The Impact of Loop Unrolling on Controller Delay in High Level Synthesis

Srikanth Kurra
Dept. of Comp. Sc. & Engg.
Indian Institute of Technology
New Delhi 110016
srikanth@cse.iitd.ac.in

Neeraj Kumar Singh
Intel Tech. India Pvt. Ltd.
136 Airport Road
Bangalore 560017
neeraj.k.singh@intel.com

Preeti Ranjan Panda
Dept. of Comp. Sc. & Engg.
Indian Institute of Technology
New Delhi 110016
panda@cse.iitd.ac.in

Abstract

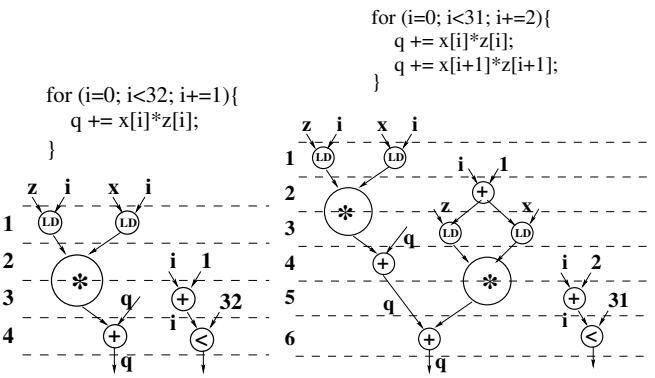
Loop unrolling is a well-known compiler optimization that can lead to significant performance improvements. When used in High Level Synthesis (HLS) unrolling can affect the controller complexity and delay. We study the effect of the loop unrolling factor on the delay of controllers generated during HLS. We propose a technique to predict controller delay as a function of the loop unrolling factor, and use this prediction with other search space pruning methods to automatically determine the optimal loop unrolling factor that results in a controller whose delay fits into a specified time budget, without an exhaustive exploration. Experimental results indicate delay predictions that are close to measured delays, yet significantly faster than exhaustive synthesis.

1 Introduction

In the early stages of the Behavioral (or High Level) Synthesis process, a number of high-level optimizations and transformations are applied on the behavioral description, many of them similar to typical compiler optimizations. A large number of well-known compiler optimizations have been adopted in HLS in the past. However, some of them have non-trivial effects and the ultimate result may be unexpected. In this paper we address an important transformation - *loop unrolling* - in the context of high level synthesis and make the following contributions:

- We study the effect of the loop unrolling factor on the delay of controllers generated during HLS.
- We propose a technique to predict controller delay as a function of the loop unrolling factor.
- We use the above prediction and other search space pruning methodologies to automatically determine the optimal loop unrolling factor that results in a controller whose delay fits into a specified time budget.

The loop unrolling transformation duplicates the body of the loop multiple times to expose additional parallelism that may be available across loop iterations. An example of loop unrolling is shown in Figure 1. Figure 1(a) shows the scheduled Data Flow Graph (DFG) of the original loop body with



(a) Original Loop (b) Original loop unrolled once

Figure 1. Loop Unrolling example

the resource constraints: 1 adder, 1 multiplier, 1 comparator, 2 load units, with the multiplier taking 2 cycles and the others being single-cycled. The load units perform both the memory address calculations and memory read operations, and are scheduled together if simultaneous memory accesses are possible. The loop requires approximately $32 \times 4 = 128$ clock cycles, ignoring the initialization part. When the loop is unrolled once (unroll factor = 2), the transformed code is shown in Figure 1(b) along with scheduled DFG with the same resource constraints. The new loop executes in roughly $16 \times 6 = 96$ clock cycles. There is a performance improvement of 25% by unrolling the loop once.

However, unrolling also leads to a code size increase and cache pollution, which indirectly affects performance adversely. In the context of behavioral synthesis also, there are non-trivial implications which make unrolling decisions not so straightforward. The DFG size increases in synthesis, which leads to indirect effects that ultimately influence both the performance of the resulting designs as well as correctness of the generated schedules if the effects are not properly accounted for.

Figure 2 shows a typical output of behavioral synthesis. In such circuits, the critical timing path passes through the controller, the MUXes at the inputs of the function units, the function units themselves, and the MUXes before registers.

978-3-9810801-2-4/DATE07 © 2007 EDAA

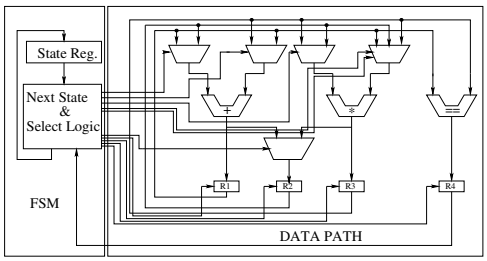


Figure 2. Behavioral synthesis output

Thus, the clock period needs to be larger than the sum of the delays through these components. Loop unrolling causes an increase in the critical path delay. First, because a larger number of nodes need to be scheduled, the controller has to perform more work, leading to more complex combinational logic in the Finite State Machine (FSM) which leads to longer delays. The number of FSM states increases leading to a larger state register and ultimately to longer FSM delays. For example, in Figure 1, the unrolled loop schedule has 6 states in comparison to 4 states in the original schedule. Further, additional operations after unrolling lead to increased sharing of resources, possibly increasing the sizes of the MUXes and hence, their delay. If the delays through the controller and MUXes are not accounted for, then the schedule resulting from high-level synthesis can easily be invalid; it would be incorrect to ignore these delays when performing scheduling.

In this paper, we use an estimation-based approach to predict the optimal unrolling factor of loops while respecting an overall FSM delay budget. Our strategy involves a search space pruning mechanism for limiting the number of unroll factors to be considered, followed by a priority function-based algorithm for exploring the loop unroll factor.

2 Related Work

Loop unrolling is a relatively well studied transformation in the compiler domain [1–3]. The topic of automatically selecting optimal loop unroll factors was addressed by [1, 2]. In [3] the authors give an algorithm for determining the number of times and the directions in which loops should be unrolled through the use of information such as dependence, reuse, and machine resources. Researchers have also evaluated the performance and power effects of transformations such as unrolling and inlining in the context of embedded processors [4, 5]. While the above analyses are relevant, the loop unrolling problem in the context of high-level synthesis is quite different, as motivated in Section 1 due to the absence of an instruction cache/memory and absence of a fixed-size register file.

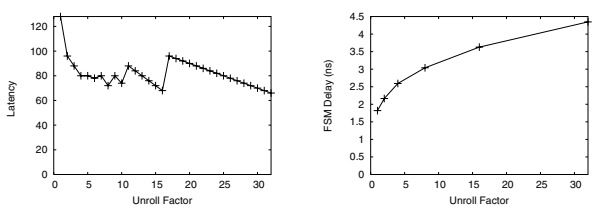
A large number of optimizations that improve timing have been identified for high-level synthesis – some of them adapted from the compiler domain, and many being hardware-specific, for example, variants of common sub-expression elimination [6, 7, 10], retiming [8], speculation [9, 10], loop shifting/compaction [13] and bit-level optimizations [11, 12]. While the authors in [13] do observe a negative impact on timing due to unrolling in their experiments, the loop unrolling factor itself is considered to be manually specified.

[14] presents a timing model for high-level synthesis consisting of a timing network for datapath and control sections. Timing analysis is included as part of the synthesis procedure to ensure that the resulting designs meet timing. [15] presents a behavioral network graph model in which HLS optimisations are evaluated by examining logic-level effects. Timing closure at the behavioral level has been addressed through introducing delay relaxation for functional units [16].

A formulation for modeling and estimation of controller delay in terms of the number of operations scheduled, number of control steps, and function unit control bits was presented in [17]. In this paper, we use this formulation and present a technique for quantifying controller delay in HLS due to various loop unrolling factors and suggest a methodology for deriving an optimal unrolling factor while respecting controller delay budget constraints.

3 Loop Unrolling and Controller Delay

Figure 3(a) shows the latency of the behaviour shown in Figure 1(a) with the same resource constraints as before. The first observation is that the performance does not monotonically increase with the unroll factor. This is because the unroll factor may not evenly divide the iteration count resulting in a trailer loop in which the remaining iterations are sequentially executed, which affects the overall latency negatively. Figure 4(a) shows the code when the unroll factor 16 evenly divides the iteration count 32 and Figure 4(b) shows the code when unroll factor 17 does not, resulting in a trailer loop. Figure 3(a) shows that the latency due to unroll factor 17 is significantly worse (greater by 41%) than the latency due to unroll factor 16.



(a) Latency Variation (b) FSM Delay Variation

Figure 3. Variation of latency and FSM delay with unroll factor

In Figure 3(b) we plot the FSM delay against a subset of the unroll factors. The delay is observed to be an increasing function of the unroll factors. In fact, as in latency, the FSM delay for evenly dividing factors is also seen to be smaller than that due to other nearby factors (not shown in figure) be-

```
for (i=0; i<32; i+=16){
   q += x[i]*z[i];
   q += x[i+1]*z[i+1];
            ⋮
   q += x[i+15]*z[i+15];
}
```

(a) Factor = 16

```
for (i=0; i<17; i+=17){
   q += x[i]*z[i];
   q += x[i+1]*z[i+1];
            ⋮
   q += x[i+16]*z[i+16];
}
for (;i<32; i+=1){
   q += x[i]*z[i];
}
```

(b) Factor = 17

Figure 4. Different unroll factors with loop bound = 32

cause of the lower number of FSM states. The FSM resulting from the unrolled code has 34 and 40 (36 in unrolled loop and 4 in trailer) states for unroll factors 16 and 17 respectively. The increasing FSM delay with unroll factor is very significant, and can critically affect the resulting designs. Arbitrarily large unroll factors, while apparently decreasing the number of cycles, can cause timing violations due to the critical path delay exceeding the clock cycle.

Clearly, an exhaustive strategy with detailed FSM and logic synthesis in the loop will be prohibitively expensive for real applications. In applications with multiple loop nests that are amenable to unrolling, every unroll factor of every loop could be a candidate for exploration. This motivates the need for a more practically efficient loop unroll factor exploration methodology that does not rely on exhaustive enumeration and synthesis.

4 Formulation and Approach

For unrolling, we consider loop constructs such as *for* and *while* with only one update (of the type 'i += k' where k is a constant) to the loop induction variable, either in the header or the body. Our objective is to determine the best set of unrolling decisions for the loops that allow maximization of performance while the controller delay fits within a time budget, being a fraction of the clock period. This time budget is an input to the algorithm, and could be arrived at by considering the clock period, and the function unit and MUX delays, but its automation is outside the scope of the paper.

4.1 Knapsack Formulation

The unroll factor optimization can be viewed as a variant of the well known *Knapsack Problem* of maximizing profits under weight constraints. In this case, we have a total delay constraint D and an initial delay D' when all loops are rolled, and the available slack $D - D'$ should be distributed among n loops, each loop being unrolled u_i times, resulting in an additional delay d_i. The delay constraint is:

$$\sum d_i \leq D - D' \tag{1}$$

If the latency due to each loop is L_i, then the total latency $\sum L_i$ needs to be minimized. However, the mapping to the traditional Knapsack problem is not straightforward, as the d_i's in our problem are not independent; the additional delay due to an unroll factor for a loop depends on the extent to which other loops have been unrolled. Nevertheless, it is possible to use heuristics similar to those used in the Knapsack context to address our problem.

Our exploration strategy consists of two components: (1) search space pruning to limit the exploration to a few promising candidate unroll factors; and (2) a fast estimation of FSM delay for the considered unroll factors without actually unrolling the loops and without detailed synthesis

4.2 Search Space Pruning

There are two important observations, based on which the search space of unroll factors to be considered can be substantially reduced.

4.2.1 Pareto-optimal Factors

Figure 3(a) shows that loop performance does not monotonically improve with increasing unroll factor. On the other hand, the FSM delay generally increases with increasing unroll factors because the larger loop body results in an increased number of states. Thus, we can restrict our attention to few *pareto-optimal* unroll factors – among these factors, latency reduces with increasing factor, but FSM delay increases.

4.2.2 Threshold Factors

Another interesting empirical observation about the performance effect of loop unrolling is that the performance improvement is marginal for relatively large unroll factors. This is generally due to the saturation of resource utilization even if parallelism theoretically exists. The results of extensive experiments we performed on the variation of performance and FSM delay with unroll factors can be summarized as follows: *a latency of around 10-15% over that of the fully unrolled loop can be achieved by relatively small threshold unroll factors that cost an average of 45% extra FSM delay over the rolled loop*. We performed this study over different applications by varying different parameters such as resource constraints and loop iteration counts, and found the above observation to be true. The experimental evidence is omitted due to lack of space, but Figure 3(a) illustrates this phenomenon – the latency is 72 cycles due to unroll factor 8 and the fully unrolled loop (factor = 32) has a latency of 66 cycles. Thus, small unroll factors such as 8 are important exploration candidates.

4.3 Controller Delay Estmation

Our estimation of the FSM delay is based on the following equation [17] that uses only high-level behavioral information from the specification.

$$\text{Delay} = A \times log(\#\text{operations}) \times (\#\text{state bits}) \tag{2}$$

where #operations is the number of DFG nodes and #state bits is the state register bits. A depends on resource constraints and the number of variables, and is found as:

$$A = x + y \times (\#\text{FU control bits}) + z \times (\#\text{Variables}) \tag{3}$$

where the constants x, y, and z are determined from a linear regression and are ASIC library-specific. The above equations give the FSM delay within a 6-7% error and do not involve actual generation and synthesis of the FSM itself, which makes them attractive mechanisms in design space exploration. For a specific unroll factor, we can arrive at an FSM delay estimate by computing: (1) the number of operations; (2) the number of basic blocks; and (3) the total number of states/control steps.

4.3.1 Operation Count

Consider a program with n unrollable loops $l_1 ... l_n$ with iteration counts $B_1 ... B_n$. Loop unrolling increases the number of operations to be managed by the FSM in the process of duplicating loop bodies. If the unroll factor for loop l_i is u_i, and the number of operations in the body of the original loop was *LOps*(l_i), then additional operations due to unrolling are $(u_i - 1) \times$ *LOps*(l_i) if u_i evenly divides B_i. If $u_i \bmod B_i \neq 0$, a trailer loop is generated with *LOps*$(l_i) + 2$ extra operations (the two extra operations are the increment and comparison operations for the trailer loop). If *Ops(P)* was the operation count in the original program, then the new operation count *LUOps*(P, U) after subjecting it to unrolling vector $U = (u_1, ..., u_n)$ is given by:

$$\textit{LUOps(P,U)} = \textit{Ops(P)} + \sum_{i=1}^{n} (u_i - 1) \times \textit{LOps}(l_i) + \textit{Even}(u_i, B_i) \times (\textit{LOps}(l_i) + 2)$$

where *Even*(u_i, B_i) = 0 if $B_i \bmod u_i = 0$ and *Even*(u_i, B_i) = 1 otherwise.

4.3.2 Basic Blocks

The FSM delay is generally independent of the number of basic blocks in a standard binary encoding. However, we have used a slightly different state register encoding in our synthesis flow – some bits are used to encode the basic block number, and other bits encode the control step within that scheduled basic block. This separation usually leads to more state bits, but has been observed in our experiments to result in slightly better delay characteristics. A basic block count resulting due to unrolling is necessary to determine a part of the state register width. (Actually, even if a straight binary encoding is done, the following computation is still necessary to determine the overall number of state bits after unrolling). If the basic block count in the original program is $BB(P)$, and the count for loop l_i is $LBB(l_i)$, the new count *LUBB*(P, U) is given by:

$$\textit{LUBB}(P, U) = \textit{BB(P)} + \sum_{i=1}^{n} (u_i - 1) \times (\textit{LBB}(l_i) - 1) + \textit{Even}(l_i) \times (\textit{LBB}(l_i) + 1)$$

The $(\textit{LBB}(l_i) - 1)$ term is due to the merging of the tail of one basic block and the head of the next one during unrolling. Note that if the original loop had only one basic block, its basic block count remains unchanged after unrolling by some u_i if *Even*$(u_i, B_i) = 0$. We have omitted discussion of some other corner cases due to lack of space.

4.3.3 Control Steps Estimation

The number of states/control steps can be estimated by a simple basic block-wise list scheduling. However, during unrolling factor exploration, it is very time-consuming to explicitly unroll the loops for different factors and perform scheduling. In our exploration, we use estimates for the number of control steps as a function of the unroll factor without explicitly performing unrolling. We use the idea of pipeline Initiation Interval (II). II for a loop depends on two main factors [18]: loop carried dependences and resource constraints. The initiation interval due to loop carried dependences is given by: $II_{LCD}(l_i) = \text{Max}\lceil \frac{\text{DistLat}(l_i)}{\text{DistIter}(l_i)} \rceil$, where DistLat$(l_i)$ is the number of cycles that the dependent nodes are apart in the basic block schedule, and DistIter(l_i) is the number of iterations that they are apart. II can also be limited by the resource constraints: $II_{RC}(l_i) = Max_r \lceil \frac{N_r C_r}{F_r} \rceil$, where N_r is the number of nodes of type r in the loop body, C_r is the latency of the function unit in cycles, and F_r is the number of resources of this type. The initiation interval estimate is now computed as: $II(l_i) = Max(II_{LCD}, II_{RC})$.

According to our FSM encoding, the basic block with the longest latency determines the second part of the state bits. This may change due to unrolling: (1) if the loop body has only one basic block, then the new latency is: *NS*(l_i) = *LCS*$(l_i) + (u_i - 1) \times II(l_i)$, where *LCS*$(l_i)$ is the latency of the original loop body; (2) if the loop has multiple basic blocks, the merging of the first and last ones during unrolling will require a rescheduling and recomputation of latency $NS(l_i)$. If the initial maximum latency basic block was $CS(P)$, the maximum control steps after unrolling *LUCS*(P, U) is given by:

$$\textit{LUCS}(P, U) = Max(CS(P), \forall i \textit{NS}(l_i)) \quad (4)$$

4.4 Unroll Factors Exploration

Because of the computational intractability of the exploration problem, we define a priority function to select a loop for unrolling, and a mechanism for arriving at a delay budget for each loop.

4.4.1 Priority Function

To capture the effect of latency reduction due to unrolling and the concomitant FSM delay increase, we use a priority function of the type:

$$\text{Pr}(l_i) = \frac{\text{Reduction in Latency}}{\text{Additional FSM Delay}} \quad (5)$$

Equation 5 allots a higher priority to loops that result in a higher latency improvement and relatively lower FSM delay. While this is similar in principle to the *value-density* heuristic (= value/weight) used in solutions to the Knapsack problem, one complication is that the value of $Pr(l_i)$ will, in general, be different for different unroll factors (whereas in the Knapsack problem the density for an object remains independent of other selections). We exploit the observation (Section 4.2.2) that we can get to within 10% of the latency of the fully unrolled loop (unroll factor B_i) with a relatively small unroll factor in typical practical loops. Let this threshold unroll factor be u_{th}. The priority function is initially defined as:

$$\text{Pr}(l_i) = \frac{\textit{LCS}(l_i) - \textit{NS}(l_i, u_{th})}{\text{Delay}(l_i, u_{th}) - \text{Delay}(l_i, 1)} \qquad (6)$$

4.4.2 Delay Budgets for Loops

Instead of spending available FSM delay budget on a single high priority loop, we attempt to distribute the delay between different loops because of the observation that latency improvements are greatest at small unroll factors. Thus, for example, if two loops have equal priority, then it is better to unroll both loops by a smaller extent than to unroll one loop to a larger extent; unrolling the loop by a larger factor leads to smaller incremental latency improvement. We arrive at an FSM delay budget for loop l_i as follows:

$$\text{Budget}(l_i) = \frac{Pr(l_i)}{\sum_{j=1}^{n} Pr(l_j)} \times (D - D') \qquad (7)$$

Equation 7 gives preference to relatively smaller unroll factors which are more profitable.

4.5 Overall Algorithm

Algorithm 1 shows our overall heuristic algorithm for arriving at the optimal unroll factors; it is based on the priority computation and delay budget allocation for loops. We make two passes through the loops. In the first pass, the priority for each loop l_i and the FSM delays are computed for the threshold factor u_{i_th}. The delay budget is allotted for the highest priority loop and it is unrolled by up to u_{i_th} from among its pareto-optimal factors. The new U is pushed on to a stack. The process continues until all loops are unrolled, or the available delay budget reduces to zero. Note that the priority functions have to be re-computed at each step because the FSM delays keep changing with new unrolling decisions.

In the second pass, if additional slack is still available, we now attempt to fully unroll all the loops, using the same priority function, but now using full unroll factors ($u_i = B_i$). Every new unroll vector U is pushed on to the stack. We continue the full unrolling until all loops are fully unrolled, or no more FSM delay budget remains. Finally, we perform a verification step involving actual synthesis using the unroll vector U at the top of the stack to confirm that the resulting FSM does fit into the allowed budget D. If not, then we go down the stack until we find a valid candidate – the optimal unroll factor.

Algorithm 1 Exploration of Optimal Unroll Factor

Input: CDFG with loops $l_1, l_2, ..., l_n$ with iteration count $B_1 ... B_n$, FSM delay constraint D
Output: Optimal unrolling factor OptU = $(u_1, u_2, ..., u_n)$

$D' = 0, U = \{1, 1, ..., 1\}$
while available FSM delay budget $D > D'$ and all loops not handled **do** {PASS 1}
    Estimate FSM delay and compute priority function for each loop at threshold unroll factor u_{i_th}
    Select maximum priority loop l_{max} and unroll by max. pareto-optimal factor up to u_{max_th}
    Let D' = new FSM delay
    Stack.Push(U)
while available FSM delay budget $D > D'$ and all loops not handled **do** {PASS 2}
    Estimate FSM delay and compute prioirity function for each loop at unroll factor B_i
    Select max. priority loop l_{max} and unroll by up to B_i
    Let D' = new FSM delay
    Stack.Push(U)
Valid = FALSE
while not Valid and not EMPTY (Stack) **do** {VERIFY}
    Synthesize FSM corresponding to OptU = Stack.Pop()
    if delay $< D$ **then**
        Valid = TRUE
return OptU

5 Experiments

We have implemented our exploration strategy on a SUIF-based behavioral synthesis framework that takes a C-style input and generates RTL-level VHDL datapath and FSM, which are then synthesized with Synopsys Design Compiler using a 0.13μ ASIC library. The exploration was carried out on examples drawn from benchmark suites such as Mediabench, MiBench, and HLSynth95.

Figure 5 shows a comparison of the actual and estimated FSM delays for different unroll factors. On an average, we observe an error of 7%, with the actual errors ranging between 4% to 9%. A comparison of the synthesis and estimation times gives an idea of the suitability of our exploration strategy. The estimate required less than 5 minutes independent of unroll factors, whereas the synthesis times exceeded 6 hours for some benchmarks when larger unroll factors were used.

In order to evaluate our exploration algorithm, we took different FSM delay budgets for each application and observed the latencies due to the solutions generated by our algorithm, and compared it with those due to a relatively exhaustive method (Figure 6). For many examples, the exhaustive algorithm (which was also subject to some pruning) did not complete within 24 hours even after using relatively small loop bounds and we used the best result generated within this time. For examples with a single loop (Figure 5(a)-(c)), our algorithm generated the same solution as that given by the

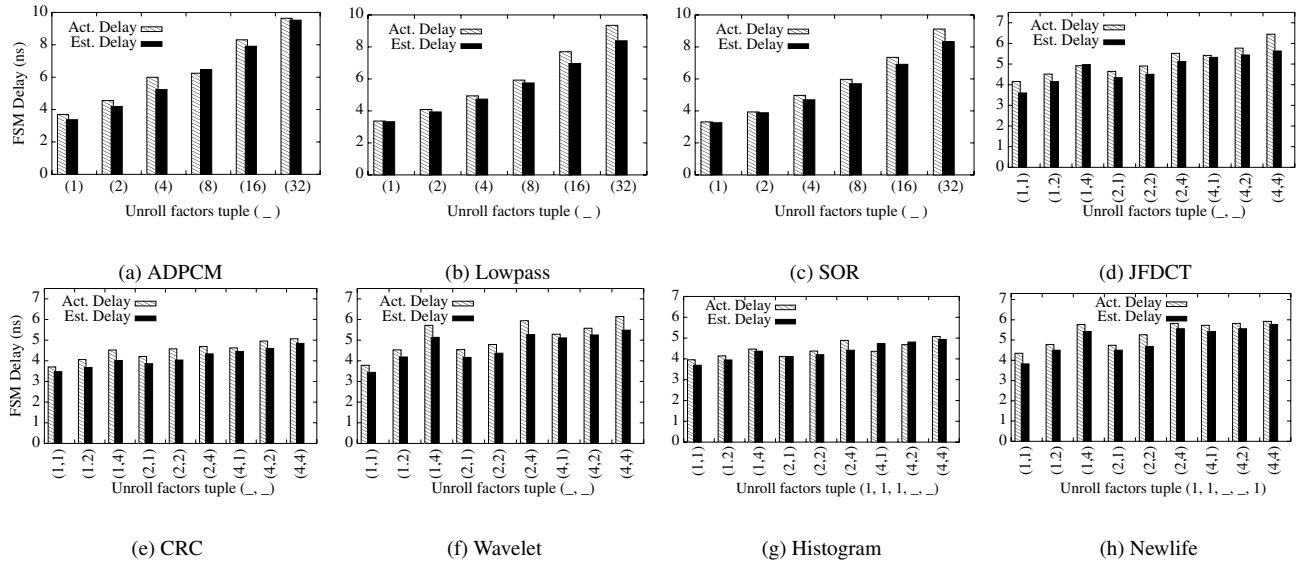


Figure 5. Predicted and Actual FSM delays

exhaustive approach, but was 2-4 orders of magnitude faster. For examples with multiple loops (Figure 5(d)-(h)), our algorithm generated solutions for which the latencies were off by an average of 8% from those generated by the exhaustive approach, with the maximum deviation being 13%. The estimation was faster by a similar magnitude.

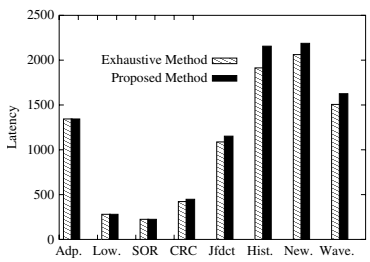


Figure 6. Exploration Results

6 Conclusions and Future Work

We presented an exploration strategy to predict the optimal loop unrolling factors in high-level synthesis that takes into account the effect on controller delay. The prediction is based on a priority function and associated techniques for pruning the overall search space to limit the exploration to promising unroll factor candidates, estimating the loop latencies without doing an actual unrolling, and estimating FSM delays from high level information. Experimental results indicate good quality delay estimations and unroll factor predictions that are significantly faster than exhaustive enumeration and synthesis. In the future, we plan to extend the current formulation to determine the FSM delay budget as part of the overall exploration, i.e., include datapath delays also. Other possible extensions include finding optimal clock periods keeping in view controller delay and incorporating the effects of other loop optimizations.

References

[1] V. Sarkar, "Optimized unrolling of nested loops," in *14th international conference on Supercomputing*, 2000.

[2] S. Carr and K. Kennedy, "Improving the ratio of memory operations to floating-point operations in loops," *ACM TOPLAS*, 16(6), 1994.

[3] A. Koseki, et al., "A method for estimating optimal unrolling times for nested loops," Intl. Symp. on Parallel Arch. ISPAN), 1997.

[4] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, "Influence of compiler optimizations on system power," DAC 2000.

[5] R. Leupers and P. Marwedel, "Function inlining under code size constraints for embedded processors," ICCAD 1999.

[6] M. Potkonjak, et al, "Multiple constant multiplications: efficient and versatile framework and algorithms for exploring common subexpression elimination," IEEE TCAD, 15(2), 1996.

[7] A. Hosangadi, et al., "Factoring and eliminating common subexpressions in polynomial expressions," in ICCAD 2004.

[8] M. Potkonjak and J. Rabaey, "Optimizing resource utilization using transformations," IEEE TCAD, 13(3), 1994.

[9] R. Cordone, et al., "Using speculative computation and parallelizing techniques to improve scheduling of control based designs," ASPDAC 2006.

[10] S. Gupta, et al., "Coordinated parallelizing compiler optimizations and high-level synthesis," ACM TODAES, 9(4), 2004.

[11] S. Park and K. Choi, "Performance-driven high-level synthesis with bit-level chaining and clock selection," IEEE TCAD, 20(2), 2001.

[12] M. C. Molina, et al, "Bitwise scheduling to balance the computational cost of behavioral specifications," IEEE TCAD, 25(1), 2006.

[13] S. Gupta, et al., "Loop shifting and compaction for the high-level synthesis of designs with complex control flow," DATE 2004.

[14] A. Kuehlmann and R. A. Bergamaschi, "Timing analysis in high-level synthesis," ICCAD 1992.

[15] R. A. Bergamaschi, "Bridging the domains of high-level and logic synthesis," IEEE TCAD, 21(5), 2002

[16] A. Srivastava et al., "Achieving design closure through delay relaxation parameter," ICCAD 2003.

[17] G. R. Gupta, M. Gupta, and P. R. Panda, "Rapid estimation of control delay from high-level specifications," DAC 2006, pp. 455–458.

[18] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software pipelining," *ACM Comput. Surv.*, vol. 27, no. 3, pp. 367–432, 1995.