Field Programmable Custom Computing Machines, FCCM '99

Dynamic Precision Management for Loop Computations on Reconfigurable Architectures*

Kiran Bondalapati and Viktor K. Prasanna
University of Southern California, Los Angeles
{kiran,prasanna}@ceng.usc.edu
http://maarc.usc.edu

Abstract

Reconfigurable architectures promise significant performance benefits by customizing the configurations to suit the computations. Variable precision for computations is one important method of customization for which reconfigurable architectures are well suited. The precision of the operations can be modified dynamically at run-time to match the precision of the operands. Though the advantages of reconfigurable architectures for dynamic precision have been discussed before, we are not aware of any work which analyzes the qualitative and quantitative benefits which can be achieved. This paper develops a formal methodology for dynamic precision management. We show how the precision requirements can be analyzed for typical computations in loops by computing the precision variation curve. We develop algorithms to generate optimal schedules of configurations using the precision variation curves. Using our approach, we demonstrate 25%-37% improvement in the total execution time of an example loop computation on the XC6200 device.

1 Introduction

Reconfigurable hardware has the potential to enhance the performance of many computer applications. The hardware resources can be tuned to the algorithm and the software overhead can be avoided to achieve superior performance compared to conventional microprocessors. Reconfigurable hardware also possesses more flexibility than ASIC hardware and can be utilized for a more diverse set of computations. There are several methods of generating custom hardware configurations suited to the computations to be performed. The ability to perform variable precision arithmetic is one of the significant advantages of reconfigurable hardware.

Reconfigurable hardware such as FPGAs [14, 16] and various custom computing machines (CCMs) [2, 4, 9, 15] contain fine-grained configurable resources. Such fine-grained configurable logic can be utilized to build computing modules of various sizes. The modules can be built to perform computations on various bit-widths. For example, it is possible to build a standard 16-bit×16-bit multiplier or a 8-bit×12-bit multiplier using reconfigurable hardware. The 8-bit×12-bit multiplier would consume less area and execute faster than the standard 16-bit×16-bit multiplier. In configurable hardware, using higher precision usually results in wastage of resources such as logic area, time and power. For example, performing 32-bit multiplications when the operands have only 8 significant bits will typically require 16 times more area and 4 times more execution time. Redundant computations also expend more clock cycles and increase the power consumption. The ability to construct modules of required precision is one of the key advantages of reconfigurable hardware. Variable precision computations can be implemented by using a *static* approach. In the *static* approach, the precision of the operands and operation is fixed at compile time and can be different from the standard precision(e.g. 8-bit, 16-bit, 32-bit, etc.) used on microprocessors. Reconfigurable architectures also support *dynamic precision*, which is the ability of the hardware to change its precision at run-time in response to variant precision demands of the algorithm.

Applications are typically developed to perform operations on standard 32-bit variables. The precision of the operands and the operations is sufficient to guarantee the correctness of the operations in the worst

*This work was supported by the DARPA Adaptive Computing Systems Program under contract DABT63-96-C-0049 monitored by Fort Hauchuca.

case. But in most applications, the actual precision required for computations is usually much lower than the precision implemented. This is typically the case in computations which accumulate values as the computations progress, as in iterative computations such as loops. The precision of the operands increases as the iterations of the loops progress. Loop computations offer the most potential for pipelining and parallelizing in most applications. Configurable hardware is an excellent match for computations with fine-grain pipelining and parallelism. In addition to the performance benefits obtained by mapping of computations in a loop onto configurable hardware, loops can also take advantage of variable precision.

Applications are currently mapped to reconfigurable hardware either by high level behavioral compilers or exhaustive hand-tooled designs. To extract the performance advantages of configurable hardware for variable precision, the trade-offs in performing computations using a very high precision versus changing the precision of computations as the execution progresses need to be evaluated. Performing this analysis by hand and tuning the implementation to the requirements of the application entails significant effort on the part of the designer. Dynamic precision management can result in implementations with lower execution times, logic area and power consumption compared to previous approaches.

For managing dynamic precision in loop computations, intelligent choices on the use of appropriate modules from the available set of modules with different precision need to be made. These configurations then have to be scheduled to achieve optimal execution schedule. We consider a schedule to be optimal if the schedule has minimum *total execution time*, which includes both the execution time in various configurations and the reconfiguration time between configurations. Automatic computation of the actual precision and configurations to be utilized in the computations is the focus of this paper. Currently, a framework for managing dynamic precision computations for any class of computations does not exist. We develop such a framework for loop computations in this paper.

In Section 2 we give an overview of our approach to the *dynamic precision* management problem. Each of the steps in our approach are then described in detail in the later sections. Analysis of the required precision for loop computations is discussed in Section 4. Section 5 describes our Hybrid System Architecture Model(HySAM) of reconfigurable architectures. The variable precision loop mapping problem is defined and our Dynamic Precision Management Algorithm(DPMA) for computing the optimal schedule is presented in Section 6. We illustrate the utility of our approach by showing an example mapping in Section 7. Conclusions and some related problems are discussed in Section 8.

2 Overview of Our Approach

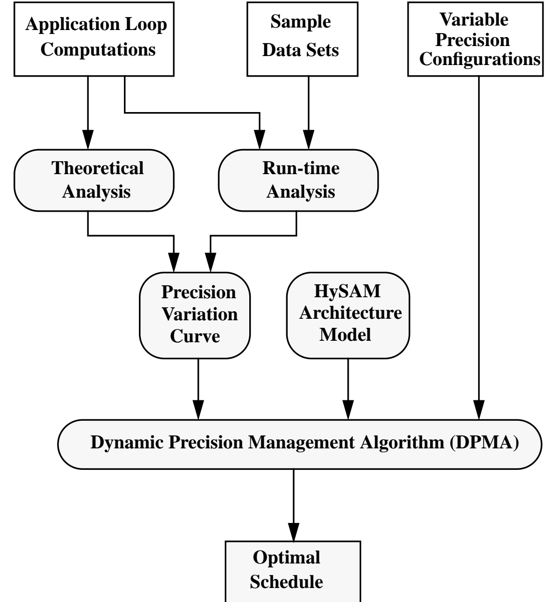

Figure 1: Overview of our approach for dynamic precision management in loops(shaded and rounded regions indicate our contributions)

This paper details an approach to managing the task of adapting the precision of the implementation to that of the application. An overview of our approach is shown in Figure 1. We focus our efforts on dynamic precision management for loop computations since they are the most compute intensive tasks in typical applications. For the loop computations in applications, we describe an approach to determine the required precision using theoretical analysis and run-time instrumentation. The required precision for the computations in a loop can be expressed as the variation in precision as the iterations of the loop progress. We introduce the concept of the *precision variation curve* to represent this variation. The *precision variation curve* for the operations and operands in the loop can be identified either by theoretical analysis or by run-time analysis as described in Section 4.

Given the required precision for the iterations of the loop, we need to determine the mapping of the iterations to a set of configurations which are used to

execute the operations in the loop. For each iteration the precision of the configuration which executes the iteration should be equal to or greater than the required precision for that iteration. The configurations are chosen from the set of library components or parameterized modules that are provided for the architecture.

Given the requirements for the precision of the computations and the available module configurations, we compute the set of configurations and the schedule of reconfigurations. We compute these by developing algorithmic techniques for precision management. First, we develop an abstract model of reconfigurable architectures, the Hybrid System Architecture Model(HySAM). This parameterized abstract model is general enough to capture a wide range of configurable systems. We define the precision management problem in loop computations using our model. A dynamic precision management algorithm is then developed to compute the optimal sequence of configurations for minimizing the total execution time including the reconfiguration time.

3 Related Work

There has been significant research in the area of mapping applications to configurable computing in the last decade [2, 5, 8, 15]. Customizing configurable hardware to suit the computations has been acknowledged as the most significant advantage of such architectures. Some researchers have adapted the hardware to perform computations with exactly the required precision for the computations [11, 13]. Such *static* approaches do not exploit the ability of configurable hardware to be adapted to the exact required precision as the computations progress. The maximum possible precision of variables which is determined in the *static* approach can still involve execution with superfluous precision and unnecessary overheads. Several efforts have also focused on developing parameterized libraries and components, precision being one of the parameters. Most FPGA device vendors provide such highly optimized parameterized libraries for their architectures. Efforts have also been made to generate such modules using high level descriptions [3, 6].

We are not aware of any formal framework to study and analyze the dynamic precision variation in applications. Algorithmic techniques to utilize configurable computing to dynamically vary the precision of computations have not been demonstrated previously.

4 Precision Requirement Analysis

The precision required for the computations in an application might not only vary with the specific operation but also change as the execution progresses. For iterative computations in which values are accumulated over the execution time of the application, the precision varies as the iterations progress. Loop computations are the most typical iterative computations which show such behavior. In addition to the varying precision, loops are the most compute intensive tasks in a program. In this paper we focus on the varying precision of operations in loop computations. This variation can be measured by analyzing the variation of the precision of the operands and the operations as the iterations progress. We represent this variation in terms of the loop iterations by using the *precision variation curve*.

4.1 Precision Variation Curve

The *precision variation curve* facilitates the representation of the notion of the variation in the precision of the operands and the operation as the execution of the loop progresses. A simple method to represent such a variation is to indicate the precision of the operand for each iteration so that the precision is defined for the whole iteration space. But as we shall show in the subsequent sections, the precision usually varies very slowly as the iterations progress. Thus the *precision variation curve* can be represented by specifying the points where the precision of the operands or the operation changes.

Definition: The *precision variation curve* for a given operation or operand in a loop computation can be represented by the sequence $<L_i, P_i>$, where $1 \leq i \leq u+1$ and $L_{u+1} = N+1$. For $1 \leq i \leq u$, P_i is the minimum precision required for the computing the iterations $L_i \ldots L_{i+1}-1$. Note that the hardware has to support at least a precision of P_i to execute the iterations $L_i \ldots L_{i+1}-1$ and produce the correct result.

Examples of *precision variation curves* are shown in Figure 3. We develop theoretical and run-time instrumentation methods for determining the *precision variation curve* in the next two sections.

4.2 Theoretical Analysis of Loops

We can theoretically determine the *precision variation curve* for the operations in a given computation. The precision of computed variables in a loop is determined by the precision of the variables before the iteration, the number of iterations and the operations

```
-----------------------------------------
 DO 10 I=1,N
   DO 20 J=1,N
     RSQ(J) = RSQ(J)+XDIFF(I,J)*YDIFF(I,J)
20   IF (MAXQ.LT.RSQ(J)) THEN
       MAXQ = RSQ(J)
   POVERR = POVERR / MAXQ
10 VIRTXY = VIRTXY + MAXQ * SCALE(I)
-----------------------------------------
```

Figure 2: Example code for simulations

performed on the variable. For each type of arithmetic operation, the maximum possible precision of the result can be expressed using the above values. For example, the precision of a variable X(initially 0) after N iterations of a loop which contains the statement $X = X + C$ is bounded by

$$Pr(X) \leq Pr(C) + \log(N+1)$$

where $Pr(X)$ denotes the bit size of the variable X. The analysis is not limited to simple expressions, but extends to complex arithmetic expressions in loops. For recursive expressions in loops where the value of the variable X in iteration i is given by X_i, if

$$X_i = c_1 * X_{j_1} + c_2 * X_{j_2} + \ldots + c_k * X_{j_k} = \Sigma_{l=1}^{l=k} c_l * X_{j_l}$$

then the upper bound on the precision of X_i is given by

$$Pr(X_i) \leq (i-1) * \log C + (i-1) * \log k + Pr(X_1)$$

where $C = max[c_1, c_2, \ldots, c_k]$, the maximum of the constant coefficients. Similarly, for the expression $X = X \times C$, the upper bound of precision for X with an initial value 1 and after N iterations is given by

$$Pr(X) \leq N * Pr(C)$$

The *precision variation curve* can be computed theoretically for all expressions in loops which are polynomials of variables and constants. Since most scientific applications consist of many such computations, theoretical analysis can be performed for all such computations. It is to be noted however, that such an analysis is not entirely feasible for floating point computations. But the analysis can be performed for integer and fixed point data and computations. This does not limit the applicability of the analysis or the algorithms we present later as many signal and image processing computations and several benchmark problems operate on integer and fixed point data. The remaining computations can be implemented with their default maximum precision.

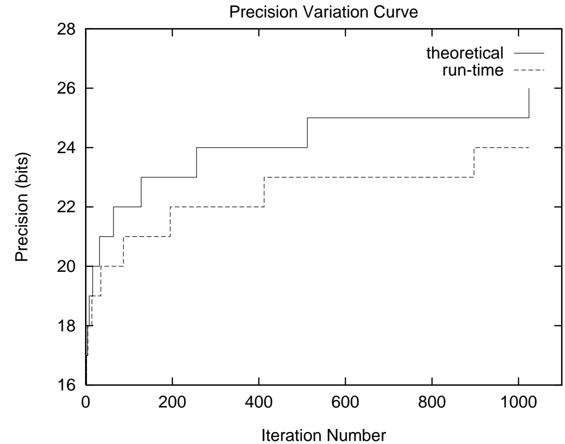

Figure 3: Precision Variation Curves for RSQ using theoretical and run-time analysis

4.3 Run-time Analysis

Theoretical analysis of expressions in loops computes the upper bounds on the precision of the variables and computations. This determines the minimum precision required to represent these variables. The estimates using theoretical analysis are conservative and can usually be much higher than the actual precision of the operands. For example, using the above analysis for the Fibonacci series $X_i = X_{i-1} + X_{i-2}$, we obtain $Pr(X_i) = i - 1$ and hence, $Pr(X_{15}) = 14$. But, $X_{15} = 610$ which needs only 10 bits. Even in the case when the bound is actually tight for expressions, the actual precision might be lower than theoretical estimate. This can occur when the data inputs are assumed to have maximum precision, but are actually randomly distributed over the complete input range. Using theoretical analysis can provide significant performance benefits by dynamic precision management. We discuss below how these benefits can be augmented by using profiling based analysis.

For example, consider the code segment shown in Figure 2. We performed simulations with uniformly distributed random values for the 8-bit non-negative data inputs $XDIFF$ and $YDIFF$. The precision of the RSQ variable was measured by tracing the earliest iteration in which a new higher significant bit was set. Since the maximum bits in the result of $XDIFF(I,J) * YDIFF(I,J)$ are 16, the iteration in which the kth most significant bit of the result is set is given by 2^{k-16}. The *precision variation curves* obtained using the theoretical and run-time analysis are plotted in Figure 3. The actual precision required for the computations is significantly lower than the theoretical estimate as evident from the graph.

This run-time measurements illustrate a very important advantage in exploiting variable precision computations. The actual $XDIFF$ and $YDIFF$ values have significantly lower precision than the maximum possible precision of 8 bits. The assumption of maximum precision for all the input $XDIFF$ and $YDIFF$ values has a rolling effect on precision of other operands and operations. The repeated accumulation of the product of these numbers results in a precision difference in the final values which is much larger than the precision difference for one value. It is clearly revealed in simulations where the actual required precision is much lower than the theoretical precision.

For computations which do not have a tight bound on the precision and for computations with complex control flow, computing the required precision by using run-time statistics is a viable alternative. The application can be instrumented to measure the precision of the different variables and the knowledge can be utilized by the mapping tool or the compiler to identify the required precision at various program points. Though we do not address the run-time mapping issues in this paper, it is also possible to determine the precision of the operands and the operations by examining the values at run-time and modifying the precision of the operations on the fly. In this paper we focus on run-time precision management based on the knowledge of the required precision at compile(mapping) time. The required precision can either be analyzed automatically or can be user specified.

5 Hybrid System Architecture Model (HySAM)

To realize a formal framework for algorithm development, we developed the Hybrid System Architecture Model(HySAM) of reconfigurable architectures. The *Hybrid System Architecture* is a general architecture consisting of a conventional microprocessor with additional Configurable Logic Unit(CLU). Figure 4 shows the architecture of the HySAM model. The architecture consists of a conventional microprocessor, standard memory, configurable logic, configuration memory and data buffers communicating through an interconnection network.

Key parameters of the Hybrid System Architecture Model(HySAM) are outlined below.

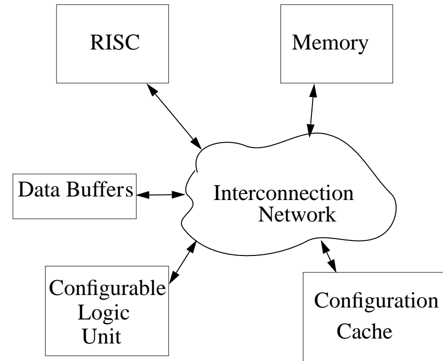

Figure 4: Hybrid System Architecture Model(HySAM)

F : Set of functions $F_1 \ldots F_n$ which can be performed on configurable logic.

C : Set of possible configurations $C_1 \ldots C_m$ of the Configurable Logic Unit.

$Pr(C_j)$: Precision of the configuration C_j.

t_{ij} : Execution time of function F_i in configuration C_j.

R_{ij} : Reconfiguration cost in changing configuration from C_i to C_j.

The parameterized HySAM models a wide range of systems from board level architectures to systems on a chip. Such systems include SPLASH [2], DEC PeRLE [15], Oxford HARP [5], Berkeley Garp [4], NSC NAPA1000 [9], Sanders CSRC [10] among others. The values for each of the parameters establish the architecture and also dictate the class of applications which can be effectively mapped onto the architecture. For example, a system on a chip architecture would have potentially faster reconfiguration times(lower k and K) than a board level architecture.

The set of functions(F) is the set of modules or library components which are available or implemented for the given architecture. Configurations(C) are developed by mapping one or more of such functions onto the available hardware architecture. A single function can have multiple configurations which can potentially execute the function. Each of the configurations might have different algorithm, area, precision, time and power characteristics. For example, a function such as division can be implemented using different algorithms such as iterative multiplication or iterative subtraction in different configurations. The execution time of a function F_i in a configuration C_j is given by t_{ij}. The cost of reconfiguring the hardware from a configuration C_i to a configuration C_j is given by R_{ij}. The reconfiguration cost includes the cost of

memory access for the configurations, the configuration data transfer cost and the cost of activating the configuration on the hardware.

5.1 Configurations for Variable Precision

Efficient modules are being developed by hand design, by automatic mapping and by generators [3, 6]. Modules for executing computations with a specified precision have also been explored. Some of the modules are parameterized which facilitate the construction of a configuration which can execute a computation of any given precision within a range of values. They are usually either statically developed designs such as the Xilinx LogicBlox or dynamically constructed using generators [3, 6]. The modules are usually optimized to exploit the nature of the computation for any given precision. Modules designed for specific architectures also exploit the hardware features which are available to enhance performance. For example, addition and multiplication modules exploit the carry chains available at nibble or byte boundaries in many FPGA architectures.

In this paper we assume that the set of modules which can execute the required arithmetic operations are available. Each function(such as multiplication) can have several configurations, each of which executes the operation with different precision. It is not necessary that a given operation have configurations which execute the operation with all the possible precision values. Note that each configuration is limited to the execution of one function in this paper though the HySAM model is actually more powerful. Hence, we represent t_{ij} as t_{C_j} is the rest of the paper.

6 Dynamic Precision Management

Given the *precision variation curve* for the loop, we need to determine the mapping of the iterations to a set of configurations which are used to execute the operations in the loop. For each iteration, the precision of the corresponding configuration which executes the iteration should be equal to or greater than the required precision for that iteration. But, reconfiguring the hardware whenever the required precision changes can result in significant reconfiguration overheads. For architectures in which the reconfiguration times are much higher than the execution times, the reconfiguration overhead might be prohibitive. Thus, it is necessary to identify the optimal set of configurations which result in minimization of the overall execution cost, including the reconfiguration cost. Also, the set of configurations which are available for executing an operation might not encompass all the possible precision values that are required. Some of the operations will have to be executed with more precision than is necessary in the absence of configurations with the exact precision.

We present the Precision Management Problem and the Dynamic Precision Management Algorithm based on the following assumptions:

- Higher precision computations require more resources such as power, logic area and computation time(t_{C_j}).
- The required precision for the computations varies monotonically. This is true for most computations which accumulate values as the loop iterations progress. The algorithms we describe can be applied to monotonic subsequences with optimal schedules for each subsequence individually.
- The algorithm determines the optimal schedule for a *given* precision variation curve. When the actual variation is different from the precision variation curve, the schedule might not be optimal.

Precision Management Problem(PMP)

Input: An operation in a loop with N iterations of the loop body and the *precision variation curve* for the operation. The *precision variation curve* is given as a sequence of pairs $<L_i, P_i>$, where $1 \leq i \leq u+1$ and $L_{u+1} = N+1$. For $1 \leq i \leq u$, P_i is the minimum precision required for computing the operation in iterations $L_i \ldots L_{i+1} - 1$.

Output: An optimal schedule of configurations $S = <Q_j, C_j>$, where $1 \leq j \leq v+1$ and $Q_{v+1} = N+1$. For $1 \leq j \leq v$, C_j is the configuration used for iterations $Q_j \ldots Q_{j+1} - 1$.

A schedule S is said to be valid if it satisfies the precision requirement for all the iterations of the loop, i.e., $\forall K \ s.t. \ 1 \leq K \leq N$, if

$$\begin{aligned} Pr_I &= P_i, \ for \ some \ i \ s.t. \ L_i \leq K < L_{i+1} \\ Pr_O &= Pr(C_j) \ for \ some \ j \ s.t. \ Q_j \leq K < Q_{j+1} \end{aligned}$$

then $Pr_I \leq Pr_O$ (see Figure 5).

An optimal schedule has the minimum total execution cost E which includes the reconfiguration cost among

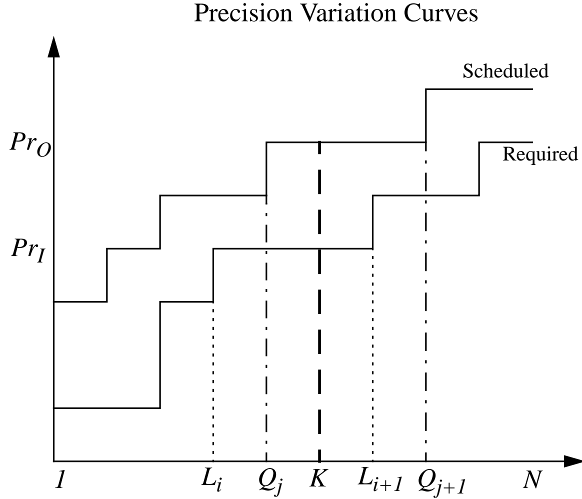

Figure 5: Constraint on required and scheduled Precision Variation Curves

all valid schedules. The cost of a schedule is given by

$$E = \sum_{j=1}^{v} [(Q_{j+1} - Q_j) \times t_{C_j} + R_{j-1j}]$$

where t_{C_j} is time for executing one iteration of the loop in configuration C_j and R_{j-1j} is the reconfiguration cost between configurations C_{j-1} and C_j. ⊙

To minimize the total execution cost, both the execution cost and the reconfiguration cost have to be examined. The set of configurations and the schedule of reconfigurations need to be determined. We first show that the points of reconfiguration are the subset of the points where the required precision changes, i.e., $Q \subseteq L$, where $Q = \{Q_1, \ldots, Q_v\}$ and $L = \{L_1, \ldots, L_u\}$.

Lemma 1. *Given the definitions in the* **PMP** *problem, the schedule S of configurations satisfies the property $Q \subseteq L$.*

Proof: Assume that $Q \not\subseteq L$ in the optimal schedule S. Then there exists at least one point of reconfiguration which is not a point of change of required precision.

$$\exists i : Q_i \notin L$$

Without loss of generality,

$$\exists j : Q_{i-1} \leq L_{j-1} < Q_i < L_j \leq Q_{i+1}$$

Consider the schedule S' where the configurations are the same as S but the reconfiguration points are different:

$$S = [Q_1 \ldots Q_{i-1} Q_i Q_{i+1} Q_{i+2} \ldots Q_n]$$

$$S' = [Q_1 \ldots Q_{i-1} L_j Q_{i+1} Q_{i+2} \ldots Q_n]$$

t_{C_i} is the cost of executing one iteration in configuration C_i. Since we assume precision variation to be monotonic, $Pr(C_{i+1}) > Pr(C_i)$ and $t_{C_{i+1}} > t_{C_i}$. The difference in execution cost of the two schedules is

$$\begin{aligned} S' - S &= (t_{C_i}(L_j - Q_{i-1}) + t_{C_{i+1}}(Q_{i+1} - L_j)) \\ &\quad -(t_{C_i}(Q_i - Q_{i-1}) + t_{C_{i+1}}(Q_{i+1} - Q_i)) \\ &= t_{C_i}(L_j - Q_{i-1} - Q_i + Q_{i-1}) \\ &\quad +t_{C_{i+1}}(Q_{i+1} - L_j - Q_{i+1} + Q_i) \\ &= (t_{C_i} - t_{C_{i+1}})(L_j - Q_i) \\ &< 0 \end{aligned}$$

Since $L_j > Q_i$ and $t_{C_i} < t_{C_{i+1}}$, $S' - S < 0$. The new schedule has lower cost and hence a schedule with reconfiguration points which is the subset of the precision change points has lower execution cost. Since S is the optimal schedule our assumption must be incorrect. Hence, $Q \subseteq L$. ⊙

6.1 Precision Management Algorithms

To determine the choice of configuration at each L_i, we can use a greedy approach where the best configuration with the required precision is chosen at each L_i. The best configuration $C_j (C_j \in C_1, \ldots, C_m)$ is given by the configuration which has the lowest execution cost t_{C_j}. But the greedy algorithm will not provide the optimal solution due to two reasons:

- The greedy approach does not consider the reconfiguration costs which are incurred at future reconfiguration points. A configuration with higher execution cost might have a lower reconfiguration cost at the next step, making it a better choice for executing the given iterations.
- With significant reconfiguration costs, it is possible that we use a higher precision configuration than required(even if exact precision configuration is available in C), to avoid a reconfiguration step in future. The greedy approach does not consider this case and thus can result in non-optimal schedule.

In the following, we present an algorithm based on dynamic programming which computes an optimal schedule having the minimum execution cost including the reconfiguration cost.

Dynamic Precision Management Algorithm (DPMA)
Let E_{ij} be the execution cost for executing up to L_i iterations with C_j being the last configuration. The

initial values of E are assigned as $E_{0j} = 0, 1 \leq j \leq m$. For each of the possible configurations C_j which can execute iterations from L_i we have to compute the optimal sequence of configurations ending in C_j. For $1 \leq j \leq m$, we compute E_{ij} by using the recursive equation:

$$\begin{aligned} E_{i+1j} &= (L_{i+1} - L_i) \times t_{C_j} + min_k(E_{ik} + R_{kj}) \\ & \qquad\qquad 1 \leq k \leq m \\ & \qquad if\ Pr(C_j) \geq P_i \\ &= \infty\ otherwise \end{aligned}$$

For each configuration, we have examined all the possible paths in executing the iterations $L_i \ldots L_{i+1} - 1$ once we have executed iterations $1 \ldots L_i - 1$. Note that we examine all configurations such that $Pr(C_j) \geq P_i$ which assures that we consider the case of using a higher precision than required$(Pr(C_j) = P_i)$. If each of the values E_{ik} is optimal then the value E_{i+1j} is optimal. Hence we can compute the optimal schedule of configurations S by computing the E_{ij} values. The minimum cost for execution of the loop is given by $min_j[E_{uj}]$.

We can use dynamic programming to compute the E_{ij} values. Computing one E_{ij} value takes $O(m)$ time since there are m configurations. The total number of values to be computed is $O(um)$, therefore the total time complexity of the algorithm is $O(um^2)$. ⊙

7 An Illustrative Example

We illustrate our approach by mapping the multiplication operation from the example code segment presented in Figure 2.

```
-----------------------------------------
 DO 10 I=1,N
   ...
10 VIRTXY = VIRTXY + MAXQ * SCALE(I)
-----------------------------------------
```

The input data $SCALE(I)$ is an 8-bit integer. The precision of $MAXQ$ has been analyzed in Section 4.3. We present the same result in the form of a table in Table 1.

We have abstracted the Xilinx XC6200 series device by using our model. The parameters specified are for the HySAM model and have been evaluated from XC6200 documentation [16, 7]. The footprint of each precision is given by the equation $4 \times row \times col$, where row and col are the precisions of the two inputs. For the configurations relevant to mapping the given operation, row is 8. Reconfiguration times are based on a 32-bit data bus running at 50MHz. It is possible to design modular configurations which can be reconfigured in lesser time using partial reconfiguration. For this mapping, we assumed that complete reconfiguration is needed for each configuration. The parameters for various multiplier configurations with different precisions are listed in Table 2.

P_i	L_i	L'_i	P_i	L_i	L'_i
Pr	Theoretical	Simulated	Pr	Theoretical	Simulated
16	1	1	22	64	195
17	2	2	23	128	412
18	4	5	24	256	897
19	8	14	25	512	-
20	16	35	26	1024	-
21	32	87			

Table 1: Theoretical and simulated iteration numbers for $N = 1024$

Configuration C_i	Precision $Pr(C_i)$	Time t_{C_i} (ns)	Reconfig. R_{0i} (ns)
C_1	8×8	140	5120
C_2	8×16	250	10240
C_3	8×20	300	12800
C_4	8×24	400	15360
C_5	8×28	520	17920
C_6	$8 \times 32^*$	*640	20480

Table 2: HySAM model parameters for XC6200 multiplier configurations(* values are estimates based on XC6264 device)

We measured the total execution time for the loop computations using five different approaches. The first two approaches do not exploit the *dynamic precision* by varying the precision of the operation at run-time. The different approaches and the schedule of configurations(<Q_j,C_j>) in each approach are described below.

- **Raw**: The first approach uses a static configuration of $8bit \times 32bit$ precision for all the iterations of the loop.
  Schedule: <1,C_6>
- **Static**: We utilize the theoretical analysis where we determine that the highest precision required for 1024 iterations is only $8bit \times 28bit$. But the configuration is still static and is used for all the iterations.
  Schedule: <1,C_5>

- **Greedy**: We used the greedy algorithm (see Section 6.1) to compute the schedule of configurations to be utilized for the computations. The precision of the operation is varied dynamically but the greedy choice is based on the lowest execution time for each configuration.
Schedule: $<1,C_2>,<2,C_3>,<32,C_4>,<512,C_5>$

- **DPMA**: Our dynamic precision management algorithm was utilized to compute the optimal schedule using the *precision variation curve*. This approach uses higher execution cost configurations for some of the computations but reduces the overall execution cost by performing lesser number of reconfigurations.
Schedule: $<1,C_4>,<512,C_5>$

- **DPMA-run**: In this approach we performed run-time analysis of the loop and utilized the *precision variation curve* from the run-time analysis as the input to the algorithm. This approach can be implemented easily by adding a run-time check of the precision, which needs very small amount of additional logic and no extra clock-cycles if the precision remains within the run-time statistics.
Schedule: $<1,C_4>$

Algorithm	Execution Time (ns)	Reconfiguration Time (ns)	Total (ns)
Raw	655360	20480	675840
Static	532480	17920	550400
Greedy	468010	56320	524330
DPMA	471160	33280	504440
DPMA-run	409600	15360	424960

Table 3: Execution times using different approaches

The execution times including the reconfiguration times are summarized in Table 3. The approaches using *dynamic precision* achieve significantly lower execution times compared to the Raw and Static approaches. We noticed that our DPMA algorithm executed all the iterations of the loop in the minimum time for the theoretical and run-time *precision variation curves*. The DPMA-run achieves significant speed-up by exploiting the fact that 28-bit precision is never required.

8 Conclusions

This paper has developed a framework for dynamic precision management for loop computations. We have shown how the variable precision in computations can be captured by using the *precision variation curve*. The paper described our approach to computing the *precision variation curve* using theoretical and run-time analysis. The information obtained from these analyses is used to develop optimal schedules for dynamic precision management. The DPMA algorithm that we have developed can compute the required optimal schedule for a given operation in a loop using the *precision variation curve* and the set of variable precision configurations. Our Hybrid System Model(HySAM) of reconfigurable architectures facilitates the development of these algorithms using a high level abstract model. The paper illustrated the performance benefits achievable for an example loop computation using our approach. We expect that the proposed approach can lead to significant improvement in performance and automatic mapping of variable precision computations on reconfigurable architectures.

The dynamic precision management framework gives rise to a wealth of issues which can potentially provide enormous benefits to mapping computations onto configurable hardware. Bit-serial and digit-serial computations are one class of computations which can exploit dynamic precision without large overheads. The control component of the design needs to execute the configurations for a variable number of steps based on the required precision. Run-time precision management where the control modifies the precision of the computations are being explored. Configurable logic can be utilized to execute multiple iterations of loops in parallel in the absence of dependencies. Reduction of the logic resources due to dynamic precision management can be exploited to execute more number of iterations in parallel. Multi-context devices and configuration caches can be utilized to reduce the reconfiguration overheads by storing variable precision configurations.

The work reported here is part of the USC MAARC project(http://maarc.usc.edu). This project is developing algorithmic techniques for realizing scalable and portable applications using configurable computing devices and architectures. We are developing computational models and algorithmic techniques based on these models to exploit dynamic reconfiguration. Some recent related results can be found in [1, 12].

9 Acknowledgments

We would like to thank Steve Casselman, David Franklin, and John Schewel of the Virtual Computer Corporation for their interest and support.

References

[1] K. Bondalapati and V.K. Prasanna. Mapping Loops onto Reconfigurable Architectures. In *8th International Workshop on Field-Programmable Logic and Applications*, September 1998.

[2] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder. *Splash 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, 1996.

[3] M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek. Object Oriented Circuit-Generators in Java. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.

[4] J. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, April 1997.

[5] A. Lawrence, A. Kay, W. Luk, T. Nomura, and I. Page. Using reconfigurable hardware to speed up product development and performance. In *5th International Workshop on Field-Programmable Logic and Applications*, 1995.

[6] O. Mencer, M. Morf, and M.J. Flynn. PAM-Blox: High Performance FPGA Design for Adaptive Computing. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.

[7] Xilinx Application Notes. A Fast Constant Coefficient Multiplier for the XC6200.

[8] R.J. Petersen and B. Hutchings. An Assessment of the Suitability of FPGA-Based Systems for use in Digital Signal Processing. In *5th International Workshop on Field-Programmable Logic and Applications*, 1995.

[9] C.R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J.M. Arnold, and M. Gokhale. The NAPA Adaptive Processing Architecture. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.

[10] S.M. Scalera and J.R. Vázquez. The design and implementation of a context switching FPGA. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.

[11] N. Shirazi, P.M. Athanas, and A.L. Abbott. Implementation of a 2-D Fast Fourier Transform on an FPGA-Based Custom Computing Machine. In *International Workshop on Field-Programmable Logic and Applications*, September 1995.

[12] R.P.S. Sidhu, A. Mei, and V.K. Prasanna. String Matching on Multicontext FPGAs using Self-Reconfiguration. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb 1999.

[13] A.F. Tenca and M.D. Ercegovac. A Variable Long-Precision Arithmetic Unit Design for Reconfigurable Coprocessor Architectures. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.

[14] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A Time-Multiplexed FPGA. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 22–28, April 1997.

[15] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, March 1996.

[16] Xilinx. XC6200 Field Programmable Gate Arrays, 1996.