PACT HDL: A C Compiler Targeting ASICs and FPGAs with Power and Performance Optimizations

Alex Jones, Debabrata Bagchi, Satrajit Pal, Xiaoyong Tang, Alok Choudhary, and Prith Banerjee

Technical Report No. CPDC-TR-2002-03-01
©2002 Center for Parallel and Distributed Computing
March 2002

Center for Parallel and Distributed Computing
Department of Electrical & Computer Engineering
Northwestern University
Technological Institute
2145 Sheridan Road, Evanston IL-60208

PACT HDL: A C Compiler Targeting ASICs and FPGAs with Power and Performance Optimizations

Alex Jones Debabrata Bagchi Satrajit Pal Xiaoyong Tang Alok Choudhary Prith Banerjee
Center for Parallel and Distributed Computing
Department of Electrical and Computer Engineering
Technological Institute, Northwestern University
2145 Sheridan Road, Evanston, IL 60208-3118
Phone: (847) 491-3641 Fax: (847) 491-4455
Email: {akjones, bagchi, satrajit, tang, choudhar, banerjee}@ece.northwestern.edu

ABSTRACT
Chip fabrication technology continues to plunge deeper into sub-micron levels requiring hardware designers to utilize ever-increasing amounts logic and shorten design time. Toward that end, high-level languages such as C/C++ are becoming popular for hardware description and synthesis in order to more quickly leverage complex algorithms. Similarly, as logic density increases due to technology, power dissipation becomes an increasing important metric of hardware design. PACT HDL, a C to HDL compiler, merges automated hardware synthesis of high-level algorithms with power and performance optimizations and targets arbitrary hardware architectures, particularly in a System on a Chip (SoC) setting that incorporates reprogrammable and application-specific hardware. PACT HDL is intended for applications well suited to custom hardware implementation such as image and signal processing codes. By making the compiler modular and flexible, optimizations may be executed in any order and at different levels in the compilation process. PACT HDL generates industry standard HDL codes, such as RTL Verilog and VHDL, which may be synthesized and profiled for power using commercial tools. Optimizations such as functional unit pipelining and reverse code levelization focus on reducing power consumed by the design without sacrificing performance.

Keywords
compiler, FPGAs, ASICs, SoC, hardware synthesis, low-power, high-performance, FSM, software pipelining, reverse levelization

1. Introduction
As chip fabrication processes continue deep into the sub-micron level and Integrated Circuits (ICs) and Field Programmable Gate Arrays (FPGAs) can support larger and larger amounts of logic, system designers require increasingly high-level tools to keep up. Recently, industry has targeted C/C++ and variants as potential long-term replacements for Hardware Description Languages (HDLs) such as VHDL and Verilog currently employed for today's hardware design. Also, as technologies increase in density in both the fabricated and reconfigurable areas, power-consumption becomes a progressively more important problem. While some work has been done in targeting C/C++ as an HDL and considering power-consumption in hardware synthesis, combining the two tasks creates a new challenge.

1.1 PACT
This paper presents PACT HDL, a compiler targeting the C language that produces synthesizable HDL usable for either FPGA designs or Application Specific ICs (ASICs) with a framework for both power and performance optimizations. PACT HDL supports arbitrary target architectures and allows for both power and performance optimizations at the C level or at an HDL level. The compiler uses a back-end to generate synthesizable Register Transfer Level (RTL) HDL codes, such as VHDL and Verilog, current industry standards. It can be easily extended to support new HDL codes as they are developed.

PACT HDL is part of the PACT (Power Aware Architecture and Compilation Techniques) project. The objective of PACT is to develop power-aware architectural techniques and associated compiler and CAD tool support that will take applications written in C and generate efficient code that runs on power-aware systems. The end goal is to generate power savings at all levels in the design process toward an overall high aggregate power savings. An overview of PACT is displayed in Figure 1.1.

More specifically, PACT HDL is a component of the PACT compiler (Task B). The PACT compiler targets a System on a Chip (SoC) style architecture consisting of reprogrammable components (FPGAs), application-specific components (ASICs), and a general-purpose processor (ARM). This paper focuses on the method for generating the descriptions for the hardware components of the SoC and describes some initial power optimizations. PACT HDL integrates with a top-level hardware/software partitioner and a power optimizing general-

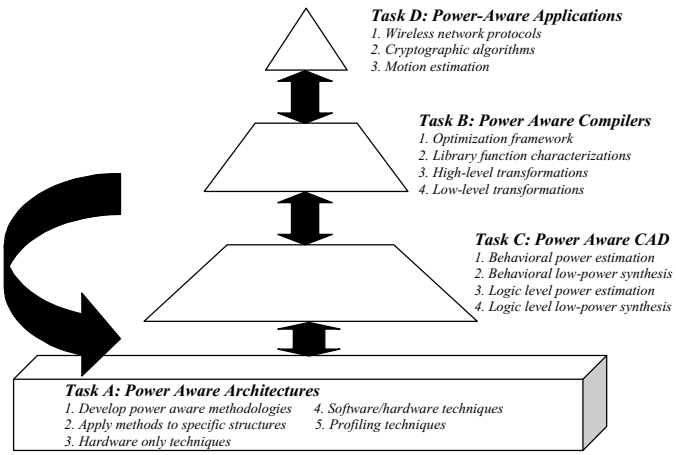

Figure 1.1. PACT Project Overview

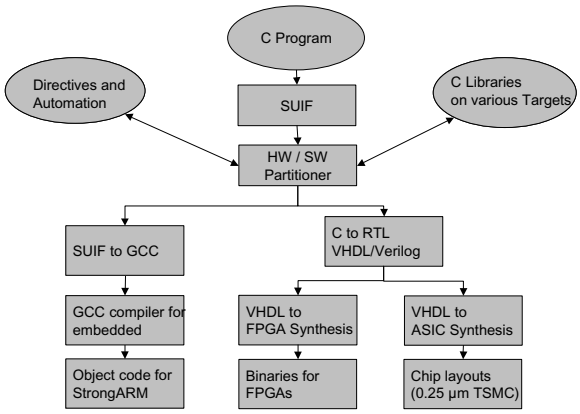

Figure 1.2. PACT Compiler Flow

purpose processor compiler to make up the larger PACT compiler. PACT HDL is designed with signal and image processing algorithms in mind. Often these sorts of problems would be handed by application-specific hardwares in the SoC setting. The flow of the PACT Compiler is illustrated in Figure 1.2.

1.2 Related Work

Recently, there has been a lot of work in the use of the C programming language and other high-level languages to generate synthesizable HDL codes or hardware implementations. David Galloway at the University of Toronto has developed the Transmogrifier C compiler. This compiler uses a subset of the C language and targets a Xilinx 4000 series FPGA[16]. Giovanni De Micheli et al. have developed HardwareC, a C like language that contains many HDL extensions. The Olympus Synthesis System synthesizes digital circuit designs written in HardwareC[12]. The Esterel-C Language (ECL) developed at Cadence Berkeley Laboratories is an HDL and compilation suite based on the C language[13]. CoWare, Inc. has developed the N2C layer for existing languages, C in particular, by adding clocking and cycle information to allow hardware modeling[4]. CynApps, Inc. has developed a suite of tools based on macro and library extensions for C/C++. These macros allow coding in C++ with an HDL style and the tools generate RTL HDL codes[10]. SystemC is another C- like language developed by Synopsys to allow C-like HDL coding that is particularly popular. The suite of tools associated with it can generate hardware directly[26]. Adelante Technologies (formerly Frontier Design) has a tool called A|RT Builder that takes a subset of C and generates VHDL[1]. The Voltaire Compiler from Get2Chip, Inc. generates RTL-Verilog from a C description[17].

Of particular interest in the synthesizable C community is the handling of pointers and other derived constructs. At Stanford, Séméria and De Micheli have done work to handle the problem by attempting to eliminate the loads and stores generated by pointers in C through special control flow[29]. C Level Design Inc. has developed the System Compiler™ that claims to handle the entire C/C++ language, including pointers, arrays, and higher-level constructs[32].

Some related work has been to target other high-level languages for hardware generation. Superlog® is a language based on both Verilog and C by Co-Design Automation, Inc[9]. The MATCH group at Northwestern University has built a synthesizable RTL VHDL compiler from the MATLAB programming language[6]. Xilinx, Inc. targeted Java as an HDL for its Forge-J HDL compiler[11].

As the proliferation of battery powered portable electronics has increased, so has the work on power-optimized hardware. Many of these approaches are applicable toward automated power optimizing compilers. At UC-Berkeley, power optimization work has been done in conjunction with the wireless research center dealing at the CMOS level[8] and using computational transformations[7]. Additional work on power-optimized synthesis has been done at Polytechnic University of Catalonia[24] and Princeton University[19][20].

The contribution of the PACT HDL compiler is to develop a C to VHDL/Verilog compiler with power optimizations for FPGAs and ASICs.

1.3 Outline

The remainder of the paper is outlined as follows: Section 2 describes the PACT HDL Compiler infrastructure. All stages of the compilation are discussed from the front-end C parsing to the back-end code generation. Implementation issues and the suitability for optimizations at different phases are discussed. Section 2.5 explains the synthesis flow for both ASICs and FPGAs using the compiler's output code. Specific optimizations for power and performance at different compiler levels are discussed in Section 3. Section 4 present some initial results as well as ongoing work. Section 5 discusses some conclusions and Section 6 relates some future work. References are listed in Section 7.

2. PACT HDL

PACT HDL is a fully modularized three-stage C to HDL Compiler. In the first stage, the C code is parsed into a high-level C type Abstract Syntax Tree (AST). At this stage, both power and performance optimizations may be executed. In general, these optimizations will not require specific information about an HDL representation or about the target architecture, such as precision analysis, constant-propagation, or loop unrolling. During the second stage, the C AST is converted into Finite State Machine (FSM) style HDL AST. Target architecture specific information is inserted into the AST in this phase. Again, power and performance optimizations can be executed. In general, these optimizations require specific information about clocks, cycles, or the target architecture, such as memory pipelining, and clock-

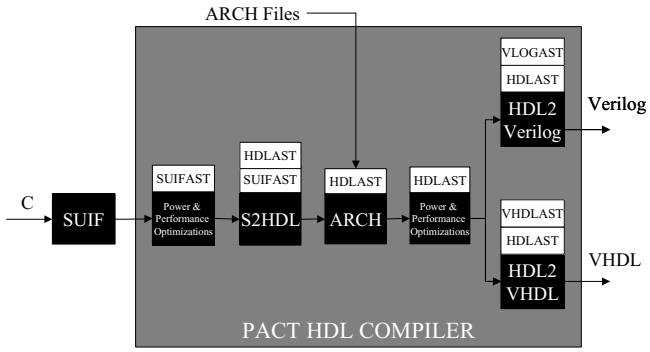

Figure 1.3. PACT HDL Compiler Flow

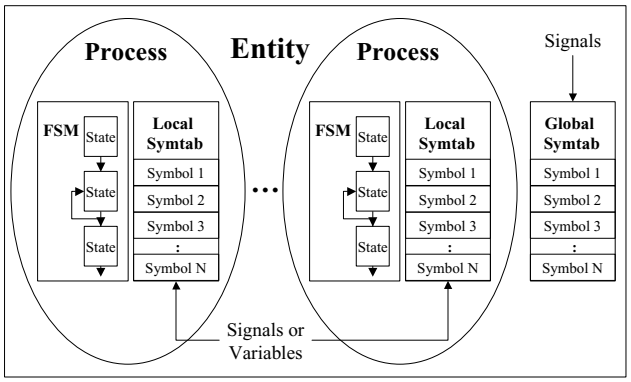

Figure 2.1. HDL AST Hierarchy

gating. Finally, RTL code is generated in the back-end phase. This phase is not designed for optimization; however, if language-specific issues arise they are handled in this phase.

The PACT HDL compiler leverages the SUIF C Compiler Version 1 from Stanford University[34] as its C front-end. SUIF was chosen for its flexible AST representation that retains high and low level information simultaneously easing the development of AST passes and optimizations that operate at this level. SUIF is not designed to handle very large, complex codes such as operating system kernels. This limitation is tolerable since the classes of problems PACT HDL targets are computationally complex rather than algorithmically complex. The PACT HDL compiler flow is shown in Figure 1.3.

2.1 HDL AST

Rather than directly translating the SUIF C like AST into a VHDL or Verilog-specific AST, an interim level of abstraction was created. This approach is very different than the related MATCH project, which directly translates the high-level language, in that case MATLAB, to a VHDL specific AST. All optimizations for hardware generation are done at the VHDL AST level[6]. In contrast, the HDL AST is designed to model the RTL HDL concept and be suitable for HDL optimizations without being language-specific. The synthesizable subset of both RTL VHDL and Verilog are similar enough to make this abstraction possible and allow optimizations that would require separate implementations to be designed only once.

As previously mentioned, PACT HDL is based on a FSM style intermediate AST. The top-level entity node corresponds to an entire C code possibly consisting of multiple C files and directories. The entity node contains a global symbol table and one or more process nodes. Each process node corresponds both to the C style function and a VHDL style process. The process also contains a local symbol table and a FSM node. The global symbol table may contain only VHDL-style signals so that they may be globally visible across processes. The local symbol table may hold either VHDL signals or variables for the different concurrency properties. VHDL style signals in a local symbol table are treated as visible only locally even though they are in fact visible globally. The FSM node describes the behavior of the C function through a list of one or more state nodes. The AST hierarchy is shown in Figure 2.1.

The core HDL AST node is the state node. A state contains one or more statements. Since the FSM requires explicit control flow,

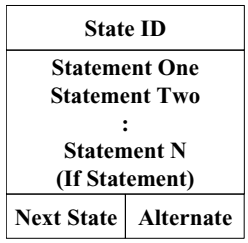

Figure 2.2. HDL AST State Node

a default and alternate exit are defined. Normally, the default exit is taken. The alternate is only required when an if statement is present. In that case, then and else correspond to the default and alternate exit, respectively. For this reason, each state is limited to one if statement, and the if statement can only be used for control flow. Each state node also has a unique identifier within each process. The state node is shown in Figure 2.2.

The rest of the AST nodes are built out of generally well-understood types of structures. For simplicity there are only five types of statements: assignments, ifs, sets, reads, and writes. Statements are built from unary, binary, and primary expressions and operators. Most symbols are some sort of variable symbol, however, some other types exist such as label symbols.

The HDL AST does not maintain direct facilities for retaining high level C structure information such as loops. If this information is required it is stored in the associated HDL AST nodes using annotations. The HDL AST is described in more detail in Jones[18].

2.2 SUIF to HDL Translation

During the conversion from the SUIF AST to the HDL AST there are two major passes. The first pass converts instructions into states and statements; the second does a control flow pass for the explicit control flow required by the FSM. The first pass is by far the most complex. During this phase, each type of possible SUIF instruction must be handled, temporary variables generated for communication of data between instructions, and HDL AST nodes generated and added to the AST. The second pass connects the states together using explicit control flow. It is during this phase that the state exits described in Section 2.1 are set. Details on the conversion are discussed in Jones[18]; however, some notable issues are described here.

PACT HDL supports pointers and arrays through memory address translation. SUIF converts pointer and array accesses into memory reads and writes. PACT retains these statements and uses a memory scheduler to initialize the memory addresses. For access to multi-dimensional arrays, Horner's Rule is used to limit the number of multiplication cycles to on the order of the number of dimensions in the array. Currently the array size must be known at compile time. While this is acceptable to the class of problems currently of interest, a method for handling dynamic arrays is being investigated.

In an effort to levelize instructions, SUIF uses temporary variables to pass data between instructions. PACT HDL retains this structure using temporaries to pass data between statements. The notion of a temporary variable is not very concrete in SUIF. Thus, PACT HDL generates concrete temporaries during code conversion. SUIF instructions are also maintained in hierarchical expression trees. This allows the compiler to maintain a scope for

each level in the SUIF expression trees. As the compiler moves into and out of each scope, temporaries can be reused and created based on need. An example of temporary creation and reuse is displayed in Figure 2.3.

PACT HDL supports function calls in a slightly different way from that of a traditional compiler. Rather than using a stack, parameters are passed using VHDL style signals because of their global properties. A global begin and end signal are created for each process. When a function is called, the caller process copies the appropriate values into the callee's parameter list. When completed, the begin signal for that process is set by the caller process and the caller waits for the end signal before proceeding. The callee process, upon receiving its begin signal, begins execution. Upon completion, the callee sets a return value and its end signal. If necessary, the caller stores the return value and then proceeds. This method mimics the behavior of function calls in C, using a process model that proceeds in parallel.

2.3 Target Architecture Independence

Hardware designs often need to interface with pre-designed components. These components may be optimized hardware cores or specialty hardware in the target. In the target class of problems, this principle is of particular importance with RAM components; however, it is applicable to other hardware components such as functional units. For PACT HDL to be an effective method for automating the process of generating hardware, this architecture specific information must be incorporated in the design creation.

In PACT HDL, interface descriptions are included in the compilation process through architecture files. The signals file specifies input and output signals for interfacing with the outside hardware components. A file is created for each method of accessing the hardware component. In the case of a RAM, a read and write file are created. Each file describes pseudo-states for accessing the device. Placeholders are used for items that require run-specific information such as memory addresses or data values.

Many core features of a hardware design are target-specific. Some examples include global clock and reset signals, and main memory. All HDL read and write instructions are converted into architecture specific memory reads and writes during an AST pass. When a statement is identified for expansion, the appropriate architecture file is parsed and the statement is expanded. This approach is not limited to memory components. Architecture files may be annotated with power and latency information for use by the compiler. More specific details on how

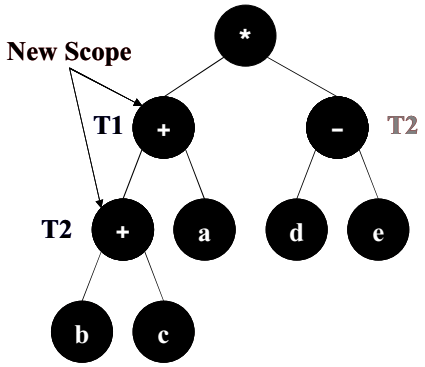

Figure 2.3. Temporary Creation Example

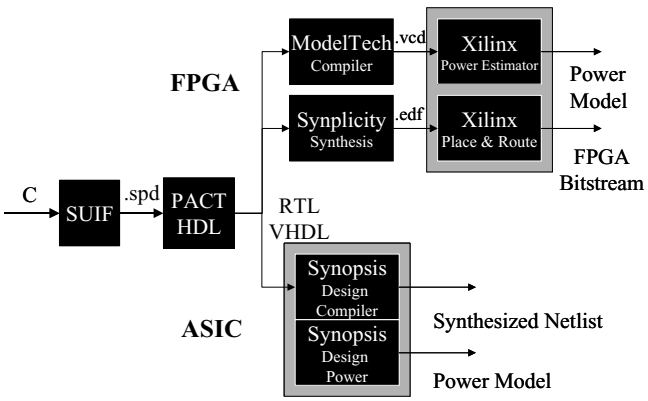

Figure 2.4. Synthesis Flow

architecture files are used in pact may be found in Jones[18].

2.4 Backend

Hardware codes in the HDL AST are of little use if that case is not synthesizable by commercial tools. For PACT HDL this means backends are necessary that generate synthesizable forms of industry standard hardware codes such as VHDL and Verilog. The HDL AST has been designed with this in mind, utilizing only a philosophy compatible with the synthesizable subset of both target languages. As a result, PACT HDL can generate any combination of VHDL, Verilog, and PACT HDL pseudo-code.

Backend code generation requires a language-specific AST for each target language. These ASTs are much simpler than the HDL AST because the only AST pass required is traversal for code generation. The language-specific AST is created during a traversal of the HDL AST generated by the compiler for a specific input code. This newly created AST is then immediately traversed to generate the appropriate code. Details on the VHDL backend may be found in Jones[18]. The Verilog code generation follows a similar method to the VHDL code generation and has been added to PACT HDL more recently.

2.5 Synthesis Flow

To prove correctness and investigate the effect of compiler optimizations, output of PACT HDL is tested with several targets: the WILDSTAR™ multi-FPGA using Xilinx Virtex components from Annapolis Micro Systems[1], standalone Xilinx Virtex FPGAs[35], a 1.5 micron process ASIC library shipped with the Synopsys Design Compiler[30], and a 0.25 micron process ASIC library from Leda Systems[21].

A combination of commercial tools is used to verify, synthesize, and analyze output of the PACT HDL compiler. All designs are simulated using Model Technology's ModelSim[23]. FPGA designs are synthesized using Synplicity by Synplify[31], placed, routed, and profiled for power consumption using Xilinx Foundation Tools. ASIC designs are synthesized and profiled for power Synopsys Design Compiler and Design Power.

The power profiling requires data from an input testbench set. This is done using ModelSim for the Xilinx tools, and is done implicitly within the Synopsys tools. The synthesis flow is outlined in Figure 2.4. More details on the synthesis flow may be found in Jones[18].

3. Optimizations for Power and Performance

Many projects have studied automated generation of hardware codes from high-level languages, see Section 1.2. For these projects, the criterion for success has been execution speed while neglecting other potentially important factors such as power consumption. Thus optimizations have focused on improving performance. Generating power-efficient hardware has also been studied, mainly in low-level circuits for applications in portable and remote devices. Unlike these other projects, PACT HDL attacks the power issue at the high-level using a series of optimizations geared toward improving power consumption while maintaining a good performance.

Power dissipated in a standard CMOS circuit is governed by the formula $P = C_{eff}V_{dd}^2 f_{clk}$ where P is the power, C_{eff} is the effective switching capacitance, V_{dd} is the supply voltage, and f_{clk} is the clock frequency[28]. Reducing the supply voltage has the greatest impact on power consumed due to its squared relationship. Unfortunately, reducing the voltage term also increases the delay of the resultant circuit thus decreasing the throughput[8]. A PACT HDL power optimization may then strive to reduce the critical path of the circuit so that the voltage may be decreased and the throughput vis-à-vis performance will be unaffected.

3.1 Functional Unit Pipelining

Consider the case where a functional unit is used extensively in a hardware design but takes several clock cycles to complete its operation. Many multi-cycle functional units used in hardware designs are pipelined, allowing multiple requests to the unit to be issued at regular before the first completes. The result from the functional unit is returned at the same interval, but after a certain initial latency. To utilize this capability of a functional unit, software pipelining techniques can be used to decrease the total number of cycles required to execute the code. The pipelined code may now run with a reduced voltage and maintain the original throughput thus reducing power consumed without sacrificing performance from the original non-pipelined design. This pipelining technique has been implemented in PACT HDL with respect to pipelined memory access.

Typical signal and image processing applications contain loops with a high incidence of memory accesses compared to relatively few computations. These loops also account for a significant proportion of the run time of the algorithm. As a result, pipelining memory accesses can significantly decrease the number of cycles required to execute such an algorithm. The loop is pipelined in two different ways. Computations are scheduled during memory accesses from later loop iterations. Memory accesses are re-ordered within a loop iteration to take advantage of pipelined memory units.

Loops meeting the following criteria are scheduled for pipeling:

- Arrays are mapped to an external pipelinable memory
- There is a single memory port
- There are no backward loop carried dependencies

In general pipelining problems, calculating the loop iteration interval (II) is NP complete[1],[21]. The assumption of a single memory port enforces a minimum II of M where M is the number of memory accesses in the loop. Non-memory operations can be pipelined considering traditional issues such as data dependencies.

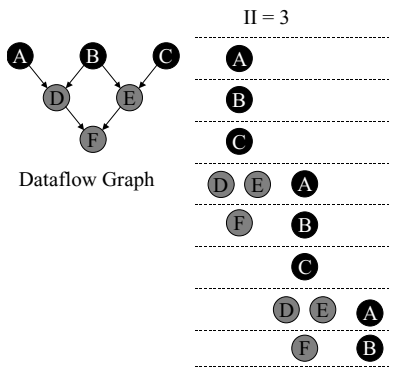

Figure 3.1. An example DFG and a possible schedule

The II of M refers to read and write instructions before expanded to reflect the target architecture as described in Section 2.3. After these instructions are expanded these memory accesses may be rather expensive, and incur stiff penalties in terms of clock cycles required when switching between read and write modes. With a pipelinable memory, these costs may be greatly reduced by minimizing transitions from read to write. In many cases consecutive reads or writes may have a throughput of one clock cycle while switching may require a delay of as many as five or ten cycles.

Before pipelining begins, an appropriate loop is selected from the input code AST that meets the appropriate pipelining criteria. This loop is converted from into a data-flow graph (DFG). PACT HDL then pipelines this loop in several stages, first scheduling the memory accesses, and then expanding the memory accesses with ARCH files, and finally scheduling non-memory computation. Figure 3.1 shows and example DFG and a possible pipeline schedule.

Figure 3.2 displays two different schedules of a loop with four memory operations. Schedule one results in four transitions between read and write. Schedule two is an optimal schedule and results in only two transitions, which is much preferable to schedule one. During the first pipelining stage, memory accesses from the DFG are separated into queues, one each for read and write. A queue is selected based on the memory operation that is topologically first in the DFG. Memory operations are scheduled

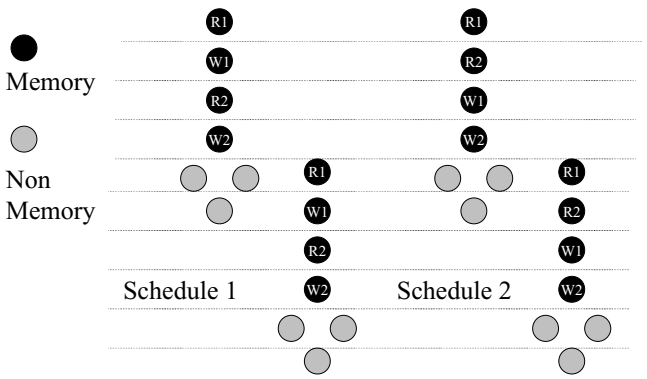

Figure 3.2. Two potential schedules of a loop

from this queue as long as no dependencies are violated. Then operations are selected from the other queue. This continues switching only when a queue is empty or a dependency is violated. The result is an optimal schedule such as schedule two.

Once both queues are empty, these operations are expanded with target architecture information in a second stage. During the third stage each non-memory operation (e.g. addition, multiplication, etc.) is scheduled. These operations correspond to the gray nodes in both Figure 3.1 and Figure 3.2. Each non-memory node is analyzed against the current schedule for memory operations. The as soon as possible (ASAP) and as late as possible (ALAP) times are determined where ASAP = 0 can occur before all scheduled operations and ALAP = ∞ can occur after all scheduled operations. These operations are ordered with least flexible (e.g. minimum of ALAP – ASAP) first. The top item is scheduled and the ASAP and ALAP labels are recomputed.

In some cases pipelining a loop can result in overlapped loop iterations reusing the same register. This can cause incorrect execution if the registers are not distinguished. The modulo variable expansion technique is used to differentiate these overlapping registers through a renaming and duplication scheme[1],[21]. Conditionally executed code inside a loop can be a barrier to pipelining. PACT HDL solves this problem using predicated execution. Both targets of the conditional branch are executed resulting in a new loop that is the union of both branches. The code resulting from the branch not taken is ignored. What were originally conditional dependencies have been converted to data dependencies and the loop may be pipelined[1],[21].

3.2 Reverse Code Levelization

The SUIF compiler is designed to target general-purpose processors, particularly load-store architectures like MIPS. The resultant SUIF AST focuses on the three-operand style instruction. For code to be stored in this format, it is necessarily levelized to fit into the two source operand one destination operand structure. While it is necessary for machine code generation, code levelization can have a significant power impact when targeting hardware designs.

Fully levelized code has the advantage of a regular structure within the FSM. Resources may be shared between states reducing the design's area and power consumed. The clock period may remain very short allowing for scaling of power factors described in Section 3. Unlevelized code may significantly increase the clock period for only few states with long control paths. Unlevelized code may also require the synthesis of additional functional units that could be shared with fully levelized code. The reduction in the number of states from fully levelized code can be significant, thus decreasing the complexity of FSM control logic.

The degree of code levelization becomes a tradeoff between the complex control logic with a shorter clock period on one hand and additional functional unit requirements and longer clock period. These factors impact not only the power consumed by the hardware code, but its performance and area requirements. The goal is find an acceptable compromise between fully levelized and unlevelized code.

Consider the case of the following C statement:

```
a=3*b+4*c-d;
e=a+2;
```

If the statement were levelized we might have the following HDL pseudo-code:

```
State1:
  Temp1 = 3 * b;
State2:
  Temp2 = 4 * c;
State3:
  Temp3 = Temp1 + Temp2;
State4:
  a = Temp3 - d;
State5:
  e = a + 2;
```

The control path for this code reads from two registers, sends the result through one functional unit and stores the value in a third register. The code requires five clock cycles to execute. Additionally, only one multiplier and one adder/subtractor need be instantiated. If the code remained completely unlevelized, it would require only two clock cycles to complete. The first statement including two multipliers, one adder, and one subtractor in the critical path would dominate the clock period. It would be necessary to instantiate two multipliers and two adder/subtractors. The PACT HDL Compiler explores the power relation between these two representations in a reverse levelization optimization.

Immediately following conversion from SUIF to HDL code, the code is fully levelized. Additionally, one notable side effect from the load-store architecture emphasis is unfortunately retained; constants are stored in temporary variables before being used in statements. The reverse levelization optimization attempts to unlevelize the code as much as possible, with the by-product of simultaneously accomplishing constant propagation.

The first stage of the optimization performs data flow analysis on the code. The compiler creates a control flow graph (CFG) from the HDL AST. From this CFG, reaching definition analysis is done using the iterative method[25]. Reaching definition analysis tells the compiler the definitions of a particular variable that may reach a use of that variable. This information is annotated to the AST using definition-use (DU)-chains[25]. DU-chains are a sparse representation of the results of reaching definition analysis. States with variable definitions store a linked list of nodes that use that variable definition. This information is stored as annotations to the appropriate HDL AST nodes.

Once the data-flow analysis is completed, this annotated information is used in the second stage of the optimization to actually do reverse code levelization. The DU-chain at a variable definition is consulted to find out where that variable is used. The variable is then replaced by the definition. Once all of the uses of the variable have been updated, the definition can be removed from the AST.

4. Results

The PACT HDL compiler has been tested with several benchmarks and has been shown to generate correct VHDL and Verilog code. A benchmark suite of appropriate image processing codes and kernels was selected for testing and profiling of optimizations. These codes correspond to functions a hardware-

	Vsum	FIR	Matmul	Laplace	Sobel
ASIC Frequency	50	50	50	50	50
ASIC Power	9.47	51.67	67.31	62.98	85.70
FPGA Frequency	94	38	22	24	25
FPGA Power	201	227	182	193	252

Table 4.1. Statistics for benchmark application synthesis without optimization targeting a Xilinx Virtex FPGA and a 0.25 µm ASIC process. Power is in mW and frequency is in MHz.

software partitioner might select for execution on a reconfigurable or application specific hardware resource in a SoC environment. The benchmarks include:

- Vectorsum
- Matrix Multiplication
- FIR Filter
- Laplace Transform
- Sobel Transform

Vectorsum is a simple benchmark that calculates the sum of all the elements of an array. It is mainly used to check for correctness and debug the design flow all the way from C to ASIC and FPGA implementation. Matrix multiplication is an $O(n^3)$ operation used in a variety of image and signal processing operations. The Finite Impulse Response (FIR) filter is a band pass filter used in image and signal processing. The Laplace transform benchmark uses the Laplace operator $\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$ to determine the gradient used for edge detection in images. The Sobel transform is another method to determine the gradient using two-dimensional linear convolution. The initial results are very promising.

All of the benchmarks were synthesized and profiled for power and area requirements using the commercial tools described in Section 2.5 targeting both the Xilinx XCV400 Virtex FPGA[35] and a 0.25 µm process ASIC using libraries from Leda Systems[21].

A summary of the synthesis results for the benchmark suite is displayed in Table 4.1. The results are from a direct translation without any optimizations. The synthesis results use the VHDL back-end.

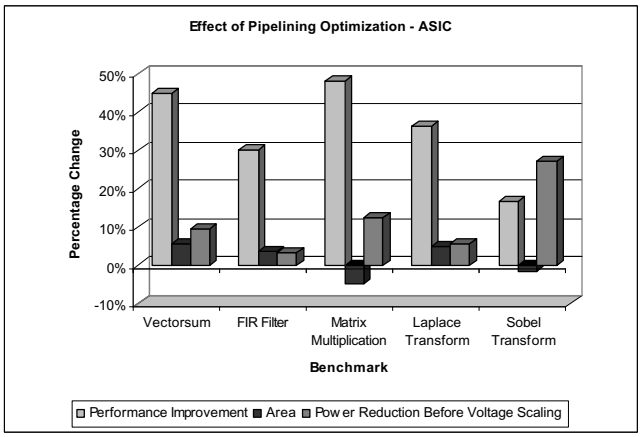

Figure 4.2. Effects of the pipelining optimization on the hardware codes generated for the benchmark suite for the 0.25 µm ASIC process

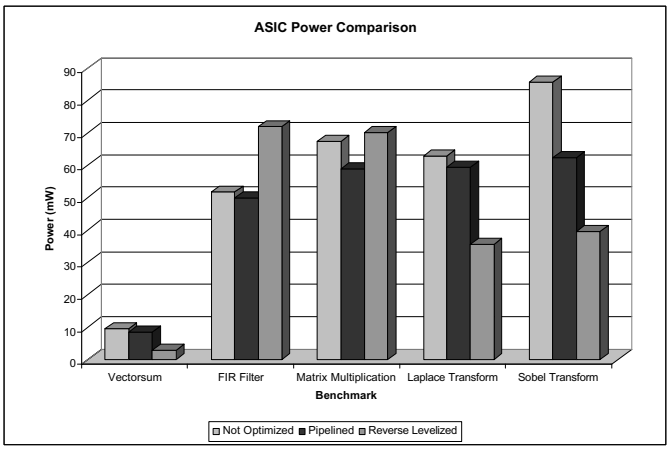

Figure 4.1. Power comparison of benchmarks with optimizations for the 0.25 µm ASIC process at 50 MHz and 2.75V

4.1 Pipelining Results

The memory pipelining in PACT HDL should not significantly impact the code size or the length of the critical path. The number of functional units required might increase due to resource conflicts that arise from multiple loop iterations occurring in parallel. The end goal is to reduce the number of cycles required to execute the loop and thus allow voltage scaling to reduce the overall power consumed by the design with a constant throughput.

Figure 4.2 shows some of the effects of the pipelining optimization on the hardware codes generated by PACT HDL for the ASIC process. Most notably, the designs performance has improved dramatically, nearly 50% in some cases. As predicted, the area has increased in some of the designs due to resource replication, although control logic was reduced which tended to offset this disadvantage.

It is important to note that while the power dissipated has decreased, voltage scaling described in Section 3 has not been done. The power savings in the designs is due to reduced number of states created when overlapping computation and memory access, reducing the control logic considerably. Because of the squared relationship between supply voltage and power it is

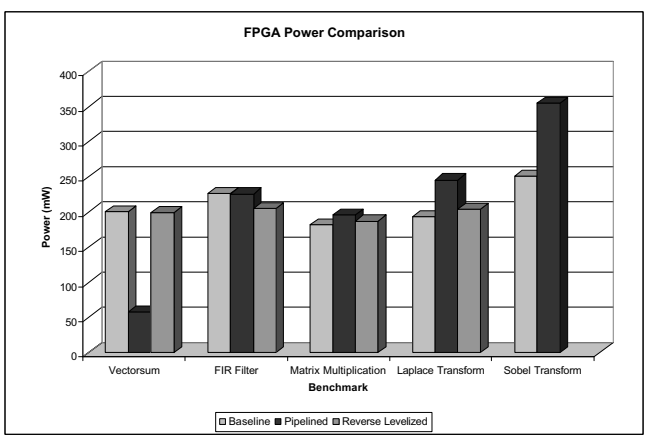

Figure 4.3: Power comparison of benchmarks with optimizations for the Xilinx Virtex XCV400 FPGA at 2.5V

expected that the power savings through pipelining and voltage scaling will be significant. The power results for the ASICs and FPGAs are shown graphically in Figure 4.1 and Figure 4.3 respectively.

The FPGA results shown in Figure 4.3 show slight power increases for some of the benchmarks. These power increases are most likely due to tradeoffs required due to FPGA area constraints and differences in algorithms used by the different synthesis tools for things like resource sharing and control extrapolation. For example, the Sobel Transform benchmark has generated the biggest power increase with the optimization and has the largest logic requirement to fit into the FPGA.

4.2 Reverse Levelization Results

Unlike pipelining, the optimization target is reduction in code size (e.g. number of FSM states) with a tradeoff of a potentially increasing critical path. For this optimization to reduce power, voltage scaling is not required. In cases where throughput is increased, voltage scaling may be applied to the design as a method of amplifying the power savings.

The power impact of the reverse levelization optimization is displayed in Figure 4.1 and Figure 4.3. This comparison is between the non-optimized code, the pipelined code and a completely reverse levelized code. As expected, some of the benchmarks show power improvement while others actually increase the power consumed. Here again a disparity between the FPGA and ASIC results is clear.

For the ASIC flow the FIR filter was able to put a relatively high number of operations in one state increasing the critical path quite a bit but with a minimal total state reduction. This required significant resource replication particularly of area and power hungry multipliers. The Sobel Transform was much more successful since it had nested loops and the effective reduction of states in runtime was significant. The resource requirements were affected but the ASIC was able to use more area. The state reduction power reduction far outweighed the extra power requirements for the additional area.

The results for FPGA have a different dynamic. Since the FIR filter was originally a relatively small design, the design was able to grow into existing FPGA resources without increasing the power required significantly. Since the design has no nested loops, the relative control savings were much less than the ASIC version of the Sobel Transform. Like the ASIC version, the FPGA version of the Sobel transform increased significantly in size for the reverse levelization optimization and was unable to be fit onto the FPGA due to resource constraints. As a result, the power results were not available for reverse levelization on this benchmark.

5. Conclusions

PACT HDL, a functional C to HDL compiler, has been presented. An abstracted HDL code layer appropriate for targeting synthesizable hardware codes and performing code analysis and optimization is discussed. The compiler has been shown to be target architecture independent and incorporate arbitrary backends for industry standard HDL codes. Compiler output is correct and synthesizable both VHDL and Verilog for either ASIC processes or FPGAs. PACT HDL is well suited for development of power and performance optimizations. The HDL pseudo-code level allows AST passes at this level to affect all potential compiler backends.

Two power emphasized optimizations have been presented, functional unit pipelining and reverse code levelization. These optimizations have been implemented and examined with an appropriate test benchmark suite of algorithms used in image and signal processing codes. The results for these optimizations show a potential for significant power improvements, particularly in larger designs, over an unoptimized code. Some of the results show that additional fine-tuning, such as resource aware reverse levelization, may be required to achieve power improvements for some designs.

6. Future Work

While PACT HDL is a functional tool, there is still interesting work to be completed. Two optimizations are presented here, however, there are a fair amount of potential optimizations to be explored for power impact. Some examples are: exploring bit-width analysis on storage and functional unit size and power requirements, exploring loop-independent code motion and voltage scaling, and common sub-expression elimination.

One main tool of hardware designers is the use of pre-designed hardware cores for optimized functional units, RAMs, etc. PACT HDL should have facilities for using these cores in the case of non-synthesizable operations (e.g. division or trigonometric functions) and potentially even synthesizable operations that may see a power improvement. The compiler may even be able to select the best functional unit for the performance and power constraints of the design.

Currently, PACT HDL assumes the processor with single memory paradigm. This assumption may be sufficient for a standalone FPGA or ASIC, it is not adequate within a SoC environment. Multiple memories and streaming data from other processors and devices need to be handled.

7. References

[1] Adelante Technologies, *A|RT Builder*, www.adelantetechnologies.com

[2] Allan, V. H. Jones, R. B. Lee, R. M. Allan, S. J. *Software Pipelining*. ACM Computing Surveys, Vol. 27, No. 3, Sep 1995.

[3] Annapolis Micro Systems. *WILDSTAR*. Xilinx Virtex Based Multi-FPGA Board. www.annapmicro.com.

[4] Arnout, G. *C for System Level Design*. Design, Automation, and Test in Europe Conference and Exhibitions 1999.

[5] Ashenden, P. *The Designer's Guide to VHDL*. Morgan Kaufmann. 1995.

[6] Banerjee, P. Shenoy, N. Choudhary, A. Hauck, S. *MATCH: A Matlab Compilation Environment for Configurable Computing Systems*. Submitted to IEEE Computer 1999.

[7] Chandrakasan, A. P. Potkonjak, M. Mehra, R. Rabaey, J. Brodersen R. W. *Optimizing Power Using Transformations*.

IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 1995.

[8] Chandrakasan, A. P. Sheng, S. Brodersen, R. W. *Low Power CMOS Digital Design*. IEEE Journal of Solid-State Circuits 1992.

[9] Co-Design Automation, Inc. *Superlog Website*. www.superlog.org.

[10] CynApps Suite. *Cynthesis Applications for Higher Level Design*. www.cynapps.com.

[11] Davis, D. Edwards S. Harris J. *Forge: High Performance Hardware from Java*. Xilinx Whitepaper, www.xilinx.com.

[12] De Micheli, G. Ku D. Mailhot, F. Truong T. *The Olympus Synthesis System for Digital Design.* IEEE Design & Test of Computers 1990.

[13] Esterel-C Language (ECL). Cadence website. www.cadence.com.

[14] Fraser, C. W. Hanson, D. R. *A Retargetable Compiler for ANSI C*. SIGPLAN Notices 1991.

[15] Free Software Foundation *GNU C Compiler*. www.gnu.org/software/gcc.

[16] Galloway, G. *The Transmogrifier C Hardware Description Language and Compiler for FPGAs.* IEEE Symposium on FPGAs for Custom Computing Machines (FCCM) 1995.

[17] Get2Chip, Inc. *Voltaire*, www.get2chip.com

[18] Jones, A., Bagchi, D., Pal, S., Banerjee, P., Choudhary, A. "PACT HDL" appears in *Power Aware Computing*. Graybill, R., Melhelm, R. Kluwer, Boston.

[19] Lakshminarayana, G. Jha, N. K. *Technical Report No. CE-J97-003: Synthesis of Power-Optimized Circuits from Hierarchal Behavioral Descriptions*. IEEE Design Automation Conference (DAC) 1998.

[20] Lakshminarayana, G. Raghunathan, A. Khouri, K. S. Jha, N. K. Dey, S. *Common-Case Computation: A High-Level Technique for Power and Performance Optimization.* Proceedings of the Design Automation Conference (DAC) 1999.

[21] Lam, M. *Software Pipelining: An Effective Scheduling Technique for VLIW Machines*. Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, Dune 1988.

[22] Leda Systems. *.25 um Process Standard Cell Libraries*. www.ledasystems.com.

[23] Model Technology. *ModelSim*. HDL Simulator. www.model.com.

[24] Musoll, E. Cortadella, J. *High-level Techniques for Reducing the Activity of Functional Units.* Proceedings of the International Symposium on Low Power Design 1995.

[25] Muchnick, S. S. *Compiler Design and Implementation*. Morgan Kaufmann. 1997.

[26] *Overview of the Open SystemC Initiative*. SystemC website. www.systemc.org.

[27] Pitas, I. *Digital Image Processing Algorithms and Applications*. John Wiley & Sons. 2000.

[28] Rabacy, I. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall Electronics and VLSI Design Series, 1996.

[29] Séméria, L. De Micheli, G. *SpC: Synthesis of Pointers in C: Application of Pointer Analysis to the Behavioral Synthesis from C.* IEEE/ACM International Conference on Computer-Aided Design (ICCAD) 1998.

[30] Synopsys. *Design Compiler*. Synthesis and Power Estimation Toolset. www.synposys.com

[31] Synplicity. *Synplify*. Synthesis Toolset. www.synplicity.com.

[32] *System Compiler: Compiling ANSI C/C++ to Synthesis-ready HDL*. Whitepaper. C Level Design Incorporated. www.cleveldesign.com.

[33] Thomas, D. E. Moorby, P. R. *The Verilog® Hardware Description Language: Fourth Edition*. Kluwer Academic. 1998.

[34] Wilson, R. P. French, R. S. Wilson, C. S. Amarasinghe, S. P. Anderson, J. M. Tjiang, S. W. K. Liao, S. W. Tseng, C. W. Hall, M. W. Lam, M. S. Hennessy, J. L. *SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers*. SIGPLAN Notices 1994.

[35] Xilinx. *Foundation Tools*. Place and route tools for Xilinx FPGAs. www.xilinx.com.

[36] Zeidman, B. *Verilog Designer's Library*. Prentice Hall. 1999.