Overview of the FREEDOM Compiler for Mapping DSP Software to FPGAs

David Zaretsky, Gaurav Mittal, Xiaoyong Tang and Prith Banerjee *Electrical and Computer Engineering Northwestern University* 2145 Sheridan Road, Evanston, IL-60208 {dcz, mittal, tang, banerjee}@ece.northwestern.edu

Abstract

Applications that require digital signal processing (DSP) functions are typically mapped onto general purpose DSP processors. With the introduction of advanced FPGA architectures with built-in DSP support, a new hardware alternative is available for DSP designers. By exploiting its inherent parallelism, it is expected that FPGAs can outperform DSP processors. However, the migration of assembly code to hardware is typically a very arduous process. This paper describes the process and considerations for automatically translating software assembly and binary codes targeted for general DSP processors into Register Transfer Level (RTL) VHDL or Verilog code to be mapped onto commercial FPGAs. The Texas Instruments C6000 DSP processor architecture has been used as the DSP processor platform, and the Xilinx Virtex II as the target FPGA. Various optimizations are discussed, including loop unrolling, induction variable analysis, memory and register optimizations, scheduling and resource binding. Experimental results on resource usage and performance are shown for ten software binary benchmarks in the signal processing and image processing domains. Results show performance gains of 3-20x in terms of reductions in execution cycles and 1.3-5x in terms of reductions in execution times for the FPGA designs over that of the DSP processors in terms of reductions of execution cycles.

1. Introduction

Recent advances in embedded communications and control systems for personal and vehicular environments efficient hardware are driving and software implementations of complete systems-on-chip (SOC). Two way radios, digital cellular phones, wireless Internet, 3G and 4G wireless receivers, MPEG 4 video, voice over IP, and video over IP are examples of applications which require digital signal processing (DSP) functions that are typically mapped onto general purpose DSP processors, such as the Texas Instruments TMS320C6000 [4], the Analog Devices SHARC, and the Motorola STARCORE processors. However, it is widely believed that such

processors will be unable to support the computational requirements of future DSP applications [24].

The conventional way to address the computational bottleneck has been to replace the DSP processor with an application specific integrated circuit (ASIC), which allows the designers to optimize for power consumption and functional parallelism. However, the design time and cost of such an implementation is very high. With the introduction of advanced Field Programmable Gate Array (FPGA) architectures that provide built-in DSP support, such as the Xilinx Virtex II [19] and the Altera Stratix [32], a new hardware alternative is now available for DSP designers. FPGAs combine the programming advantage of a general purpose DSP processor with the performance advantage of an ASIC. By exploiting its inherent parallelism, it is expected that FPGAs can provide more flexible and optimal solutions for DSP applications in a hardware/software co-design system.

Generally, the path from DSP algorithms to FPGA implementations is a complex and arduous task. Hardware design teams take the specifications created by the DSP engineers (in the form of a fixed point C, C++, or MATLAB,) and create a register transfer level (RTL) model in a hardware description language (HDL) such as VHDL and Verilog. The RTL HDL is synthesized by a logic synthesis tool, and placed and routed onto an FPGA using backend tools. Recently, there have been some behavioral synthesis tools that can automatically generate RTL descriptions from high-level descriptions in C [28,29,30] and MATLAB [31].

This paper describes the process and considerations for automatically translating software binaries that are targeted for general DSP processors into Register Transfer Level (RTL) VHDL or Verilog code to be mapped onto commercial FPGAs. The motivations for developing a translator from assembly code or binary to hardware are as follows:

- 1. As more DSP applications begin to require further computational power than what DSP processors can provide, there will be a need to migrate applications to hardware in the form of FPGAs.
- 2. There is a large established code base of DSP algorithms optimized for specific processor families.

Some of it is hand-coded for better performance or for reduced memory requirements.

- 3. Tools are available to implement C/C++, MATLAB and SIMULINK designs on DSP processors. Hence, the binary or assembly language can be used as an intermediate language from all high-level languages.
- 4. One may choose to migrate sections of code in a DSP processor application to hardware in order to obtain better performance results using a co-design system.

There has been previous work on binary translation from one processor's instruction set to another. There has also been work on decompilation, i.e. translating software binaries into high level programming languages such as C. Finally, there has been recent work in behavioral synthesis that takes a design written in a high-level language such as C and automatically generates hardware. Our paper is the first complete system that directly translates software binaries to hardware systems using FPGAs.

A simple approach to translating software binaries and assembly to RTL VHDL would be to map each assembly instruction onto one RTL operation per state in a finite state machine. Clearly, there will be no performance benefit when mapping such a design to hardware. The real benefit of migrating applications from a DSP processor onto an FPGA is in exploiting the on-chip parallelism. The question is whether or not it is possible to automatically infer the high-level control structure of a given application, perform all the necessary data flow and parallelism analysis at the assembly level, and manage to obtain a performance in a hardware implementation that is at least an order of magnitude faster than a software implementation. The true test in this approach is in experimentally evaluating the quality of the synthesized hardware results in terms of area and performance. The key contribution of this paper is in answering this fundamental question.

Our goal in this paper is not to compete with the best manual hardware implementation of a DSP algorithm on an FPGA [24], nor are we suggesting this approach as a better alternative to high-level synthesis. Rather, we wish to show that it is possible to migrate legacy assembly code for a state-of-the-art DSP processor seamlessly to hardware and still obtain an order of magnitude improvement in performance.

The rest of the paper is organized as follows. Section 2 reviews related work in the area. An overview of the compiler is presented in Section 3. Section 4 describes various optimizations used in the compiler. Section 5 described the experimental framework used to evaluate our compiler and experimental results on ten benchmark applications. Finally, conclusions and future work are described in Section 6

2. Related Work

The problem of translating a high-level or behavioral language description into a register transfer level representation is called high-level synthesis [1]. In contrast to traditional behavioral synthesis tools that take a behavioral description of an application in a language such as C/C++ or MATLAB and generate a RTL HDL implementation automatically, our compiler maps software binaries and assembly language codes into RTL VHDL for mapping onto FPGAs.

There has been some related work in the field of binary translation for converting assembly or binary codes written for one processor to another processor's ISA. Cifuentes et al [7,8] have done a lot of fundamental work in binary translation and decompilation. The Transmeta Crusoe processor performs dynamic code translation on the fly using a technique called CodeMorphing by translating code from an Intel x86 ISA and targeting the Crusoe processor, a VLIW machine [9]. Bala et al [16] have developed the Dynamo system for dynamic binary optimization for the HP architecture. Gschwind [17] has developed a similar system called BOA for the PowerPC architecture. Cooper et al [20] have reported methods to construct Control and Data Flow Graphs from scheduled assembly code. Baily and Davidson [21] introduced a formal model to specify procedure-calling conventions. Cifuentes and Simon [23] have described a Procedure Abstraction Language that can be used to specify the calling conventions for different architectures. Van Emmerik described the use of patterns to identify library functions in executables [23].

The above-mentioned work deals mainly in translating binary codes from one fixed ISA to another. However, our compiler differs in that it automatically translates binary codes from one ISA into hardware in the form of RTL VHDL and Verilog.

There has been related work in the area of hardwaresoftware co-designs. Stitt and Vahid [11] have reported work on hardware-software partitioning of binary codes. They took kernels from frequently executed loops at the binary level for a MIPS processor and investigated their hardware implementations on a Xilinx Virtex FPGA; this study was done manually. Stitt et al [12] have recently dynamic reported on partitioning work of hardware/software of software binaries for a MIPS processor. They have developed an approach to take kernel functions consisting of simple loops and automatically map them onto reconfigurable hardware. The hardware used is significantly simpler than FPGA architectures. The automatic commercial generation of RTL code is limited to only combinational logic. Hence the loops that must be implemented on the hardware are implemented in a single cycle. This



Figure 1. Overview of the FREEDOM compiler.

approach only works for sequential memory addresses and fixed size loops. The focus of their work is on fast dynamic hardware software partitioning, whereas the focus of our work is on the actual automated synthesis of software binaries onto hardware.

Levine and Schmidt [13] have proposed a hybrid architecture called HASTE, which consists of an embedded processor and a reconfigurable computational fabric (RCF) inside a chip. Instructions from the processor are dynamically compiled onto the RCF using a hardware compilation unit (HCU). Ye et al [14] have developed a compiler for the CHIMAERA architecture with a similar architecture of a general-purpose processor connected to a reconfigurable functional unit (RFU).

CriticalBlue, an Electronic Design Automation (EDA) start-up [15], has recently announced the launch of its Cascade Tool Suite. Cascade synthesizes a hardware co-processor specifically designed to accelerate software tasks selected by the user. However, there is no description of the technology or any published benchmarking results that would enable us to compare this compiler to their approach.

3. Overview of the Compiler

We now provide an overview of the FREEDOM compiler's infrastructure, which is illustrated in Figure 1. The compiler was designed to have a common entry point for all assembly languages. To this effort, the front-end requires a description of the processor ISA in order to configure the assembly language parser. It uses ISA specifications written in SLED from the New Jersey Machine-Code toolkit [5,6], coupled with a new semantic description language designed for this project. The parser generates a virtual assembly representation called the Machine Language Abstract Syntax Tree (MST).

The MST representation is similar to the MIPS processor ISA in syntax, and is generic enough to encapsulate most ISAs, including but not limited to ARM, MIPS, Intel IA-32 and IA-64, and Texas Instruments TI 320TMSC6000 processors. It supports both predicated and parallel instruction sets. All MST instructions are three-operand, predicated instructions, in which one or more of the operands may be null. An MST Instruction is defined as:

Label: *if* (**pred** *OR* (**pred**==**null**)) *then do* $dst \leftarrow op (src1, src2)$

Operand types include: memory addresses, registers, immediate values, and labels. Operator types include, but are not limited to: logical operators (AND, NAND, NEG, NOR, NOT, OR, SLL, SRA, SRL, XNOR, XOR), arithmetic operator (ADD, DIV, MULT, SUB), branch operator (BEQ, BGEQ, BGT, BLEQ, BLT, BNEQ, GOTO, JMP, CALL), comparison operators (CMPEQ, CMPNEQ, CMPLT, CMPLE, CMPGT, CMPGE), assignment operator (LD, ST, MOVE, UNION), and general operators (NOP).

The Control and Data Flow Graph (CDFG) is generated from the MST, and represent the data dependencies and the flow of control. Static-single variable assignment (SSA) is used to break the register name dependencies.

Several traditional optimizations are performed on the CDFG [2]. Scheduling and resource binding are also preformed on the CDFG, where computations in each basic block are mapped onto various resources (adders, multipliers, etc) in different states within a finite state machine. Figure 2 shows an example assembly code and the corresponding CDFG representation of a dot-product application.

DOTPROD:	MVK ZERO	.S1 .L1	500, A1 A7
	MVK	.SI	2000, A3
LOOD:	LDW	.DI	^A4++,AZ
	LDW	.Dl	*A3++,A5
	NOP		4
	MPY	.M1	A2,A5,A6
	SUB	.Sl	A1,1,A1
	ADD	.L1	A6,A7,A7
[A1]	В	.S2	LOOP
	NOP		5
	STW	.Dl	A7,*A3
(a) TI C	6000 A	ssem	bly code



(b) CDFG representation

Figure 2. Assembly code and CDFG for dot-product.

The optimized CDFG is translated into another intermediate abstract syntax tree, analogous to a highlevel Hardware Description Language (HDL). The HDL models processes, concurrency, and finite state machines. Additional optimizations and customizations are performed on the HDL to enhance the efficiency of the output and to correctly support the target device's architecture. Architecture-specific information is acquired via the Architecture Description Language (ADL) files. This includes data pertaining to resource availability, signal names, etc. Memory models are generated in the HDL as required by backend synthesis tools, such as Synplify Pro [3], to automatically infer both synchronous and asynchronous RAMs. Memory pipelining is used to improve the throughput performance.

The complete HDL is translated directly to RTL VHDL and Verilog to be mapped onto FPGAs, while automatically generating a testbench to verify the correctness of the output. The testbenches are used to guarantee bit-true behavior in the synthesized hardware, compared to that of the original TI assembly code versions.

4. Compiler Optimizations

It is obvious that without optimizations, the performance of an FPGA would be much worse than that of a DSP processor. The uniqueness of our compiler is in the methodologies and the various compiler optimizations that we have developed in order to exploit the inherent parallelism of the FPGA. A key step in applying some of these optimizations is the recognition of high-level language constructs such as loops, and arrays. We now discuss these optimizations in detail.

4.1. Analyzing Data Dependencies in Scheduled **Software Binaries**

The fixed number of physical registers on processors necessitates the use of advanced register reuse algorithms by compilers. This introduces false dependencies based on the register names, and results in difficulties when determining correct data dependencies, specifically when dealing with scheduled or pipelined binaries and parallel instruction sets. As a solution, each MST instruction is assigned a timestamp, specifying a linear instruction flow. Each cycle begins with an integer base stamp 'T'. Parallel instructions are assigned the timestamps ' $T_n = T + 0.01 x$ n' in succession. Assembly instructions that expand to more than one MST instruction are assigned timestamp values ' $T_n = T + 0.0001 x n$ '. Each MST instruction is also assigned an operation delay, equivalent to the number of execution cycles. The write-back time for the instruction, or the cycle in which the destination register is valid, is defined as wb = timestamp + delay.

Figure 3 illustrates how the timestamp and delay are used to determine data dependencies. In the first instruction, the MPY operation has one delay slot and therefore requires two cycles to complete. The new value of register A4 is not written until the end of cycle 1, and may only be used at the beginning of cycle 2. Consequently, the first three instructions are dependant on the same source register A4. Similarly, the ADD instruction at cycle 2.00 is dependant on registers A4 in cycle 0.00 and A2 in cycle 1.00, but not on register A2 of the LD instruction at cycle 1.01.

instructions.							
Figure 3.	Timestamp	os and del	ays for MST				
2.00	ADD (1)	\$A4, \$A2	, \$A3				
1.01	LD (5)	*(\$A4),	\$A2				
1.00	ADD (1)	\$A4, 4,	\$A2				
0.00	MPY (2)	\$A4, 2,	\$A4				

Scheduled or pipelined software binaries present many difficulties when attempting to analyze data dependencies. In the Vectorsum example in Figure 4, each branch instruction is executed in consecutive iterations of the loop. Furthermore, the dependencies of the ADD instruction in the loop body changes with each iteration of the loop. In order to correctly determine data dependencies in scheduled or pipelined assembly codes, one must linearize the assembly code, which can be accomplished in three steps. The first step is to build a correct control flow graph representation, using the algorithm developed by Cooper et al [20].

The second step is to linearize pipelined branch operations by moving a copy of the branch instruction to all possible write-back times in the CFG, and then setting its delay to zero. The source and predicate operands of the branch instruction are stored in temporary virtual registers. If a branch operation is not predicated, but its execution time falls outside its current block in the CFG, a predicate is added to the instruction. An example is shown in cycle 3 of Figure 5. If the normal control flow passes through the branch instruction's original position in cycle 3, the virtual predicate operand P0 is set. When the control flow reaches the branch instruction's intended execution stage at the end of cycle 8, P0 is reset in parallel with the branch execution, thus preventing the branch instruction from further execution outside the normal control flow.

The third step is to levelize pipelined operations. This is accomplished by breaking up a multi-cycle instruction whose write-back time occurs in another block into multiple single-cycle instructions. Virtual registers are introduced at each state of the levelization process. For an instruction with *n* delay slots, the original instruction is written to a temporary virtual register R_n and the delay on the instructions is changed to one cycle. In each successive cycle, we move virtual registers $R_{n-1} \leftarrow R_n$, $R_{n-2} \leftarrow R_{n-1}$, ... $R_0 \leftarrow R_1$, where R_0 is the original register name. This approach assumes that no two instructions are ever scheduled to write back to the same register in the same cycle. When the end of a block is reached, the assignments are propagated to the target and fall-through blocks. We may eliminate redundant virtual register assignments by keeping track of the cycles to which they have been written.

VSUM:	MVK	. S1	500. Al
	ZERO	.L1	A7
	ZERO	.L1	A4
	В	.S2	LOOP
	LDW	.D1	*А4++, Аб
	В	.S2	LOOP
	LDW	.Dl	*А4++, Аб
	В	.S2	LOOP
	LDW	.Dl	*А4++, Аб
	В	.S2	LOOP
	LDW	.D1	*А4++, Аб
	В	.S2	LOOP
LOOP:	ADD	.Ll	A6, A7, A7
[A1]	LDW	.Dl	*А4++, Аб
[A1]	SUB	.S1	Al, 1, Al
[A1]	В	.S2	LOOP
	MVK	.Ll	2000, A5
	STW	.Dl	A7, *A5
	-		

Figure 4. TI C6000 Assembly code for Vectorsum

3.00		MOVE	(1)	1, \$P0
4.00		LD	(1)	*(\$A4), \$A6_4
5.00 5.01		MOVE LD	(1) (1)	\$A6_4, \$A6_3 *(\$A4), \$A6_4
: 6.00 6.01		: MOVE	(1)	\$A6_3, \$A6_2
6.02		LD	(1)	*(\$A4), \$A6_4
7.00 7.01 7.02		MOVE MOVE MOVE	(1) (1) (1)	\$A6_2, \$A6_1 \$A6_3, \$A6_2 \$A6_4, \$A6_3
:		:	(1)	^(\$A4), \$A6_4
8.00 8.01 8.02 8.03 8.04	LOOP:	MOVE MOVE MOVE ADD	(1) (1) (1) (1) (1)	\$A6_1, \$A6 \$A6_2, \$A6_1 \$A6_3, \$A6_2 \$A6_4, \$A6_3 \$A6, \$A7, \$A7
8.05 8.06	[\$A1]	LD MOVE	(1) (1)	*(\$A4), \$A6_4 0, \$P0
8.07 :	[\$P0]	GOTO :	(0)	LOOP

Figure 5. Selected MST Instructions for Vectorsum

Figure 5 shows selected MST instructions for the Vectorsum of Figure 4. We determine that the LD instruction in cycle 4 with four delay slots has its writeback stage in the fall-through block (LOOP). The LD instruction is now written to virtual register A6_4 and the instruction delay is changed from five cycles to one cycle. In cycle 5, A6 4 is written to A6 3; in cycle 6, A6 3 is written to A6_2; in cycle 7, A6_2 is written to A6_1. The path continues to the fall-through block, where A6_1 is written to the original register A6 in cycle 8. Similarly, we determine the write-back stage of the LD instruction in cycle 5 occurs at the end of the second iteration of the LOOP block, and perform the same procedure as above. Although this LD instruction writes to register A6_4 in parallel with the assignment of A6_4 to A6_3, nevertheless, the one cycle delay on the former forces the latter to be correctly dependant on the previous value of A6_4 in cycle 4. The final two virtual register assignments for this instruction both occur in cycle 8 of the LOOP block.

4.2. Traditional Optimizations

Several traditional optimizations have been implemented at the CDFG level of the compiler [2] as shown in Figure 6. Many of the optimizations utilize reaching definitions to determine definition-use dependencies [2]. Prior to running the optimizations, input and output ports are identified using reaching definitions. An input port is defined as a node that is used, but has no prior definition; an *output* port is defined as a node that has a definition but no subsequent uses; an *inout* port is a variable that has been defined as both an *input* and *output* port. The CDFG is then converted into Static single variable assignment (SSA) form, which is essential in removing dependencies among registers and decreasing the lifetime of variables. The effects of SSA are apparent in register allocation, dead code elimination and resource binding.

The goal of the CDFG optimizations is to reduce the code size and resource usage, and increase the frequency of the design. The compiler runs the following optimizations repeatedly until the design converges.

Undefined variable elimination assigns a value of zero to a node that is used prior to having been defined, which would otherwise produce erroneous results in hardware.

Common sub-expression elimination uses hash-defined strings in determining redundant operations that frequently arise after loop unrolling is performed. Similarly, redundant memory access elimination uses hash-defined strings to represent memory addresses for determining redundant memory operations that often occur after unrolling a loop or due to memory spilling optimizations. A memory operation is said to be redundant if: two *consecutive* memory read operations access the same address, the second memory read operation is eliminated and the result of the first memory read operation is forwarded; two *consecutive* memory write operations access the same address, the first memory write operation is eliminated; a memory read operation immediately follows a memory write operation in which both access the same address, the memory read operation is eliminated and the value written to memory is forwarded.



Figure 6. Compiler Optimizations at the CDFG level.

Copy propagation is a transformation that given an assignment of variables $x \leftarrow y$, replaces later uses of x with y, as long as the intervening instructions have not changed the value of either x or y [2]. Constant propagation is similar to copy propagation, in that given an assignment $x \leftarrow c$ for a variable x and a constant c, the optimization replaces later uses of x with c, as long as the intervening instructions have not changed the value of x [2]. Constant folding solves for operations on constants at compile time.

Strength reduction and algebraic simplification are performed simultaneously to replace assembly operations that may produce costly hardware structures in RTL HDL with simplified operations. For instance, a multiplication or division by a constant value that is a power of two is replaced by a shift operation to save cycle delays and resource utilization.

Constant predicate elimination solves for predicated instruction in which the predicate condition is constant and may be solved at compile time. These conditions often arise after constant propagation is performed. Predicate reduction is an optimization that reduces multiple sequences of conditional set operations, generally used as predicates for other operations. The result of this optimization leads to a reduction in the number of multiplexers implemented in a hardware design, saving the cost of area and critical path.

Merging block sets is an optimization that merges the nodes in a set of consecutive blocks, where the first block is the only predecessor to the second block, and the second block is the only successor to the first block. The result produces more efficient scheduling techniques, allowing for more parallelism in a design.

Finally, dead code elimination removes operations in which there are no subsequent uses of the resulting value.

4.3. Procedure Extraction

Procedures are extracted from the linked assembly using an idiomatic approach. Three passes are used to identify function bodies within the binary. They use the procedure calling convention [21] to recognize caller prologues and callee epilogues. Details of the procedure extraction algorithms are described in a related paper [26].

4.4. Exploiting Fine-Grain Parallelism through Scheduling

As was mentioned in the introduction, the real benefit of migrating applications from a DSP processor onto an FPGA is in exploiting the on-chip parallelism. For example, the Xilinx Virtex II Pro [19] XC2Vp125 has 556 embedded multipliers that can potentially exploit 556-way parallelism in each clock cycle. Hence, one needs to explore the fine grain parallelism that is inherent in the CDFG through data scheduling.

The scheduling and binding pass performs behavioral synthesis on the CDFG representation by scheduling the computations of nodes in each basic block in the data flow graph onto various resources (adders, multipliers, etc). The delay and resource availability is used to schedule as many operations in parallel as possible. The type and quantity of each of these architectural resources are described using the Architecture Description Language (ADL) of the target FPGA. The high-level synthesis handle algorithms multi-cycle operators during scheduling, as well as multi-cycle memory read and write operations.

Two simple scheduling algorithms have been implemented, namely, As Soon As Possible (ASAP) and As Late As Possible (ALAP) scheduling [27]. Using ALU operation chaining, one is able to schedule many more simple operations per state, mainly those that do not affect the frequency of the design. Complex structures, such as multiplication, are generally not chained, as they tend to increase the critical path of the design [27].

4.5. Register Allocation

Register allocation is an optimization that is performed after scheduling to reduce the number of registers, which generally leads to smaller design size. Unlike DSP processor architectures, FPGAs are not limited to a small, fixed number of registers. Since they are capable of handling significantly more registers, one does not need to be concerned with issues such as memory spilling. However, one must realize that the scheduling of operations affect the number of possible register reuses.

Register allocation was implemented on the FREEDOM compiler using the Linear-Scan (left-edge) algorithm [33]. We assume the nodes in the CDFG are in SSA form, in which data dependencies are broken. We also assume an unbound number of register resources in the target FPGA, and our task is to assign the variable lifetimes to the smallest subset of registers. Prior to running the Linear-Scan algorithm, one must determine the *liveness* of each variable, or the time from the variable's first definition until its last use. This information is obtained from the nodes in the CDFG after scheduling [27].

4.6. Loop Unrolling

The unrolling optimization helps exploit on chip parallelism. It takes a loop body and generates an unrolled form by attaching successive copies of all the instructions inside. For example, the CDFG shown in the basic block of Figure 2(b) can be unrolled four times to get four times as many operators to schedule in a given time step. A key step in applying optimizations in the compiler is to recognize high-level language constructs such as loops, and array subscripts. Loop unrolling is performed through recognizing loop constructs using interval analysis on the flow graph. Loop variables are adjusted for unrolling by using induction analysis within the loop body.

4.7. Memory and Register Optimizations

General-purpose DSP processors have a limited number of memory ports (typically 2) for supporting memory loads and stores, and a small number of functional units (2-4) that can operate in parallel. The advantage of mapping applications to FPGAs is that one can exploit the parallelism in a design by implementing a large number of computations in parallel. However, in doing so, one also requires a greater storage area. One can store the data for an FPGA design in registers, or embedded RAMs.

If we map all variables to registers, we can access all the data in parallel, however, the results of these registers have to be distributed through large multiplexers to various functional units. This can increase the amount of area required by a design. However, if one maps all these variables onto embedded RAMs, one can reduce the cost of the multiplexers on the FPGAs, but at the cost of reduced memory bandwidth. We use a simple heuristic in that all scalar variables are mapped to registers, and all array variables that are greater than 128 bytes are mapped to embedded memories. By using the embedded block RAMs on the FPGA it is therefore possible to support larger data bandwidths through parallel memory access.

Towards this effort, the FREEDOM compiler generates RTL VHDL and Verilog codes that allow backend synthesis tools, such as Synplify Pro [9], to automatically infer both synchronous and asynchronous RAMs. Memory pipelining is used to improve the throughput performance. Through alias analysis, we are able to automatically partition data into different memories, thus increasing the number of memory access per state and the parallelism in the design.

4.8. Memory and Data Partitioning through Alias Analysis

When compiling high-level language programs onto a DSP processor, global variables are generally mapped onto the data memory and local variables are placed on the stack. The limited size of the register file often requires that variables be spilled to the memory, thus generating numerous memory accesses. When such codes are translated into RTL VHDL or Verilog, it severely limits the performance on the FPGA since the FPGA is capable of supporting an extensive number of registers far

beyond the scope of a DSP processor. Furthermore, when this code is unrolled, successive iterations of a loop must wait for the preceding memory writes to complete before performing their operations. Other limitations are apparent when loop indices are placed on the stack. These problems can be resolved through alias analysis and partitioning data across different memories.

A simple aliasing technique has been devised for the stack and memory. It requires that any memory access have addresses of the type *B[x*R+y], where *B* is the base address, *R* is a register, and *x* and *y* are numeric constants. Two address expressions refer to the same location if *x*, *y* and *R* are identical. The simplicity of the technique relies on the fact that memory addresses are usually modified by immediate values only. Within a loop, the array base would remain the same while the offset would change based on the loop index, represented by *R*. It is aided by induction analysis, by removing some of the register name dependence.

In a simple 2D-array, stored in the row-major form A[I,J], the address expression would be "x*I+J+y". Here, x would be the row length and y the base. The arrays are distinguished by the difference in their bases. In our experiments, data on array sizes and offsets has been used when convenient. User input of minimum array sizes has also been used effectively in some cases. The occurrence of address expressions within loops and the presence of loop iterators within the expressions are indicators to the presence of arrays.

5. Experimental Results

The FREEDOM compiler was tested using the Texas Instruments C6000 DSP processor architecture [4] and assembly language as the DSP processor platform, and Xilinx Virtex II [19] as the target FPGA platform.

The TI C6000 processor (model C64x) has 64 general-

to Xilinx Virtex II FPGAs.							
	TI C600	0 processor	Xilinx Virtex II FPGA				
Benchmark	Cycles	Freq (MHz)	Cycles	Area (LUT)	Freq (MHz)	Speedup (cycles)	Speedup (exec time)
dot_prod	12516	300	1555	1063	83.9	8.0	2.3
iir	22987	300	2105	2170	87.4	10.9	3.2
fir16tap	113285	300	16400	1301	84.3	6.9	1.9
fir_cmplx	72856	300	9637	3690	84.3	7.6	2.1
matmul	1799064	300	138602	1672	87.9	13.0	3.8
laplace	74673	300	5174	2655	120.9	14.4	5.8
sobel	127495	300	11744	4213	106.4	10.9	3.9
gcd	268	300	78	862	152.1	3.4	1.7
ellip	335	300	105	1402	122	3.2	1.3
diffeq	2318	300	119	1831	88.3	19.5	5.7

Table 1. Results of FREEDOM compiler translating TI C6000 DSP assembly programs to Xilinx Virtex II FPGAs.

purpose 32-bit registers, 2 multipliers, and 6 ALUs. It can execute up to 8 simultaneous instructions. It supports 8/16/32-bit data, and can additionally support 40/64 bit arithmetic operations. It has two sets of 32 general-purpose registers, each 32 bits wide. Two multipliers are available that can perform two 16x16 or four 8x8 multiplies each cycle. It has special support for non-aligned 32/64-bit memory access. The C64x has support for bit level algorithms and for rotate and bit count hardware.

We now report on the results on ten benchmark examples that were originally available in C. We used the TI Code Composer Studio to generate the TI software assembly version for those codes. The RTL HDL codes generated by the compiler were synthesized using the Synplify Pro 7.2 logic synthesis tool [3] from Synplicity and mapped onto Xilinx Virtex II XC2V250 devices [19]. These synthesis results were used to obtain estimated frequencies and area utilization for each benchmark. The areas of the synthesized designs were measured in terms of Look Up Tables (LUTs) for the Xilinx FPGAs. The RTL HDL codes were also simulated using the ModelSim 5.6 tool from Mentor Graphics. In each case the bitaccuracy of the results was confirmed. The execution times on the FPGAs were measured by counting the number of clock cycles needed to simulate the designs on the FPGAs using ModelSim. The execution time for the software implementation on the embedded processor was measured using the TI C6000 simulator.

Table 1 shows the results of the implementations of the 10 benchmarks on a DSP processor and on a Xilinx Virtex II XCV2V250 FPGA. The first column lists the benchmarks. The second column shows the execution time of the benchmark in clock cycles on a TI C6000 DSP processor using the TI Code Composer Studio instruction level simulator. The third column shows the maximum frequency of operation of the TI C6000 processor. The fourth, fifth, and sixth columns show the results of our FREEDOM compiler mapping DSP assembly programs onto FPGAs in terms of execution time in cycles on a Xilinx FPGA (measured by ModelSim), area of the FPGA implementation (measured in Look Up Tables by Synplify Pro 7.2), and frequency of the design (measured in MHz estimated by Synplify Pro). We can see that using our automated compilation techniques, it is possible to obtain performance gains of about 3-20x with an FPGA over the DSP processor with respect to clock cycles. In terms of actual execution times, including the impact of the clock frequencies, the performance gains are about 1.3-5x with an FPGA over an embedded processor. The impact of various optimizations on the area, frequency and performance are described in a related publication [25].

One may speculate that decompiling software binaries into a high-level language, such as C, and using a

behavioral synthesis tool to generate hardware would produce better results. In order to test this hypothesis we took all ten example benchmarks in C, and used the PACT compiler [10] to generate a hardware implementation on an FPGA. Both the FREEDOM and PACT compilers use very similar optimizations and are therefore comparable. Table 2 shows a comparison of the PACT compiler results with our FREEDOM compiler in terms of area, frequency and cycles. It is clear that the results of the two approaches are comparable. This validates our claim that one does not need to decompile software binaries or assembly code to a high-level language in order to obtain quality results. Rather, assembly and binary codes may be used as an intermediate language from any high-level language to generate efficient hardware implementations.

Table 2. Performance Comparison between PACT and FREEDOM compilers.

TREEDOM compliers.							
	PACT			FREEDOM			
	(C te	D FPGA)	(Assembly to FPGA)			
Benchmark	Cycles	Area	Freq	Cycles	Area	Freq	
dot_prod	3357	2447	69.2	1555	1063	83.9	
iir	3010	5873	98.4	2105	2170	87.4	
fir16tap	115209	547	69.7	16400	1301	84.3	
fir_cmplx	8499	6083	57.9	9637	3690	84.3	
matmul	277703	2155	70.2	138602	1672	87.9	
laplace	8467	10000	78.2	5174	2655	120.9	
sobel	81418	7873	57.6	11744	4213	106.4	
gcd	48	322	158.2	78	862	152.1	
ellip	43	1222	180.0	105	1402	122.0	
diffeq	79	1396	69.4	119	1831	88.3	

6. Conclusions

This paper described the process and considerations for designing a compiler that translates DSP algorithms written in the assembly language or binary code of a DSP processor into Register Transfer Level (RTL) VHDL or Verilog code for FPGAs. Experimental results were shown on ten assembly language benchmarks from signal processing and image processing domains. Results showed performance gains between 3-20x in terms of reductions of execution cycles and 1.3-5x in terms of reductions of execution times for the FPGA designs over that of the DSP processor.

The preliminary results are very encouraging. Future work includes a look at more complex benchmarks (e.g. MPEG4, JPEG2000, MP3 decoders, Viterbi, Turbo decoders, 3G and 4G wireless applications), other optimizations for area, delay and power reduction. Finally, we will investigate the issues of hardware/software co-design and function partitioning.

7. References

- [1] G. DeMicheli, Synthesis and Optimization of Digital Circuits, McGraw Hill, 1994.
- [2] Steven S. Muchnick. Advanced Compiler Design Implementation. Morgan Kaufmann, San Francisco, CA.
- [3] Synplicity. Synplify Pro Datasheet, www.synplicity.com.
 [4] Texas Instruments. TMS320C6000 Architectu
- [4] Texas Instruments, TMS320C6000 Architecture Description, www.ti.com
- [5] N. Ramsey, and M.F. Fernandez, "Specifying Representations of Machine Instructions", ACM Transactions on Programming Languages and Systems, May 1997.
- [6] N. Ramsey, and M.F. Fernandez, "New Jersey Machine-Code toolkit", Proceedings of the 1995 USENIX Technical Conference, January 1995.
- [7] C. Cifuentes and K.J. Gough, "A Methodology for Decomposition", XIX Conferencia Latinoamericana de Informatica, August 1993.
- [8] C. Cifuentas and V. Malhotra, "Binary Translation: Static, Dynamic, Retargetable?", Proc. Int. Conf. On Software Maintenance, Monterey, CA, Nov. 1996.
- [9] A. Klaiber, "The Technology Behind Crusoe Processors," Transmeta Corp., White Paper, Jan. 2000, www.transmeta.com
- [10] A. Jones et al, "PACT HDL: A C Compiler with Power and Performance Optimizations," Proc. CASES 2002, Grenoble, France, October 2002.
- [11] G. Stitt and F. Vahid, "Hardware/Software Partitioning of Software Binaries," Proc. Int. Conf. Computer Aided Design (ICCAD), Santa Clara, CA, Nov. 2002, pp. 164-170.
- [12] G. Stitt et al, "Dynamic Hardware/Software Partitioning: A First Approach," Proc. Design Automation Conf., Anaheim, CA, Jun. 2003, pp. 250-255.
- [13] B. Levine, H. Schmidt, "Efficient Application Representation for HASTE: Hybrid Architectures with a Single Executable", Proc. IEEE Symp. FCCM, Apr. 2003.
- [14] Z. Ye et al, "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit," Proc. 27th International Symposium on Computer Architecture, Vancouver, CANADA, June 10-14, 2000.
- [15] CriticalBlue, Cascade Tool Set, www.criticalblue.com
- [16] V. Bala et al, "Dynamo: A Transparent Dynamic Optimization System," Proc. ACM SIGPLAN Conf. On Programming Language Design and Implementation (PLDI), June 2000.
- [17] M. Gschwind et al, "Dynamic and Transparent Binary Translation," IEEE Computer Magazine, Vol. 33, No. 3, pp. 54-59, March 2000.
- [18] David Callahan et al, "Constructing the procedure call multigraph", *IEEE Trans. Software Engineering*, April 1990.
- [19] Xilinx VirtexII Datasheets, www.xilinx.com
- [20] K. Cooper et al, "Building a Control-Flow Graph from Scheduled Assembly Code," Dept. of Computer Science, Rice University.
- [21] M. Bailey and J. Davidson, "A formal model and specification language for procedure calling conventions".

ACM Symposium on Principles of Programming Languages, Jan. 1995.

- [22] Van Emmerik M. J., "Identifying Library Functions in Executable Files Using Patterns", *Proceedings of the 1998 Australian Software Engineering Conference*, Adelaide, 9th to 13th November, 1998, IEEE-CS Press, pp 90-97.
- [23] C. Cifuentes and D. Simon, "Procedure Abstraction Recovery from binary code", Dept. of Computer Science, University of Queensland, 1999.
- [24] Xilinx Corp, "Extreme DSP: DSP Implementation Techniques for FPGAs," http://support.xilinx.co.jp/ support/training/abstracts/v4/dsp_final.pdf
- [25] G. Mittal, D. Zaretsky, X. Tang, and P. Banerjee, "Automatic Translation of Software Binaries onto FPGAs," Proc. Design Automation Conference (DAC 2004), San Diego, Jun. 2004.
- [26] G. Mittal, D. Zaretsky, P. Banerjee, "Automatic Extraction of Function Bodies from Software Binaries," Submitted to Int. Conf. Computer Aided Design (ICCAD), Santa Clara, CA, Nov. 2004,
- [27] D. Zaretsky, G. Mittal, X. Tang, P. Banerjee, "Evaluation of Scheduling and Allocation Algorithms While Mapping Software Assembly onto FPGAs," Proc. Great Lakes Symp. on VLSI (GLSVLSI 2004), Apr 2004, Boston, MA, USA.
- [28] Adelante Technologies, "A|RT Builder," www.adelantetechnologies.com
- [29] Celoxica Corp, Handle C Design Language, www.celoxica.com
- [30] Forte Design Systems, Behavioral Design Suite, www.fortedesign.com
- [31] P. Banerjee, M. Haldar, A. Nayak, V. Kim, J. Uribe, "AccelFPGA: A DSP Design Tool for Making Area Delay Tradeoffs While Mapping MATLAB Programs onto FPGAs," *Proc. Int. Signal Processing Conference (ISPC)*, Mar. 31-Apr. 3, 2003, Dallas, TX.
- [32] Altera Stratix Datasheets, www.altera.com
- [33] M. Poletto and V. Sarkar, "Linear Scan Register Allocation," ACM Trans. on Programming Languages and Systems, Vol. 21, No. 5, pp. 895-913, Sept. 1999.