Machine. The input() method relays the input to the FilterPipeline to the first stage of the internal pipeline, and then cascades the output of each stage to the input of the following stage. The step() method need not do anything here (and could be omitted), since all of the work is done by the embedded LowPassFilters (this happens automatically, since machines are active). Finally, the output() method delivers the output of the entire pipeline by simply relaying the output of the final stage. Thus the internal structure of FilterPipeline is completely hidden from the user, and FilterPipeline may be used just like any other fundamental Machine.

1 This is the only instance of deviation from standard OO semantics in our language, but remains backwards compatible with the base OO language since we disallow inheritance from classes other than the Machine class.

3.4 Rules of the model

A parallel program in our programming model consists of a *single instance* of a class that extends the Machine base class. Since machines may be nested, the total number of machines can be quite large. All Machine instances execute concurrently, regardless of where they are declared in the hierarchy, and behave as though the following pseudo-code were executed:

```
Machine rootMachine = new SomeMachine();
while (runnable) {
   rootMachine.input(...);
   for each machine in hierarchy
      machine.step();
}
```

The boolean variable runnable allows the machines to step for a predetermined number of steps, or until some condition has been met internally within the program.

Machine data members are always private to other Machines: a Machine may receive data from another Machine only through invocation of an output method.

Protected data members are writable only by an input() method, if any. An input() method may write protected data members, but may not read unprotected data members or perform any other computation.

Many issues have not been addressed here, among them: returning multiple outputs from a machine; implementing the runnable predicate; and integration with a host application.

4. COMPILER ISSUES

Our compiler generates configuration bits directly from source code without using an intermediate representation, such as VHDL, or other tools, such as a VHDL compiler, floorplanner, router, etc. Our current implementation targets the Virtex family of FPGA's and uses the JBits system from Xilinx to create the actual configuration bitfiles. Although JBits includes a router, we have implemented our own A* (depth-first search) router.

The compiler, written in Java, is conventionally structured into multiple passes, and separates target-dependent and -independent code using interfaces and the Factory Method design pattern [26] (Figure 1).

The remainder of this section focuses on three areas of compiler construction where design decisions differ from a compiler for a sequential machine: (1) internal representation of the computation; (2) optimization (target-independent and -dependent); and (3) code generation.

4.1 Internal representation

Since we desire to compile applications to hardware configurations that rival circuits designed by hand in performance, it is important

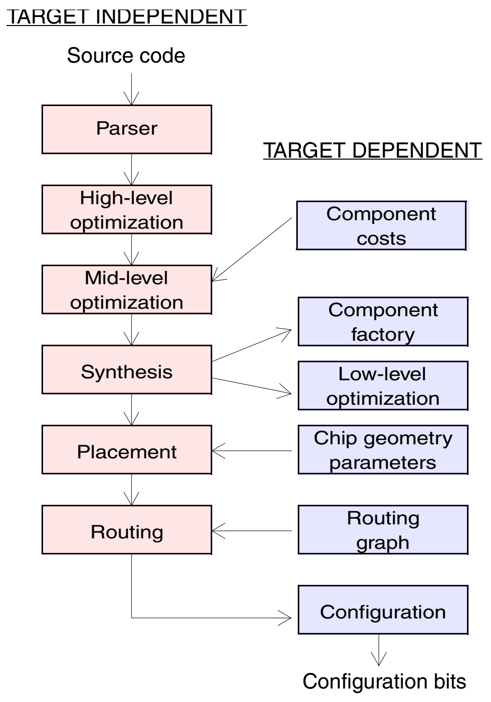


Figure 1: Compiler organization: The compiler has a multi-pass structure with separation between target-dependent and target-independent code to facilitate technology porting.

to squeeze out as much parallelism as possible, and eliminate sequential flow control, from the user's source code. We do this by converting the source to a directed hypergraph[2] internal representation (IR) that is pure data flow, containing no control flow structures whatever. This is done with a combination of techniques which we will describe:

- *Static single assignment (SSA) form*—a form that makes some optimizations much easier to implement.
- *Predication*—a technique for converting control flow to data flow.
- *Loop unrolling*—replacing a loop with the loop body replicated.
- *Array representation*—determining the best way to implement an array depending upon its access pattern.

Static single assignment (SSA) form. This form of IR requires that every variable within the computation be assigned to exactly once. Since application languages rarely require SSA form, the source must be transformed to SSA form during the construction of the IR. In spite of the ease with which many optimizations can be done in this form, it is often avoided in production compilers because of the increased number of assignments that it creates, causing problems with program size and register allocation [27]. In the hardware domain, though, this disadvantage does not exist: variables for intermediate results correspond to nothing more than

2 A directed hypergraph is a directed graph with one difference: an edge is defined to be a set of two *or more* vertices, with one vertex designated as the source.

Original code

```
int a = ...;
if (p)
    a = 5
else
    a = a + 1

int b = a;
```

SSA

int a_0 = ...;
if (p)
    a_1 = 5;
else
    a_2 = a_0 + 1;
a_3 = p ? a_1 : a_2;
int b_0 = a_3;

Figure 2: Static single assignment (SSA) form. The original source code, which has multiple assignments to the variable "a," is transformed by the compiler to a form where each variable is written only once.

wires that are required anyway to perform the computation. We should emphasize that the input language does *not* have to be in SSA form—our compiler accepts conventional multi-assignment code.

Conversion to non-minimal SSA form is easy: variable names in the original source are appended with an index; every time that variable is assigned to, the index is incremented (Figure 2). Correct referencing of the indexed variables, accomplished through careful construction of the symbol table, guarantees that the semantics of the program are preserved.

Predication. With predication [34], every statement in the original computation is tagged with a *guard* that conceptually controls whether or not that statement actually gets executed at runtime. In our IR, the guard is of the form if(p) where p is a *predicate* (a boolean expression) that is generated as part of the computation. The if(p) guard allows its statement to execute if its predicate, p, is true; otherwise it prevents its execution. The value of the predicate often cannot be deduced at compile time and, when necessary, must be computed at runtime.

Using predication, control flow statements (if-then-else, loops) can be completely eliminated from the IR [35]. Converting the IR to predicated SSA form is straightforward (Figure 3). Note that

Original code

```
int a = ...;
if (p)
    a = 5
else
    a = a + 1

int b = a;
```

Predicated SSA

int a_0 = ...: if (true)
a_1 = 5 : if (p)
a_2 = (a_0 + 1) : if (!p)
a_3 = p ? a_1 : a_2 : if (true)
b_0 = a_3 : if (true)

Generated circuit

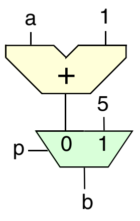


Figure 3: Predicated static single assignment (PSSA): Converting the original code into PSSA form eliminates the control flow of the if statement and exposes all dependences. In this case, both arms of the if statement are executed, and the predicate p is used only by the multiplexer. Since a value cannot be consumed before it is produced (e.g. b_0 = a_3 cannot be executed until a_3 has been computed), the PSSA from is actually an implicit hypergraph, and its sequential representation above could be arbitrarily reordered without affecting the semantics.

guards on instructions can be ignored during synthesis as long the guarded instruction does not write a user-visible state variable. When guards are ignored, all intermediate results (usually the vast bulk of the computation) will be computed regardless of the predicate state, generally requiring more area but providing a faster implementation. Guards for instructions which write user-visible state, though, must be implemented, usually with a multiplexer or a write-enable on a memory unit, or the semantics of the program will be destroyed (Figure 4).

Loop unrolling. Replicating each body of a loop by the number of times that the loop is executed enables extraction of additional parallelism. Of course, if the loop is executed a large number of times, the replicated body might require the synthesis of more circuitry than can fit in the target device, especially if the loop contains recurrences (where the computation of one iteration depends upon a result computed in a previous iteration). Such loops, in this programming model, must be implemented using a machine for the body of the loop. A simple example of this is the low-pass IIR filter example in section 3.3; such a filter might be implemented in a sequential model like this:

```
lowPassFilter(int in[1000000], int out[1000000]) {
    int state = 0;
    for (int i = 0; i < 1000000; i++) {
        state = in[i] / 2 + state / 2;
        out[i] = state;
    }
}
```

The loop here clearly cannot be unrolled because of the enormous code expansion. Transforming this loop into a Machine (as in section 3.3) circumvents this problem.

Array representation. Arrays may be implemented in hardware in many different ways: ROMs, RAMs, multiported register files, or a set of registers coupled with multiplexers. The appropriate implementation depends upon whether or not the array is ever written, how the array is accessed in the step() function, and the degree of micro-pipelining that has been used in the implementation of the machine containing it. If no micro-pipelining is used (so that the step() function can be executed in a single clock cycle), the array can be implemented in a RAM only if step() performs no

Original code

```
class Foo ... {
    int state;
        :
    void step() {
        if (state != 0)
            state --;
    }
}
```

Predicated SSA

int $state_0$

p = ($state_0$!= 0) : if (true);
$state_1$ = $state_0$ - 1 : if (p);
$state_2$ = p ? $state_1$: $state_0$
    : if (true);
state = $state_2$: if (true);

Generated circuit

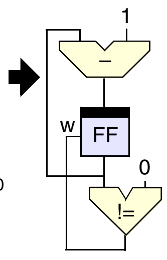


Figure 4: PSSA and state variables: In this case, the multiplexer required by the predicate can be eliminated if the storage element FF has a write enable w.

Arithmetic reductions

$x + 0$	=	x	$x - (-y)$	=	$x + y$
$x + (-y)$	=	$x - y$	$x - K$	=	$x + (-K)$
$-(-x)$	=	x	$x / 1$	=	x
$x \% 2^i$	=	$x \& (2^i - 1)$	$x / 2^i$	=	$x >> i$
$1 ? x : y$	=	x	$x << 0$	=	x
$0 ? x : y$	=	y	$x << K$	=	extract, zero fill
$x * 1$	=	x	$x >>> K$	=	extract, zero fill
$x * 0$	=	0	$x >> K$	=	extract, sign extend
$x - 0$	=	x	$x >> 0$	=	x
$0 - x$	=	$-x$			

Logical reductions

$x \& -1$	=	x	$x \mid -1$	=	-1	x ^ 0	=	x
$x \& 0$	=	0	$x \mid 0$	=	x	x ^ x	=	0
$x \& x$	=	x	$x \mid x$	=	x	x ^ !x	=	-1
$!x \& x$	=	0	$x \mid !x$	=	-1			
$!(!x) \& x$	=	x	x ^ -1	=	$!x$			

Figure 5: Arithmetic and logical identities for reducing the complexity of the IR. "K" represents a constant value. Commutative laws nearly double the number of identities shown here. The reductions are performed by traversing the IR, looking for the patterns on the left hand side of the above equations and replacing them with the simpler right hand side.

more than a single read or write of that array. Similarly, a constant array in a non-micro-pipelined realization can be implemented in a ROM if it is read only once, or in replicated ROMS if read more than once. Otherwise such arrays must be implemented as a set of registers, or as a multi-ported register file. Micro-pipelining relaxes this constraint and allows area/speed trade-offs to be made.

4.2 High-level optimizations

High-level (target-independent) optimization is performed by repeatedly traversing the IR hypergraph and using pattern matching to detect and apply transformations that reduce the size of the graph. Many standard compiler transformations are used: dead code elimination, constant folding (i.e., precomputing at compile time the results of constant expressions in the user's source, such as "5 * 2"), common subexpression elimination, logical/arithmetic identity reductions (Figure 5), constant propagation and strength reductions (e.g., replacing a multiplication by a constant with a small number of adds). Because our IR is free from control flow, we escape having to do any branch optimizations.

As mentioned before, loops are always completely unrolled (meaning that the body of the loop is replicated to produce one copy per iteration) to eliminate sequential control flow. Because of this, all loop bounds must be known at compile time.

Bit width reductions. Because of type casting, not all bits of the result of an arithmetic operation may be used. Consider the following Java code:

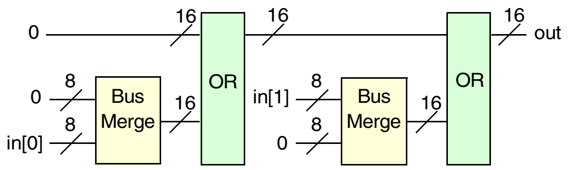
```
short a, b;
int c = a + b;
```

Synthesizing a 32-bit adder to produce c from the 16-bit input values would be wasteful. The compiler can detect this situation and synthesize the smallest needed adder.

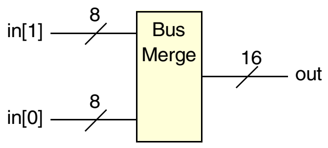
Bit optimizations. A class of optimizations not usually performed in compilers for sequential machines is bit-by-bit optimization of operands produced by logical operations. As an example of how this works and why it's useful, consider the following fragment of Java code:

```
byte in[2];
short out = 0;
for (int i = 0; i < 2; i++)
    out = out | (((int) in[i]) << i*8);
```

At first glance, this might seem difficult to parallelize because of the recurrence in the loop (the value of out read in the second iteration is the value of out produced by the first iteration), but this loop is actually just a C-style idiom for bus concatenation of two 8-bit values into a single 16-bit value—the recurrence is illusory. The lack of an operand concatenation operator in the C family of languages is another example of a semantic gap that must be compensated for in the compiler. After unrolling this loop and doing strength reduction of the shifts and multiplies, we wind up with a computation graph that looks like this:



When the optimizer considers the output of this graph, it recognizes that the 16 bits of out are driven by a logical operation, "OR," and is thus eligible for bit-by-bit optimization. By chasing back each output bit as far as it can to its source, and using the logical identity (x | 0 == x), the optimizer can determine that each bit is just a renaming of a bit in the variables in[0] or in[1], thus reducing the computation graph to this:

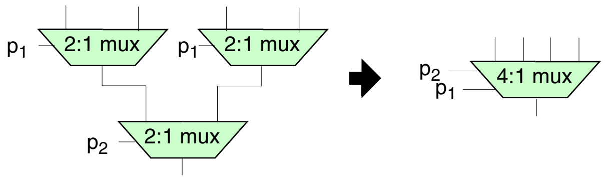


The bus-merger, though, requires no computation in an FPGA, so the original loop is optimized away to nothing but wires.

Multiplexer reductions. Nested if statements and switch statements can generate trees of 2:1 multiplexers. These can be reduced by traversing the IR, doing pattern matching to detect the trees, and replacing them with higher order multiplexers such as the following:

4.3 Mid-level optimizations

The mid-level (target-dependent) optimizations allow the compiler to make speed/area tradeoffs in the implemented circuit. The choices made here are target-dependent, but can be directed by a set of parameters determined by the target architecture. Some of



the optimizations which we will describe in the following subsections (Figure 6) include: micro-pipelining, digit-serial arithmetic, and micro-sequencing

Micro-pipelining. In programs containing no feedback, it is possible to insert additional register stages in the synthesized circuit which can increase the clock rate (since the paths between register stages are shorter) as well as the latency (since the number of register stages is increased). For programs that execute a large number of iterations, the latency is usually unimportant, and hence micro-pipelining can greatly increase the throughput[3]. Some languages, such as Handel-C, allow the user to pipeline their design by rewriting their code to include additional variable declarations for holding intermediate results (since a named variable gets mapped to a register). That approach is somewhat problematic in that it requires the programmer to either make educated guesses about the critical paths in the synthesized circuit, or to examine the actual generated circuit to find them. The rewriting also introduces opportunities for errors, and can could compromise the portability of the application to a different target platform. We believe that micro-pipelining is best automated in the compiler.

Digit-serial arithmetic. Because of their bit level granularity, FPGAs are usually not good candidates for numeric-intensive applications. However, if a numeric application is pipelineable, it's possible that a digit-serial implementation of the arithmetic operations could lead to an area-efficient implementation. The latency would be large, but overall throughput could be quite high.

Micro-sequencing. Programs that require more computational resources than are available in the target platform can sometimes be transformed so that expensive resources (such as pipelined multipliers) are time-shared by several independent computations. This slows down the overall computation, but is compensated by the corresponding reduction in area. The sharing factor is usually called the *initiation interval,* II, and represents the number of different computations that can time-share the resource, or, equivalently, the number of cycles that must separate distinct instances of the same computation. The mapping of the computation onto the shared resources addresses essentially the same problems as modulo scheduling [28] and space-time scheduling [5]. This optimization is generally less useful for FPGA's than for custom silicon if each function unit is individually multiplexed. The reason: multiplexers tend to be expensive in FPGA's relative to the cost of function units. However, this approach can pay off if entire machines are micro-sequenced rather than individual function units.

4.4 Code generation

Since the output of the compiler is a circuit rather a sequence of instructions, a directed hypergraph is a more appropriate represen-

3 We call this *micro-pipelining* to distinguish it from user-level pipelining, as in the FilterPipeline example.

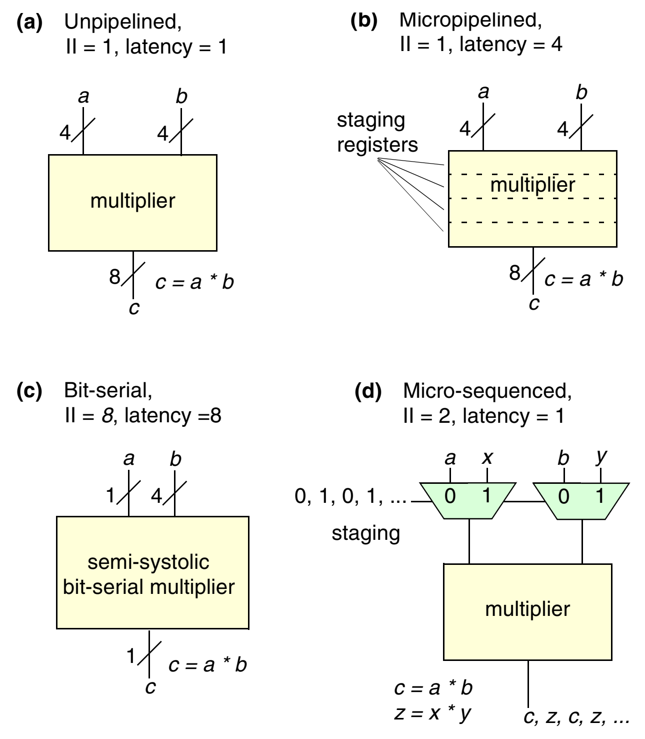


Figure 6: Mid-level Optimizations. To make speed/area trade-offs, the compiler may transform a simple implementation of a multiplier (a) into a pipelined version (b) that increases clock speed but also increases latency; a bit-serial version (c) that is very area efficient, but with long latency; or a micro-sequenced multiplier (d) that is shared by multiple computations, reducing the effective area per computation.

tation of the "code" than the usual sequential list of machine instructions.

To isolate the main body of the compiler from platform-dependent knowledge, code generation is done through an abstract interface that hides the idioms of the target. For example, the compiler may request the interface to construct an *adder* component; the target-dependent implementation may do this in any way it likes (e.g., using carry chains, if available, generating carry look-ahead adders, doing table lookup, etc.), but to the main part of the compiler, the result is just an *adder*. Thus porting code generation to a different platform only requires implementation of this interface.

4.5 Low-level optimizations

Low-level (target-dependent) optimizations must be custom crafted for each target platform to fully exploit the hardware. We have done the following three optimizations for our initial Xilinx Virtex target: LUT/register merging, register/LUT merging, and LUT combining.

LUT/register merging. Each lookup table (LUT) in Virtex is paired with a flip-flop; we call this combination a "FLUT." When a LUT within a FLUT containing an unused flip-flop drives the input of a flip-flop in a different FLUT with an unused LUT, the used LUT and flip-flop may be merged into a single FLUT (Figure 7a).

Register/LUT merging. Essentially the mirror image of the previous case: a flip-flop driving a LUT in a different FLUT gets merged with it into the same FLUT (Figure 7b). This is less effective than LUT/register merging since it compresses the circuit without reducing the number of signals that need to be routed. The compressed circuit usually takes much longer to route and sometimes executes at a lower clock rate (probably because of an inferior routing).

LUT combining. Connected LUTS that share inputs or that collectively require no more inputs than the amount available on a single LUT, can often be combined into a single LUT with a new truth table (Figure 7c).

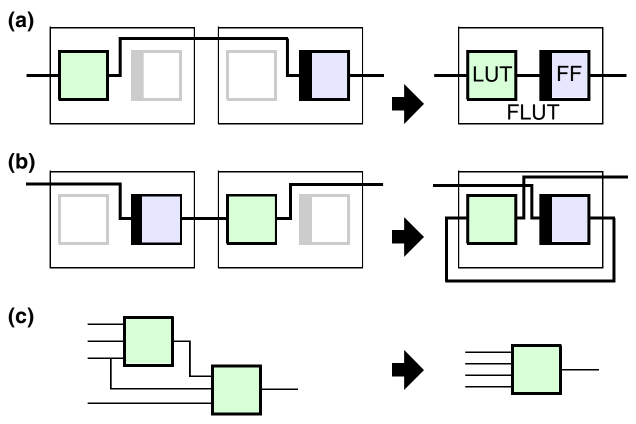


Figure 7: Virtex-specific optimizations: (a) merging a LUT driving a flip-flop into a single FLUT; (b) merging a flip-flop driving a LUT into a single FLUT; (c) LUT merging.

5. PERFORMANCE

We have chosen the Serpent encryption algorithm as our first performance benchmark for four reasons: (1) it is a large design, requiring more than half of the resources on our target FPGA (the Virtex XCV1000); (2) it has a large number of bus rotations, bus splits and bus merges, which must be expressed in C++ with loops, shifts and logical ORs—an excellent example of semantic mismatch to challenge the compiler; (3) it has fair amount of medium-grained parallelism that is naturally expressed as a pipeline of 32 "rounds"; and (4) Elbirt and Paar have done an independent implementation in VHDL [29], providing us with a baseline for comparison. We implemented the algorithm as described by Elbirt and Paar in ECB mode with the same degree of pipelining (one register stage per round in their implementation, which corresponds to one Machine per round in our model). We did not do any mid-level optimizations to keep the number of pipeline stages the same (Serpent in ECB mode can easily be micro-pipelined to increase its clock rate). We executed the compiled design on an ESL RC1000-PP card. Since we are using JBits for generating configuration bits, and since (as of this writing) JBits does not yet support I/O pins, we were forced to simulate external memory with a small Machine that generated a word stream to be encrypted. Implementing the algorithm in C++ required about one day. Source code for a "round" of the algorithm is shown in Figure 8.

Compilation time, from source code to configuration bits, was about 16 minutes on a 600 MHz Pentium processor, using the Symantec JIT compiler (Table 1). Based on some simple experiments, we estimate that the compiler (implemented in Java) would execute 2 to 6 times faster if recoded in C++. The vast bulk of compile time was consumed by placement and routing.

```
class Round : public Machine {
    uint128 outBlock;        // Data block output of round
    uint4 sbox[16];          // Sbox used by this round.
protected:
    uint128 inBlock;         // Data block input to round.
    uint128 key;             // Subkey for round.
public:
    // Construct a round with the given sbox.
    Round(uint4 sboxToUse[16]) {
        for (int i = 0; i < 16; i++)
            sbox[i] = sboxToUse[i];
    }

    // Collect and save inputs to the round.
    void input(uint128 inputBlock, uint128 subkey) {
        inBlock = inputBlock;    key = subkey;
    }

    // Advance the state of the Machine (1 round of encryption).
    void step() {
        uint128 mixed = inBlock ^ key;

        // Sbox mapping.
        uint32 x0 = 0, x1 = 0, x2 = 0, x3 = 0;
        for (int i = 0;  i < 32;  i++) {
            uint4 shiftAmount = (mixed >> (4*i)) & 0xf;
            uint4 mappedNibble = sbox[shiftAmount];
            // Form words (x0, x1, x2, x3) for linear transformation.
            x0 = x0 | ((uint32)((mappedNibble >> 0) & 0x1)) << i;
            x1 = x1 | ((uint32)((mappedNibble >> 1) & 0x1)) << i;
            x2 = x2 | ((uint32)((mappedNibble >> 2) & 0x1)) << i;
            x3 = x3 | ((uint32)((mappedNibble >> 3) & 0x1)) << i;
        }
        // Linear transformation.
        x0 = (x0 << 13) | (x0 >> 19);      x2 = (x2 << 3) | (x2 >> 29);
        x1 = x1 ^ x0 ^ x2;                 x3 = x3 ^ x2 ^ (x0 << 3);
        x1 = (x1 << 1) | (x1 >> 31);       x3 = (x3 << 7) | (x3 >> 25);
        x0 = x0 ^ x1 ^ x3;                 x2 = x2 ^ x3 ^ (x1 << 7);
        x0 = (x0 << 5) | (x0 >> 27);       x2 = (x2 << 22) | (x2 >> 10);

        // Rescramble bits into inter-round format.
        uint1 bits[128];
        for (i = 0;  i < 32;  i++) {
            bits[4*i] = (uint1)(x0 >> i);      bits[4*i+1] = (uint1)(x1 >> i);
            bits[4*i+2] = (uint1)(x2 >> i);    bits[4*i+3] = (uint1)(x3 >> i);
        }
        uint128 result = 0;
        for (i = 0;  i < 128;  i++)
            result = result | (((uint128) bits[i]) << i);
        outBlock = result;
    }

    // Get output of the Machine.
    uint128 output() { return outBlock; }
};
```

Figure 8: Source code for a non-final round of Serpent encryption. Our C++ library supports arbitrary-width integers (such as "uint128," an unsigned, 128 bit integer) implemented with templates, operator overloading and typedefs; this is not possible in Java. Loops are completely unrolled —none of the loops here have true recurrences, so there is no performance penalty. Note that the last two "for" loops generate only wires when optimized.

Table 1: Breakdown of compiler execution time for the Serpent encryption algorithm.

Compiler Pass	*Execution time (sec)*
Parsing, translation to IR	6
High-level optimizations	71
Circuit synthesis	3
Low-level optimizations	6
Placement	117
Routing	743
Configuration	23
Total	16m 9s

Table 2 shows the importance of optimizations when compiling a sequential language to a highly parallel machine. Without optimizations, a straightforward translation of the source code to hardware required 3 orders of magnitude more area! This comes about from the difficulty in expressing the inherently parallel algorithm in a C-family language, as mentioned earlier.

Table 2: Effectiveness of optimizations on the synthesized implementation of the Serpent encryption algorithm.

Optimization Level	*Circuit Size (CLBs)*
No optimizations	4,040,744
High-level only	10,494
High-level + low-level	3,200

Elbirt and Paar estimated the clock rate of their implementation using timing analysis tools that we did not have access to. To determine the performance of our implementation, we downloaded and executed the implementation on the Virtex XCV1000 FPGA on our RC1000-PP card, and increased the clock rate until the implementation stopped working correctly. A straightforward implementation of the algorithm (Figure 8) ran at 77 MHz (Table 3). By examining the circuit generated by the compiler, it was possible to slightly alter the source to achieve 95 MHz operation.[4] However, this type of tuning (which is analogous to examining the assembly code emitted by a conventional compiler and modifying the source in response) is not something we envision a typical user doing.

6. CONCLUSIONS

1. There is a semantic gap between the expressiveness of application programming languages and the inherent parallelism of reconfigurable hardware. The challenge is to squeeze the sequential specification of the application into a parallel implementation.

4 The optimization involved moving the exclusive-OR with the Round key (which normally occurs at the beginning of a Round) to a point in the previous Round following the linear transformation, and reordering the key bits.

Table 3: Performance of the Serpent encryption algorithm when implemented in VHDL and C++.

Implementation	*Circuit Size (CLBs)*	*Clock Rate*
VHDL	4,502	38 MHz
C++	3,200	77 MHz
C++ (tuned)	3,200	95 MHz

Automated extraction of fine-grained (instruction-level) parallelism is easy to do, but automated extraction of medium-grained parallelism is an active research topic that is still limited in applicability (to affine loop nests, primarily).

2. Hardware design languages, such as Verilog or VHDL, do not suffer this semantic gap, but tend to be too low-level for applications—they don't offer enough abstraction. Application programmers do not want to know about, and should not have to know about, implementation issues that hardware designers must confront.

3. Using an unmodified application language (preserving its semantics) combined with a simple programming model for expressing medium-grained parallelism: (1) allows the development of parallel applications on sequential machines, and (2) should allow compilation to hardware competitive with a handcrafted hardware design.

4. Optimizations are of course helpful when compiling onto sequential machines, but are absolutely essential when compiling onto parallel, fine-grained machines. The optimizer is a critical component in closing the semantic gap.

5. The following compiler techniques are especially applicable in this domain: predication, static single assignment (SSA), loop unrolling, and modulo and space-time scheduling (for II > 1). The newer techniques (at least in the realm of conventional compilers) of bit-width analysis, array representation, bit optimization, micropipelining, micro-sequencing and serial arithmetic offer interesting research opportunities.

7. ACKNOWLEDGEMENTS

We wish to thank Delon Levi and Eric Keller of Xilinx for the tremendous support they gave us for JBits.

8. REFERENCES

[1] R. Barua, W. Lee, S. Amarasinghe, A. Agarwal, "Maps: A compiler-managed memory system for Raw machines," *Proceedings of the Twenty-Sixth International Symposium on Computer Architecture (ISCA-26)*, June 1999.

[2] T. J. Callahan and J. Warzynek, "Instruction-level parallelism for reconfigurable computing," *International Workshop on Field Programmable Logic*, September 1998.

[3] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, A. Agarwal, "The RAW benchmark suite: computation structures for general purpose computing," *Proceedings, IEEE Workshop on FPGA-based Custom Computing Machines*, Napa, Ca, April 1997.

[4] J. Babb, M. Rinard, C.A. Moritz, W. Lee, M. Frank, R, Barua, S. Amarasinghe, "Parallelizing applications into silicon," *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 70–80, April 1999.

[5] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, S. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a Raw machine," *Proceedings of the 8th International Conference on Architectural support for Programming Languages and Operating Systems*, pp. 46–57, October 1998.

[6] S. Vernalde, P. Schaumont, I. Bolsens, "An object oriented programming approach for hardware design," *Proceedings of the IEEE Computer Society Workshop on VLSI'99*, April 1999.

[7] N. Wirth, "Hardware compilation: translating programs into circuits," *Computer*, pp 25–31, June 1998.

[8] M. Gokhale, B. Schott, "Data-parallel C on a reconfigurable logic array," *Journal of Supercomputing*, no. 9, pp. 291–313, 1995.

[9] J. Hammes, R. Rinker, W. Bohm, W. Najjar, B. Draper, "A High-Level, Algorithmic Programming Language and Compiler for Reconfigurable Systems," Computer Science Department, Colorado State University.

[10] J.M.P. Cardoso, H.C. Neto, "Fast hardware compilation of behaviors into an FPGA-based dynamic reconfigurable computing system," *Proceedings of the XII Symposium on Integrated Circuits and Systems Design (SBCCI'99)*, Natal-RN, Brazil, pp. 150–153, Sept. 29–Oct. 2, 1999.

[11] R. Schreiber, S. Aditya, B.R. Rau, V. Kathail, S. Mahlke, S. Abraham, G. Snider, S. Anik, "High-level synthesis of non-programmable hardware accelerators," *submitted to CODES2000.*

[12] BRASS Research Group, "Automatic C compilation to SW + reconfigurable HW," *http://brass.cs.berkeley.edu/compile.html.*

[13] D. Galloway, "The transmogrifier C hardware description language and compiler for FPGAs," *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '95),* Napa, California, pp. 136–144, April 1995.

[14] C. Iseli, E. Sanchez, "A C++ compiler for FPGA custom execution units synthesis," *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '95)*, pp. 173–179, April 1995.

[15] D. Davis, S. Edwards, J. Harris, "Forge-J," LavaLogic white paper available from: *http://www.lavalogic.com/product/wp_forge.html*, pp. 1–8, 2000.

[16] D. Davis, "Java for digital circuit design," LavaLogic ratepayer available from: *http://www.lavalogic.com/product/wp_java.html*, pp. 1–5, 2000.

[17] P. Clarke and R. Goering, "C Level fields C++ class library," *eeTimes.com*, available from: http://www.eet.com/story/OEG20000103S0020, Aug. 30, 2000.

[18] M. Brown, "Handel-C and the Alex APAC509," *Embedded Solutions Application Note 002*, pp. 1–4, Feb. 6, 1998.

[19] X. Zhu and B. Lin, "Hardware compilation for FPGA-based configurable computing machines," *Proceedings of the 36th Design Automation Conference*, pp. 697–702, June 1999.

[20] D. Ku, "HardwareC—a language for hardware design version 2.0," *HardwareC Language Manual*, available from: http://www.ics.uci.edu/~jian/olympus/hardwarec.html#manual.

[21] L.E. Thon, K. Rimey, L. Svensson, "From C to silicon,"

Chapter: 17, The PUMA Processor, available from: http://infopad.eecs.berkeley.edu/~burd/software/maker2html/ex10/CtoSilicon.html, Sept. 1995.

[22] P.L. Flake, S.J. Davidmann, D.J. Kelf, "Superlog, A Next Generation Systems Design Language," *International HDL Conference*, March 2000.

[23] S.E. Mitchell, "TAO—A model for the integration of concurrency and synchronisation in object-oriented programming," *Ph.D. Thesis, University of York, UK*, pp. 1–167, March 1995.

[24] J Heath, P. Kuekes, G. Snider, R. Williams, "A Defect-Tolerant Computer Architecture: Oppotunities for Nanotechnology," *Science*, June 12, 1998.

[25] T. Toffoli and N. Margolous, "Cellular Automata Machines," *MIT Press*, 1987.

[26] E. Gamma, "Design Patterns," *Addison-Wesley*, 1995.

[27] S. Muchnick, "Advanced Compiler Design and Implementation," *Morgan Kaufmann*, p. 258, 1997.

[28] B. Rau, "Iterative Modulo Scheduling," *Hewlett-Packard Laboratories Technical Report HPL-94-115*, 1994.

[29] A. Elbirt, "An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher," *Proceedings, Eighth International Symposium on Field Programmable Gate Arrays*, pp.33–40, February 9-11, 2000.

[30] C. Ebeling, "Mapping Applications to the RaPiD Configurable Architecture," *Proceedings, IEEE Workshop on FPGA's for Custom Computing Machines*, April, 1997.

[31] S. Goldstein, "PipeRench: A Coprocessor for Streaming Multimedia Accelleration," *Proceedings, 26th International symposium on Computer Architecture*

[32] O. Mencer, M. Morf, M. Flynn, "PAM-Blox: High Performance FPGA Design for Adaptive Computing," IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98), pp. 167-174, April 1998.

[33] P. Bellows, B. Hutchings, "JHDL—An HDL for Reconfigurable Systems," IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98), pp. 175-184, April 1998.

[34] S. Mahlke, "Exploiting Instruction-Level Parallelism in the Presence of Conditional Branches," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana IL, Sept. 1996.

[35] M. Brandis, "Optimizing Compilers for Structured Programming Languages," Ph.D. dissertation, Swiss Federal Institute of Technology Zurich, 1995.