The Challenges of Hardware Synthesis from C-like Languages

Stephen A. Edwards*
Department of Computer Science
Columbia University, New York

Abstract

The relentless increase in the complexity of integrated circuits we can fabricate imposes a continuing need for ways to describe complex hardware succinctly. Because of their ubiquity and flexibility, many have proposed to use the C and C++ languages as specification languages for digital hardware. Yet, tools based on this idea have seen little commercial interest.

In this paper, I argue that C/C++ is a poor choice for specifying hardware for synthesis and suggest a set of criteria that the next successful hardware description language should have.

1 Introduction

Familiarity is the main reason C-like languages have been proposed for hardware synthesis. Synthesize hardware from C, proponents claim, and we will effectively turn every C programmer into a hardware designer. Another common motivation is hardware/software codesign: today's systems are very often implemented as a mix of hardware and software, and it is often unclear at the onset which portions to implement in hardware and which can be software. Here, the claim is that using a single language for both simplifies the migration task.

In this paper, I argue that these claims are largely false and that C is a poor choice for specifying hardware. On the contrary, I claim that the semantics of C-like imperative languages are so distant from those of hardware that C-like thinking is actually detrimental to hardware design.

Executing a given piece of C code on a traditional sequential processor can be thought of as synthesizing hardware from C, but the techniques presented here instead strive for more customized implementations that exploit more parallelism, hardware's main advantage. Unfortunately, as discussed below, the C language has no support for parallelism and as such either the synthesis tool is responsible for finding it (a very difficult task) or the designer is forced to modify the algorithm and insert explicit parallelism. Neither solution is particular satisfactory and the latter deviates significantly from the objective of trivially turning C programmers into hardware designers.

The thesis of this paper is that giving C programmers tools is not enough to turn them into reasonable hardware designers. I show that efficient hardware is usually impossible to describe in an unmodified C-like language because the language inhibits the specification or automatic inference of adequate concurrency, timing, types, and communication. The most successful C-like languages, in fact, bear little syntactic or semantic resemblance to C, effectively forcing users to learn a new language anyway. As a result, techniques for synthesizing hardware from C either generate inefficient hardware or propose a language that merely adopts part of C syntax.

For space reasons, this paper is focused purely on the use of C-like languages for synthesis. I deliberately omit discussion of other important uses of a design language, such as validation and algorithm exploration. C-like languages are much more compelling for these tasks, and one in particular (SystemC) is now widely used, as are many ad hoc variants.

2 A Short History of C

Dennis Ritchie developed C in the early 1970 [18] based on experience with Ken Thompson's B language, which had itself evolved from Martin Richards' BCPL [17]. Ritchie describes all three as "close to the machine" in the sense that their abstractions are very similar to the data types and operations supplied by conventional processors.

A core principle in BCPL is its undifferentiated array of words memory model. Integers, pointers, and characters are all represented in a single word; the language is effectively typeless. This made perfect sense on the word-addressed machines BCPL was targeting at the time, but wasn't acceptable for the byte-addressed PDP-11 on which C was first developed.

Ritchie modified BCPL's "array of words" model to add the familiar character, integer, and floating-point types now supported by virtually every general-purpose processor. Ritchie considers C's treatment of arrays to be characteristic of the language. Unlike other languages that have explicit array types, arrays in C are almost a side-effect of its pointer semantics. While this model leads to simple, very efficient implementations, Ritchie observes that the prevalence of pointers in C means that compilers must use careful dataflow techniques to avoid aliasing problems while applying optimizations.

Ritchie lists a number of infelicities in the language caused by historical accident. For example, the use of *break* to separate cases in *switch* statements arose because Ritchie copied an early version of BCPL; later versions used *endcase*. The precedence of bitwise-AND is lower than the equality operator because the logical-AND operator was added later.

Many aspects of C are greatly simplified from their BCPL counterparts because of limited memory on the PDP-11 (24K, of which 12 were devoted to the nascent Unix kernel). For example, BCPL allowed arbitrary control-flow statements to be embedded within expressions. This facility does not exist in C because limited memory demanded a one-pass compiler.

*sedwards@cs.columbia.edu http://www.cs.columbia.edu/~sedwards
Edwards is supported by an NSF CAREER award, a grant from Intel corporation, an award from the SRC, and from New York State's NYSTAR program.

Language	Comment
Cones [25]	Early, combinational only
HardwareC [12]	Behavioral synthesis-centric
Transmogrifier C [7]	Limited scope
SystemC [8]	Verilog in C++
Ocapi [19]	Algorithmic structural descriptions
C2Verilog [23]	Comprehensive; company defunct
Cyber [26]	Little made public (NEC)
Handel-C [2]	C with CSP (Celoxica)
SpecC [6]	Resolutely refinement-based
Bach C [10]	Untimed semantics (Sharp)
CASH [1]	Synthesizes asynchronous circuits

Table 1: The languages/compilers considered in this paper.

Thus, C has at least four defining characteristics: a set of types that correspond with what the processor directly manipulates, pointers instead of a first-class array type, a number of language constructs that are historical accidents, and many others that are due to memory restrictions.

These characteristics are troubling when synthesizing hardware from C. Variable-width integers are natural in hardware, yet C only supports four sizes, all larger than a byte. C's memory model is a large, undifferentiated array of bytes, yet many small, varied memories are most effective in hardware. Finally, modern compilers can assume available memory is easily 10 000 times larger than what was available to Ritchie.

3 C-like Hardware Synthesis Languages

A variety of C-like hardware languages have been proposed since the late 1980s (Table 1). This section describes each in roughly chronological order. See also De Micheli [3].

Stroud et al.'s Cones [25] was one of the earliest. From a strict subset of C, it synthesized single functions into combinational blocks. It could handle conditionals; loops, which it unrolled; and arrays treated as bit vectors.

Ku and De Micheli developed HardwareC [12] for input to their Olympus synthesis system [4]. It is a behavioral hardware language with a C-like syntax and has extensive support for hardware-like structure and hierarchy.

Galloway's Transmogrifier C [7] is a fairly small C subset. It supports integer arithmetic, conditionals and loops. Unlike Cones, it generates sequential designs by inferring a state at function calls and at the the beginning of *while* loops.

SystemC [8] is a C++ dialect that supports hardware and system modeling. While its popularity seems to stem mainly from its facilities for simulation (it provides concurrency with lightweight threads [13]), a subset of the language can be synthesized. It uses the C++ class mechanism to model hierarchical structure and models hardware using combinational and sequential processes, much like Verilog or VHDL. The Cynlib language from Forte Design Systems is similar.

IMEC's Ocapi system [19] is also C++-based but takes a very different approach. Instead of being parsed, analyzed, and synthesized, the C++ program is run to generate in-memory data structures that represents the structure of the hardware system. Supplied classes provide mechanisms for specifying datapaths, finite-state machines, and similar constructs (Paško et al. [16] describes adding an extension for RAM interfaces). These data structures are then translated into languages such as Verilog and passed to conventional synthesis tools. Lipton et al.'s PDL++ system [14] takes a very similar approach.

The C2Verilog compiler developed at CompiLogic (later called C Level Design and, since November 2001, part of Synopsys) is one of the few that can claim broad support of ANSI C. It can translate pointers, recursion, dynamic memory allocation, and other thorny C constructs. Soderman and Panchul describe their compiler in a pair of 1998 papers [23, 24] and hold a broad patent covering C-to-Verilog-like translation [15] that describes their compiler in detail.

NEC's Cyber system [26] accepts a C variant dubbed BDL. While details of the language are not publicly available, the available information suggests it is a somewhat ad hoc translation that probably puts restrictions on the language, insists its users write C in particular idioms, and probably requires a fair amount of coaxing from users.

Celoxica's Handel-C [2] is a C variant that extends the language with constructs for parallel statements and OCCAM-like rendezvous communication. Handel-C's timing model is uniquely simple: each assignment statement takes one cycle.

Gajski et al.'s SpecC language [6] is a superset of ANSI C augmented with many system- and hardware-modeling constructs, including ones for finite-state machines, concurrency, pipelining, and structure. The latest language reference manual [5] lists thirty-three new keywords. SpecC imposes a refinement methodology. As such, the whole language is not directly synthesizable but through a series of both manual and automated rewrites, a SpecC description can be refined into one that can be synthesized.

Like Handel-C, Sharp's Bach C [10] is an ANSI-C variant with explicit concurrency and rendezvous-style communication. However, Bach C only imposes sequencing rather than assigning a particular number of cycles to each operation. And although it supports arrays, it does not support pointers.

Budiu et al.'s CASH compiler [1] is unique among the C synthesizers because it generates asynchronous hardware. It accepts ANSI C, identifies instruction-level parallelism, and generates an asynchronous dataflow circuit.

4 Concurrency

The biggest difference between hardware and software is its execution model: software is based on a sequential, memory-based execution model derived from Turing machines, while hardware is fundamentally concurrent. Thus, sequential algorithms that are efficient in software are rarely the best choice in hardware. This has serious implications for any software programmer designing hardware: his familiar toolkit of algorithms is suddenly useless.

Why is so little software developed for parallel hardware? The plummeting cost of parallel hardware would make this appear even more attractive, yet concurrent programming has had limited success compared to its sequential counterpart. One fundamental reason is the conceptual difficulty humans have

in conceiving of parallel algorithms, and many more sequential algorithms are known than parallel algorithms. Another challenge is disagreement about the preferred model for parallel programming (e.g., shared memory versus message-passing); the panoply of parallel programming languages, none of which has emerged as a clear winner [22], is indicative of this.

Instead of exposing concurrency to the programmer and encouraging the use of parallel algorithms, the most successful approach has been automatically exposing parallelism in sequential code. Since C does not naturally support concurrency, using such a technique is virtually mandatory for synthesizing efficient hardware. Unfortunately these techniques are limited.

4.1 Finding Parallelism in Sequential Code

There are three main approaches to exposing parallelism in sequential code, distinguished by their granularity. Instruction-level parallelism (ILP) dispatches groups of nearby instructions simultaneously. While this has become the preferred approach in the computer architecture community, it is recognized that are fundamental limits to the amount of ILP that can be exposed in typical programs [27, 28], and that adding hardware to approach these limits, nowadays most often through speculation, results in diminishing returns.

Pipelining, the second approach, requires less hardware than ILP but may be less effective. A pipeline dispatches instructions in sequence but overlaps them: the second instruction is initiated before the first completes. Like ILP, inter-instruction dependencies and control-flow transfers tend to limit the maximum amount of achievable parallelism. Pipelines work well for regular loops, such as those in scientific or signal processing applications [11], but are less effective in general.

Instead of single instructions, process-level parallelism dispatches multiple threads of control simultaneously. This can be much more effective than finer-grain parallelism depending on the algorithm, but it is very difficult to identify automatically. Hall et al. [9] attempt to invoke multiple iterations of outer loops simultaneously, but unless the code is written to avoid dependencies, this may not be ineffective. Exposing process-level parallelism is thus usually the programmer's responsibility, and is usually controlled through the operating system (e.g., POSIX threads) or the language itself (e.g., Java).

4.2 Approaches to Concurrency

The C-to-hardware compilers considered here take two approaches to concurrency. The first approach adds parallel constructs to the language, thereby forcing the programmer to expose most concurrency. HardwareC, SystemC, and Ocapi all provide process-level parallel constructs; Handel-C, SpecC, and Bach C additionally provide statement-level parallel constructs. SystemC's parallelism resembles that in standard hardware description languages such as Verilog: a system is a collection of clock-edge-triggered processes. Handel-C, SpecC, and Bach C's approaches are more software-like, providing constructs that dispatch collections of instructions in parallel.

The other approach lets the compiler identify parallelism. While the languages that provide parallel constructs also identify some parallelism, Cones, Transmogrifier C, C2Verilog, and CASH rely on the compiler to expose all possible parallelism. The Cones compiler takes the most extreme approach, flattening an entire C function with loops and conditionals into a single two-level combinational function evaluated in parallel. The CASH compiler, by contrast, takes an approach closer to compilers for VLIW processors, carefully examining inter-instruction dependencies and scheduling instructions to maximize parallelism. None of these compilers attempt to identify process-level parallelism.

```
for (i = 0 ; i < 8 ; i++) {
  a[i] = c[i];
  b[i] = d[i] || f[i];
}
```

Figure 1: In how many cycles does this execute?
Cones says one (it is combinational), Transmogrifier C says eight (one per iteration), and Handel-C says twenty-five (one per assignment).

Neither approach is satisfactory. The latter group places the burden on the compiler and therefore limits the parallelism achievable in general if normal, sequential algorithms are used. While this could be mitigated by careful selection of algorithms that can be easily parallelized, such thinking would be foreign to most software programmers, and may actually be harder than thinking in an explicitly concurrent language.

The former group, by adding parallel constructs to C, introduces a fundamental and far-reaching change to the language, again demanding substantially different thinking on the part of the programmer. Even if s/he is experienced with concurrent programming with, say, POSIX threads, the parallel constructs in hardware-like languages differ greatly from the thread-and-shared-memory model of threads typical of software.

Any reasonable hardware specification language must be able to express parallel algorithms since they are the most efficient for hardware. C-like languages, because of their inherent sequentiality, fail this requirement.

Which concurrency model the next hardware design language should employ remains an open question, but it seems clear that the usual software model—asynchronously running threads communicating through shared memory—is not it.

5 Timing

The C language is mute on the subject of time. It guarantees causality among most sequences of statements, but says nothing about the amount of time it takes to execute each. This flexibility simplifies life for compilers and programmers alike, but makes it fairly difficult to achieve specific timing constraints. C's compilation technique is transparent enough to make gross performance improvements easy to understand and achieve, and differences in efficiency of sequential algorithms is a well-studied problem, but wringing another 5% speed-up from any piece of code can be quite difficult.

Achieving a performance target is fundamental to hardware design. Miss a timing constraint by 3% and the circuit will fail to operate or the product will fail to sell. Achieving a particular

performance target under power and cost constraints is usually the only reason to implement a particular piece of functionality in hardware as opposed to using an off-the-shelf processor. Thus, any reasonable technique for specifying hardware needs mechanisms for specifying and achieving timing constraints.

This disparity leads to yet another fundamental question in using C-like languages for hardware design: where to put the clock cycles. With two exceptions (Cones only generates combinational logic; CASH generates self-timed logic), the compilers described here define generate synchronous logic in which the clock cycle boundaries have been defined.

5.1 Approaches to Timing Control

The compilers considered here use a variety of techniques for inserting clock cycle boundaries, ranging from fully explicit to a variety of rules for fully implicit.

Ocapi's clocks are the most explicit: a designer specifies explicit state machines and each state gets a cycle. At some point in the SpecC refinement flow, the state machines are also explicit, although clock boundaries may not be explicit earlier in the flow. The clocks in the Cones system are also explicit, although in an odd way: since Cones generates only combinational logic, clocks are implicitly at function boundaries. SystemC's clock boundaries are also explicit: like Cones, combinational processes' clock boundaries are at the edges and in sequential processes, explicit *wait* statements delay a prescribed number of cycles.

HardwareC allows the user to specify clock constraints, an approach common in high-level synthesis tools. For example, a user can require that three particular statements should execute in two cycles. While this presents a greater challenge to the compiler and is sometimes more subtle for the designer, it allows flexibility that may lead to a more optimal design. Bach C takes a similar approach.

Like HardwareC, the C2Verilog compiler also inserts cycles using fairly complex rules and provides mechanisms for imposing timing constraints. Unlike HardwareC, however, these constraints are outside the language.

Transmogrifier C and Handel-C use fixed implicit rules for inserting clocks. Handel-C's are the simplest: assignment and *delay* statements each take a cycle; everything else executes in the same clock cycle. Transmogrifier C's rules are nearly as simple: each loop iteration and function call takes a cycle. Such simple rules can make it difficult to achieve a particular timing constraint, unfortunately: assignment statements may need to be fused to speed up a Handel-C specification, and Transmogrifier C may require loops to be manually unrolled.

The ability to specify or constrain detailed timing in hardware is another fundamental requirement. While slow software is an annoyance, slow hardware is a disaster. When something happens in hardware is usually as important as what happens. This is another big philosophical difference between software and hardware, and again hardware requires different skills.

The next hardware specification language needs the ability to specify detailed timing, both explicitly and through constraints, but perhaps should not be mandatory everywhere. Unfortunately, the best-effort model of software is inadequate.

6 Types

Data types are another central difference between hardware and software languages. The most fundamental type in hardware is a single bit traveling through a memoryless wire. By contrast, the base types in C and C++ are bytes and multiples thereof stored in memory. While C's base types can be implemented in hardware, C has almost no support for types smaller than a byte[1]. As a result, straight C code can easily be interpreted as bloated hardware.

The situation in C++ is better. C++ supports a one-bit *bool* type and its class mechanism makes it possible to add more types such as arbitrary-width integers to the language.

The compilers considered here take three approaches to introducing hardware types to C programs. The first approach, and perhaps the purest, neither modify nor augment C's types but allow the compiler or designer to adjust the width of the integer types outside the language. For example, the C2Verilog compiler provides a GUI that allows the user to set the width of each variable used in the program. The width of each integer in Transmogrifier C can be set through a preprocessor pragma.

The second approach is to add hardware types to the C language. HardwareC, for instance, adds a boolean vector type. Both Handel-C and Bach C add integers with an explicit width. SpecC adds all these types and many others that cannot be synthesized, such as pure events and simulated time.

The third approach, taken by the C++-based languages, is to provide hardware-like types through C++'s type system. The SystemC libraries include variable width integers and an extensive collection of types for fixed-point fractional numbers. Ocapi, since it is an algorithmic mechanism for generating structure, also effectively takes this approach, allowing the user to explicitly request wires, buses, and whatnot.

Each approach, however, constitutes a fairly radical departure from the "call it an integer and forget about it" approach of C. Even the languages that support only C types compel a user to provide the actual size of each integer. Worrying about the width of each variable in a program is not something a typical C programmer does.

Compared to timing and concurrency, however, adding appropriate hardware types is a fairly easy problem to solve when adapting C to hardware. C++'s type system is flexible enough to accommodate hardware types, and minor extensions to C suffice. A bigger question, which none of the languages adequately addresses, is how to apply higher-level types such as classes and interfaces to hardware description. SystemC has some facilities for inheritance, but since its mechanism is simply the one used for software, it is not clear that it is convenient for adding or modifying behavior of existing pieces of hardware. Incidentally SystemC has supported templates, more abstract modeling of communication channels, and so forth since version 2.0, but they are not typically synthesizable.

The next hardware description language needs a rich type system that allows precise definition of hardware types, but it should also assist in ensuring program correctness.

[1] With one exception: the number of bits for each field in a *struct* may be specified explicitly. Oddly, none of these languages even mimic this syntax.

7 Communication

C-like languages are built on the very flexible random-access memory model of communication. The language models all memory locations as being equally costly to access, but modern memory hierarchies make this a lie. At any point in time, it make take hundreds or even thousands of times longer to access certain locations. And although the behavior of these memories, specifically caches, can often be predicted and used more efficiently, this is very difficult and C-like languages provide scant support for it.

In hardware, long, nondeterministic communication delays are anathema. Timing predictability is mandatory, so large, uniform-looking memory spaces are rarely the primary communication mechanism. Instead, a variety of mechanisms are used, ranging from simple wires to complex protocols, depending on the more precise needs of the system. An important characteristic of this approach is the need to understand a system's communication channels and patterns before it is running, since communication channels must be hard-wired.

7.1 The Problem with Pointers

Communication patterns within software, unfortunately, are often very difficult to determine a priori because of the frequent use of pointers. These are memory addresses computed at run-time, and as such are often data-dependent and simply cannot be known completely before a system is running. Implementing such behavior in hardware mandates at least small memory regions.

Aliasing, when a single value can be accessed through multiple sources, is an even more serious problem. Without a good understanding of when a variable can be aliased, a hardware compiler is forced to place that variable into a large, central memory, which is necessarily slower than a small memory local to the computational units that read and feed it.

One of C's strengths is its memory model that allows complicated pointer arithmetic and essentially uncontrolled access to memory. While very useful for systems programs such as operating systems, such abilities make it especially difficult to analyze the communication patterns of an arbitrary C program. This problem is so great, in fact, that software compilers often have a much easier time analyzing a FORTRAN program rather than an equivalent C program.

Any technique that implements a C-like program in hardware must either analyze the program to understand all possible communication pathways, resort to large, slow memories, or some combination of both.

Luc Semeria et al. [20, 21] have applied pointer analysis algorithms from the software compiler literature to estimate the communication patterns of C programs for hardware synthesis. Pointer analysis identifies to which data each pointer may refer, allowing memory to be divided. While an impressive body of work, it illustrates the difficulty of the problem. Exact pointer analysis is undecidable, so approximations are used. These are necessarily conservative and hence may miss opportunities to split memory regions, leading to higher-cost implementations. Finally, pointer analysis is a costly algorithm with many variants.

7.2 Communication Costs

The event-oriented style of communication in software is another key difference. Every bit of data communicated among parts of a software program has a cost (i.e., a read or write operation to registers or memory) and as such, communication must be explicitly requested in software. Communicating the first bit is very costly in hardware because it requires the addition of a wire, but after that, communication is actually more costly to disable than continue.

This difference leads to a fairly different set of concerns. Good communication design in hardware amounts to trying to minimize the number of pathways among parts of the design, where good design in software minimizes the number of transactions. For example, good design in software tries to avoid forwarding through copying, preferring instead to pass a reference to the data being forwarded. This is a good strategy for hardware that stores large blocks of data in a memory, but rarely in other cases. Instead, good design in hardware considers alternate encodings of data, such as serialization.

7.3 Approaches to Communication

The languages considered here fall broadly into two groups: those that effectively ignore C's memory model and look only at communication through variables, and those that adopt the full C memory model.

Languages that ignore C's memory model do not support arrays or pointers and instead only look at how local variables are used to communicate between statements. Cones is the simplest: all variables, arrays included, are interpreted as wires. HardwareC and Transmogrifier C do not support arrays or memories. Ocapi also falls into this class, although arrays and pointers can be used to assist during system construction.

The other group of languages go to great lengths to preserve C's memory model. The CASH compiler is the most brute-force: it synthesizes one, large memory and puts all variables and arrays into it. The Handel-C and C2Verilog compilers can split up memory into multiple regions and assign each to a separate memory element. Handel-C adds explicit constructs to the language for specifying these. SystemC also supports the explicit declaration of separate memory regions.

Other languages provide communication primitives whose semantics differ greatly from the memory style of communication used in C. HardwareC, Handel-C, and Bach C all provide blocking rendezvous-style (i.e., unbuffered) communication primitives for communicating between concurrently-running processes. SpecC and later versions of SystemC provide a large library of communication primitives.

Again, the difference between appropriate design for software and hardware is substantial. Software designers usually ignore memory access patterns. Although this can slow overall memory access speed, it is usually acceptable. Good hardware design, by contrast, usually starts with a block diagram detailing every communication channel and attempts to minimize communication pathways.

So software designers usually ignore the fundamental communication cost issues common in hardware. Furthermore, automatically extracting efficient communication structures from

software is challenging because of the pointer problem in C-like languages. While pointer analysis can help to mitigate the problem, it is imprecise and cannot improve an algorithm with poor communication patterns.

The next hardware specification language should make it easy to specify efficient communication patterns.

8 Meta-data

Any given high-level construct can be implemented in many different ways. However, because hardware is at a much lower level than software, there are many more ways to implement a particular C construct in hardware. For example, consider an addition operation. A processor probably only has one useful addition instruction, whereas in hardware there are a dizzying number of different adder architectures (e.g., ripple-carry, carry lookahead, and carry save).

The translation process for hardware therefore has more decisions to make than translation for software. Making many decisions correctly is fundamentally difficult and computationally expensive. Furthermore, the right set of decisions varies with design constraints. For example, a designer might prefer a ripple-carry adder if area and power were at a premium and speed was of little concern, but a carry lookahead adder might be preferred if speed were of greater concern.

While much effort has been put into improving optimization algorithms, it remains unrealistic to expect all of these decisions to be automated. Instead, designers need mechanisms that allow them to ask for exactly what they want.

Such designer guidance takes two forms: through manual rewriting of high-level constructs into the desired lower-level ones (e.g., replacing a "+" operator with a collection of gates that implements a carry-lookahead adder), or through annotations such as constraints or hints about how to implement a particular construct. Both approaches are common register-transfer level design. Designers routinely specify complex datapaths at the gate level instead of using higher-level constructs. Constraint information, often supplied in an auxiliary file, usually drives logic optimization algorithms.

One of the anonymous reviewers suggested that C++'s operator overloading mechanism could be used to specify, for example, when a carry-lookahead adder was to be used to implement an addition. Unfortunately, I believe it would be very difficult. C++'s overloading mechanism uses argument types to resolve ambiguities, which is natural when you want different data types to be treated differently, but the choice of particular algorithm in hardware is usually driven by resource constraints (e.g., area or delay) rather than data representation (although of course data representation matters). Concurrency is the fundamental problem: in software, there is little reason to have multiple implementations of the same algorithm, but it happens all the time in hardware. Not surprisingly, C++ does not really support this sort of thing.

The languages considered here take two approaches to specifying such meta-data. One group places it within the program itself, hiding it in comments, pragmas, or added constructs, and the other group places it outside the program, either in a text file or in a database populated by graphical user interface.

Challenge	**Comment**
The concurrency model	Specifying parallel algorithms
Specifying timing	How many clock cycles?
Types	Need bits and bit-precise vectors
Communication patterns	Want isolated memories
Hints and constraints	How to implement something

Table 2: The big challenges in hardware languages

C does have a standard way of supplying extra information to the compiler: the `#pragma` directive. By definition, a compiler ignores such lines unless it understands them. Transmogrifier C uses it to specify the width of integers, and Bach C uses it to specify timing and mapping constraints.

HardwareC provides three language-level constructs: timing constraints, resource constraints, and arbitrary string-based attributes, whose semantics are much like a C `#pragma`.

SpecC takes the other approach: many tools for synthesizing and refining SpecC have the user to use a GUI to specify how various constructs are to be interpreted.

Constructs such as addition that are low-level in software are effectively high-level in hardware and as such, there must be some mechanism for conveying designer intent to any hardware synthesis procedure, regardless of the source language. The next hardware specification language needs some way of guiding the synthesis procedure to select among different implementations, trading off between, say, power and speed.

9 Conclusions

Table 2 lists the key challenges of a successful hardware specification language. In this paper, I discussed how a variety of C-like languages and compilers proposed for specifying hardware tried to address these issues and argued why choosing a C-like language over one specifically designed to specify hardware is not helpful. In particular, giving experienced software designers hardware synthesis tools is unlikely to turn them into effective hardware designers.

Why bother generating hardware from C? It is clearly not necessary since there are many excellent processors and software compilers. This is certainly the cheapest and easiest way to run a C program. So why consider using hardware?

Efficiency is the logical answer. Although general-purpose processors get the job done, well-designed customized hardware can always do the job faster using fewer transistors and less energy. Thus the utility of any hardware synthesis procedure depends on how well it is able to produce efficient hardware specialized for the particular application.

9.1 Summary

To recap the difficulties with C-like languages, concurrency is fundamental for efficient hardware, yet C-like languages impose sequential semantics and nearly demand the use of sequential algorithms. Unfortunately, automatically exposing concurrency in sequential programs is limited in its effectiveness, so an effective language almost surely requires explicit concurrency, something missing from most C-like languages.

Adding such a construct is easy, but teaching software programmers to use concurrent algorithms is much harder.

Careful timing design is also required for efficient hardware, yet C-like languages provide essentially no control over timing, so the language needs to have timing control added to it. The problem amounts to where to put the clock cycles, and the languages have a variety of solutions, both implicit and explicit. The bigger problem, though, is changing programmer habits to consider such timing details.

Using software-like types (e.g., bytes and multiples thereof) is also a problem in hardware, which wants to manipulate individual bits for efficiency, but a much easier problem to solve for C-like languages. Some languages add the ability to specify the number of bits used for each integer, for example, and C++'s type system is flexible enough to allow hardware types to be defined. The type problem is the easiest to address.

Communication also presents a challenge. C's very flexible global-memory communication model is not very efficient for hardware. Instead, memory should be broken into smaller regions, often as small as a single variable. Compilers can do this to a limited degree, but efficiency often demands explicit control over this. A fundamental problem, again, is that C programmers generally do not worry about memory, and that C programs are rarely written with memory behavior in mind.

A high-level hardware description language must allow the designer to provide constraints or hints to the synthesis system because of the wide semantic gap between a C program and efficient hardware. Since there are many ways to implement in hardware a construct such as addition, the synthesis system needs some way to select an implementation. Constraints and hints are the two main ways to control the algorithm, yet standard C has no such facility.

While presenting designers with a higher level of abstraction is obviously desirable, presenting them with an inappropriate level of abstraction (e.g., one in which they cannot effectively ask for what they want), is of little help. Unfortunately, C-like languages do not lend themselves to the synthesis of efficient hardware.

In summary, I believe the next great hardware specification language will not closely resemble C or any other familiar software language. Software languages work well only for software, and a hardware language that does not produce efficient hardware is of little use. It is my hope that this paper will prompt a more effective discussion about the design of the next great hardware description language.

Another important future issue will be the ability of the language to effectively build systems from existing pieces (a.k.a. IP-based design), which none of these languages really addresses. This appears necessary to raise designer productivity to the level needed for the next generation of chips.

Lording over all these issues, however, is verification. What we really need are languages that let us create correct systems more quickly, by making it easier to check, identify, and correct mistakes. Raising the level of abstraction and facilitating efficient simulation are two well-known ways of achieving this, but are there others?

References

[1] Mihai Budiu and Seth C. Goldstein. Compiling application-specific hardware. In *Proceedings of the 12th International Conference on Field Programmable Logic and Applications (FPL)*, volume 2438 of *Lecture Notes in Computer Science*, pages 853–863, Montpellier, France, September 2002. Springer-Verlag.

[2] Celoxica, http://www.celoxica.com. *Handel-C Language Reference Manual*, 2003. RM-1003-4.0.

[3] Giovanni De Micheli. Hardware synthesis from C/C++ models. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 382–383, Munich, Germany, March 1999.

[4] Giovanni De Micheli, David Ku, Frédéric Mailhot, and Thomas Truong. The Olympus synthesis system. *IEEE Design & Test of Computers*, 7(5):37–53, October 1990.

[5] Rainer Dömer, Andreas Gerstlauer, and Daniel Gajski. *SpecC Language Reference Manual*. SpecC consortium, version 2.0 edition, March 2001.

[6] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Methodology*. Kluwer, Boston, Massachusetts, 2000.

[7] David Galloway. The Transmogrifier C hardware description language and compiler for FPGAs. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 136–144, Napa, California, April 1995.

[8] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer, Boston, Massachusetts, 2002.

[9] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, December 1995.

[10] Takashi Kambe, Akihisa Yamada, Koichi Nishida, Kazuhisa Okada, Mitsuhisa Ohnishi, Andrew Kay, Paul Boca, Vince Zammit, and Toshio Nomura. A C-based synthesis system, Bach, and its application. In *Proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC)*, pages 151–155, Yokohama, Japan, 2001. ACM Press.

[11] Ken Kennedy and Randy Allen. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2001.

[12] David C. Ku and Giovanni De Micheli. HardwareC: A language for hardware design. Technical Report CSTL-TR-90-419, Computer Systems Lab, Stanford University, California, August 1990. Version 2.0.

[13] Stan Liao, Steve Tjiang, and Rajesh Gupta. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Proceedings of the 34th Design Automation Conference*, Anaheim, California, June 1997.

[14] Richard J. Lipton, Dimotrios N. Serpanos, and Wayne H. Wolf. PDL++: an optimizing generator language for register transfer design. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, pages 1135–1138 vol. 2, New Orleans, Louisiana, May 1990.

[15] Yuri Panchul, Donald A. Soderman, and Denis R. Coleman. System for converting hardware designs in high-level programming language to hardware implementations. US Patent 6,226,776, May 2001.

[16] Robert Paško, Serge Vernalde, and Patrick Schaumont. Techniques to evolve a C++ based system design language. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 302–309, Paris, France, March 2002.

[17] Martin Richards and Colin Whitby-Strevens. *BCPL: The Language and its Compiler*. Cambridge University Press, 1979.

[18] Dennis M. Ritchie. The development of the C language. In *History of Programming Languages II*, Cambridge, Massachusetts, April 1993.

[19] Patrick Schaumont, Serge Vernalde, Luc Rijnders, Marc Engels, and Ivo Bolsens. A programming environment for the design of complex high speed ASICs. In *Proceedings of the 35th Design Automation Conference*, pages 315–320, San Francisco, California, June 1998.

[20] Luc Séméria and Giovanni De Micheli. Resolution, optimization, and encoding of pointer variables for the behavioral synthesis from C. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):213–233, February 2001.

[21] Luc Séméria, Koichi Sato, and Giovanni De Micheli. Synthesis of hardware models in C with pointers and complex data structures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(6):743–756, dec 2001.

[22] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[23] Donald Soderman and Yuri Panchul. Implementing C algorithms in reconfigurable hardware using C2Verilog. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 339–342, Los Alamitos, CA, April 1998.

[24] Donald Soderman and Yuri Panchul. Implementing C designs in hardware: a full-featured ANSI C to RTL Verilog compiler in action. In *Proceedings of the 1998 International Verilog HDL Conference (IVC)*, pages 22–29, Santa Clara, California, March 1998.

[25] Charles E. Stroud, Ronald R. Munoz, and David A. Pierce. Behavioral model synthesis with cones. *IEEE Design & Test of Computers*, 5(3):22–30, July 1988.

[26] Kazutoshi Wakabayashi. C-based synthesis experiences with a behavior synthesizer, “Cyber”. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 390–393, Munich, Germany, March 1999.

[27] David. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, SIGPLAN Notices, 26(4):176–189, New York, NY, 1991. ACM Press.

[28] David W. Wall. Speculative execution and instruction-level parallelism. Technical Report TN–42, DEC Western Research Laboratory, Palo Alto, California, March 1994.