EVALUATION OF SCHEDULING AND ALLOCATION ALGORITHMS WHILE MAPPING ASSEMBLY CODE ONTO FPGAS

ABSTRACT

Migration of software from older general purpose embedded processors onto newer mixed hardware/software Systems-On-Chip (SOC) platforms is becoming an increasingly important topic. Automatic translation of general purpose software binaries and assembly code onto hardware implementations using FPGAs require sophisticated scheduling and allocation algorithms to maximize the resource utilization of such hardware devices. This paper describes the effects of scheduling and chaining of node operations in a CDFG onto an FPGA. The effects of register allocation on scheduled nodes are also discussed. The Texas Instruments C6000 DSP processor architecture was chosen as the DSP processor platform and assembly code, and the Xilinx Virtex II XC2V250 was chosen as the target FPGA. Results are reported on ten benchmarks, which show that scheduling with chaining operations produces the best results on FPGAs, while the addition of register allocation in fact generates poorer designs in terms of area and frequency.

I. INTRODUCTION

Recent advances in embedded communications and control systems for personal and vehicular environments are driving efficient hardware and software implementations of complete systems-on-chip (SOC). These applications require digital signal processing (DSP) functions that are typically mapped onto general-purpose DSP processors, such as the Texas Instruments C6000 [3] and the Motorola 56600. However, it is widely believed that such processors will be unable to support the computational requirements of future DSP applications. It is therefore desirable to migrate fragments of code or functions to a hardware implementation on FPGAs. The benefit of migrating to such devices is in utilizing its inherent parallelism via scheduling more computations per cycle than possible on a DSP processor. Towards this effort, we developed the FREEDOM compiler, which automatically translates software binaries targeted for general DSP processors into Register Transfer Level (RTL) VHDL or Verilog code to be mapped onto commercial FPGAs. The designs are optimized and scheduled to achieve maximum utilization of the FPGA’s resources.

The classical high-level synthesis problem is one of transforming a behavioral model in a high-level application into a set of multi-cycle operations, which have been scheduled for optimal performance. It is common for one to explore alternate methods of scheduling to improve the performance of the design. In doing so, we attempt to exploit the parallelism in the design as much as possible for implementations on FPGAs. Operation chaining is a technique that is most effective, in which the result of the operation is used immediately rather than being stored in a register until the next cycle. Register allocation is another optimization that complements scheduling by reducing the number of registers in a design. The number of possible register reuses is dependant upon the type of scheduling routine that is implemented.

The problem of translating software binaries and assembly to FPGAs is interesting because assembly code consists of scheduled instructions on a fixed processor architecture having a fixed number of functional units, a fixed number of registers, and loads and stores of variables from external memory. It is quite challenging to translate these prescheduled list of instructions onto commercial FPGAs, where one can exploit a great deal of parallelism using a much larger number of functional units, embedded multipliers, registers and on-chip embedded memories. The contribution of this paper is in evaluating a wide range of scheduling, operator chaining and register allocation algorithms within the context of the problem of translating software assembly to FPGAs.

The remainder of the paper is organized as follows: Section II presents the motivation for this work. Section III discusses related work. An overview of the compiler framework for binary to hardware translation is presented in Section IV. Sections V, VI, and VII describe the effects of scheduling, operation chaining, and register allocation optimizations implemented on FPGA designs that were translated from software binaries. Section VIII reports experimental results on ten benchmarks. Conclusions and future work are discussed in Section IX.

II. PROBLEM MOTIVATION

Consider the example Texas Instruments C6000 DSP processor [3] assembly code in Figure 1. The DSP processor has eight functional units (.L1, .S1, .M1, .D1, etc.), and therefore may execute at most eight instructions in parallel. In the example code, the MPY instructions require two cycles to execute and all other instructions require one cycle; hence, the instruction following the MPY is a NOP in this example. The symbol in certain instructions signify that the instruction is executed in parallel with the previous instruction. As a result, the section of code requires seven cycles to execute.

A simple translation of this code onto an FPGA by assigning one operation per state in an RTL finite state machine would produce no cost benefit. It is simplest form, the design would require eight cycles to complete on an FPGA since there are eight instructions, excluding NOPs. Rather, one must explore...
the parallelism in the design through scheduling techniques in order to exploit the fine-grain parallelism inherent in the FPGA architecture, thereby reducing the number of execution clock cycles. Consequently, very little work has been done in analyzing the efficiency of different methods of scheduling and operation chaining in FPGA designs. Likewise, the impact of register allocation after such scheduling techniques requires investigation as well.

\[
\begin{align*}
&\text{MV .L1 A0, A1} \\
&\text{MV .L2 B0, B1} \\
&\text{MPY .M1 A1, A2, A3} \\
&\text{MPY .M2 B1, B2, B3} \\
&\text{NOP 1} \\
&\text{MPY .M1 A3, A6, A7} \\
&\text{MPY .M2 B3, B6, B7} \\
&\text{NOP 1} \\
&\text{ADD .L1X A7, B7, A8} \\
&\text{ADD .L1 A4, A8, A9}
\end{align*}
\]

Figure 1. Example TI C6000 DSP Processor assembly code.

III. RELATED WORK

The problem of translating a high-level or behavioral language description into a register transfer level (RTL) representation 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 translates software binaries and assembly language codes into RTL HDL for mapping onto FPGAs.

Stitt and Vahid [11,12] have reported work on hardware-software partitioning of binary codes. CriticalBlue [13] has recently announced the Cascade Tool that synthesizes a hardware co-processor specifically designed to accelerate software tasks selected by the user.

Scheduling is a very important problem in behavioral synthesis. For a given data flow graph, scheduling determines the concurrency of the resulting implementation by assigning operations in a CDFG to specific cycles assuming either constrained or unconstrained resources. Numerous algorithms for scheduling have been developed over the years by various researchers [1]. Some example scheduling algorithms are As-Soon-As-Possible (ASAP), As-Late-As-Possible (ALAP), various versions of list scheduling, force directed scheduling, and scheduling based on integer linear programming [1]. We study the use of resource constrained and unconstrained ASAP and ALAP algorithm in this paper within the context of scheduling software binaries to FPGAs.

Regardless of the scheduling routine performed, register allocation is an optimization that is generally performed after scheduling to minimize the registers in the design. The graph-coloring method, adapted by Chaitin et al. [7,8], is a widely used approach to register allocation. It iteratively builds an interference graph and heuristically attempts to color it. While graph-coloring usually results in very effective allocations, it can be very expensive since compilers may generate numerous register candidates with temporary variables [9].

A simpler and significantly faster approach to register allocation is the Linear-Scan method, developed by Poletto and Sarkar [10], which is not based on graph coloring. Given a range of lifetimes for each variable, the greedy algorithm allocates the variables to registers in a single pass. Traub et al. [9] have developed a more efficient algorithm based on Linear-Scan called Binpacking, which allocates registers and rewrites the instruction stream in a single scan, and also makes use of lifetime holes in temporary variables.

While much research has been done in terms of comparing the quality of results between different approaches in register allocation, many have failed to ascertain the quality of these results when implemented on different reconfigurable hardware devices that have restrictions in routing. The importance of this research is in quantifying the results on FPGAs, which have fixed architectures and routing.

IV. OVERVIEW OF THE COMPILER

This section provides an overview of the FREEDOM compiler that automatically translates software binaries into Register Transfer Level (RTL) VHDL or Verilog code to be mapped onto commercial FPGAs. The FREEDOM compiler’s infrastructure is shown in Figure 2.

![Diagram of compiler infrastructure](image-url)

The compiler was designed to have a common entry point for all assembly languages. Towards 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 [4,5], 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). This representation is similar to the MIPS ISA in syntax, and is 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 operands may be null.

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

Several traditional optimizations, such as constant propagation and dead code elimination, have been implemented on the CDFG [2]. These optimizations serve to reduce the execution clock cycles, design area and power...
consumption, while increasing the frequency of the design. The compiler runs these optimizations repeatedly until the design becomes stable. Resource binding is applied to operation nodes in the CDFG based on the available resources of the target FPGA. Architecture-specific information, such as resource and signal names, is acquired via the Architecture Description Language (ADL) files. Finally, a scheduling pass assigns individual operation nodes in the CDFG to states in a finite state machine.

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. Memory models are generated in the HDL as required by backend synthesis tools, such as Synplify Pro [6], to automatically infer both synchronous and asynchronous RAMs. The complete HDL is translated directly to RTL VHDL and Verilog to be mapped onto FPGAs. A testbench is also generated for verification of bit-true accuracy.

V. SCHEDULING

The scheduling pass is implemented at the CDFG level after all optimizations are complete. We assume the CDFG is in static-single variable assignment (SSA) form. The nodes in the CDFG are distinguished in five different types: Constants, Variables, Values, Control, and Memory. Constant and Variable nodes are inputs to operations. Value nodes are operation nodes, such as addition and multiplication. Control nodes represent branch conditions in the control flow. Memory nodes refer to memory read and write operations. The nodes in a basic block are scheduled based on a time-step scheduling and As-Late-As-Possible (ALAP) scheduling [1].

If scheduling were not performed on the CDFG, each node operation would be mapped to an independent state of an RTL finite state machine. This would produce a very costly design in terms of clock cycles. Assuming a single clock cycle delay for each operation, the design would require eight states (or clock cycles) to complete, which is slower than the original implementation of seven cycles on the DSP processor. Unlike the DSP processor, an FPGA is capable of executing a much larger number of operations per cycle, and therefore can benefit from parallel node scheduling to optimize the utilization of the FPGA’s resources. Towards this effort, we implemented two well-known scheduling routines for basic block nodes, namely As-Soon-As-Possible (ASAP) scheduling and As-Late-As-Possible (ALAP) scheduling [1]. The nodes in a basic block are scheduled based on a time-step and the operation delay of the FPGA’s resource.

VI. OPERATION CHAINING

Designs that are run on DSP processors are limited to the processor’s architecture in terms of the number of operations that can execute in a single cycle. Conversely, FPGA designs are dynamic in the sense that one has the freedom to increase or decrease the number of operations executed per state cycle. In addition to operation scheduling discussed above, RTL HDL allows one to chain a number of operations in sequence within a single state in order to maximize the number of operations executed per cycle. The chaining process is performed prior to scheduling, and is accomplished by assigning the delay on the node operator to zero. The effect of chaining is generally a tradeoff between larger critical paths and reduced frequencies versus a significant reduction in execution clock cycles.

In RTL VHDL and Verilog, chained operations are distinguished by using blocking assignments as opposed to non-blocking assignments. Blocking assignments are analogous to wires, in that the result is not stored in registers and may be read within the same state cycle in which it was written. Conversely, non-blocking assignments store their results in registers, and may be read in the next state cycle.

When chaining RTL HDL operations, one must consider the impact on the critical path of the design. Ideally, it is best not to chain complex structures, such as multipliers, for they cause a significant increase in the critical path and reduction in frequency. However, chaining operations also reduces the number of registers, which may decrease the design area. One must also consider the restrictions imposed by backend synthesis tools, which do not allow mixing blocking and non-blocking assignments to the same destination operand. It is therefore necessary for one to first determine which nodes are valid for chaining.

Figure 4 describes a routine for generating a map of valid registers for chaining. Lines 1-3 iterate through the nodes of each block in the CDFG. Lines 4-6 determine if a node is
valid for chaining. The is_valid_op function returns true if the operation node meets certain qualifications for chaining, which is discussed below in more detail. Lines 7-8 map the valid flag to the register name while ensuring the value is not overwritten if it was already determined that the register cannot be chained. The algorithm runs in O(V) time, where V is the total number of nodes in the entire CDFG.

The Chaining routine in Figure 5 iterates through the nodes in each basic block, and sets the operation delay to zero if it is a valid chaining register. The algorithm also runs in O(V) time.

```
Generate_Valid_Chain_Map( blocks, map )
1    for each block b in blocks do
2        let node_list = b->nodes
3        for each node n in node_list do
4            let valid = false
5                if is_valid_op(n) then
6                    valid = true
7                    if map[n->name] != false then
8                        map[n->name] = valid
```

Figure 4. Pseudo-code for generating a map of valid chains

```
Chaining( blocks, map )
1    for each block b in blocks do
2        let node_list = b->nodes
3        for each node n in node_list do
4            if map[n->name] == true do
5                n->delay = 0
```

Figure 5. Chaining routine for a CDFG basic block.

We consider three approaches to chaining: simple chaining, complex chaining, and unconstrained chaining.

In simple chaining, only simple ALU operations are chained. These include, but are not limited to logical operations, conditional set operations, addition, and subtraction. Complex structures, such as multiplication, are isolated in separate states so as to reduce the critical path. In this case, the is_valid_op function returns true if the following conditions are true: the node has successors, the operation is not a complex structure, the node does not have a successor that is a complex structure, and the node is not a memory write operation.

```
State: s0
B1 := B0
A1 := A0
Next State: s1

State: s1
Next State: s2

State: s2
B7 := B3[15:0] * B6[15:0]
A7 := A3[15:0] * A6[15:0]
Next State: s3

State: s3
A8 := A7 + B7
A9 := A4 + A8
Next State: s4
```

Figure 6. CDFG and HDL representation of ASAP scheduling with simple chaining for TI code in Figure 1.

In complex chaining, operations up to and including a single complex structure are chained. When considering DSP applications, it is more common to find sequences of multiply-accumulate operations than accumulate-multiply sequences. Consequently, one would expect this approach to produce larger critical paths if multipliers were chained with successive nodes rather than preceding nodes. The function is_valid_op returns true if the following conditions are true: the node has successors, the operation is not a complex structure, and the node is not a memory write operation.

```
State: s0
B1 := B0
A1 := A0
Next State: s1

State: s1
Next State: s2

State: s2
B7 := B3[15:0] * B6[15:0]
A7 := A3[15:0] * A6[15:0]
Next State: s3

State: s3
A8 := A7 + B7
A9 := A4 + A8
Next State: s4
```

Figure 7. CDFG and HDL representation of ASAP scheduling with complex chaining for TI code in Figure 1.

In unconstrained chaining, all operations may be chained. In this case, the function is_valid_op returns true if the following conditions are true: the node has successors, and the node is not a memory write operation.

```
State: s0
B1 := B0
A1 := A0
Next State: s1

State: s1
Next State: s2

State: s2
B7 := B3[15:0] * B6[15:0]
A7 := A3[15:0] * A6[15:0]
Next State: s3

State: s3
A8 := A7 + B7
A9 := A4 + A8
Next State: s4
```

Figure 8. CDFG and HDL representation of ASAP scheduling with unconstrained chaining for TI code in Figure 1.
the clock frequency of the design will be much lower than the design without chaining in Figure 3. We would like to experimentally evaluate the impact of these chaining algorithms on the total execution time (product of clock cycles and cycle latency).

VII. REGISTER ALLOCATION

Register allocation is an optimization that is performed after scheduling to reduce the number of registers. Reducing registers in circuit designs 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 and chaining of operations affect the number of possible register reuses. One can perform register allocation to reduce the number of registers. However, by reusing a small number of registers one may require more complex multiplexers in front of functional units and longer interconnects. In this study, we want to experimentally evaluate the impact of register allocation while mapping software assembly code onto FPGAs. 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. Structural analysis is performed beforehand to identify loops and other constructs. 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.

Our approach for calculating the live intervals of registers in a CDFG is as follows: The nodes in each basic block are sorted in depth-first order. We then iterate through the nodes in each basic block and map a list of nodes sharing the same name along with a time interval to the name in a register table. Each node that is encountered is added to the table under its corresponding name. The lifetime interval is updated by comparing the current start time with the node’s timestamp and comparing the current end time with the timestamp of all successive uses of the node. If a node is a Variable type and exists inside a loop, the time interval is extended to the loop body’s time boundaries. The register table is used in the Linear-Scan algorithm by renaming the list of nodes in each mapping with a newly allocated register name. We use a simple approach that does not make use of lifetime holes or necessitate memory spilling. The algorithm runs in O(V) time, where V is the total number of nodes in the CDFG.

VIII. EXPERIMENTAL RESULTS

This section reports the results of the FREEDOM compiler on a set of ten benchmarks from the signal and image processing domains, shown in Tables 1-4. The benchmarks were originally available in C and compiled into the TI C6000 assembly code using the Code Composer Studio from Texas Instruments. The designs were unrolled several times where applicable in order to increase the number of operation nodes.

The RTL HDL codes generated by the FREEDOM compiler were synthesized using the Synplify Pro 7.2 logic synthesis tool from Synplicity and mapped onto Xilinx Virtex II XC2V250 devices. 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 were 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.

Tables 1-4 show results in terms of clock cycles, area, frequency and final execution times, respectively. The first column of each table shows the results of the compiler’s base case (B) without scheduling. The use of scheduling optimizations facilitates a reduction in the number of states (clock cycles) and design area. The results are apparent when comparing the base case (B) and the scheduling alone (B+S); frequency results are comparable between the two. Consequently, when combining scheduling with simple chaining (B+S+SC), complex chaining (B+S+CC) and unconstrained chaining (B+S+UC), results show increasingly improved performance in clock cycles and design area due to reduction in registers. The frequency decreases due to larger critical paths, which is an effect of operation chaining. As was expected, most cases showed unconstrained chaining producing better results in terms of clock cycles and area, while scheduling without chaining produced the best frequency results.

The final column (B+S+UC+R) shows the effects of register allocation on area and frequency after chaining. It is interesting to note that this optimization in fact caused a negative effect in almost all designs and in all forms of scheduling; results showed design areas increased while frequencies decreased. We speculate the effect of register...
reuse causes the backend synthesis tools to insert additional multiplexers in order to support the multiple uses of the register across different combinational logic blocks. This in fact causes the design area, interconnect and the critical path to increase, thus affecting the frequency of the design as well. Accordingly, Table 4 shows that the optimal method for FPGA designs would be to use scheduling and chaining alone, leaving the design in SSA form by not implementing register allocation.

<p>| Table 4. Final execution times in µs for Xilinx Virtex II FPGA. |
|-----------------------|------------------|------------------|------------------|-----------------|-------------------|</p>
<table>
<thead>
<tr>
<th>dot_prod</th>
<th>B</th>
<th>B+S</th>
<th>B+S+SC</th>
<th>B+S+CC</th>
<th>B+S+UC R</th>
</tr>
</thead>
<tbody>
<tr>
<td>ir</td>
<td>101.7</td>
<td>52.7</td>
<td>49.6</td>
<td>41.3</td>
<td>39.3</td>
</tr>
<tr>
<td>fisttap</td>
<td>1526.4</td>
<td>462.3</td>
<td>279.5</td>
<td>236.2</td>
<td>236.2</td>
</tr>
<tr>
<td>cmplx</td>
<td>376.9</td>
<td>53.0</td>
<td>55.3</td>
<td>54.2</td>
<td>54.2</td>
</tr>
<tr>
<td>matmul</td>
<td>1727.1</td>
<td>3943.0</td>
<td>2948.9</td>
<td>3812.7</td>
<td>3812.7</td>
</tr>
<tr>
<td>laplace</td>
<td>3859.6</td>
<td>324.6</td>
<td>202.2</td>
<td>201.2</td>
<td>201.2</td>
</tr>
<tr>
<td>sobel</td>
<td>1069.5</td>
<td>371.8</td>
<td>226.3</td>
<td>226.3</td>
<td>226.3</td>
</tr>
<tr>
<td>gcd</td>
<td>0.9</td>
<td>0.7</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>ellip</td>
<td>1.6</td>
<td>1.2</td>
<td>0.9</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>diffreq</td>
<td>8.8</td>
<td>5.4</td>
<td>3.6</td>
<td>3.3</td>
<td>3.3</td>
</tr>
</tbody>
</table>

IX. CONCLUSIONS

The FREEDOM compiler 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. This paper evaluated a wide range of scheduling, operator chaining and register allocation algorithms within the context of the problem of translating software binaries to FPGAs using the framework of the FREEDOM compiler.

Experimental results were shown on ten assembly language benchmarks from signal processing and image processing domains. Results show performance gains on the FPGA designs that used a combination of scheduling and chaining with respect to area, frequency and execution clock cycles. Register allocation has shown to produce poorer results than those designs left in SSA form.

X. REFERENCES