Early Performance-Cost Estimation of Application-Specific Data Path Pipelining

Jelena Trajkovic
Computer Science Department
École Polytechnique de Montréal, Canada
Email: jelena.trajkovic@polymtl.ca

Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine, USA
Email: gajski@uci.edu

***Abstract*—Application-specific processors (ASPs) are increasingly being adopted for optimized implementation of embedded systems. ASP design automation tools are, therefore, critical for meeting the time-to-market goals for ASP-based embedded systems. This paper targets the problem of determining the optimal data path pipeline configuration from a given application C code. We propose a technique for automatically estimating the application execution time on an ASP for various data path pipeline configurations based on estimated clock cycle length and estimated number of cycles. In addition, we compute the cost of each pipelined design, thereby characterizing the ASP by its performance and cost. Our estimation enables fast, accurate and early analysis of trade-offs between different data path pipeline configurations, without the need for creating either a prototype or a cycle-accurate model of the ASP. Our experimental results, based on industrial applications, demonstrate high fidelity for the performance estimation.**

I. Introduction

Pipelining improves performance by increasing the throughput of instructions and reducing the clock cycle time. However, there are practical constraints ([1]) that prevent such improvement:

1) The data path usually can not be perfectly divided into n stages of equal length; therefore the length of the longest stage determines the clock cycle length.
2) Pipeline latches/registers add overhead to the cycle time (latch/register setup time).
3) Data dependent, control dependent or resource dependent instructions can not be overlapped. Therefore the dependent instruction needs to wait (*stall*) until the previous one finishes execution.

We take the factors 1 and 2 into account while computing the cycle length, and we focus on effects of the factor 3 on the execution time (in number of cycles) for a particular pipelined configuration. Our algorithms operate on three-address code representation derived from C application code to determine the desired pipeline configuration. We start from predefined data path ([2], [3]) and propose following steps for various pipelined configurations of the input data path:

- Estimate cycle length;
- Estimate the number of execution cycles;
- Compute execution time and cost for each created pipelined configuration.

This work was done while Jelena Trajkovic was at the Center for Embedded Computer Systems, University of California, Irvine.

The designer may select a few optimal pipeline configurations, simulate and synthesize them in order to find the best one. The designer chooses from significantly smaller number of optimal configurations instead of traversing a very large design space of all possible pipeline configurations.

Application Specific Instruction-set Processors (ASIP) and Instruction Set (IS) extension processors, such as Xtensa [4] and Stretch processor [5], allow the designer to configure processor features, including pipeline. In both cases, the designer selects the pipeline configuration based of the description of the pipeline behavior (produced by tools, such as Xtensa Pipeline Viewer). On the contrary, our technique automatically produces performance and cost metrics for all configurations, allowing for early trade-off analysis. Also, our technique is complementary to use of custom IS extensions and HW accelerators.

C-to-RTL tools, such as Catapult Synthesis [6], generate the data path 'on the fly' while inserting the (pipeline) registers in places where those are needed to store a value. To the best of our knowledge, during the data path generation, there is no analysis if the generated pipeline configuration is pareto-optimal. Moreover, those tools are applicable to fairly small code size (480 lines of C code [7]). Our technique can handle any size of C code (10K lines of C code) making it valuable assets in pipelined data path generation.

II. Selection of Candidate Configurations

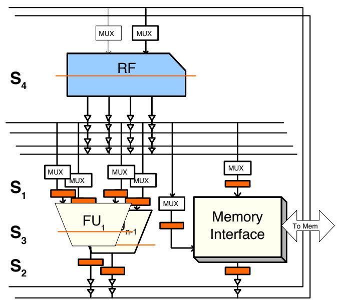


Fig. 1. Different Pipeline Confi guration Under Consideration

The pipeline estimation takes in the non-pipelined (optimized) data path and divides it by inserting the pipeline stages, creating several different pipeline configurations. In this work we consider uniformly pipelined configurations, i.e. the configurations in which inputs/outputs of all components have the same number of pipeline registers attached to them, such as seen in Fig. 1. In general, it is possible to insert n pipeline stages in a data path. Any such configuration may be described as a set $(S_1, S_2, ...S_n)$ where, S_i is a bit indicating existence of registers in pipeline level i. All pipeline configurations may be explained using Fig. 1 ($n = 4$): $S_1 = 1$ represents pipeline registers connected to the inputs of functional and memory interface units, $S_2 = 1$ pipeline registers connected to the outputs of corresponding units, $S_3 = 1$ corresponds to pipelined functional units and memory interfaces, and $S_4 = 1$ corresponds to pipelined register file. The considered configurations are: non-pipeline (0,0,0,0), input pipeline (1,0,0,0), output pipeline(0,1,0,0), two stage pipelining (1,1,0,0), three stage pipelining (1,1,1,0) and four stage pipelining (1,1,1,1). The goal of this work is to find the values of n and $(S_1, S_2, ...S_n)$ that deliver the best execution time for the given C code.

III. Clock Cycle Length Estimation

For any pipeline configuration described with (S_1, S_2, S_3, S_4) the clock cycle length is determined by the value of a discrete function $T_{clk}(S_1, S_2, S_3, S_4)$ given by the Equation 1.

$$T_{clk}(S_1, S_2, S_3, S_4) = max\{T_{clk1}, T_{clk2}, T_{clk3}, T_{clk4}, T_{clk5}\} \quad (1)$$

Variables T_{clk1}, T_{clk2}, T_{clk3}, T_{clk4} and T_{clk5} correspond to possible cycle lengths considered for our data path. Due to the limited space, we present here only the equation for T_{clk1}:

$$\begin{aligned} T_{clk1} = {} & (c_0 \vee c_1 \vee c_2 \vee c_3 \vee c_4)\} \cdot [max\{RF.prop\_delay\} \\ & + max\{interconnect.src\}\} \\ & + (c_1 \vee c_3 \vee c_4) \cdot (pipe\_reg(T_1).setup\_time) \\ & + (c_0 \vee c_2) \cdot [max\{(FU_k \vee MemI_k) | k = 1, 2, ...q\} \\ & + (interconnect.dst)] \\ & + c_0 \cdot max\{RF.setup\_time\} \\ & + c_2 \cdot (pipe\_reg(T_2).setup\_time) \\ & + c_5 \cdot max\{RF(stage\_1).prop\_delay\} \end{aligned} \quad (2)$$

where binary coefficients c_0, c_1, ... c_5 are function of S_1, S_2, S_3, and S_4 and represent each pipeline configuration and the remaining values are: *RF.prop_delay* - maximum time between an active clock edge to the valid output for selected register file; *interconnect.src* or *interconnect.dst* - propagation delay from latched output via *source* or *destination* interconnect to the following latched input; $pipe\_reg.(T_i).setup\_time$, *RF.setup_time* - *input arrival time*; FU_k, $MemI_k$ - propagation delay for functional units and memory interface, respectively; $FU_k(stage\_i)$, $MemI_k(stage\_i)$ or $RF(stage\_i).prop\_delay$ - propagation delay for stage i of pipelined functional units, memory interface and register file, respectively.

IV. Estimation of the Number of Execution Cycles for a Basic Block

In this section we present the algorithms for estimating the number of execution cycles. In order to do so, we make following assumptions for a data path model with n pipeline stages.

A1 Memory access is done only via special (load/store) operations, i.e. we use load-store architecture model;

A2 Reading a source operand from the register file is performed in the second half of the first stage of the pipeline (results from A1);

A3 Writing a destination operand is performed in the last pipeline stage (also results from A1), during the first half of the clock cycle;

A4 There are no forwarding paths;

A5 There is no support for hardware dynamic scheduling.

These assumptions result in the following conclusions:

C1 From A2 and A3, the only possible dependency is true data dependency. True data dependency, also known as 'read-after-write', is when an operation o_a generates a source for a subsequent operation o_b.

C2 From A4 and A5 true data dependency will cause a *stall* in the pipeline *if* an operation o_a appears in any of the $(n-1)$ cycles before o_b. Therefore, we need to examine a 'window' of all the operations in previous $(n-1)$ cycles.

Since we aim to exploit all available parallelism, we also consider having multiple operations per cycles wherever the available resources allow. If w is the maximum number of operations per cycle, the 'window' that we observe contains up to $w \cdot (n-1)$ operations.

A. Data Path Model (DPM)

The *DP Model (DPM)* consists of: pipeline configuration type and reservation station (*RS*) for each functional units, register files, memory interface and ports. The common fields in RS for all the types are:

- *Delay* - time required from the moment the operands are available on the input ports till the moment when a valid result is available on the output ports of the component;
- *Tti* (time-to-issue) - time between start of the operation o_a and earliest start of a consecutive operation o_b; value for this parameter is specified only for *pipelined* components;
- *FreeCycle* - a cycle number in which a component may start the next operation; for non pipelined components: it is equal to a sum of a starting cycle and a delay; and for pipelined components: it is a sum of a starting cycle and a time-to-issue.

The type-specific RS fields are:

- For functional unit RS: *Operations* - a list of all the operations that the unit performs;
- For register file RS: *Type* - register file (RF) type form the Component Library (e.g. RF-4x2); *InPort* - a list of input ports; *OutPort* - a list of output ports;
- For memory interface RS: *Type* - same as for RF; *DelayL* and *DelayS* - delay for load and store operations; *TtiL* and *TtiS* - time-to-issue for load and store operation, respectively; *AddPort*, *InPort* and *OutPort* - lists of address, input and output ports;

- For Port RS: *Type* - port type form the Component Library; does not have *Delay* and *Tti* field, but inherits the appropriate values from the RS for the component that the port belongs to.

B. Estimation Algorithm

After the data structure for DPM has been created and initialized, we create a priority ready list (*PR list*), of all operations. Each node of a PR list consists of an opcode; a pointer for each operand that points to a node that generates its value; mobility computed as a difference between values in ASAP ('as soon as possible') and ALAP ('as late as possible') schedules; execution time of the slowest component executing the operation; and the number of operations that use the result. Operations are inserted into the PR list according to the descending order of priority:

1) Mobility - a node with a *smaller* mobility has greater priority.
2) Execution time - a node with a *longer* execution time has greater priority.
3) Number of users - a node with a *larger* number of users has greater priority.

We define a function *FindPath* that takes as an input the operation *op*, its earliest start time, *DPM* and *PR List*. It searches for all the data path components C_j available in *DPM* that are required for execution of the operation *op*. The RS for the selected component is updated with the operation that uses it, the start time of the operation and the next time that the component becomes available. This function returns a list of all components reserved *CompList* for the operation *op*,and corresponding start and end execution time, for each component.

Algorithm 1 ExCyEst(DPM, PR List)

```
out: $Estimated\ Schedule$
for all ($node\ K \in PR\ List$) do
  for all ($i = 1, max\ inputs$) do
    $t_{K_i} = cycle\ where\ operand\ i\ is\ available$
    if $any\ K.op \in (previous\ n-1\ cycles)$ then
      insert stall
    end if
    $t_{Kstart} = max\{t_{K_i}\}$
  end for
  $CompList = FindPath(K.op,\ t_{Kstart},\ DPM,\ PR\ List)$
  for all ($C \in CompList$) do
    create a node in $Estimated\ Schedule$
    for all ($cycle = t_{Kstart}, t_{Kend}$) do
      insert($K.op$, C)
    end for
  end for
  $remove\ node\ K\ form\ PR\ List$
end for
```

The execution cycle estimation function *ExCyEst*, based on list scheduling algorithm [8], is outlined in the Algorithm 1. For each node in PR list, we first determine earliest starting time (t_{Kstart}) based on the availability of its operands. If an operand value is produced by an operation within previous $(n-1)$ cycles, we need to wait until its value is available. This results in insertion of stall cycles in the *Estimated Schedule*. Then the *FindPath* function is called to find a path from a registered output to a registered input and reserve the components on the path. The function returns a list of reserved components and a corresponding starting time (t_{Kstart}) and an ending time (t_{Kend}). For each cycle from t_{Kstart} to t_{Kend} and for each component we create a new node in the Estimated Schedule and populate it with the name and identifier for both component and the operation. Node for each processed operation is finally removed from the *PRlist*.

V. Execution Time and Cost Computation

The estimated total execution time ET_{exe} for a configuration $(S_1, S_2, ...S_n)$ is given by:

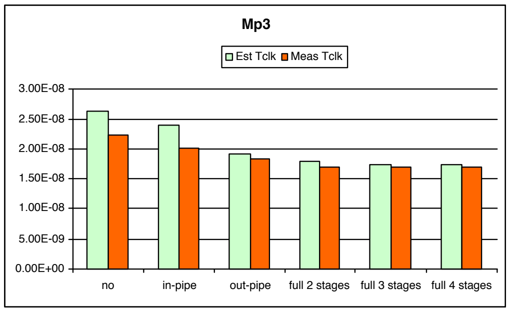
$$ET_{exe}(S_1,S_2,...S_n) = T_{clk}(S_1,S_2,...S_n) \cdot N_{exec}(S_1,S_2,...S_n)$$
$$N_{exec}(S_1,S_2,...S_n) = \sum_{BasicBlock} (f_i \cdot N_{Block_i}(S_1,S_2,...S_n)) \qquad (3)$$

where $T_{clk}(S_1,S_2,...S_n)$ is the estimated clock cycle length, $N_{exec}(S_1,S_2,...S_n)$ is the estimated number of execution cycles, f_i is frequency for the basic block $Block_i$ (obtained by application profiling) and $N_{Block_i}(S_1,S_2,...S_n)$ is a number of cycles for the block. We define a cost function C_{design}, simply as a sum of slices and a sum of RAMs for all the data path components.

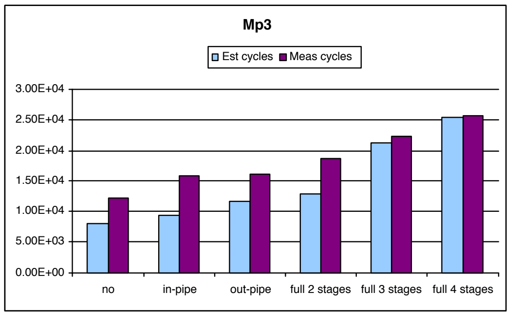
VI. Experimental Results

We implemented the proposed technique for performance and cost estimation in C++. All pipeline configurations use programmable controller unit implementation, and we used the NISC tool-set to generate schedule and Verilog RTL model of the processor [9]. For synthesis and simulation of all configurations, as well as for synthesis of components in the Component Library, we used Xilinx ISE 8.1i and ModelSim SE 6.2g. The target implementation device was a Xilinx Virtex II FPGA xc2v2000 in FF896 package with speed grade -6. We used a reference implementation of *Mp3* decoder, with 13898 lines of code, as our test applications. The initial data path for *Mp3* consists of a register file with three input and four output ports, with 128 registers, 2 ALUs and one multiplier, comparator and divider each. The initial data path is obtained using standard allocation algorithms ([2], [3]). Total run-time for the estimation tool was 25 minutes.

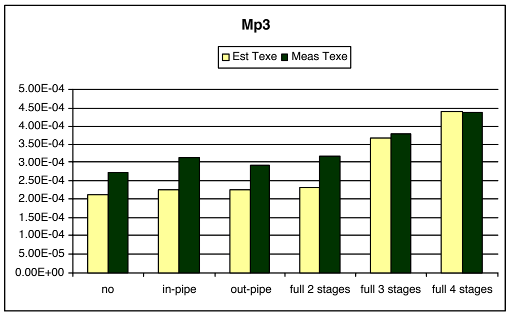
Fig. 2 shows estimated and measured (synthesized) clock cycle length (T_{clk}), number of cycles, execution time (T_{exe}) and estimated cost for *Mp3*. Estimated cost of pipelined configuration is shown as *relative* to the cost of the non-pipelined configuration. The presented pipelined configurations are annotated, on each x-axis, in all the graphs with *no*, *in-pipe*, *out-pipe*, *full 2 stage*, *full 3 stage*, *full 4 stage*, corresponding to the non-pipeline, input pipeline, output pipeline, two stage pipelining, three stage pipelining and four stage pipelining, respectively.



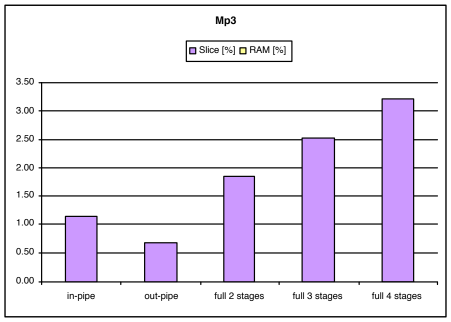
(a) $Tclk$



(b) Number of Cycles



(c) T_{exe}



(d) Estimated Cost

Fig. 2. Mp3 - Estimated and Measured (synthesized) Clock Cycle Length (T_{clk}), Number of Cycles, Execution Time (T_{exe}) and Estimated Cost

As it may be seen in the Fig. 2a, our algorithm uniformly overestimates the clock cycle length. On the other hand, the number of cycles (Fig. 2b) is slightly underestimated, but the rate of increase in estimated number of cycles follows the rate of increase in the simulated number of cycles. Consequently, the first three configurations, namely *in-pipe*, *out-pipe* and *full 2 stage*, have noticeably smaller estimated execution time than the synthesizes/simulated one (Fig. 2c). Still, the total execution time for all pipelined configurations is larger than for the initial, non-pipelined one. There are two reasons behind this; firstly, the code was originally written for a general purpose processor, with many sequential functions. Secondly, having the register file with three input and six output ports, two ALUs, one multiplier, one divider and a comparator enable all the available parallelism to be fully exploited. Cost increase for any pipelined configuration is less than 3.5% as seen in the Fig. 2d.

For a larger set of applications [10], the absolute error for the total execution time T_{exe} is 20%, on an average, and 30% maximum. However, the fidelity of the estimate is much higher. Experiments on a set of five applications show that the fidelity of the method is 94%.

VII. Conclusions

This paper presents a technique for estimating the application execution time and ASP pipeline configuration cost from C code. The fidelity of our estimation of the total execution time is 94% on average. Therefore, the designer may perform time-consuming synthesis and RTL simulation for only a few selected configurations. In future, we plan to support more pipeline configurations, including the non-uniform pipelining, and investigate methods for automatic pipeline optimization. We will also add support for function pointers, global pointers and standard libraries in C specification.

A. Acknowledgment

We would like to thank Gabriela Nicolescu for her constructive feedback on this paper.

References

[1] J. Hennessy and D. Patterson, *Computer Architecture A Quantitative Approach*. Third Edition, The Morgan Kaufmann Series in Computer Architecture and Design, 2006.
[2] P. Marwedel, “The MIMOLA system: Detailed description of the system software,” in *Proceedings of Design Automation Conference*. ACM/IEEE, Jun. 1993.
[3] J. Trajkovic and D. Gajski, “Custom processor core construction from c code,” in *Proceedings of* The Sixth IEEE Symposium on Application Specifi c Processors (SASP), 2008.
[4] 2005, Tensilica: Xtensa LX http://www.tensilica.com/products/xtensa_LX.htm.
[5] 2005, stretch. Inc.: S5000 Software-Confi gurable Processors http://www.stretchinc.com/products/ devices.php.
[6] 2008, Mentor Graphics Catapult Synthesis http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/index.cfm.
[7] 2008, Mentor Graphics Technical Publications http://www.mentor.com/training_and_services/tech_pubs.cfm.
[8] A. Sllame and V. Drabek, “An effi cient list-based scheduling algorithm for high-level synthesis,” in *Proceedings of* Euromicro Symposium on Digital System Design, 2002.
[9] NISC Technology - NISC Toolset Online Demo, 2009. http://www.ics.uci.edu/ nisc/demo/.
[10] J. Trajkovic, “Automatic design and optimization of processor data path and memory hierarchy,” Ph.D. dissertation, University of California, Irvine, 2009.