AUTOMATIC TRANSLATION OF SOFTWARE BINARIES ONTO FPGAS

ABSTRACT

Applications such as digital cell phones, 3G and 4G wireless receivers, MPEG 4 video, voice over IP, and video over IP require digital signal processing (DSP) functions that are typically mapped onto general purpose DSP processors. With the introduction of advanced FPGA architectures, which provide built-in DSP support, a new hardware alternative is available for DSP designers. This paper describes how software binaries that are targeted to execute on general purpose DSP processors can be automatically translated into Register Transfer Level (RTL) VHDL or Verilog code which in turn can be synthesized onto FPGAs using commercial logic synthesis tools. In order to develop some concrete concepts, we have used the Texas Instruments C6000 DSP processor architecture and assembly language as our DSP processor platform, and the Xilinx XC2V250 Virtex II and the Altera Stratix as example target FPGAs. The paper describes various optimizations during the translation process including automatic recognition of high-level loop constructs and array subscripts, loop unrolling, induction variable analysis, memory and register optimizations, and scheduling and binding during high-level synthesis, to get improved performance. Experimental results on resource usage and performance improvements are shown for ten software binary benchmarks in the signal processing and image processing domains.

I. INTRODUCTION

Recent advances in embedded communication and control systems for personal and vehicular environments are driving efficient hardware and software implementations of complete systems-on-a-chip (SOC). Applications such as two way radio, software defined radio, digital cellular phones, wireless Internet, 3G and 4G wireless receivers, MPEG 4 video, voice over IP, and video over IP require digital signal processing (DSP) functions that are typically mapped onto general purpose DSP processors such as the Texas Instruments TMS320C6000 and the Motorola 56600 processors. However, it is widely believed that the general purpose DSP processors are not going to be able to keep up with the computational requirements of future DSP applications.

The conventional way to address the computational bottleneck is to replace the DSP processor with an application specific integrated circuit (ASIC). ASICs have the advantage of having the fastest performance, however the design time and cost of an ASIC implementation is very high. Design and fabrication times are of the order of 12-18 months, and costs can be as high as $5-10 million for an ASIC in 0.13 micron technology. With the introduction of advanced Field-Programmable Gate Array (FPGA) architectures which provide built-in DSP support, such as embedded multipliers and block RAMs on the Xilinx Virtex-II, and DSP Blocks on an Altera Stratix device, and newer reconfigurable computing architectures from Chameleon Systems, QuickSilver Technologies and RapidChip from LSI Logic, a new hardware alternative is available for DSP designers. FPGAs combine the programming advantage of a general purpose DSP processor with the performance advantage of an ASIC. By integrating FPGA technology along with a embedded processor in a SOC, one can enable applications such as a mobile communicator...
whose hardware functionality can be easily adapted "on-the-fly" by downloading software applications -- that include hardware descriptions -- from the Internet.

DSP designs have traditionally been divided into two types of activities – systems/algorithm development and hardware/software implementation. The majority of DSP system designers and algorithm developers use the C/C++ or MATLAB languages for prototyping their DSP algorithm. If one wishes to use a TI DSP processor, they can use the CodeWarrior Tools from Metrowerks or the Mathworks Real Time Workshop to compile the code onto a TI DSP processor in the form of an assembly code. Some DSP designers implement parts of their DSP application at the assembly language level in order to get the maximum performance or the save memory.

If the performance of the implementation on a general purpose DSP processor is insufficient, a hardware implementation of the algorithm is investigated on an FPGA. Hardware design teams take the specifications created by the DSP engineers (in the form of a fixed point C code, fixed point MATLAB code, or an assembly code) 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. The process of creating an RTL model and a simulation testbench takes approximately one to two months for average designs with the tools currently used today. Hence, it is very difficult to perform various area-performance tradeoffs quickly for complex designs. While the rest of the hardware synthesis steps are automated, the manual creation of RTL HDL is an enormous bottleneck in this design process. Behavioral synthesis tools have been developed to take a behavioral description of an application in a language such as C/C++ or MATLAB and generate a RTL HDL implementation automatically.

This paper describes a compiler, which takes DSP algorithms implemented in the assembly language or binary machine code of general-purpose DSP processors, and automatically generates RTL VHDL or Verilog code to be mapped onto custom hardware using FPGAs.

The motivations for developing a translator from assembly code or binary to hardware are as follows:

1. The $4 billion DSP market sector is one of the fastest growing sectors in the semiconductor industry. As these DSP processors run out of steam in terms of computational bandwidth, there will be a need to migrate these applications from general purpose processors to hardware in the form of FPGAs.
2. There is a large established code base of DSP algorithms that are optimized for DSP processor such as the Texas Instruments C6000 DSP family. Some of the code is available in assembly language, and others are available in the form of binary code in the form of libraries that can be compiled along with a user application.

3. If DSP engineers wish to have an automated path from high-level languages to hardware, there are currently excellent tools to implement MATLAB and SIMULINK to DSP processors as part of the Real Time Workshop from The Mathworks, and tools to implement C/C++ to DSP processors as part of the CodeWarrior Tools from Metrowerks. Hence, the assembly language can be used as an intermediate language from all high-level languages.

4. Many DSP engineers develop DSP code directly in assembly for optimized performance.

5. There have been some previous efforts to take DSP applications written in MATLAB, C or C++ and mapping them into hardware using behavioral synthesis tools such as MATCH, AccelFPGA, or SystemC compilers. One problem with this approach is that they require the users to convert floating point DSP algorithms to fixed-point versions in order to produce low-cost hardware. This conversion often causes large round off and truncation errors that have to be dealt with by the hardware designers. The compiler will take DSP algorithms implemented in the assembly language or binary machine code of general-purpose DSP processors, such as the Texas Instruments, and automatically generate RTL VHDL or Verilog code to be mapped onto custom hardware using FPGAs. The advantage with this approach is that the DSP code has already been converted from a floating point to fixed-point representation, hence the generation of the hardware RTL code is relatively straightforward.

6. Many recent System-on-a-Chip (SOC) architectures such as the Xilinx Virtex II Pro and the Altera Excalibur have an integrated chip which contains an embedded processor (PowerPC on the Virtex II Pro and ARM on the Excalibur) and some programmable logic. Hence it is natural to look at hardware/software partitioning of binary codes that are originally purely software implementations on these embedded processors, and move selected parts of these codes to hardware. Our compiler will enable such a smooth transition of automatic translation of hardware software partitioning.
II. RELATED WORK

The problem of translating a high-level or behavioral language description into a register transfer level representation is called high-level synthesis [6]. 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 [7,9]. There have been several commercial efforts to develop compilers translating C/C++ into VHDL or Verilog. Examples are Adelante [1], Celoxica [3], C Level Design [4] and Cynapps [5]. SystemC is a new language developed by the SystemC consortium, which allows users to write hardware system descriptions in a language similar to C++ [10].

Banerjee et al have developed two behavioral synthesis tools, the MATCH compiler [8] which translates MATLAB to VHDL for FPGAs, and the PACT compiler [23] which translates C to VHDL/Verilog for ASICs. AccelChip has developed the AccelFPGA compiler that takes MATLAB and SIMULINK programs and produces RTL VHDL and Verilog for mapping onto FPGAs [24]. Recently, FPGA vendors such as Xilinx and Altera have developed system level tools for DSP applications called the System Generator [13] and the DSP Builder [14], which allow DSP designers to design a DSP system using a SIMULINK graphical user interface.

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. Bala et al [30] have developed the Dynamo system for dynamic binary optimization for the HP architecture. Gschwind [31] has developed a similar system called BOA for the PowerPC architecture. The Transmeta Crusoe processor performs dynamic code translation on the fly by translating code from an Intel x86 ISA and targeting the Crusoe processor [22]. Ramsey et al [18,19] has done work on the SLED tool, a front end for binary codes for various Instruction Set Architectures (ISAs). Cifuentes et al [20,21] have done a lot of fundamental work in binary translation algorithms. Our compiler is different because it does not translate from one fixed ISA to another. Rather, it translates code from one ISA and generates hardware in the form of RTL VHDL automatically.

Stitt and Vahid [25] have reported work on hardware/software partitioning of binary codes. They took kernels from frequently executed loops from some Powerstone benchmarks at the binary level for a MIPS processor, and investigated the effects of moving parts of these codes into hardware implementations on a Xilinx Virtex FPGA. However, this study was done manually and there was no automation in the form of a compiler tool. Even though in
their work it was mentioned that loops in kernels would be moved to the FPGA, it was not clear how this step could be automated. In contrast, our work presents results of an actual compiler that takes binary and assembly language programs and maps them onto FPGAs.

Stitt et al [26] 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 from a software binary and automatically map them into hardware in the form of a configurable logic fabric, which 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 have to be implemented in a single cycle. The approach also only works for sequential memory addresses and fixed size loops. The focus of their work is on fast dynamic hardware software partitioning, whereas our focus is on the actual automated synthesis of software binaries onto hardware. Our compiler takes software binaries and generates RTL VHDL and Verilog code that can be mapped onto general sequential circuits implemented on commercial FPGAs from Xilinx and Altera by integrating with commercial backend synthesis tools.

Levine and Schmidt [27] 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). The contribution of their work is in the design of an ISA for such an architecture, which would enable such hardware compilations onto the RCFs that they have designed. Our compiler is different in that it takes an existing ISA of a commercial DSP processor and maps the output RTL VHDL and Verilog code onto commercially available FPGAs.

Ye et al [28] have developed a compiler for the CHIMAERA architecture with a similar architecture of a general purpose processor connected to a reconfigurable functional unit (RFU). Certain sequences of instructions of the MIPS processor were automatically converted into operations for the RFU. Our compiler is different in that it takes binary and assembly codes from a DSP processor and maps them automatically onto commercially available FPGAs.

CriticalBlue, an Electronic Design Automation (EDA) start-up [29], has recently announced the launch of its Cascade Tool Suite. Cascade consists of a set of co-processor synthesis tools that accelerate software in embedded microprocessor applications. It analyzes application software at the compiled binary level on the main microprocessor (for an ARM processor) and, under the control of the user, maps various functions to hardware. It
subsequently 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 which would enable us to compare our compiler to their approach.

III. EXPERIMENTAL SETUP

In this paper we describe a compiler which takes DSP algorithms implemented in the assembly language or binary machine code of general-purpose DSP processors and automatically generates RTL VHDL or Verilog code to be mapped onto custom hardware using FPGAs. In order to develop some concrete concepts, we have used the Texas Instruments C6000 DSP processor architecture [15] and assembly language as our DSP processor platform, and the Xilinx XC2V250 Virtex II [14] and the Altera Stratix [2] as our target FPGA platforms. We now provide a brief description of the TI C6000 DSP processor [15], the Xilinx XC2V250 Virtex II FPGA [14] and the Altera Stratix FPGA [2], which are used to evaluate the flow of the compiler.

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. It has two multipliers that can perform two 16x16 or four 8x8 multiplies every 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. An overview of the processor architecture is shown in Figure 1.

![Figure 1. Overview of the TI C6000 DSP Processor Architecture.](image)
Our compiler generates RTL HDL for any FPGA with support for both synchronous and asynchronous on-board RAMs. However, it was tested specifically on two target FPGAs. The Xilinx Virtex II XC2V250 device consists of 1536 Combinational Logic Blocks, 48 Kbits of distributed RAM, 24 embedded multipliers, 24 embedded RAM blocks.

The Altera Stratix EPS1S10 device consists of 94 M512 RAM Blocks, 60 M4K RAMs, 1 MegaRAM Blocks, 6 DSP Blocks, and a 40 X 30 array of Logic Array Blocks.

We will report results of our compiler on a set of 10 benchmarks from the signal and image processing domains:

- A dot product application that takes two vectors of integer elements and performs a dot product on them (data size of 500 elements)
- A 16 tap Finite Impulse Response (FIR) filter (data size of 256 elements)
- A complex FIR filter using complex numbers (data size of 64 elements)
- An Infinite Impulse Response (IIR) filter (data size of 500 elements)
- A matrix multiplication program which multiplies two integer matrices (data size 16 X 16 elements)
- A matrix multiplication program which multiplies two integer matrices (data size 32 X 32 elements)
- A Laplace transform operating on a two dimensional image (data size of 32 X 32 elements)
- A Sobel transform, which determines the gradient using two-dimensional linear convolution (data size of 32 X 32 elements)
- An elliptic filter (data size of one element)
- A greatest common denominator (GCD) kernel

The benchmarks were originally available in C and were compiled into the TI 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 assembly codes were compiled by our compiler into RTL VHDL and Verilog codes. The RTL VHDL codes were synthesized using the Synplify Pro 7.2 logic synthesis tool [13] from Synplicity and mapped onto Xilinx Virtex2 XC2V250 [14] devices. The areas of the synthesized designs were measured in terms of Look Up Tables (LUTs) for the Xilinx FPGAs. The RTL VHDL codes were also simulated using the ModelSim 5.6 tool from
Mentor Graphics. In each case, it was verified that the RTL VHDL simulations provided bit-accurate simulation results with the original assembly language programs using the TI C6000 emulator. 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.

IV. OVERVIEW OF OUR COMPILER

We will now provide an overview of our compiler which takes DSP applications implemented in the assembly language or binary code of a commercial DSP processor, and automatically maps them to FPGAs by generating RTL VHDL and Verilog code.

We have implemented a front-end that parses the assembly and binary code and generates an abstract syntax tree representation of the program in a form called the Machine language Syntax Tree (MST). Various optimizations are performed at the MST level. Next, the MST representation is converted into a Control and Data Flow Graph (CDFG) representation. Various optimizations and scheduling routines are performed on the CDFG representation, which in turn is converted into a Hardware Design Language (HDL) representation. The HDL models processes, concurrency, and states in a finite state machine, and is ultimately translated into RTL VHDL and Verilog. An overview of the compiler infrastructure is shown in Figure 2.

![Figure 2. Overview of the compiler infrastructure.](image)
The front-end takes as input a description of the processor ISA in order to configure the assembly language parser. The parser uses ISA specifications written in SLED, from the New Jersey Machine-Code toolkit [21,22], and our own semantic description language.

An example of a dot product application using the TI C6000 assembly code is shown in Figure 3. Figure 4 shows the corresponding the MST representation.

Several optimizations have been implemented at the MST level of the compiler. Resource simplification replaces assembly operations that may produce costly hardware structures in RTL HDL with simplified operations. For instance, zeroing a register may be performed sometimes by subtracting the register’s value from itself. In RTL HDL this would produce an unnecessary adder that would cost area and time, so it is replaced with an MST equivalent MOVE instruction to assign the value as zero. Similarly, 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 area. Predicate levelization is required for control data flow analysis. The objective of this optimization is to replace predicated assembly instructions with branching operations to either execute or skip the instruction based on the result of the predicated
condition. Read and write conflicts may arise when assembly instructions are executed in parallel. They are handled on the MST level by adding temporary variables in case of conflicts. Finally, instruction re-sequencing is used to linearize the order of execution for instructions in branch delay slots. During the control flow analysis, this procedure plays an important role in order to take full advantage of the fine grain parallelism allowed in FPGAs, in contrast to that allowed in DSPs. Consequently, more instructions will be able to be run in parallel on the FPGAs than on the DSPs.

Once the optimizations are completed, the low-level MST is converted into a Control and Data Flow Graph (CDFG). Scheduling routines are used on the CDFG to reduce area and time, increase power efficiency and thus improve the overall performance of the design. In the conversion to the CDFG, MST instructions are grouped into different basic blocks and then broken up into node representations. Figure 5 shows the CDFG representation of the dot-product application.

![Figure 5. CDFG representation for dot product.](image)

The Scheduling and Binding pass will perform behavioral synthesis on the CDFG representation. It will schedule the computations (nodes in the data flow graph) in each basic block onto various resources (adders, multipliers, etc).

The post-scheduled CDFG is translated into another intermediate abstract syntax tree, analogous to a high-level Hardware Description Language (HDL). 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, such as in memory read and write operations. Architecture-specific information is acquired via the Architecture Description Language
(ADL) files. This includes data pertaining to resource availability, memory control, signal names, and other necessary information.

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 automated synthesized hardware, compared to that of the original TI assembly code versions.

A. Experimental Results

We now report results of our basic compiler on the set of benchmarks listed in Section IV. Table 1 shows the resource requirements measured using Look Up Tables (LUTS) while synthesizing the eight benchmarks on a Xilinx Virtex2 XC2V250 device.

Table 1. Resource usage in LUTS for the compiler on a Xilinx Virtex2 XC2V250 FPGA.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Data Size</th>
<th>LUTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>dot_prod</td>
<td>500</td>
<td>383</td>
</tr>
<tr>
<td>lir</td>
<td>500</td>
<td>684</td>
</tr>
<tr>
<td>fir16tap</td>
<td>256</td>
<td>1163</td>
</tr>
<tr>
<td>fir_cmplx</td>
<td>64</td>
<td>2511</td>
</tr>
<tr>
<td>matmul (16x16)</td>
<td>16x16</td>
<td>1609</td>
</tr>
<tr>
<td>matmul (32x32)</td>
<td>32x32</td>
<td>1378</td>
</tr>
<tr>
<td>laplace</td>
<td>32x32</td>
<td>1951</td>
</tr>
<tr>
<td>sobel</td>
<td>32x32</td>
<td>2913</td>
</tr>
<tr>
<td>gcd</td>
<td>1</td>
<td>655</td>
</tr>
<tr>
<td>ellip</td>
<td>1</td>
<td>4181</td>
</tr>
<tr>
<td>diffeq</td>
<td>1</td>
<td>1426</td>
</tr>
</tbody>
</table>

Figure 6 shows the execution time results, in cycles, as a bar chart for the eleven benchmarks. The first bar for each benchmark shows the normalized execution time used by the TI DSP processor shown as 100%. The second bar shows the normalized (with respect to the DSP execution time) execution time in cycles for the FPGA using un-optimized VHDL code. It can be seen that the execution time of the un-optimized code on an FPGA is greater than the DSP. We will discuss several optimizations in the next section to reduce the execution time on the FPGAs.
Figure 6. Comparison of Execution Time in Cycles between DSP processor and FPGA using basic compiler.

V. COMPILER OPTIMIZATIONS

It is clear from the results of the basic compiler that the performance of the implementation on the FPGA is worse than the original DSP processor code. We will now describe various compiler optimizations that we have developed in order to improve the performance of the code.

A. Exploiting Fine Grain Parallelism using Scheduling

The basic compiler takes DSP assembly and directly maps each instruction onto one RTL operation per state in a finite state machine representation. Figure 7 illustrates the mapping of some example code onto a hardware implementation. This would show no performance benefits of mapping to a custom hardware. The real benefit of migrating applications from a DSP processor onto a hardware implementation in the form of an ASIC or an FPGA is in exploiting on chip data parallelism. Hence one needs to explore the fine grain parallelism that is inherent in the CDFG of the program and schedule the operations onto resources on an FPGA.

The Scheduling and Binding pass will perform 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 type and quantity of each of these architectural resources will be described using the Architecture Description Language (ADL) of the target FPGA.
Several scheduling algorithms such as As Soon As Possible (ASAP) and As Late As Possible (ALAP) were developed as part of this framework. It should be noted that our high-level synthesis algorithms can handle multi-cycle operators during scheduling, as well as multi-cycle memory read and write accesses.

Experimental results of ASAP and ALAP scheduling with unlimited resources are shown in Figure 8. It can be seen from the results that the execution time of the FPGA implementation has now gone down.
B. 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. Consider the dot product example shown in Figure 4. Without the unrolling optimization, the assembly code has three additions, one multiplication and a subtract operation in the data flow graph of its loop body. If scheduled as such, the compiler would generate an RTL VHDL or Verilog using the same number of functional units, running the loop a hundred times. If the code were to be unrolled by 4, the loop body would be replicated four times with the loop index being adjusted as A1, A1-1,A1-2 and A1-3. Scalars, carrying values from one iteration to the next, could also be renamed. The new loop body would now be scheduled to run across 25 iterations, using 12 + 4 adders, and 4 multipliers.

A key step in applying optimizations in the compiler is to recognize high-level language constructs such as loops, and array subscripts. We have developed methods for automatically recognizing these constructs from the assembly and binary codes. The approach we have taken is based on flow graphs and common idioms used by compilers to translate high-level language constructs into the assembly code. Loops are recognized from backward branches using interval analysis, and array subscripts from base register and offset pairs.

Experimental Results of the unrolling optimization are shown in Figure 9.
The resources required in terms of Look Up Tables on the Xilinx Virtex2 for these optimizations are shown in Table 2. Three sets of results are shown. The third column shows the LUTs of the un-optimized VHDL code that is generated by converting each TI assembly instruction into an HDL statement with one RTL operation per state (no parallelism). The fourth column shows the results of using ASAP scheduling on the operator nodes of the CDFG. We perform this scheduling under no resource constraints. The fifth column shows the results using ALAP scheduling, again with unconstrained resources.

**Table 2. Results of LUTS used in various optimized codes on Xilinx Virtex II device.**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Data Size</th>
<th>Unroll Factor</th>
<th>Unopt</th>
<th>ASAP</th>
<th>ALAP</th>
<th>ASAP-Unrolled</th>
</tr>
</thead>
<tbody>
<tr>
<td>dot_prod</td>
<td>500</td>
<td>10</td>
<td>383</td>
<td>333</td>
<td>331</td>
<td>1914</td>
</tr>
<tr>
<td>fir</td>
<td>500</td>
<td>10</td>
<td>684</td>
<td>573</td>
<td>567</td>
<td>3815</td>
</tr>
<tr>
<td>fir16tap</td>
<td>256</td>
<td>8</td>
<td>1163</td>
<td>682</td>
<td>697</td>
<td>2717</td>
</tr>
<tr>
<td>fir_cmplx</td>
<td>64</td>
<td>8</td>
<td>2511</td>
<td>2189</td>
<td>2207</td>
<td>6094</td>
</tr>
<tr>
<td>matmul (16x16)</td>
<td>16x16</td>
<td>4</td>
<td>1609</td>
<td>1366</td>
<td>1395</td>
<td>2764</td>
</tr>
<tr>
<td>matmul (32x32)</td>
<td>32x32</td>
<td>4</td>
<td>1378</td>
<td>1189</td>
<td>1141</td>
<td>2120</td>
</tr>
<tr>
<td>laplace</td>
<td>32x32</td>
<td>10</td>
<td>1951</td>
<td>1562</td>
<td>1674</td>
<td>5067</td>
</tr>
<tr>
<td>sobel</td>
<td>32x32</td>
<td>10</td>
<td>2913</td>
<td>2217</td>
<td>2395</td>
<td>12584</td>
</tr>
<tr>
<td>gcd</td>
<td>1</td>
<td>2</td>
<td>655</td>
<td>653</td>
<td>627</td>
<td>928</td>
</tr>
<tr>
<td>ellip</td>
<td>1</td>
<td>NA</td>
<td>4181</td>
<td>3231</td>
<td>3613</td>
<td>3231</td>
</tr>
<tr>
<td>diffeq</td>
<td>1</td>
<td>3</td>
<td>1426</td>
<td>1189</td>
<td>1190</td>
<td>2607</td>
</tr>
</tbody>
</table>
C. Memory
The TI C6000 DSP processor has two parallel memory ports (LD1 and LD2) for supporting loads, and another two (ST1 and ST2) for stores. The processor also has 8 functional units that can execute in parallel. The maximum instruction level parallelism on the TI processor is therefore limited to 8. On an FPGA, if the compiler can identify significant parallelism, it is possible to schedule 24 multiply operations in parallel on each of the 24 embedded multipliers of the Xilinx XC2V250 device. However, in order to support these 24 parallel operations one has to feed the data from 24 parallel data ports. By using the embedded RAMs of the Xilinx FPGA it is possible to support such high data bandwidths.

The compiler generates RTL VHDL and Verilog codes that map variables onto the embedded RAMs of the Xilinx and Altera devices. For example, the Xilinx Virtex II XC2V250 device has 48 Kbits of distributed RAM, and 24 embedded RAM blocks. The Altera Stratix EP1S10 device has 94 M512 RAM Blocks, 60 M4K RAMs, 1 MegaRAM Blocks. The generated RTL VHDL and Verilog use coding styles that allow backend synthesis tools, such as Synplify Pro [13], to infer both synchronous and asynchronous RAMs automatically.

In the future we plan to try to map large arrays to separate memories. This should allow more accesses to be made in parallel. Currently we use memory pipelining to improve the memory performance.

D. Register Versus Memory Optimization

When an application is compiled for a fixed general purpose processor such as a TI C6000 CPU, it is compiled for a fixed number of registers. For example, the TI C6000 is limited to only 32/64 registers. These registers are typically accessed through a limited number of read and write ports. While compiling code for an FPGA, it is possible to use as many registers as are needed by the application. If a register output is needed to multiple destinations such as adders or multipliers, or if a particular operation unit such as an adder or a multiplier needs to accept input from multiple registers, suitable multiplexers and demultiplexers are automatically instantiated by the logic synthesis tools from the RTL VHDL or Verilog code.

A processor usually has a single address space divided into different sections like the data and code sections. These sections are frequently accessed using section registers like the Data Pointer (DP), Stack Pointer (SP), etc. When one compiles high-level language programs such as C programs onto a TI DSP processor, global variables are generally mapped onto the data memory and accessed by the DP register. The local variables are put on the stack and
accessed by an offset from SP. In addition, the limited size of the register file often requires that variables be spilled to the memory.

Figure 10 shows an example of a C code having four variables a, b, c, d, and some simple computations on them.

When compiled into C6000 assembly this straight line code gets transformed into a code using stack pointers.

```
int sum = 0;
void func(int a, int b)
{
    int c;
    int d;
    c = a * b;
    d = c + a;
    sum = c + d;
    /* more code using c & d */
    /* code to manage stack pointer */
    ....
    LD .D1 *SP[0x0], A1; a
    LD .D1 *SP[0x1], A2; b
    MPY .M1 A1, A2, A3; calc c
    ST .D1 A3, *SP[0x2]; store c
    ADD .S1 A1, A3, A5; calc d
    ST .D1 A5, *SP[0x3]; store d
    ADD .S1 A3, A5, A6; calc sum
    ST .D1 A6, *DP[0x0]; store sum
    ....
    /* rest of the code */
    /* return call, etc. */
}
```

**Figure 10. Example of mapping local variables of C program onto memory accesses using stack pointer.**

Unfortunately when such a code gets directly translated into RTL VHDL or Verilog, it generates a lot of memory accesses on the FPGA and severely limits its performance. As shown in the CDFG for this above code, the multiply operation cannot execute until both the inputs are loaded from memory. Such a code within a loop will become memory bound and not give any performance enhancement when mapped onto an FPGA.

When this code is unrolled the successive loops have to wait for the writes to terminate before performing their reads and writes. This is because of the common address space and the difficulty in distinguishing addresses from one another. This situation can be prevented sometimes by alias analysis. The fact that the loop indices are often stored on the stack makes aliasing essential for unrolling to succeed.

We have implemented a simple aliasing technique for the stack. It requires that any accesses to the stack have addresses of the type *SP[x * REG + y], where REG is the same register for all accesses and, x and y are numeric constants. It relies on the fact that the stack pointer is usually modified by immediate values only.

Induction variable analysis provides some flexibility to this technique. Within an unrolled loop, writes are converted to moves (Fig 11.b line 8) for all unrolled blocks, except the last one (line 21). Similarly LDs are converted to moves
Induction variables are also adjusted for each unrolled block. Line 4 sets up the loop index variable in $SPaddr_temp_30unrolls_ind_copy. Line 18 adjusts it for loop 2. We generate unique register names based on the address to help this process.

Figure 12 compares results of loop unrolling with and without this optimization. The execution times are normalized with respect to the DSP run times.
Figure 12. Results of the register optimization

E. Mapping to Different FPGA Architectures

Our compiler generates RTL VHDL and Verilog code that can be synthesized onto various FPGAs. Table 3 shows the results of our mapping of various designs on a Xilinx Virtex2 and an Altera Stratix FPGA. It can be seem that our compiler can be used to very rapidly evaluate the performance of translating the binary code onto different FPGA architectures with different resource and frequency results.

Table 3. Comparison of designs on a Xilinx Virtex2 and an Altera Stratix FPGA in terms of area (LUTS) and frequency of the design (MHz).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Data Size</th>
<th>Xilinx Virtex2</th>
<th></th>
<th>Altera Stratix</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LUTS</td>
<td>Freq (Mhz)</td>
<td>LUTS</td>
<td>Freq (MHz)</td>
<td></td>
</tr>
<tr>
<td>dot_prod</td>
<td>ASAP-Unrolled</td>
<td>1914</td>
<td>103.1</td>
<td>ASAP-Unrolled</td>
<td>1789</td>
</tr>
<tr>
<td>lir</td>
<td>ASAP-Unrolled</td>
<td>3815</td>
<td>90.0</td>
<td>ASAP-Unrolled</td>
<td>5954</td>
</tr>
<tr>
<td>fir16tap</td>
<td>ASAP-Unrolled</td>
<td>2717</td>
<td>72.2</td>
<td>ASAP-Unrolled</td>
<td>3156</td>
</tr>
<tr>
<td>fir_cmplx</td>
<td>ASAP-Unrolled</td>
<td>6094</td>
<td>59.8</td>
<td>ASAP-Unrolled</td>
<td>11010</td>
</tr>
<tr>
<td>matmul (16x16)</td>
<td>ASAP-Unrolled</td>
<td>2764</td>
<td>106.0</td>
<td>ASAP-Unrolled</td>
<td>2926</td>
</tr>
<tr>
<td>matmul (32x32)</td>
<td>ASAP-Unrolled</td>
<td>2120</td>
<td>107.9</td>
<td>ASAP-Unrolled</td>
<td>2095</td>
</tr>
<tr>
<td>laplace</td>
<td>ASAP-Unrolled</td>
<td>5067</td>
<td>76.2</td>
<td>ASAP-Unrolled</td>
<td>6141</td>
</tr>
<tr>
<td>sobel</td>
<td>ASAP-Unrolled</td>
<td>12584</td>
<td>33.7</td>
<td>ASAP-Unrolled</td>
<td>11188</td>
</tr>
<tr>
<td>gcd</td>
<td>ASAP-Unrolled</td>
<td>928</td>
<td>86.1</td>
<td>ASAP-Unrolled</td>
<td>778</td>
</tr>
<tr>
<td>ellip</td>
<td>ASAP-Unrolled</td>
<td>3231</td>
<td>71.7</td>
<td>ASAP-Unrolled</td>
<td>3808</td>
</tr>
<tr>
<td>diffeq</td>
<td>ASAP-Unrolled</td>
<td>2607</td>
<td>74.2</td>
<td>ASAP-Unrolled</td>
<td>3835</td>
</tr>
</tbody>
</table>

Table 3. Comparison of designs on a Xilinx Virtex2 and an Altera Stratix FPGA in terms of area (LUTS) and frequency of the design (MHz).
VI. CONCLUSIONS

This paper described a compiler which takes DSP algorithms written in the assembly language or binary code of a DSP processor such as the Texas Instruments C6000, and automatically generates Register Transfer Level (RTL) VHDL or Verilog code for synthesizing designs on to Xilinx Virtex II and Altera Stratix FPGAs. Experimental results were shown on eight assembly language benchmarks from signal processing and image processing domains.

The preliminary results of our compiler infrastructure are very encouraging. In the future, we plan to look at the following issues. First, we will continue to look at more complex benchmarks (e.g. MPEG, JPEG, Viterbi, Turbo decoders, 3G and 4G wireless applications, image processing) and more optimizations and see how close we come to manual design. We will look at some example benchmarks for which there are System Generator blocks available (i.e. FIR filter, FFT) for Xilinx FPGAs, and compare the quality of results (area in LUTS, frequency, latency, throughput) with the compiler. We realize that it is unlikely that an automated compiler will be able to compete with a hand optimized design, but we would like to experimentally evaluate how much worse the compiler output is compared to the best manual designs.

A very important aspect that we plan to study is whether it is possible to recreate high-level programming language information such as loop constructs and array accesses that are often lost when one looks at assembly or binary codes. Such high level constructs can improve the automatic detection of parallelism and also enable the use of various high-level loop transformations such as loop unrolling, loop distribution, loop fusion, loop interchange, and loop tiling. We will experimentally compare the differences in the quality of synthesis between a high-level language approach to an assembly approach. In the first approach, we will take C or MATLAB versions of a DSP application, and synthesize the designs onto FPGAs or ASICs using PACT or MATCH compiler previously developed by our group. In the second approach, we will take the C or MATLAB versions of the same applications, compile into assembly, and then use the compiler to compile onto an FPGA or ASIC. We will compare the two approaches in terms of quality in area and performance.

The real value of mapping designs onto FPGAs or ASICs is in exploiting on-chip parallelism and tuning the data path widths to the applications. Hence we will implement the unroll optimization, pipeline optimization, data-path width optimization, and memory tiling optimization and evaluate the impact of these optimizations on a wide range of benchmarks.
Finally, we will investigate the issues of hardware/software co-design in the future.

VII. REFERENCES


