Automatic Translation of Software Binaries onto FPGAs

Gaurav Mittal David C. Zaretsky Xiaoyong Tang
P.Banerjee
[mittal, dcz, tang, banerjee]@ece.northwestern.edu
Northwestern University,
2145 Sheridan Road, Evanston, IL-60208
847 467 4610, U.S.A.

ABSTRACT

The introduction of advanced FPGA architectures, with built-in DSP support, has given DSP designers a new hardware alternative. By exploiting its inherent parallelism, it is expected that FPGAs can outperform DSP processors. This paper describes the process and considerations for automatically translating binaries 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 is chosen as the DSP processor platform, and the Xilinx Virtex II as a target FPGA. Various optimizations are discussed, including data dependency analysis, procedure extraction, induction variable analysis, memory optimizations, and scheduling. Experimental results on resource usage and performance are shown for ten software binary benchmarks. Results show performance gains of 3-20X in the FPGA designs over that of the DSP processors in terms of reductions of execution cycles.

Categories and Subject Descriptors

B.6.3 [**Logic Design**] Design Aids – automatic synthesis, hardware description languages, and optimization. B.5.0 [**Register Transfer Level Implementation**] General. D.3.4 [**Programming Languages**] Processors – *code generation, optimization and retargetable compilers*.

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

Reconfigurable computing, compiler, hardware-software co-design, binary translation, decompilation.

1. Introduction

Increasing demands for cell-phones, PDAs, and network devices have provided opportunities for the growth of embedded software, operating systems and development tools. According to a 2002 Venture Development Corporation study, there are nearly 230,000 embedded software developers, and 130,000 embedded hardware developers; the number of embedded devices has tripled from 130 million units in 1999 to 416 million units in 2003. Software developers are under increased pressure to develop a large number of more complex embedded software projects in less time. As newer processor architectures are announced, there is a need to reuse and migrate the software from older generation processors to newer processors to have better performance and cost. In many cases, one may not have access to the high-level source code when migrating applications; only the assembly or binary code may be available since the code was developed for performance or memory reasons at the assembly language level for the embedded processor. Often, users will hesitate to migrate an application to a newer platform owing to the time and effort required in migrating the code. It is therefore highly desirable to have a technology to automatically translate the binary or assembly code from an older processor onto a newer hardware platform.

In general for many computationally intensive real-time applications such as voice-over-IP, video-over-IP, 3G and 4G wireless communications, JPEG and MPEG encoding/decoding applications, there will be parts of the application that will be run in software on a general purpose processor and other portions that need to run on an application-specific hardware for performance reasons. For these reasons, there will be an increasing set of software applications that will need to be migrated to hardware.

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

1. As the computational requirement grows, there will be a need to migrate more software applications to hardware. The translator can be used as the primary tool or as an aid to design optimization for hardware implementations.
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.
3. Tools are available to implement C/C++, MATLAB and SIMULINK designs on DSP processors. The generated binary or assembly language can be used as an intermediate language.
4. The close-to-hardware nature of assembly might allow for better HW/SW partitioning decisions.

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. One may speculate on the possibility of decompiling binaries to a high-level language such as C

DAC 2004, June 7–11, 2004, San Diego, California, USA

and using behavioral synthesis to generate hardware from that. The true test of that approach is in experimentally evaluating the quality of the synthesized hardware in terms of area and performance. The key contribution of this paper is in answering that fundamental question.

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 if 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 get a performance in a hardware implementation that is an order of magnitude faster than a software implementation. Our goal in this paper is not to compete with the best manual implementation of a hardware implementation of a DSP algorithm on an FPGA, but instead to show that it is possible to seamlessly migrate legacy assembly code for a state-of-the-art DSP processor to hardware and still get 2-5X improvement in performance.

2. Related Work

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

There has been some related work in the field of binary translation in converting assembly or binary code written for one processor to another processor's ISA [5,6,7,816,17]. Rather than translating from one fixed ISA to another, our compiler automatically translates code from one ISA into hardware in the form of RTL HDL.

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 reported work on dynamic partitioning 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 commercial FPGA architectures. The automatic 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 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, as seen 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 Syntax Tree (MST), similar to the MIPS ISA in syntax and generic enough to encapsulate most ISAs, including those that support predicated and parallel instruction sets. All MST instructions are three-operand, predicated instructions. One or more of the operands may be null.

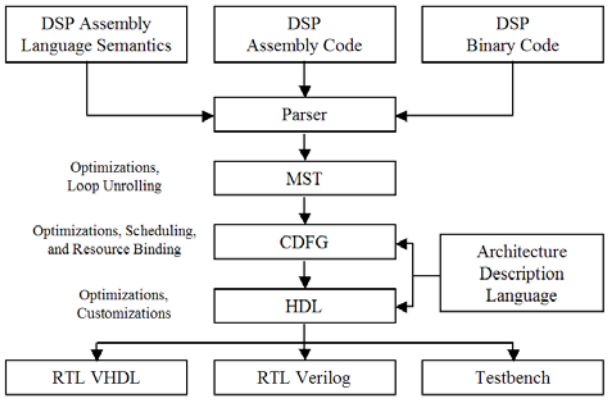

Figure 1. Overview of the compiler infrastructure.

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 *'$T_n = T + 0.01 \times n$'* in succession. Assembly instructions that expand to more than one MST instruction are assigned timestamp values *'$T_n = T + 0.0001 \times n$'*. 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 2 shows 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.

```
0.00   MPY (2) $A4, 2, $A4
1.00   ADD (1) $A4, 4, $A2
1.01   LD  (5) *($A4), $A2
2.00   ADD (1) $A4, $A2, $A3
```

Figure 2. Timestamps and delays for MST instructions.

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 were implemented on the CDFG [2]. These optimizations serve to reduce the design area and power consumption, while increasing the frequency of the design. The compiler runs these optimizations repeatedly until the design becomes stable. Most of the optimizations utilize DU-chains and UD-chains to determine definition-use dependencies [2]. The optimizations include constant propagation, constant folding, copy propagation, common sub-expression elimination, strength reduction, constant predicate elimination, dead-code elimination, and register allocation. 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. 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. The benefits of scheduling are reduced area and time through increased parallelism.

The optimized CDFG is translated into another intermediate abstract syntax tree, analogous to a high-level 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 optimization, 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 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 Binaries

Scheduled or pipelined software binaries present many difficulties when attempting to analyze data dependencies. In the Vectorsum example in Figure 3(a), 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. Two steps are required to correctly determine data dependencies in scheduled assembly codes. The first step is to build a correct control flow graph representation, using the algorithm developed by Cooper et al [20]. We determine there are two blocks in the control flow graph, namely the VSUM and LOOP code sections. The VSUM block contains an edge to the fall-through LOOP block, while the latter contains a single edge to itself. The second step is to introduce virtual temporary registers to break up multi-cycle instructions into multi-single-cycle instructions if the instruction's write-back time occurs in another block. 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 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 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   500, A1          |        LDW   *A4++,A6
       ZERO  A7               ||       SUB   A1,1,A1
       ZERO  A4               ||       B     LOOP
       LDW   *A4++,A6         |        LDW   *A4++,A6
||     SUB   A1,1,A1          ||       SUB   A1,1,A1
||     B     LOOP             ||       ADD   A6,A7,A7
       LDW   *A4++,A6         ||       B     LOOP
||     SUB   A1,1,A1          LOOP:    ADD   A6,A7,A7
||     B     LOOP             || [A1]  LDW   *A4++,A6
       LDW   *A4++,A6         || [A1]  SUB   A1,1,A1
||     SUB   A1,1,A1          || [A1]  B     LOOP
||     B     LOOP             |        STW   A7,*A4
```

(a) TI C6000 Assembly code

```
4.0003          LD    (1) *($temp), $A6 4
   :
5.0000          MOVE  (1) $A6 4, $A6 3
5.0004          LD    (1) *($temp), $A6_4
   :
6.0000          MOVE  (1) $A6 3, $A6 2
6.0001          MOVE  (1) $A6 4, $A6 3
6.0005          LD    (1) *($temp), $A6 4
   :
7.0000          MOVE  (1) $A6_2, $A6_1
7.0001          MOVE  (1) $A6 3, $A6 2
7.0002          MOVE  (1) $A6 4, $A6 3
7.0006          LD    (1) *($temp), $A6 4
   :
8.0000    LOOP: MOVE  (1) $A6 1, $A6
8.0001          MOVE  (1) $A6_2, $A6_1
8.0002          MOVE  (1) $A6_3, $A6_2
8.0003          MOVE  (1) $A6 4, $A6 3
8.0004          ADD   (1) $A6, $A7, $A7
8.0005    [$A1] LD    (1) *($temp_1), $A6_4
```

(b) Selected MST Instructions

Figure 3. Assembly code and MST for Vectorsum

Figure 3(b) shows selected MST instructions for Vectorsum. We determine that the LD instruction in cycle 4 with four delay slots has its write-back stage in the fall-through block (LOOP). The LD 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 Procedure Extraction

Procedures are extracted from the linked assembly using an idiomatic approach. Function bodies are identified within the binary in three passes. They use procedure calling conventions [21] to recognize caller prologues and callee epilogues.

In the first pass a list of the all the discernible functions is generated from the entire MST instruction list, using caller prologues. A function is discernable if the call instruction clearly denotes the first instruction (destination) within the target function body, e.g. jump to a label or an immediate. In some cases, if the destination is stored within a register or a memory location, its position within the instruction list is not clear. This is most often the case when function pointers are passed as arguments to other functions. Caller prologues are also used to identify the return addresses.

The second pass is used to identify function bodies and remove erroneous function prologues. It assumes that the function bodies are not scattered as fragments within the binary. Callee epilogues are used to recognize functions returns. The pass looks for all possible return instructions, callee epilogues, within a function body. If none is found and the body of another discerned function is impinged, then it is assumed that the impinged function was identified erroneously. Its body is merged with the function that was being processed. This leads to the pruning of the function list.

The third pass generates individual, CDFGs for each recognized function. It uses structural analysis to identify high-level structures. If the only common block between two branches in the CDFG is the exit block, then they are broken into separate function bodies. This, in essence, is disjoint set identification.

Finally a function call graph is generated. This is used to identify procedures that can be moved to hardware. The list of return addresses is used to generate switch-case structures to mimic function returns. Ongoing work on hardware-software partitioning will try to automate the selection process.

4.3 Exploiting Fine-Grain Parallelism with Scheduling

As 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 inherent in DFGs through data scheduling.

The scheduling and binding pass performs behavioral synthesis on the CDFG representation by scheduling the computations of data flow graph nodes, in each basic block, 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 algorithms handle multi-cycle operators during scheduling, as well as multi-cycle memory read and write operations.

Several simple scheduling algorithms have been developed, namely, unconstrained and constrained versions of As Soon As Possible (ASAP) and As Late As Possible (ALAP) scheduling [1]. Using ALU operation chaining, one is able to schedule many more simple operations per state, mainly those that do not effect the frequency of the design. Multiplication, memory writes and predicated instructions are among those that are not chained. Work is in progress for developing more efficient scheduling routines, such as software pipelining, modulo scheduling and hyperblock scheduling.

4.4 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 * REG + y], where B is the base address, REG is a register, and x and y are numeric constants. Two address expressions refer to the same location if x, y and REG 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 REG. 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 of the form "$x*I+J+y$". Here, x would be the row length and y the base. The arrays are distinguished from one another by the difference in their bases. Hard-coded array sizes and offsets are used when that information is available. User input of minimum array sizes has also been used. The occurrence of address expression within loops and the presence of loop iterators within the expressions are indicators to the presence of arrays.

5. Experimental Setup

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.

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

Table 1. Execution clock cycle results for Xilinx VirtexII XC2V250 as compared to the TI C6000 DSP processor. Percentages are normalized to that of the DSP's clock cycles.

Benchmark	DSP		B		B+O		B+O+S		B+O+S+U		B+O+S+U+A	
dot_prod	12516	100%	20008	160%	11005	88%	2505	20%	2505	20%	1505	12%
iir	22987	100%	37438	163%	23958	104%	4496	20%	4505	20%	1605	7%
fir16tap	113285	100%	272852	241%	151604	134%	33270	29%	23630	21%	15918	14%
fir_cmplx	72856	100%	92776	127%	60997	84%	11429	16%	9637	13%	8613	12%
matmul	1799064	100%	2956509	164%	1691243	94%	372074	21%	195946	11%	134506	7%
laplace	74673	100%	119179	160%	64578	86%	21014	28%	6794	9%	5174	7%
sobel	127495	100%	162122	127%	91792	72%	22432	18%	21622	17%	11744	9%
gcd	268	100%	239	89%	152	57%	96	36%	87	32%	78	29%
ellip	335	100%	462	138%	238	71%	108	32%	108	32%	105	31%
diffeq	2318	100%	1428	62%	840	36%	315	14%	213	9%	98	4%

The results are reported for the compiler on a set of 10 benchmarks from the signal and image processing domains, shown in Tables 1-3. The benchmarks were originally available in C and compiled into the TI C6000 assembly code using the Code Composer Studio from Texas Instruments. The execution time for the assembly codes were measured using the TI C6000 simulator.

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 bit-accuracy 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.

6. Experimental Results

This section reports the results of the compiler on the set of benchmarks that were compiled to the Xilinx Virtex II XCV2V250 FPGA. Table 1 shows the number of clock cycles and normalized percentages with respect to the DSP execution times. The benchmarks were run with different combinations of optimizations to show their individual effects.

Table 2. Frequency results in MHz for Xilinx Virtex II XC2V250.

Benchmark	B	B+O	B+O+S	B+O+S+U	B+O+S+U+A
dot_prod	95.0	95.0	87.9	87.9	66.3
iir	95.0	95.0	87.9	87.9	55.9
fir16tap	71.5	95.0	149.7	121.8	62.1
fir_cmplx	83.0	94.2	87.9	82.3	62.6
matmul	70.6	95.0	72.0	68.1	62.1
laplace	64.6	116.0	101.4	97.6	90.6
sobel	82.6	102.0	114.5	42.2	76.1
gcd	120.7	133.7	142.1	141.2	118.5
ellip	54.2	61.9	102.4	102.4	102.4
diffeq	81.6	94.2	60.9	59.6	58.8

The first two columns of Table 1 show the execution times and the normalized percentage for each benchmark simulated on the TI Code Composer Studio for the C6000 DSP processor. The next two columns show the results of the compiler's base case (B) without scheduling or any optimizations. Clearly, there is very little or no performance gains without any optimizations. The use of traditional optimizations facilitates a reduction in the code size. The results are apparent when comparing the base case (B) and the optimizations alone (B+O). Consequently, when combining traditional optimizations with scheduling (B+O+S) and loop unrolling (B+O+S+U), results show 60-80% improvement in execution times over that of the DSP processor. Finally, we show that through alias analysis and array separation (B+O+S+U+A) we are we are able to break dependencies and schedule more memory optimizations in parallel. The effect shows performance gains between 70-90% over that of the DSP processor with respect to clock cycles.

Table 3. Area results in LUTs for Xilinx Virtex II XC2V250.

Benchmark	B	B+O	B+O+S	B+O+S+U	B+O+S+U+A
dot_prod	321	181	165	1326	274
iir	697	449	282	2333	1936
fir16tap	1179	595	197	230	673
fir_cmplx	1155	853	696	3837	2990
matmul	1670	776	601	1741	974
laplace	2073	1003	799	2942	2573
sobel	1909	1174	898	8268	4160
gcd	631	440	375	612	793
ellip	3735	2987	1894	1894	1304
diffeq	1415	1171	829	1883	1814

Table 4. Comparison of normalized execution times between the TI C6000 DSP and the Xilinx Virtex II XC2V250.

Benchmark	DSP	B	B+O	B+O+S	B+O+S +U	B+O+S +U+A
dot_prod	100.0%	504.8%	277.7%	68.3%	68.3%	54.4%
iir	100.0%	514.3%	329.1%	66.8%	66.9%	37.5%
fir16tap	100.0%	1010.6%	422.6%	58.9%	51.4%	67.9%
fir_cmplx	100.0%	460.3%	266.6%	53.5%	48.2%	56.7%
matmul	100.0%	698.3%	296.9%	86.2%	48.0%	36.1%
laplace	100.0%	741.2%	223.7%	83.3%	28.0%	22.9%
sobel	100.0%	461.8%	211.8%	46.1%	120.6%	36.3%
gcd	100.0%	221.7%	127.3%	75.6%	69.0%	73.7%
ellip	100.0%	763.3%	344.3%	94.4%	94.4%	91.8%
diffeq	100.0%	226.5%	115.4%	66.9%	46.3%	21.6%

Tables 2 and 3 show frequency in MHz and area results in LUTs respectively of the 10 benchmarks on the Xilinx Virtex II FPGA for the same set of optimizations. One can see the tradeoff between area (LUTS) and performance (cycles and frequency) for various optimizations. Using the clock cycles and frequencies in Tables 1 and

2 respectively, and assuming the DSP code is running at 300 MHz, we compare final runtime results between the TI DSP C6000 and the Xilinx Virtex II FPGA in Table 4. It is interesting to note that while the DSP processor's frequency is 3 times faster than the FPGA designs, results show that the FPGA runtimes are comparable or faster than that of the DSP processor. This is achieved through exploiting the inherent parallelism in the FPGA.

Table 5. Performance comparison with the PACT compiler.

	PACT			FREEDOM		
Benchmark	**Cycles**	**Area**	**Freq**	**Cycles**	**Area**	**Freq**
dot_prod	3357	2447	69.2	1505	274	66.3
iir	3010	5873	98.4	1605	1936	55.9
fir16tap	115209	547	69.7	15918	673	62.1
fir_cmplx	8499	6083	57.9	8613	2990	62.6
matmul	277703	2155	70.2	134506	974	62.1
laplace	8467	10000	78.2	5174	2573	90.6
sobel	81418	7873	57.6	11744	4160	76.1
gcd	48	322	158.2	78	793	118.5
ellip	43	1222	180.0	105	1304	102.4
diffeq	79	1396	69.4	98	1814	58.8

One may speculate on the possibility of generating better results by decompiling software binaries into high-level languages, such as C, and using a behavioral synthesis tool to generate hardware. In order to test this hypothesis, we took all ten example benchmarks in C, and used the PACT compiler [10] to generate hardware implementations on an FPGA. Table 5 shows a comparison of the PACT compiler results with our FREEDOM compiler in terms of area, frequency and cycles. It is apparent the results of the two approaches are comparable. Hence decompilation is not a necessity to generate quality results. This validates our claim that binary or assembly code can act as an intermediate form, from any high-level language, and be used to generate efficient hardware implementations. It is also worth noting the questionable quality of decompiled code [7].

7. 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 3-20 times in the FPGA designs over that of the DSP processor in terms of reductions of execution cycles.

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.

8. 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 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 Decompilation", 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] 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.

[23] 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.