

The Hybridthreads Compiler

Jim Stevens, Fabrice Baijot
Information and Telecommunication Technology Center - University of Kansas
2335 Irving Hill Road, Lawrence, KS
{jstevens,bricefab}@ittc.ku.edu

Abstract

The ability to build custom hardware on an application-specific basis has been greatly improved through the advent of Field Programmable Gate Arrays (FPGAs) and Hardware Description Languages (HDLs). However this ability is still best achieved if domain-specific knowledge of hardware design is known. Unfortunately most software engineers have limited knowledge of the hardware design process, making FPGA programming a challenging and sometimes impossible task. To facilitate the use of FPGAs by software engineers, a tool that translates traditional programming languages into hardware is needed. The Hybridthreads Compiler is designed to solve this problem by implementing full support for mapping high-level languages into HDLs in the context of the Hybridthreads system.

1 Introduction

The purpose of this document is to describe the requirements, design, and implementation of the Hybridthreads Compiler (HTC). HTC generates a custom hardware thread core from a software implementation of a thread. Currently, this works by translating the programming language C into the hardware description language VHDL. The resulting thread(s) can then be integrated into a build of the Hybridthreads system and executed on an FPGA.

The Hybridthreads platform (Hthreads) is an integrated tool suite and hardware/software co-design runtime kernel. Hthreads allows programmers to drive the generation of custom multi-core systems on programmable chip architectures from a higher level multi-threaded programming model. With Hthreads programmers design, implement, and test their application with a POSIX threads (Pthreads) compatible API. Their code can then be compiled to either run on a general-purpose CPU or be synthesized into a thread-specific hardware core. This hardware thread is generated using HTC. Communication and synchroniza-

tion between software and hardware threads are specified by the programmer's code in the form of Hthreads API calls. These calls are implemented for the programmer in either library routines that are linked in during the build process for software threads, or by the Hardware Thread Interface (HWTI) that is synthesized with each hardware thread. More details on the Hthreads system may be found at [7].

HTC is actually two independent modules: HIFGEN and HIF2VHDL. The two modules communicate using the Hardware Intermediate Form (HIF) language that was defined for this purpose. HIF acts as both an intermediate form for HDL generation and an object file to allow for separate compilation of hardware threads. HIFGEN is a modified version of GCC that generates HIF from a high-level language source file at the compilation stage. HIF2VHDL is run at link time to complete the hardware thread translation from HIF to VHDL. The driver program *htc* encapsulates both modules in a GCC-like command line interface.

HTC's direct hardware generation capability allows the compiler to not only create the implementation for a particular program, but also modify the underlying architecture that the program will run on. HTC generates behavioral VHDL that causes the needed thread logic to be inferred by the synthesizer. A state machine is used to handle control flow for a thread and to handle communication with the HWTI. Since the HWTI provides primitive memory and communication operations and the generated user logic can implement any other computations needed, full support for programming languages such as C is possible. Although FPGA size is a constraint that heavily affects what can be migrated to hardware, many real world applications can benefit from running certain threads in hardware [reference some paper here and expand on real world apps somewhere in the paper].

HTC is designed with the goal of opening FPGAs up to software developers. With the ability to create custom parallel thread processors without being concerned with the low-level details of the hardware, software engineers may be able to use FPGAs for many new and exciting applications. Our goal is not to break FPGA performance records

or to create a general-purpose hardware development tool [cite Synopsis and XST], but to simply create a tool that allows a software developer to write programs that can automatically be mapped to FPGA logic with reasonable performance.

In the next section we sketch the main requirements of HTC. In section 3 we discuss the HIF language: the justification for using it as an intermediate form as well as its requirements, syntax, and semantics. In sections 4 and 5 we describe the HIFGEN and HIF2VHDL modules, respectively. Section 6 outlines HTC's VHDL model and the assumptions it uses. In section 7 we give the current status of HTC's support of the C language. We demonstrate HTC's capabilities and show our results in section 8. Section 9 discusses the short term and long term goals for the compiler, and we offer our final thoughts in section 10. More details on the Hybridthreads compiler, including information on how to obtain the source tree, can be found at [6].

2 Requirements

The two primary requirements for HTC are 1) to allow a software developer to create hardware threads using familiar programming languages and models, and 2) to integrate the generated hardware threads with the capabilities of Hthreads.

Several similar systems that generate HDL code from high-level programming languages (HLLs) actually modify the input language heavily to make a more flexible hardware development tool [reference these systems]. They do so at the cost of introducing many low-level operations that may not be familiar to software engineers. For instance, Handel-C uses C-like syntax but has operations to specify explicit parallelism and signal declarations [3]. These extensions require the user to understand hardware development, as the input language effectively becomes an HDL with different syntax. Furthermore, having hardware-specific constructs in source code forces the programmer to keep duplicate copies of the same thread: one for hardware and one for software.

Another common method for implementing HLL to HDL compilers is to use a proper subset of a particular language. For example, a number of systems that implement the C programming language have proper C syntax and semantics but do not handle common constructs such as pointers or generalized loops [1][C2H?]. The problem is that focusing on particular hardware optimizations (e.g., pipelining inner loops) forces other aspects of the programming model (e.g., addressable memory) to become difficult to implement. In this case, although the input language is unmodified, to move a thread from software to hardware a programmer must remove the unsupported constructs and rewrite the source code, potentially changing the meaning

of the program in the process. Full support for the primitive operations, memory model, and function call model of the input programming language is desired to keep the consistency between software and hardware threads. For the development of programs that run on the FPGA fabric to become commonplace, the compilers must support applications that can run in either software or hardware without any changes to the source code. In the short term, FPGA size may limit the feasibility of full applications that can live in either software or hardware (e.g., double precision floating point support). However, assuming Moore's Law continues to hold, we can foresee FPGAs becoming large enough to provide support for complex constructs at the low level (similar to the on-chip multipliers currently available). Consequently, the computational model used should not limit what features can be supported.

The Hthreads OS implements a Pthreads-compatible API that allows communication and synchronization between threads in the system, regardless of whether the threads are implemented in hardware or software. Since the hardware threads in the system all have a copy of the Hardware Thread Interface (HWTI) to provide system services, the HTC compiler must support the full set of capabilities provided by the HWTI.

Given that the Hthreads system allows for multiple custom thread cores to run in parallel and independently of the CPU, the potential performance benefit from true thread level parallelism is significant. Therefore, focusing on thread level parallelism before instruction level parallelism and enforcing independence from the CPU are important features of the compiler. The compiler is free to focus on general support for the programming model instead of heavily optimizing certain aspects of the generated hardware and giving up generalized capabilities such as the systems referenced above [maybe show an example of our system compared to another system and argue that our programming model outperforms theirs while still being more general-purpose!]

3 HIF Language

Our approach to designing a HLL to HDL compiler is to create an intermediate form that is both easy to generate from programming languages and easy to translate to HDLs. This is the essence of our Hardware Intermediate Form (HIF) language. [To generate HIF, a programmer simply runs a modified version of the familiar tool GCC (HIFGEN). For each C file, a .o and a .hif file is generated. The .o files can be used for linking the software part of the application. The set of HIF files that compose a hardware thread then linked together and translated to VHDL by the HIF2VHDL module.]

The HIF language itself is a very simple linear three ad-

dress code intermediate form that provides the necessary primitives to support GCC's intermediate form GIMPLE [4]. Starting from GIMPLE has the additional benefit of supporting all of the languages that GCC implements. The HIF language should not be thought of as an ends but as a means within our compilation system. It exists to provide a mechanism for separate compilation and HDL generation, but is not very interesting from a research point of view. The HIF language will evolve as more features are supported in the low-level hardware thread model. At the current time, it provides operations for arithmetic and logic, control flow, memory access, and function calls.

At this time, the experienced compiler writer may be wondering why the HIF language is needed. Since we are starting from the GCC toolset with its well-defined and complete GIMPLE intermediate form, it would seem as more than an adequate intermediate form for hardware generation. However, given that GIMPLE is a relatively recent addition to GCC and it is still evolving, we feel that defining our own intermediate representation will better isolate us from changes in the GCC front end. If GIMPLE were to significantly change, it would only affect the GIMPLE to HIF translation stage rather than the entire compiler. The flexibility inherent in defining our own intermediate form also allows us to enrich the HIF syntax to support hardware constructs directly, thereby simplifying the HDL generation process.

The requirements of the HIF language were specified explicitly by the Hthreads group in early 2006. The requirements list the purpose of HIF, the independence of HIF from the output hardware description languages and targeted FPGA architectures, and the basic rules of the HIF syntax. For the complete list of HIF requirements, please see [5]. The syntax of HIF is defined formally with a BNF grammar in Figure x (include `hif_grammar.txt` as Figure x). Since the semantics of HIF are straightforward we will give only an informal description of the HIF semantics in this document.

Each HIF thread compilation run gathers the required HIF files that consist of a set of HIF functions. A HIF function begins with an instance of the *function* keyword and ends with either the next instance of the *function* keyword or the end of the current file. Each function name, as the first argument to the *function* keyword, must be unique in the entire set of HIF files passed to the HIF2VHDL module. The function name must be an identifier, which is defined as a `"_"`, `"_"`, or a letter [A-Za-z] followed by one or more characters that are a `"_"`, `"_"`, a letter, or a digit [0-9]. The second argument to the *function* keyword must be a non-negative integer that tells HIF2VHDL how many arguments to expect when the function is called.

One HIF function is distinguished as the thread's main function and will automatically start execution when the

thread is started. The Hardware Thread Interface (HWTI) maintains a stack that allows threads to make function calls. Every time a function is executed, a new frame is generated on the call stack. A frame consists of the state that is used to handle the run time details of function calls, the arguments passed to the function, and additional local storage beyond the thread's register set for the function.

Within a HIF function variables and labels can be declared, and operations can be performed. Variables and labels must be identifiers and must have unique names among other variables, labels, and functions. Variables are used to define storage locations and can be either a scalar or an array. Labels define specific locations in a program that can be referred to in control flow operations. Operations are statements that change the state of the thread and include arithmetic and logic operations, control flow operations, function operations, and memory access operations. It is important to note that declarations are scanned at compile time and ignored at run time. The scope of a variable declaration is any function operation that follows the declaration within the same function. The scope of a label declaration is the entire function in which the label appears. The scope of a function declaration is all of the HIF files that are included in a particular thread compilation. Another important item worth noting is that the name *returnVal* and the names of all the system calls (see below) are special names; functions, variables, and labels cannot have these names.

A scalar variable is a single 32-bit value and is declared with the *declare* keyword. The first argument to *declare* is the new variable's name. Scalar variables are used as both 32-bit signed integers and 32-bit unsigned addresses. An array can be declared with the *declarearray* keyword. The first argument is the array's name and the second argument is the number of 32-bit signed integers to allocate. Scalars and arrays are local variables and can be implemented either in the thread's register set or on the thread's call stack. The compiler is responsible for determining a consistent way to implement a function's storage. In the HIF abstract model, each variable and array member is given a unique location on the call stack that is addressable by a 32-bit address. The array's name is an additional storage location that points to the front of the array in memory. In practice, the addresses for many storage locations are not needed and the values only appear temporarily in the register set of the thread. If the address of a variable is needed, then that variable's value will be placed on the call stack by the compiler.

The arithmetic and logic operations available in HIF are based on those provided by C. In the current version of the HIF language, all operations operate on signed 32-bit numbers except for the logical shift right. The arithmetic and logic operations are subdivided into two classes: binary and unary. Binary operations are in three address form consisting of a result and two operands. The binary operations

include addition, subtraction, multiplication, division, modulo, arithmetic shift right, logical shift right, logical shift left, bitwise and, bitwise or, and bitwise exclusive or. The unary operations take a result and a single operand, and include negation, move, and bitwise not. The operands can be previously declared scalar variables, integer literals, or the special *returnVal* variable, and the result must be a previously declared scalar variable. In the logical shift right, a '0' will always be shifted into the most significant bit.

The control flow within a HIF function is sequential; instructions are executed in the order they appear, unless a control flow operation is executed. Labels are the key to HIF's control flow operations: they define points at which operations can transfer control. HIF provides three control flow operations: the *goto*, *if*, and *switch* statements. The *goto* keyword is an unconditional jump that takes a label as an argument and transfers control to the first operation following the label. The *if* statement is a conditional jump that has a predicate and a label. The predicate consists of two operands and a relational operator. If the predicate holds true then control is transferred to the first operation following the label. Otherwise execution proceeds to the operation following the *if* operation. The *switch* statement is the most complicated of the control flow operations. A *switch* statement consists of a scalar variable that is the guard, a set of one or more *case* statements, and an optional *default* statement. Each *case* statement has a literal integer value and a label. If the control variable's value is equal to the literal for a case then control is transferred to the label associated with that case. If none of the cases holds, then control is transferred to the label identified by the *default* statement. If the *default* statement does not appear and none of the cases applies then control goes to the instruction following the *switch* statement.

Another type of control flow manipulation is performed by function operations. These operations switch the function currently being executed by transferring the position of control and manipulating the call stack and registers that are needed to maintain the thread's state. There are two types of functions that can be called: system calls and user-defined HIF functions. The valid system calls currently are *hthread_self*, *hthread_yield*, *hthread_mutex_lock*, *hthread_mutex_unlock*, and *hthread_exit*. The semantics for these calls are defined in the Hardware Thread Interface technical report [2]. More system calls will be added in the future as the system matures. System calls can be invoked using the *syscall* keyword in the current version of the HIF language. This will cause the thread to halt while the system call is handled by the Hardware Thread Interface or VHDL code inserted into the user logic, depending on the system call. In the near future, the *syscall* keyword will be dropped and system calls will be made with the *call* keyword.

A user-defined HIF function must be declared in one of

the files that are included during compilation and can be called from other HIF functions by using the *call* keyword. This keyword will transfer control from the calling routine to the invoked routine, creating a new frame on the call stack for the callee and updating any necessary thread state variables in the process. The *return* keyword is used to end execution of the current function and return control to the caller. If the current function is the main function of the thread, then *return* has the same effect as the *hthread_exit* system call and will end the current thread's execution. The *return* keyword takes the return value as an operand, which is always a signed 32-bit integer. After control is transferred back to the calling function, the return value will appear in the special *returnVal* variable, which can be read as an operand, but never written to.

The memory operations allow a thread to use the memory resources of the system. All memory addresses are 32-bit unsigned integers. The basic operations are *read* and *write*. The *read* keyword takes three arguments: a result scalar variable, a base address, and an offset. The base address is added to the offset to form the read address and then the Hardware Thread Interface is queried to fetch the value at the read address. When the operation completes the read value is placed in the result variable. The *write* keyword takes as arguments a base address, an offset, and a data operand. The write address is formed by adding the base address to the offset, and the Hardware Thread Interface is ordered to set the value located at the write address to the value in the data operand. Another memory operation provided is the *addressof* operation. This operation takes a result scalar variable and a target scalar variable. The address of the target scalar variable is computed by the run time environment and placed in the result variable. The *addressof* keyword is useful for creating the pointer aliases that are common in C. The last memory operation available in HIF is the *readarg* statement. This operation takes a result scalar variable and a non-negative integer index. The index is the argument number to read. The run time system will read the function argument at the given index and place the value in the result variable. Note that the main function of a thread must have exactly one argument that can also be read with the *readarg* operation.

The HIF specification also includes an *extern* statement that works similarly to C's *extern* statement. Although global variables cannot be declared in a thread, the *extern* statement will allow global variables that are declared in the software portion of the system to be accessed by the current thread. This feature has not yet been implemented but will be in the future; currently, to access global data within a thread, pointers to all global variables must be passed through a struct that is pointed to by the thread's argument. This constraint requires a minor rewrite to the HLL code and will be fully resolved once a proper linking

stage is implemented (see C Language Support section for more details).

[A number of example HIF files are attached to this report in Appendix X. The C files that were used to generate the HIF files are also given. The examples include a factorial thread, a quicksort thread, and others.]

4 HIFGEN

4.1 Introduction

As previously described, the Hybridthreads compiler (HTC) uses a two-step process to generate hardware cores from a variety of high-level languages. The front end of the compiler consists of a modified version of GCC that translates high-level constructs into a custom Hardware Intermediate Form (HIF). The back end links HIF files and applies the final transformations to produce synthesizable VHDL code. This part of the document describes the front end of the compiler; the tasks it performs, its limitations, and the general direction of its implementation efforts.

The front end of the compiler is called the HIFGEN module. It uses the popular and familiar GCC toolset to scan, parse, and optimize C programs. HIFGEN is a set of C functions that uses the GIMPLE intermediate form as a starting point to produce HIF by defining a useful mapping from GIMPLE to HIF. GIMPLE will be described in more detail in section 4.2, and its mapping to HIF will be explained in section 4.4.

The HIFGEN module serves two purposes: it isolates the back end of HTC from front end compilation and it applies the initial transformations and optimizations necessary to generate hardware from high-level user code. HTC currently uses GCC as a front end compiler, but a HIFGEN module could be written for any number of compilers, making HTC a very flexible tool. GIMPLE being a relatively recent addition to the GCC suite, it was crucial to write a flexible front end module that can easily be adapted to fit newer versions of GCC. Finally, the HIFGEN module uses information provided by GCC to help the back end in the generation of efficient hardware that matches the user's specifications and targets the chosen architecture.

4.2 GCC

In order to discuss the HIFGEN module, we must first examine the internals of the GCC compiler. We only mean to describe the components of GCC that are relevant to our discussion; for a complete description of the compiler, please see [4].

During the compilation process, a program will go through three main intermediate forms: GENERIC, GIMPLE, and RTL. GENERIC is a language-independent inter-

mediate form generated by each front end of the compiler and is used as the interface between the parser and the optimizer [4]. GENERIC, as its name implies, is meant to be a common abstraction that is able to represent programs written in all the languages supported by GCC. Strictly speaking, the GENERIC representation of a program could be used as the input to HTC, but this would unnecessarily complicate the compiler as very few optimizations have been performed at this point. Since HTC is using GCC's capabilities "for free," it would be wasteful not use its resources to the fullest.

GIMPLE is heavily influenced by the SIMPLE IL used by the McCAT compiler project at McGill University. It is a tree representation that is lowered from the GENERIC tree. In a sense it is simplified subset of GENERIC that is used as the entry point to the optimizer for target and language independent optimizations (e.g., inlining, constant propagation, tail call elimination, redundancy elimination, etc) [4]. In GIMPLE all expressions—aside from function calls—are in three-address form with temporaries used to hold intermediate values. There are no control flow structures and all statement and variable attributes are stored in annotations.

RTL stands for register transfer language, and this is where the bulk of the compiler's work is done. Statements and expressions are run through language-dependent and architecture-dependent optimizations and the result is used as the input to the code generator. The RTL representation of a program is too low-level and architecture-specific for HTC, as many of its constructs—such as stack and memory manipulation—are CPU-oriented and do not translate well to hardware.

Consequently, GIMPLE represents an appropriate compromise between a high-level data structure that lacks local optimizations (GENERIC) and a low-level representation that is architecture-specific (RTL). Furthermore GIMPLE, unlike GENERIC, is free of nested expressions and explicit lexical scopes that complicate instruction-level parallelism in hardware. Additionally, implicit jumps derived from control flow constructs such as if statements are made explicit in GIMPLE. This is an especially attractive feature as these jumps easily translate to state transitions in hardware state machines. Finally, important attributes such as variable aliasing (for hardware support of pointers) and size (for space optimizations) are available in annotations.

A detailed description of GIMPLE and its structure can be found at <http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html#GIMPLE>.

4.3 The HIF Language

Just as GCC uses an intermediate representation in its compilation process, HTC generates its own Hardware Intermediate Form (HIF).

One of our goals in designing an HLL to VHDL was to create an intermediate language that facilitates the transition from GIMPLE to hardware. In essence, our HIF is both an intermediate form for compilation and an object file for separate linking. We required the HIF to be independent of hardware description languages to allow for future extension to new languages or language constructs. We also needed the HIF to be independent of specific FPGA hardware architectures to simplify porting between architectures and to accommodate future technologies. The back end of the compiler is therefore responsible for applying HDL-specific transforms and component models. (For the complete list of HIF requirements, please see http://wiki.ittc.ku.edu/hybridthread/HIF_Requirements.)

The complete HIF syntax can be found at http://wiki.ittc.ku.edu/hybridthread/HIF_Syntax.

Operations in the HIF are specified in abstract form so that only the meaning of an operation is explicitly expressed. The back end of the compiler is responsible for expanding abstract operations into low-level implementations derived from the component model libraries. Variables are represented using an infinite register set and specified bit widths to allow the back end to create custom register sets that are specific to particular programs and FPGA architectures and to carry out semantics driven operations. Finally, control flow is implied by the ordering of operations or through explicit control flow statements within the HIF. State transitions and boundaries will then be inferred by the back end of the compiler and customized for the given application and architecture. Furthermore, implicit control flow allows the compiler to choose the most appropriate computation model.

4.4 Implementation

This section briefly discusses the implementation details of the front end of HTC.

C and C++ programs do not follow the conventional compilation process through the GCC compiler; they are directly converted from front end trees to GIMPLE and handed off to the back end rather than first being converted to GENERIC. For a detailed description of this process please consult the online documentation of the GCC internals at [4].

All languages supported by GCC have to be lowered to GIMPLE by their front end. Our modified version of GCC uses the `fdump-tree-gimple-raw` debugging option to signal the compiler to produce HIF. If this flag is chosen at the command line, the function `hif_dump_node` is called with a GIMPLE function tree and the target dump file as parameters. The function will then linearly walk down the tree and print HIF code to the dump file by mapping GIMPLE nodes to HIF statements. The function `hif_dump_node` is called

for every function in the original C code. The first time it is called, it creates the dump file and prints HIF. Any subsequent call to the function simply appends the new HIF to the dump file. It is important to note that generating HIF does not alter GCC's compilation process in any other way than printing the HIF to a file. In other words the compiler will still output executables or object files depending on the compilation options. This is a valuable feature of HTC as the linker will make use of those object files for address resolution and the Hthreads system will potentially run the executables as software threads.

As previously stated, the HIF is produced on a function-by-function basis. In other words, the tree that is passed to the `hif_dump_node` function is a tree that represents one whole function in the original C code. The node that serves as the entry point to the HIF translation process is the `FUNCTION_DECL` node. If the function has any arguments, they can be accessed by calling the macro `DECL_ARGUMENTS` on the `FUNCTION_DECL` node. Each function contains a top-level `STATEMENT_LIST` node that dominates all statement nodes in the GIMPLE tree. HIF statements are generated by walking down each path out of this node and translating GIMPLE along the way by mapping it to HIF.

The similarities between GIMPLE and HIF are not accidental. After choosing GCC as the starting point for HLL compilation, it was only natural to design our intermediate form to match GCC's intermediate representation (IR) fairly closely. Since both IRs are in three address form, most simple statements are easily mapped from GIMPLE to HIF, as shown in Table 1.

Control flow statements match up in a similar way; GIMPLE includes the `if`, `switch`, and `goto` statements, which are also part of HIF. Labels are slightly different and are generated in three ways, depending on the situation. Labels that are part of the original C source code will not be modified in any way and will appear in their appropriate location in the HIF. Labels that are generated by GCC for explicit GIMPLE control flow will be copied to the HIF. Finally, the translation process from GIMPLE to HIF sometimes requires statements to be reordered, and appropriate `gotos` and labels are inserted to maintain the semantics of the program.

Functions were briefly described in the previous subsection but their translation will be explained in more detail here. The HIF syntax requires a function declaration to include its name and its return type. In GIMPLE, the name of a function is obtained by calling the `DECL_NAME` macro on the `FUNCTION_DECL` node and its return type is fetched by calling the `TREE_TYPE` macro on the same node. When calling a function, however, the list of parameters passed to the function is in the second operand of the call node. In this case, the parameters are contained in a `TREE_LIST` node that can be expanded to extract them, and

GIMPLE	HIF
NEGATE_EXPR	neg
BIT_NOT_EXPR	not
TRUNC_MOD_EXPR	mod
BIT_AND_EXPR	and
BIT_XOR_EXPR	xor
LSHIFT_EXPR	shl
TRUNC_DIV_EXPR	div
BIT_IOR_EXPR	or
MINUS_EXPR	sub
MULT_EXPR	mul
PLUS_EXPR	add
RSHIFT_EXPR	shr
LROTATE_EXPR	rol
RROTATE_EXPR	ror
EQ_EXPR	==
LT_EXPR	<
GT_EXPR	>
NE_EXPR	!=
GE_EXPR	>=
LE_EXPR	<=
UNLT_EXPR	<
UNLE_EXPR	<=
UNGT_EXPR	>
UNGE_EXPR	>=
LTGT_EXPR	!=

Table 1. Mapping GIMPLE Operations to HIF

the variable to which the return value is assigned is in the first operand of the call node. In HIF, the return value of a function is always assigned to the special register returnVal, from which we subsequently read it.

Memory access is where GIMPLE and HIF begin to diverge. Since explicit memory transactions are at a lower level than GIMPLE and since memory structure and hierarchy are significantly different in hardware, a nontrivial amount of work is required to gather the information the HIF2VHDL back end requires. For example, since array initialization is not built into the HIF syntax, it requires explicit pointer arithmetic and multiple writes with the help of registers. This of course implies that the bit length of the array type be extracted to assign the values to the correct memory locations. In HIF, knowing the size of a variable is always required when reading from or writing to memory.

At the basic level, translating from GIMPLE to HIF is straightforward; for each construct in the GIMPLE grammar we find and print the corresponding HIF statement. Starting from the entry node in the GIMPLE function tree, each node is expanded until a leaf is reached. When a leaf is reached in the GIMPLE function tree, it is printed and the next branch of the last inner node is expanded until the next leaf is reached. When all nodes have been visited, the

function returns, which completes front end translation.

The command that generates HIF from a C source file is as follows:

```
htc -c source.c
```

For more information on how to generate HIF, see [6].

4.5 Limitations

The HTC is by no means an industrial strength compiler and suffers from a variety of limitations. Some of these limitations are due to the current state of its implementation. As the back end progresses and matures, most of these limitations will vanish. However, there is a set of constructs that simply do not map well to hardware and therefore will most likely never be supported by the compiler (e.g., polymorphism). There are also limitations that are a consequence of the tools and methodology used by the compiler that reside outside of its internals (e.g., simulation and profiling information).

There are various GIMPLE constructs that are not supported by the front end of HTC due to design decisions, HIF limitations, and hardware conflicts. For instance, types, function pointers and static variables have not yet been integrated into the current implementation.

The current version of HTC is completely dependent on the internal structure of GIMPLE. Consequently, statements in the GIMPLE tree are not always optimally ordered to produce HIF. This limitation results in a two-step process in the generation of HIF statements from GIMPLE: the first step is to output HIF-like statements that are GIMPLE-ordered, and the second step is to apply simple transformations to these statements to produce syntactically correct HIF. This solution is obviously less than optimal and the next version of the compiler will address this issue. (A list of future implementation efforts can be found in the future work section.)

There are several known bugs in the front end of the compiler:

- HLL constructs that are not supported will either produce illegal HIF and cause the back end to fail or cause GCC to seg fault.
- Pointer dereferencing is sometimes interpreted as a reference.
- The front end does not guarantee the correct number of parameters passed in a function call, causing the back end to fail.

5 HIF2VHDL

The purpose of the HIF2VHDL module is to read a set of HIF files that represent a single thread, link the files together, and transform the files into an efficient thread core implementation in VHDL. This module has a number of responsibilities including generating a VHDL core that can communicate properly with the hardware thread interface, take advantage of FPGA resources to implement a thread efficiently, and properly implement the the HIF language specification.

The user interface of HIF2VHDL is hidden by the HTC driver program. The module can be invoked by using the "-o" option of the HTC command. This option expects an output file name to be given and accepts C or HIF files as input files. At least one input file must be given. The option also expects the user to have supplied the name of the thread's main function with the "-m" option. Any C files that are provided when the "-o" option is invoked will be automatically compiled to HIF with HIFGEN. After this is complete, the HIF2VHDL module will be executed.

The HIF2VHDL module itself is a Python script called hiftrans.py that accepts a configuration file as input. The configuration file contains information such as the thread's main function, the memory base address for local memory, the output file name, and the list of input HIF files. The configuration file is automatically generated by the HTC command based on the information it collected from the user and then HTC invokes hiftrans.py with the configuration file.

Due to the necessity to transform HIF into VHDL, the HIF2VHDL module is an entire compilation system in itself. It has all of the standard components that one expects in a compiler: a lexer, a parser, analysis/optimization passes, and a code generator. However, due the fact that the original design of the system was done while the development team was still learning standard compiler techniques, the currently available version of the HIF2VHDL module should be considered a prototype version. Although the internal implementation is non-standard, the system is fully functional for the features that we describe in this document. Future versions of the compiler will change the internals to match standard compiler methods and add further features such as optimizations. Please see the Future Work section for more information.

As was mentioned above, the HIF2VHDL module is implemented in the Python programming language. Python was selected because it allows for extremely flexible and rapid application development and it has a large standard library that is particularly well-suited to the needs of HIF2VHDL. The first major task that the HIF2VHDL module does is read the configuration file. From the configuration file the compiler has a list of input HIF files, so the

next step is to read the HIF files. Lexical analysis on the HIF files is done next and is implemented with Python's string manipulation libraries. The parser module is executed next and due to the simplicity of the current definition of the HIF language, the parser is a simple hand-written DFA. The parser will be rewritten to use conventional parser generators when the HIF language becomes complex enough to require it. The next stage is an analysis/optimization module that does tasks such as statically trace the call graph of the thread and perform register allocation. The final stage is the code generator, which uses VHDL templates to implement each operation in the thread.

The following sections will provide a detailed description of the implementation of the HIF2VHDL modules listed in the previous paragraph. The hiftrans.py file sequences each of the major modules. Some of the modules also have sub-modules that will be discussed (such as the register allocation sub-module of the analysis module).

The configuration parser is implemented in configparser.py. The main entrance function to this module is the getConfigInfo function. This function accepts the argv list (which contains the configuration file name) and returns a dictionary structure called the configuration dictionary. The expected structure of the configuration dictionary is an attribute name and the associated value on a single line for the output file name, the main function, and the memory base address. The input files are specified by writing the keyword "inputfiles" on its own line, then giving an input file on on each subsequent line. The configparser module works by detecting the keywords and placing the appropriate values in the configuration dictionary. Error checking is then performed to confirm that the data read is of the proper type.

The lexical analyzer in HIF2VHDL is implemented in the file hif2tokens.py. The main entrance function to this module is getTokenLines. This function takes the configuration dictionary as input and returns the tokenlines data structure as output. The tokenlines datastructure is a dictionary with an entry for each input file. The value for each entry is a dictionary with an entry for each line in the file. Each line in the file is split up on the whitespace in the program. The first task this module does is read each input file into the lines dictionary. The next task is to eliminate comments by scanning each line that was read in and splitting on the HIF comment symbol, "#", and dropped everything after the comment. The tokenizeLines function is then executed and calls the tokenizeList function in the hifutil library. The tokenizeList function removes any whitespace lines from the list and then returns a dictionary that has entries for each line that remains. The values for each entry have been split on the white space to generate tokens. Once the tokenization process has been performed for each file, the hif2tokens module completes and returns the tokenlines

data structure to `hiftrans`.

The main file for the parser module of HIF2VHDL is `parser.py`. The parser calls several submodules including `syntaxcheck`, `splittokens`, `parseglobals`, `parsefunctions`, and `semanticscheck`. As mentioned above, the current design was created while the compiler team was still learning standard compiler techniques and the parser reflects that fact more than any other module in the compiler. As can be derived from the name, this module does slightly more than parsing (i.e. checking semantic rules after the parsing is complete). The basic structure of this parser is to check the syntax using a simple DFA in the `syntaxcheck` module, then split the components of a thread into smaller pieces until a control flow graph is formed for each function (`splittokens`, `parseglobals`, and `parsefunctions`), and finally to perform some basic semantic checks on the resulting thread data structures. The parser module's main function is called `runParser` that takes as arguments the `tokenlines` structure and the configuration dictionary and returns the CFG data structure.

Before going into further detail about how each submodule in the parser works, it is necessary to understand the CFG data structure that the parser module returns. This data structure is a large aggregate data structure that combines control flow graphs for each function with a symbol table for the thread. As the compiler performs more analysis and transformations, the CFG data structure is eliminated. Please make sure to make the distinction between the CFG data structure that stores information about the thread as a whole and the control flow graphs for each function, which are parts of the CFG data structure. This poor naming convention will be resolved in a future implementation of the compiler. This data structure is rather complex and clunky and it is important to note that this is a prototype version and future versions of the system will likely drop this structure all together and generate abstract syntax trees for the output of the parsing phase.

The syntax check module of the parser defines a dictionary called the `grammarDict` that holds a regular expression for various aspects of the HIF language. Since HIF is defined in a very simple way, the grammar is a regular language and it can be analyzed line by line. Therefore, each rule in the `grammarDict` simply defines a regular expression that will be applied to a line. The `grammarDict` is compared to the tokens in each HIF file by a set of functions that implement the check. This set of functions combined with the `grammarDict` is equivalent to a large DFA that returns true or false for the entire file. The syntax check module does not return anything. It will cause an error that will end the program if the parse fails.

The split tokens module uses the fact that each file has now been verified to be correct HIF and simply breaks each file up into a global part and a set of functions. This works

by finding the "function" keywords and splitting accordingly. The resulting pieces are placed into the CFG data structure. This function also initializes many of the fields in the CFG data structure that will be used by later modules.

The `parseglobals` and `parsefunctions` modules are used to walk the split tokenlines structures that were generated by `splittokens` placing each operation into control flow graph nodes and placing each declaration into the symbol table. These modules do some minor semantic checking such as not allowing certain HIF operations to be used in the global scope and making sure that all declarations in the same scope are unique. The modules also do some simple processing such as replacing declared constants with their values.

The semantics check module checks a number of semantic rules for constructs such as switch statements, labels, declarations, function calls, argument reads, and the thread's main function. The primary check that is done on these constructs is uniqueness of symbols (for declarations, case statements in switches, labels, etc). It also checks existence of symbols (e.g. a label in an if statement exists somewhere in the current function). All of these checks operate by examining the contents of the CFG data structure. This completes the parser module.

The analysis module of HIF2VHDL is implemented in `analysis.py`. It currently performs basic transformations that are required to prepare a thread for VHDL generation. In the future, this module will be expanded with a full optimization suite. The current tasks performed include replacing labels with state numbers, resolving offsets for address-of calculations, a static derivation of the call graph for the thread, and register allocation.

The call graph derivation is a simple fixed point computation that uses a work list consisting of functions to expand and a loop that expands a function. The goal of this analysis is to find what functions can be called from a thread. This is a simple optimization that allows the compiler to not include functions in the final thread that are never called. This works by placing the thread's main function in the work list. The front of the work list is then popped and expanded by adding all functions called in the current function that have not been marked as expanded into the worklist. The algorithm completes when the work list is empty and returns the call graph. The nodes of the call graph are the possible functions that can be called during the thread and an edge exists for each function call instance.

The register allocation algorithm is implemented in the file `registeralloc.py`. The register allocation scheme currently used in HTC is straightforward and takes advantage of the fact that a thread can allocate as many registers as it needs because of the underlying FPGA architecture. Spills do not happen with this register allocation scheme and this allows for extremely efficient register usage at run-

time. When combined with other optimizations, this can be a very powerful concept. The trade off is that there is currently no upper limit on the number of registers that will be created when using this technique as currently implemented. The core idea used in this register allocation scheme will be maintained in future versions of the compiler but with refinements to take into account FPGA space usage as well.

The code generation phase of HIF2VHDL works by reading in a set of VHDL templates, replacing the placeholders in those templates, and writing the results to the output file. The main VHDL module is implemented in `vhdlgen.py`. This module calls `vhdlcopy.py`, which copies the templates and replaces the placeholders, and `vhdlemit.py`, which writes the result to file. The copy and replace algorithm is straightforward, but does have to handle a lot of details such as translating to proper VHDL syntax for constants, computing addresses for the hardware thread properly, generating state names to be used in the VHDL state machine, and sequencing the next state assignments for each node with the proper generated state names.

Overall, the HIF2VHDL module is fairly primitive as far as compilation techniques, but it has great potential in the fact that it can take advantage of the Hthreads OS services and the FPGA architecture to create customized hardware thread cores on a per-application basis. This will require improving the hardware model that is generated by the compiler as well as optimizing the intermediate forms of the program as much as possible. However, the current version of the compiler does work well as a prototype. It successfully demonstrated the ability to compile HIF into VHDL and run the resulting VHDL on Xilinx Virtex 2 Pro FPGAs.

6 VHDL Thread Model

Body missing.

7 C Language Support

The full ANSI-C standard needs to be implemented by HTC to satisfy the requirement of opening the FPGA up to a software engineer. If the software engineer does not have to change how a program is written, then porting the application to the FPGA is just a matter of recompiling the software. As with most architectures, modifying the C program in certain ways will lead to a more efficient implementation, but writing standard C will at least always execute in the expected manner.

Currently, HTC supports a broad subset of the C standard. Many of the constructs supported by HTC have not been implemented for C to HDL compilation by past projects, which either rely on the CPU to handle what

cannot run in hardware [cite Garp] or restrict certain constructs in some way [cite something, probably Spark or Streams-C]. HTC provides an implementation implementation for pointers, arrays, function calls, structs, all C control flow constructs including variable bounded while loops, and support for communication and synchronization with other threads in the system.

There are many important parts of C that are not currently supported by HTC. The lack of support for a given construct is due to one of three reasons: the current model can support the construct and the construct simply has not been implemented yet, it is not practical to implement the construct in current FPGAs due to size constraints, and the construct may not ever be practical to implement in an FPGA. Fortunately, most of the constructs that are not currently supported are due to the first reason.

The most obvious item missing in the current implementation of HTC is proper support for the type system of C. At this time, the only type allowed is the 32-bit signed integer. The other integer types (e.g. char, short, long, etc.) can be supported by modifying the HIF language to have annotations for bit length. Floating point types are one of the constructs that fall into the category which is not practical for current FPGAs. A fully functional floating point unit takes up a large amount of the FPGA and leaves little room for the actual application. The assumption that FPGA density will track Moore's Law should allow full FPUs to fit on FPGAs with real world applications in the relatively near future. Other type related constructs such as the union, enum, and bit-field structs can be supported in HTC's current model but are not implemented yet.

Another important construct missing from HTC is the ability to use global variables directly in a hardware thread. This is due to the fact that a complete linking stage has not been implemented. The workaround is to create a struct that contains pointers to all global variables that need to be referenced in the hardware thread and then to pass that struct into the thread via the thread argument. Once the linker is implemented properly, then global variables will work as expected.

Related to global variables, static function variables are not implemented in the current version of HTC. A static annotation for declarations is needed in the HIF language to support static variables. In the hardware thread model, static variables can be allocated at the base of a thread's call stack and will work as expected.

Function pointers are not supported in HTC at the current time. The HIF language specification supports function pointers by using the `addressof` HIF instruction on a function name to get the function's address and then using a call HIF instruction on the address. Unfortunately, an implementation detail of the hardware model prevents function pointers from being allowed in the current model. The state num-

ber that starts a function is the hardware thread equivalent of a function's base address in software. The correct way to implement function pointers is to allow the state number to be treated in the same way that a function address acts in software. However, since the state numbers are not integrated into the global address space, great care has to be taken in the compiler implementation to make sure that function pointers from the software side of the system are not used in the hardware thread and vice versa. In a more advanced implementation, the state numbers can be integrated into the global address space. Since each function that is implemented in hardware is always going to exist in software, if a thread can know the software function addresses of the hardware functions that it implements, then function pointer calls could be "captured" by a translation table and the appropriate function in the hardware thread could then be called. This could even be extended in such a way that if a function pointer is called that does not exist in hardware, then the hardware thread interface could call that function via a call back to a software thread. Of course, the call back scheme needs to be used with care by the programmer because it makes the hardware thread depend on the CPU. The HTC project plans on implementing simple support for function pointers during the next iteration of development.

The C standard library introduces a new level complexity to supporting all of C in hardware. The requirement that the hardware threads not depend on the CPU means that ideally a local version of the standard library is needed. This is impractical with current FPGA architectures due to size constraints and will likely remain impractical for the foreseeable future. The approach of HTC is to support certain vital parts of the standard library. The latest version of the HWTI supports standard library functions such as `memcpy` and `malloc` [cite Erik's design document]. The next iteration of HTC will have full support for the new standard library functions in the HWTI. Other standard library functions may never be practical to place in a hardware thread. These include standard library functions that heavily depend on the operating system such as `signal` and `stdio`. The solution is to use the CPU call back method mentioned above (which the Hthreads project plans to implement in the future) to at least give the hardware thread the ability to use the entire standard library. However, if a thread is using the standard library extensively, then the thread is not a very good candidate for hardware in the first place.

The C99 standard includes many more features that were not mentioned above that are not currently implemented. The long term goal of this project is to implement everything in the C99 standard that is practical to run in hardware. This includes new features such as the complex number type and the `restrict` keyword. The standard has been studied in detail by the Hthreads group and priorities have

been set for the implementation plan. The HIF language and the HIF2VHDL model will evolve as each feature is implemented.

8 Results

Body missing.

9 Future Work

The current version of the HTC is in its infancy. Our goal is to develop it into a fully functional compiler that can support virtually all high-level programming constructs. Many of the compiler's limitations have already been mentioned in this paper, but a complete list of future design aspects will help the reader get a solid grasp on the direction of its implementation.

9.1 HIF

To be filled in.

9.2 HIFGEN

The front end of the compiler suffers from many drawbacks. Many of these drawbacks are due to the limited knowledge of the internals of GCC at the time of the initial conception of HIFGEN. After taking a few months to more thoroughly investigate the inner workings of GCC, the front end of HTC will be redesigned from the ground up.

Currently, the generation of HIF involves unnecessarily visiting every node in the function tree twice: once for enqueueing a certain node, and once for printing the corresponding HIF. This method is inefficient as it wastes time and space resources in an already large compiler. In the next revision of the compiler, a new method will be implemented using better data structures to generate HIF by visiting each node only once.

An additional benefit of using better data structures will be to completely eliminate the need to apply transformations to the HIF-like statements generated by the GCC module (as discussed in the limitations section). Using agile data structures that can work with the GIMPLE format and directly produce syntactically correct HIF will be a major improvement to the compiler. Furthermore, this method would prove more flexible if the HIF were to change significantly.

Another issue with the current version of the compiler is its lack of support for many GIMPLE expressions. For example, having function pointers in the source code currently causes GCC to seg fault, and multi-dimensional arrays have not been implemented yet. In the next version of HTC we

hope to remedy this situation by ensuring that every element of the GIMPLE grammar maps to a HIF statement. Accomplishing this goal will be a major milestone for the compiler.

An additional improvement to HTC, since it uses GCC as a starting point, would be to integrate Hthreads debug flags into GCC's compilation process. The present method of generating HIF relies on an already existing GCC debug flag (fdump-tree-gimple-raw) but a more elegant approach would be to create a new debug option and add it to GCC, thereby eliminating the side effect of outputting the raw GIMPLE when compiling. We can even envision having several different flags for target selection, optimizations, and file linking, just to name a few.

[C. Seamless marriage to HIF2VHDL module]
To be filled in.

9.3 HIF2VHDL

To be filled in.

10 Conclusion

We have made a system for translating threads written in the C programming language into VHDL. The resulting hardware thread cores can be executed within the context of the Hybridthreads FPGA operating system. This document has discussed the motivation, design, implementation, and future of the Hybridthreads Compiler (HTC). As this compilation system matures, it will hopefully help reveal the potential of the Hybridthreads system by allowing programmers to write software that can then be translated into extremely efficient hardware descriptions that can execute on an FPGA with proper runtime support by the OS modules and API provided by Hthreads.

References

- [1] Spark project.
- [2] E. Anderson. Hardware Thread Interface Specification and Implementation Document.
- [3] celoxica. HandelC.
- [4] Free Software Foundation, Inc. GNU Compiler Collection (GCC) Internals. <http://gcc.gnu.org/onlinedocs/gccint/>.
- [5] KU HybridThreads. Hif Requirements. http://wiki.ittc.ku.edu/hybridthread/HIF_Requirements. Last accessed February 20, 2007.
- [6] KU HybridThreads. Hybridthreads Compiler Wiki. http://wiki.ittc.ku.edu/hybridthread/HybridThreads_Compiler. Last accessed February 20, 2007.
- [7] KU HybridThreads. Project Wiki. http://wiki.ittc.ku.edu/hybridthread/Main_Page. Last accessed February 20, 2007.