

Reconfigurable Architecture Exploration for Speeding Up Execution of Code Generated from High-Level Specifications

December 11, 2001

Frank Gennari and Yatish Patel

Mentors: William Y. Jiang, Luciano Lavagno, and Massimo Baleani

Abstract

Software generated from finite state machines targeted for standard embedded systems processors generally displays poor execution speeds. This paper evaluates an architecture that couples a standard processor with a reconfigurable unit in order to improve the execution speed of the generated code. A method for automatically partitioning the code is presented along with results obtained from simulation and profiling.

1 Introduction

In the software-based design methodologies for embedded systems, systems are designed and programmed in a functional computation model. This can be realized with finite state machines, data flow networks, Petri nets, discrete events, etc., depending on the target application. Software synthesis techniques are used to derive implementation code, like C, from these models.

This project evaluated C code derived from an FSM represented in Esterel. The Esterel was fed into Cadence conversion tools to generate a multi-value BLIF (BLIF-MV) file. The BLIF-MV file was then processed with MVSIS [2] to create C code [5].

The automatically generated code displayed poor performance in terms of execution speed partly due to the control portion of the FSM that runs slowly on normal ALU-based architectures. In addition, multi-step complex data functions require many cycles to compute. The performance of the generated code can be improved by modifying the standard processor architecture.

There exist many variations to the standard processor architecture that could improve the performance of the code. This paper describes two approaches that both involve the addition of a reconfigurable array. One method is to add the reconfigurable array to the datapath of the processor while the other is to add the reconfigurable array as a coprocessor. Advantages

and disadvantages of both methods are discussed in Section 2.

Section 3 describes the enhanced version of GCC [3] used to simulate and profile the generated C code. Section 4 presents some background on MVSIS. Section 5 describes the algorithm used to decide how to group FSM nodes together and Section 6 illustrates some examples that were used to test the system.

2 Architecture

Two main reconfigurable architectures were considered to improve the performance of the generated C code. The first architecture implements a reconfigurable array as another functional unit in the datapath of the processor. The second architecture uses a standard processor coupled with a reconfigurable array as a coprocessor. Each architecture has advantages and disadvantages that effect performance.

A reconfigurable array can be added to the datapath of a standard processor to allow custom instructions to be added to the instruction set. The reconfigurable array can be located parallel to the other functional units such as the ALU. Unused opcodes in the processor's instruction set are assigned to the instructions that run on the reconfigurable array allowing the user to essentially create new instructions. The reconfigurable array obtains input values directly from the register file of the processor. When the array is finished computing the output it is written

back into the register file in a similar fashion to output generated by the ALU. An example of this approach is the Tensilica Xtensa processor.

An alternate approach is to use the reconfigurable array as a coprocessor. Input values for the reconfigurable array are sent from the processor using a communication method such as a bus or memory mapped I/O. The processor instructs the reconfigurable array to execute a specific function and then can either continue executing any code that is not dependent on the reconfigurable array, or it can stall and wait for the response from the reconfigurable array. An example that uses a MIPS processor core with an FPGA on the same die is Garp [6].

Due to the fine granularity of the finite state machine code we evaluated, we decided to target the code for an architecture that has a reconfigurable array as an additional functional unit. Since the reconfigurable array is part of the processor's datapath there is minimal communication overhead to transfer data compared to the coprocessor implementation. The coprocessor implementation requires many cycles to transfer every input to the reconfigurable array and additional cycles to obtain the result. In the best case a cycle is required to write each input to a memory mapped address. Then an additional cycle is necessary to instruct the reconfigurable array to begin processing. Once the array is complete each output will require a cycle to read back into the registers in the processor. If asynchronous operation is desired additional cycles are required to run an interrupt handler to begin reading in the output values.

Some disadvantages to the functional unit approach include the limitation on the number of ports present in the register file, and the requirement that the instructions implemented on the reconfigurable array be synchronous. Due to the limited number of ports on the register file the reconfigurable array is only capable of reading and writing a few values. The reconfigurable array is also required to execute instructions synchronously in order to maintain the flow of data in the processor. This forces only small blocks with few inputs and outputs to be implemented on the reconfigurable array. Fortunately, this works very well with the application space we are evaluating. Finite state machines are generally constructed of

nodes that have only a few inputs and outputs and perform simple calculations.

3 Enhanced GCC

In order to simulate the chosen architecture we used a modified version of GCC. The modified version of GCC targets a simple MIPS R3000 architecture assuming a hardware block is present to implement user-defined instructions (the reconfigurable array). The user tags blocks of C code that should be implemented on the reconfigurable array. The user can then simply specify the number of processor cycles that the instruction should consume along with the number of inputs and outputs required for the instruction.

The tool set also includes a simulator and profiler. After the tags are added to the C code, the simulator will simulate the code using the cycle counts specified by the user for the tagged blocks. The profiler will then return the number of cycles used to execute each line of code.

4 MVSIS

In order to tag blocks of code that should be implemented on the FPGA, MVSIS needed several modifications. MVSIS performs the conversion of BLIF-MV to C by reading a BLIF-MV input file, building a network, and then generating the C file upon user request through the `gen_c` command. Once the network has been built, MVSIS provides a wide variety of operations that can be performed on it such as Boolean optimization with `espresso` or minimization with `scripts`. The network can also be written out to any file format supported in MVSIS. User-defined functions or packages can be created to complement the internal MVSIS functions. We have modified two existing packages to achieve the functionality necessary to evaluate our FPGA-based architecture. The first is the `codegen` package, written by William Jiang to convert an MVSIS network into a C file with the use of Multi-Valued Decision Diagrams (MDDs). The second is the `club` package, written by Sunil Khatri to group the nodes in the network for mapping onto a PLA. The modifications we made to the `club` package are discussed in the next section.

MVSIS stores a design as a flattened network of nodes representing combinational and sequential logic. The MVSIS node has a variety of fields, including name, type, fanin and fanout pointers,

and an expression or cover. Valid node types are primary input (PI), primary output (PO), latch, constant, control, data, predicate, and mux. Latch nodes are built directly into the network and represent the sequential parts of the design, while all other nodes are combinational. MVSIS stores these nodes in a hash table so that they can be referenced by name as well as by using the fanin and fanout pointers. MVSIS also provides common algorithms and data structures along with network-specific function calls for processing the network.

Internal node types can be classified into either control or data functions, which could possibly be constants. Control nodes have multi-valued (MV) inputs and outputs and include such functions as Boolean operations and MV if/else statements. These nodes determine how data is routed through, used in, and produced by the design. Data nodes consist of predicates, muxes, or data expressions, and in this case operate on 32-bit integers. Note that in the following descriptions the data input may in fact be a 32-bit integer constant node. Predicate nodes have one or more data inputs, a single MV (usually binary) output, and perform a comparison operation on the inputs such as equality or inequalities. Mux nodes contain two or more data inputs, a MV select input, and one data output. As in combinational logic muxes, these mux nodes propagate the value of the individual data input selected by the control value to the data output. Data expressions have any number of data inputs and one data output and can include equations, hierarchical function calls, math library calls, or anything that doesn't fit into the previous two data node categories.

5 Clubbing

We define clubbing as the process of grouping well-connected nodes of compatible types into a supernode with a specified maximum input and output (I/O) count. A club can only be targeted for the reconfigurable array if the entire club meets the I/O constraints of the target architecture and does not exceed the area limitations of the reconfigurable array. We place several constraints on the clubbing algorithm so as to make the addition of reconfigurable array designators more straightforward. A node cannot be added to a club if:

1. It is a PI or latch node.
2. It is a control(data) node while the club consists of data(control) nodes.
3. The resulting club exceeds a predetermined maximum number of inputs and outputs.
4. The addition of the node introduces latchless circular I/O dependencies within the club.

Since PI and latch nodes are not implemented on the reconfigurable array, it is incorrect to add them to a club of other node types and a waste of computation time to club them together with similar types, thus condition (1). Furthermore, adding latch nodes to a club may violate condition (4). Condition (2) is introduced to keep control and data nodes in separate clubs so that we can independently evaluate the effect of implementing the control and the data portions of the design on the reconfigurable array. Condition (3) is necessary because the reconfigurable array instruction contains a limited number of I/O location fields and the architecture has a limited amount of hardware to support information flow between it and the processor. If a club exceeds this I/O count, then it cannot possibly be implemented as an instruction on the reconfigurable array. In the case of the MIPS R3000 architecture used in the modified version of GCC, the limit is three inputs and two outputs. Condition (4) guarantees that the design is deterministic and the resulting C code does not try to use variables before they are assigned correct values. A club violating condition (4) could result in an incorrect C implementation of the design or even the use of uninitialized variables. We guarantee that condition (4) is satisfied by adding nodes to a club in the order of a topological traversal of the network from the PIs to the POs. The network is assumed to be acyclic such that latch nodes, which are not added to the clubs, break all of the cycles. There is no equivalent C file for a cyclic combinational network.

Sunil Khatri has written a clubbing algorithm that matches the above definition of clubbing but does not satisfy our clubbing constraints. The simplest version of Sunil's clubbing code was intended to group nodes for mapping onto a PLA, so his algorithm assumes a different set of constraints. We have rewritten most of the code to make it compatible with the codegen algorithm, including the levellizing algorithm, the process of

adding nodes to a club, and the steps involved in producing the final club data structure.

The network is levlized before clubbing so as to satisfy condition (4). The nodes in the network are iterated through, and each PI, latch, or constant (0 fanin) node is both added to the end of the “clubbing” array and inserted into a “node_added” hashtable. Each time a node is added to the array a recursive function call is made on each of the node’s fanouts. In the recursion, if all of a node’s fanins have previously been added to the hashtable, this node is also added to the clubbing array and inserted into the hashtable. In this way the final topologically ordered clubbing array contains every node in the network such that each non-latch node i does not depend on (fanout from) any non-latch node j , for all $i < j$.

The main clubbing loop iterates through this clubbing array in order, maintaining a current club as an array of pointers to nodes in the club. At each step, if the addition of a node to the current club violates one of conditions 1-4, then a new club is begun and that node is inserted into the new club. If the node violating the condition is a PI or latch, it is inserted into its own single-node club and “written” to the final array of clubs, allowing the current club to continue building in the next step. If the node is not a PI or latch, then the current club is written to the club array and the new club becomes the current club.

Inputs to and outputs from the current club plus one potential member node must be counted at each step while neglecting internal node connections. This is accomplished by inserting each club member node into a pointer-based hashtable and iterating through the fanins and fanouts of each node in the club (plus the potential member) in two separate loops. For each node and each fanout, if the fanout node is not in the hashtable, then it is outside of the club. If any of the fanouts of a node exit the club, then the club output counter is incremented. For each fanin of every node, if that fanin is (a) not in the hashtable and (b) not constant (0 fanins and not a PI), then the club input counter is incremented. The fanin node is also added to the hashtable so that it is not counted more than once in the case that the same node is a fanin to multiple members of the club. The final values of the input and output counters are used to determine if the addition of the new club will exceed the I/O limitations of the FPGA

instruction. If a single node exceeds these limitations then it cannot be clubbed and cannot be implemented on the reconfigurable array.

When all of the nodes have been processed and all clubs have been written to the club array, the other data structures are freed and the club array is returned for use in the codegen algorithm. Clubbing can be performed on any legal MVSIS network and does not alter the network data structure or any node fields.

When all candidate clubs are generated a first run simulation is run through the modified version of GCC. The cycle counts returned from the profiler are fed back into the code generation and used to tag the clubs that will best improve performance of the generated C code.

We assume that the small nodes we implement on the reconfigurable array will take a minimal amount of time to execute. We use a simple heuristic that employs the number of inputs and number of outputs to determine the number of cycles the instruction will require.

6 Results

In order to demonstrate our clubbing algorithm, we chose to use an FSM that implemented a multi-injection driver for engine control systems [7]. In addition to the engine control system we also used two additional automobile control system examples. All three examples contained both control and data portions in the FSM.

	Clubs	Estimated LUTs	Cycle Count
No FPGA	0	0	5,263,012
All Control	122	542	4,682,820
All Data	99	3304	2,169,596
Mix	221	3846	1,530,823

Table 1 - Injection Driver

	Clubs	Estimated LUTs	Cycle Count
No FPGA	0	0	452,268
All Control	18	18	346,882
All Data	58	2100	306,641
Mix	76	2118	218,146

Table 2 - Cross Display

	Clubs	Estimated LUTs	Cycle Count
No FPGA	0	0	923,384
All Control	75	128	536,729
All Data	76	2596	736,794
Mix	151	2724	406,695

Table 3 - Shock Dance

The tables show the number of cycles required to execute the FSM, depending on which clubs were implemented on the reconfigurable array. Four variations were tried. The first implemented all of the C code on the processor. The second targeted all control nodes to the reconfigurable array, and the third targeted all data nodes to the array, while the fourth option targeted all possible clubs to the array.

A simple algorithm was used to estimate the number of LUTS (reconfigurable array units) the clubs would require if they were implemented in hardware. Node patterns that commonly appeared were synthesized using Xilinx logic synthesis tools to determine the LUT counts. Some examples of the commonly occurring node patterns include multiplexors and comparators. These values were then used in the code generation algorithm to provide a rough estimation of the required LUTs. Any unidentifiable node types such as function calls were estimated manually. Since control nodes have a fine granularity, many of them could be implemented with very few LUTS and low delay. However, data nodes require more complicated arithmetic operations and present a challenge in LUT and delay estimation. The number of LUTS required and delay is very closely related to the type of reconfigurable array that is used.

The results show that generally implementing the data nodes on the reconfigurable array benefits the performance the most. However the data nodes require a significantly higher number of LUTs. In certain cases such as Shock Dance the delay incurred by the complex data nodes prevented the decreased cycle count from being realized.

7 Future Work

The architecture evaluated was targeted for an FSM that required multi-valued control variables. This required us to use the full bit width of the registers that fed the reconfigurable array and limited us to three inputs and two outputs. However in many of the designs we evaluated, a

majority of the nodes only required a few bits to represent the largest value. A possible improvement to our algorithm could bit-pack multiple control variables into a single integer width allowing us to implement more complex instructions on the reconfigurable array.

In addition, our algorithm for clubbing used a very simplistic estimation of the area and delay required for each club that was implemented on the reconfigurable array. Due to the inaccurate area estimations, the full potential of our clubbing algorithm was not realized. The algorithm is capable of constraining the number of instructions implemented on the reconfigurable array by limiting the area available for the instruction logic. An improved method of generating the required area of an instruction would allow us to more efficiently decide which instructions should be created to optimally benefit performance.

8 Conclusion

By inserting a reconfigurable array into the datapath of a standard processor, the performance of finite state machines implemented in C can be significantly improved. Both control and data nodes can be targeted to the reconfigurable array, drastically reducing the number of instructions required for the execution of the code.

The data nodes seem to benefit most from the use of the reconfigurable array; however, tradeoffs do exist. The more complicated data nodes require a significantly larger amount of area on the reconfigurable array compared to the simpler control nodes.

References

- [1] Tsutomu Sasao, Munehiro Matsuura, and Yukihiro Iguchi, "A Cascade Realization of Multiple-Output Function for Reconfigurable Hardware", *IWLS*, May 2001.
- [2] M. Gao, J.-H. Jiang, Y. Jiang, Y. Li, S. Sinha, and R. K. Brayton, "MVSIS," *IWLS*, May 2001.
- [3] Alberto La Rosa, Luciano Lavagno and Claudio Passerone, "A Software Development Tool Chain for a Reconfigurable Processor," *CASES*, Nov. 2001.
- [4] "MVSIS Programming Guide"

- [5] Y. Jiang and R. K. Brayton, "Logic Optimization and Code Generation for Embedded Control Applications," in *Proc. of the Intl. Symposium on Hardware/Software Codesign*, Apr. 2001.
- [6] J. Hauser, and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 1997.
- [7] A. Nardi, and F. Mo, "HW/SW Co-Design of a Multi-Injection Driver for Engine Control Systems," EE249 Project, Fall 1999.