Evaluation Of Scheduling And Allocation Algorithms While Mapping Assembly Code Onto FPGAs

David Zaretsky, Gaurav Mittal, Xiaoyong Tang, Prith Banerjee

Northwestern University
Department of Electrical and Computer Engineering
2145 N. Sheridan Road
Evanston, IL 60208-3118

{dcz, mittal, tang, banerjee}@ece.northwestern.edu

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.

Categories and Subject Descriptors

B.5.0 [**Register-Transfer-Level Implementation**]: *General.*
B.7.1 [**Types and Design Styles**]: *Algorithms implemented in hardware, Gate Arrays, VLSI (very large scale integration).*

General Terms

Algorithms, Performance, Design, Experimentation, Verification.

Keywords

Hardware Synthesis, Optimizations, Scheduling, Chaining, Compilers, Binary Translation, FPGAs.

GLSVLSI'04, APRIL 26-28, 2004, BOSTON, MASSACHUSETTS, USA

1. 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 and 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 2 presents the motivation for this work. Section 3 discusses related work. Sections 4, 5, and 6 describe the effects of scheduling, operation chaining, and register allocation optimizations implemented on FPGA designs that were translated from software binaries. Section 7 reports experimental results on ten benchmarks. Conclusions and future work are discussed in Section 8.

2. PROBLEM MOTIVATION

Consider the example Texas Instruments C6000 DSP processor [3] assembly code in Figure 1. The TI DSP processor has eight functional units (.L1, .S1, .M1, .D1, etc.), and therefore may execute at most eight instructions in parallel. 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. In it's 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.

```
        MV    .L1   A0,A1
||      MV    .L2   B0,B1
        MPY   .M1   A1,A2,A3
||      MPY   .M2   B1,B2,B3
        NOP         1
        MPY   .M1   A3,A6,A7
||      MPY   .M2   B3,B6,B7
        NOP         1
        ADD   .L1X  A7,B7,A8
        ADD   .L1   A4,A8,A9
```

Figure 1. Example TI C6000 DSP Processor assembly code.

3. 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.

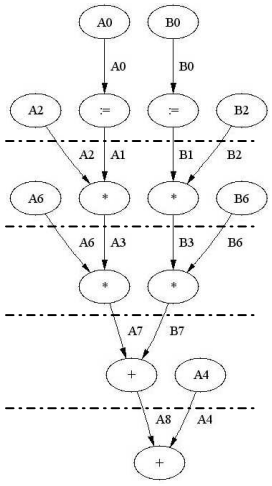
Scheduling is a very important problem in behavioral synthesis. For a given data flow graph, scheduling determines the concurrency of the resulting implementation by assigning operations in a CDFG to specific cycles assuming either constrained or unconstrained resources. Numerous algorithms for scheduling 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 using graph coloring. 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.

4. SCHEDULING

The scheduling pass is implemented at the CDFG level after all other optimizations. If scheduling were not performed on the CDFG, each node operation would be mapped to an independent state of an RTL finite state machine, which would produce a very costly design in terms of clock cycles. 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.




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

State: s1
    B3 <= B1[15:0] * B2[15:0]
    A3 <= A1[15:0] * A2[15:0]
    State <= s2

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

State: s3
    A8 <= A7 + B7
    State <= s4

State: s4
    A9 <= A4 + A8
    State <= s5
```

Figure 2. CDFG and HDL representation of ASAP scheduled code in Figure 1.

Towards this effort, we implemented two well-known scheduling routines, 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. Figure 2 shows the CDFG and corresponding HDL after running ASAP scheduling on the assembly code in Figure 1. The MV, ADD and MPY operations, along with its destination register, are each represented as *Value* nodes, while the input source registers (A0, A2, A6, etc.) are represented as *Variable* nodes. We assume each *Value* type operation has a single clock cycle delay on the FPGA; *Constant* and *Variable* types have a delay of zero. The design runs in only five clock cycles, as opposed to eight clock cycles without scheduling.

5. OPERATION CHAINING

Designs that are run on DSP processors are limited to the processor's architecture in terms of the number of operations that may be executed in a single cycle. Conversely, FPGA designs are allow 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, effectively making the operation instantaneous. The effect of chaining is generally a tradeoff between a significant reduction in execution clock cycles and area versus larger critical paths and reduced frequencies.

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.

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, addition, and subtraction. Complex structures, such as multiplication, are isolated in separate states so as to reduce the critical path. When *simple chaining* is applied to the CDFG in Figure 2, the first set of assignment operations are assigned to the first state; the two sets of multiplication operations are isolated in the second and third states; the last two addition operations are chained together in the fourth state. The resulting design takes four cycles to complete.

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. When *complex chaining* is applied to the CDFG in Figure 2, the first set of assignment and multiplication operations are chained together in the first state; the second set of multiplication operations is isolated in the second state; the addition operations are chained in the third state. The resulting design takes three cycles to complete.

In *unconstrained chaining*, all operations are valid for chaining. Therefore, all operations in Figure 2 are chained together in a single state, and the resulting design takes only a single cycle to complete. However, even though the number of clock cycles in this design is reduced to one, the critical path delay requires five operations, including two multipliers in sequence. Hence, the clock frequency of the design will be much lower than that with scheduling alone.

6. 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.

Register allocation was implemented on the FREEDOM compiler using the Linear-Scan (left-edge) algorithm [10]. We use a simple approach that does consider lifetime holes or necessitate memory spilling. We assume the nodes in the CDFG are in SSA form, in which data dependencies are broken. We also assume an unbound number of register resources in the target FPGA, and our task is to assign the variable lifetimes to the smallest subset of registers.

Prior to running the Linear-Scan algorithm, one must determine the *liveness* of each variable, or the time from the variable's first definition until its final use in a CDFG. 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 its start time with the node's timestamp, and comparing its end time with the timestamp of all successive uses of the node. If a variable is used inside a loop and its definition exists outside the loop body, 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. The algorithm runs in O(V) time, where V is the total number of nodes in the CDFG.

7. EXPERIMENTAL RESULTS

This section reports the results of the FREEDOM compiler on a set of ten benchmarks from the signal and image processing domains. 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. Estimated frequencies and area utilization were obtained from these synthesis results. 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 clock cycles were measured by counting the total number of cycles in the ModelSim simulations of each FPGA design.

Tables 1-2 shows results in terms of final execution times (clock cycles / frequency) and area. The first column 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 because the critical path has not changed. 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, an effect of operation chaining. As 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 due to smaller critical paths.

Table 1. Final execution times in μs for Xilinx Virtex II FPGA.

	B	B+S	B+S+SC	B+S+CC	B+S+UC	B+S+UC+R
dot_prod	101.7	52.7	49.6	41.3	39.3	41.1
iir	238.7	79.1	72.3	65.1	57.9	66.0
fir16tap	1526.4	462.3	279.5	296.2	296.2	403.7
fir_cmplx	376.9	53.0	55.3	54.2	54.2	79.3
matmul	17217.1	3943.0	2948.9	3812.7	3812.7	4272.5
laplace	589.6	324.6	201.2	201.2	201.2	196.3
sobel	1069.5	371.8	226.3	226.3	226.3	238.0
gcd	0.9	0.7	0.4	0.4	0.4	0.4
ellip	1.9	1.2	0.9	0.9	0.9	0.9
diffeq	8.8	5.4	3.6	3.3	3.5	4.1

Table 2. Area results in LUTs for Xilinx Virtex II FPGA.

	B	B+S	B+S+SC	B+S+CC	B+S+UC	B+S+UC+R
dot_prod	1958	1807	1544	1563	1578	3033
iir	4025	3694	2634	2606	2544	8232
fir16tap	2813	2426	1938	1922	1880	2899
fir_cmplx	4361	3740	3324	3295	2873	9013
matmul	3222	2633	1765	1906	1871	3101
laplace	6089	5609	3949	3949	3949	10327
sobel	11582	9531	4379	4379	4379	8629
gcd	766	673	456	456	456	712
ellip	3082	3112	2023	2023	2023	2744
diffeq	3274	2914	2381	2474	2283	5120

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. 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, 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.

8. 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 execution times and area. Register allocation produced poorer results than those designs left in SSA form.

Future work would include heuristical methods for operation chaining in which one would consider a more accurate representation of an operation's delay on the FPGA.

9. 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] Texas Instruments, TMS320C6000 Architecture Description, www.ti.com

[4] N. Ramsey, and M.F. Fernandez, "Specifying Representations of Machine Instructions", ACM Transactions on Programming Languages and Systems, May 1997.

[5] N. Ramsey, and M.F. Fernandez, "New Jersey Machine-Code toolkit", Proceedings of the 1995 USENIX Technical Conference, January 1995.

[6] Synplicity. Synplify Pro Datasheet, www.synplicity.com.

[7] G. Chaitin et al., "Register Allocation via Coloring," Computer Languages, 6, pp. 47-57, 1981.

[8] G. J. Chaitin, "Register Allocation and Spilling via Graph Coloring," SIGPLAN Notices, 17(6):201-107, June 1982.

[9] O. Traub et al., "Quality and Speed in Linear-scan Register Allocation," ACM SIGPLAN 1998 Conf. On Programming Language Design and Implementation, pp. 142-151, June 1998.

[10] M. Poletto and V. Sarkar, "Linear Scan Register Allocation," ACM Trans. on Programming Languages and Systems, Vol. 21, No. 5, pp. 895-913, Sept. 1999.

[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] CriticalBlue, Cascade Tool Set, www.criticalblue.com