Path-based Scheduling in a Hardware Compiler

Ruirui Gu
Department of Electrical and Computer Engineering
University of Maryland at College Park
College Park, MD, USA
rgu@umd.edu

Alessandro Forin, Neil Pittman
Microsoft Research
Microsoft Corporation
Redmond, WA, USA
{sandorf, pittman}@microsoft.com

Abstract—Hardware acceleration uses hardware to perform some software functions faster than it is possible on a processor. This paper proposes to optimize hardware acceleration using path-based scheduling algorithms derived from dataflow static scheduling, and from control-flow state machines. These techniques are applied to the MIPS-to-Verilog (M2V) compiler, which translates blocks of MIPS machine code into a hardware design represented in Verilog for reconfigurable platforms. The simulation results demonstrate a factor of 22 in performance improvement for simple self-looped basic blocks over the base compiler.

Keywords—compiler; MIPS; Verilog, FPGA, scheduling

I. INTRODUCTION

Hardware acceleration uses hardware to perform some computation faster than it is possible in software on embedded processors or general purpose CPUs. Applications of hardware acceleration [4][5] can be found in a wide range of embedded systems such as graphics processing units (GPUs) in mobile devices. Hardware accelerations improve computationally intensive software codes and/or frequently executed programs exploiting the spatial concurrency of hardware circuits. FPGAs are well known for their flexibility and configurability [1] and are therefore attractive candidates for hardware accelerators.

Graph based methods are powerful tools to analyze data dependencies and explore optimization strategies. We propose to use these tools in a compiler that automatically generates hardware accelerators from software. The work described in this paper uses a special kind of compiler for hardware acceleration. Rather than using hardware description languages such as C [9] or Scheme [3], the MIPS-to-Verilog (M2V) compiler [18] translates blocks of MIPS machine code into a hardware design represented in Verilog. In a sense, M2V is a hardware assembler, usable for the last translation phase of any high-level language compiler. M2V can be applied to any reconfigurable platform, e.g. eMIPS [17] from Microsoft Research. We optimized the compiler by taking advantage of both control-flow-based analysis and dataflow-based static scheduling. The optimizations we applied to M2V use only four basic control patterns to assemble arbitrary sets of basic blocks. We tested M2V within an automatic tool chain that generates Verilog from software binaries, using a 64-bit divider, accelerating only three small basic blocks. Simulations demonstrate a factor of 22 in performance improvement for a simple self-looped basic block case, and a 16% performance improvement for an infrequent branch-away case.

II. RELATED WORK

There is a long history of developing techniques for compilers. Compilers [6] usually transform a high-level language to a lower language, and perform various optimizations along the way such as semantic analysis and code optimization. There is also extensive research [7] [8] on synthesizing hardware directly from a high-level behavioral model. Edwards [7] presents techniques to automatically generate Verilog codes from a high-level synchronous language called Esterel. A more direct approach to generate code for an extensible processor is to modify an existing C compiler. Soderman [9] proposes implementing C algorithms in reconfigurable hardware using C2Verilog.

M2V is a hardware compiler that accepts as input binary machine code rather than higher level source code. One of the major advantages is that any application can be accelerated, even applications where the source code is not available to the system developer. A disadvantage is that some of the information that has been discarded must be laboriously reconstructed. The fact that the binary code has already been optimized (register allocation, loop unrolling, etc) during its compilation makes identifying parallelism and other compilation optimization more difficult. The approach is similar to FREEDOM [14] or Cascade [10], but M2V targets a general-purpose environment and therefore must support safe memory access, TLB misses, interrupts and traps.

Graphs are often used to assist Verilog code generation, and dataflow modeling is attracting more attention. Synchronous dataflow (SDF) is streamlined for efficient representation and benefits from static scheduling [11]. In order to increase the expressiveness to a wider range of applications, a variety of dynamic dataflow modeling techniques have been developed, including the Functional DIF [12] and the CAL actor language [13]. Our work on M2V utilizes both control flow representations and dataflow static scheduling by judiciously integrating compiler techniques with architecture-specific insights.

III. M2V COMPILER ARCHITECTURE

The M2V compiler uses as input a decorated and human readable text file, describing a set of basic blocks. Each basic block is a directed acyclic graph (DAG) [6] -- a set of machine instructions that do not contain branches except at the end of the block, and are branched-to only at the very first instruction. The best candidate blocks are those that require a lot of
computation and occur at a high frequency in the application. These candidates are identified by executing the application under the NetBSD operating system, using the Giano full-system simulator [15], in concert with the data obtained via static analysis of the application binary. The M2V compiler is a four-pass compiler, as shown in Figure 1. The first pass builds the control flow graph (CFG) of the input (set of basic blocks). The second pass parses the actual machine instructions, semantically analyzes them, and builds and connects register/instruction nodes in a dataflow graph, which we call a circuit graph (CG), revealing the data dependencies. The third pass of the M2V compiler creates the schedule for register and memory accesses by doing a constrained depth-first traversal of the dependency graph created in the second pass. Based on the dependency graph, the fourth pass generates the synthesizable Verilog. For details on the architecture of M2V, how it interfaces with the main processor, and how it handles traps and interrupts please refer to [16][17][18][19].

![Figure 1: The four passes of the M2V compiler.](image)

The existing M2V compiler only handled the case of a single basic block. To reap the benefits in larger applications we added support for multiple basic blocks, and for the control transfers between them. We decompose the general input graph using the four kinds of basic control patterns shown in Figure 2. Any graph can be decomposed into a combination of these patterns. Control flow analysis makes it possible for each pattern to be considered as a single block and optimized for static scheduling. In the following, a block means a combined block other than basic block if not specified.

![Figure 2: Four kinds of basic control patterns.](image)

The M2V compiler constrains the design of the accelerator such that it does not interfere with the instructions flowing through the pipeline of the main processor, before and after the extended instructions execute. Figure 3 shows a standard five stage RISC pipeline, with IF, ID, EX, MA, and WB stages, and how the (extended) instructions proceed through the pipeline. The main processor recognizes the extended instruction, such as m in Figure 3, and waits for the execution results returned by the extension circuit. Inside the accelerator, the single instruction m is actually a set of basic blocks. In order to distinguish the traditional meaning of “pipeline stage”, here we call state an intermediate stage in which a set of instructions are executing in a fully parallel way, provided only that data dependencies are still met and resource constraints are not violated. Each state may last for several cycles if the cost of some instruction, say a memory load, is more than one cycle. A state may become active multiple times during the execution of one extended instruction, such as in the self-loop cases.

![Figure 3: Interaction between extension and main pipeline.](image)

IV. COMBINATION OF CONTROL FLOW GRAPH AND DATA FLOW SCHEDULING

A. Path Based Static Scheduling

M2V automatically generates the CFG that encodes the relationship among blocks. In a CFG, each node represents a basic block, and a directed edge between nodes represents the ordering between the two basic blocks. An edge between the source node and the sink node indicates the necessary condition for the sink node to be executed. It may be a sufficient condition, such as in the sequential case, and may not be, such as in the branch case.

In the path detection step, we compute different paths based on pre-determined entry blocks and exit blocks. A path in CFG is a sequence of basic blocks from one entry to one exit. We set an upper bound on the total number of paths to schedule, because scheduling too many paths simultaneously will result in inefficiency on execution time. Amongst MIPS instructions, the branch instructions are important indicators of control structure. The result of a branch either determines the program counter (PC) pointing to the instructions outside of the graph, or determines the paths inside the graph. In the current version, we assume there is only entry block but possibly multiple exit blocks. As a result, all the paths share the same entry block. It is straightforward to extend this technique to handle multiple entry blocks by using a multiplexer on the input side to select the appropriate entry block.

In the second step of our scheduling strategy, we apply a single static scheduling tree to the CG expanded from CFG. Figure 4 shows the CG that is generated from the CFG of a branch away case. There are two paths from the entry block to the exit blocks. Each register/instruction node in the CG belongs to either path 1, or path 2, or both paths. In the static scheduling tree, all the paths are scheduled together based on data dependencies and resource constraints. As a result, it is possible to read/write registers and execute instructions from different paths in the same state.

When we combine different basic blocks together, it is possible to further optimize the scheduling strategy by eliminating some “redundant” instructions, in the sense that the instruction is only used to combine two blocks together, such as an unconditional jump. Another kind of code elimination is to remove redundant operation to the same set of registers without any further modification to the output registers.

Code motion impacts execution time since the sequence of code affect compile-time scheduling directly. M2V determines the sequence of code execution for all the paths at compile time. The advantage is that no resource is idle waiting for results from conditional branches; some instructions along the
expected path are already executed before determination of the path. The disadvantage is that executing instructions along unused paths may result in waste of clock cycles. In actual execution, all the paths are executed at the same time until one conditional branch instruction is met, and unexpected paths are discontinued. At the end, only one path is traversed from entry to exit. To avoid the waste of clock cycles as much as possible, M2V applies two kinds of code motions. The first is probability-based path relocation. When we put those paths together, the instructions are listed sequentially. Therefore the path whose instructions are located earlier in the sequence will have a higher priority to be scheduled, since the statically scheduling tree is traversed from top to bottom. In M2V, each path is given a parameter of “statProb”, which represents the statistical probability that the path is executed in practical applications. We arrange the order of the paths in the list based on the value of “statProb”. By default, we assume the “statProb” of each path equals to one another, so we list the paths just as indexed. The second is to move the instructions related to determining conditional branches upward. The movement happens only if it does not affect result of execution on all the paths. Once we determine the exact path to be executed, we can stop executing other paths immediately.

solve this problem, we create different sets of registers to be updated for the different paths. If one updated register is shared by all the write back sets, then this register is committed immediately. If there are conflicts for the same register, we delay the write back until the end of the inner branch, when the exact path is determined.

B. Control-flow Based Self Loop Basic Block

The previous version of M2V [16] depended on the processor’s main pipeline to implement loops. The loop body (e.g. the basic block) was executed only once, and then all resources were returned back to the processor; the processor then re-fetched and re-executed the extended instruction, activating the basic block again and again. This is especially inefficient for self-loops, because the processor has to restart the pipeline when it is in fact known at compile time that the block will just re-activate itself. The timing overhead between the executions of consecutive loops incurs inefficiencies for hardware accelerators.

![Figure 5: State machine of the branch-away case in Figure 4.](image)

After static scheduling, the third step in the compiler is a control-flow analysis that determines which path gets executed. In this step, we generate a flat state machine in Verilog. The state machine for Figure 4 is shown in Figure 5. Both paths share some common states, such as S1, S2, S3 and S6. In this example, path 2 has two distinguishing states S4 and S5.

![Figure 6: Circuit graph for a self-loop basic block. The big arrow represents the looping transition from state 4 back to state 1.](image)

We now realize loops directly in the reconfigurable logic, using the control-flow based approach shown in Figure 6. In this implementation, the accelerator executes all the loop iterations before returning the resources back to the processor. At the end of each loop, the branch condition is calculated. If the target PC points to the beginning of the block, the accelerator will execute the same logic again. The processor is not involved in the execution; all the computation of the (self-) loop is realized in one pipeline stage, simply with more cycles assigned to the EX stage. Intermediate data between loops is kept in the internal registers in the accelerator, except for memory stores which are still retired in program order. The execution time is greatly reduced due to (a) the reduction in communication between processor and accelerator, (b) elimination of all instruction fetches, and (c) avoidance of costly pipeline restarts.

V. SIMULATION AND EVALUATION

For evaluation we used simulations in the combined environment of the Xilinx ISE 10.1 and ModelSim 6.4. The test program is run using Giano to simulate the full system,
and ModelSim to simulate the eMIPS processor with reconfigurable extensions. We selected a 64-bit divider “mmldiv” in video games from Microsoft Xbox applications as an example. The divider is composed of 8 basic blocks, as shown in Figure 7 and discussed in more details in [18]. We simulated two scenarios: with (ACCEL) and without accelerator (NO_ACCEL). In the NO_ACCEL scenario we execute the entire mmldiv software on the eMIPS processor. In the ACCEL scenario we execute just the relevant instructions in hardware, while the rest remains on the eMIPS processor. Two different circuits realize either the branch-away case or the whole self-loop block on the accelerator.

![Control flow graph: example of mmldiv](image)

The test program for the branch-away case is run 60 times with different inputs, and the self-loop case is run 30 times. The average wall time for the simulation across all runs is shown in Table 1. The branch-away case shows only a 16% improvement. There are total of 31 instructions in mmldiv, and only 8 instructions (Blocks 78-80 in Figure 7) account for the branch-away case. The mmldiv example spends most of the time in Blocks 75-76 instead, which are covered by the self-loop case. With an average of 32 repetitions per activation, this block provides a more significant performance improvement; the overall simulation time without acceleration is 22.3 times higher. The previous version of M2V was able to speed up this application only by 2.3x.

Table 1: Average Simulation Time Comparison (Sec)

<table>
<thead>
<tr>
<th>Simulation case</th>
<th>NO ACCEL</th>
<th>ACCEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmldiv (Branch-away)</td>
<td>2773</td>
<td>2337</td>
</tr>
<tr>
<td>Single self-loop block</td>
<td>1027</td>
<td>46</td>
</tr>
</tbody>
</table>

VI. CONCLUSIONS

By taking advantage of both control flow analysis and dataflow scheduling, M2V automatically generates the Verilog code that efficiently implements dynamic control transfers, such as branch, join and self-loop blocks. In the case of a simple loop, we realized a 22x performance improvement, or about 10 times better than the previous version of the compiler. M2V, along with the Giano simulator, provides the solution to automatically generate hardware accelerators from software binaries and hence complements the extensible eMIPS processor platform by easily generating hardware solutions for the reconfigurable logic. M2V can be applied to other reconfigurable platforms simply modifying the boilerplate interface code for the generated Verilog modules.

One possible extension for M2V is to extend the current version for single processor into an advanced version for parallel processors[20].

REFERENCES