Mapping Loops onto Reconfigurable Architectures [*]

Kiran Bondalapati and Viktor K. Prasanna

Department of Electrical Engineering Systems, EEB-200C
University of Southern California
Los Angeles, CA 90089-2562, USA
{kiran, prasanna}@usc.edu
http://maarc.usc.edu

Abstract. Reconfigurable circuits and systems have evolved from application specific accelerators to a general purpose computing paradigm. But the algorithmic techniques and software tools are also heavily based on the hardware paradigm from which they have evolved. Loop statements in traditional programs consist of regular, repetitive computations which are the most likely candidates for performance enhancement using configurable hardware. This paper develops a formal methodology for mapping loops onto reconfigurable architectures. We develop a parameterized abstract model of reconfigurable architectures which is general enough to capture a wide range of configurable systems. Our abstract model is used to define and solve the problem of mapping loop statements onto reconfigurable architectures. We show a polynomial time algorithm to compute the optimal sequence of configurations for one important variant of the problem. We illustrate our approach by showing the mapping of an example loop statement.

1 Introduction

Configurable systems are evolving from systems designed to accelerate a specific application to systems which can achieve high performance for general purpose computing. Various reconfigurable architectures are being explored by several research groups to develop a general purpose configurable system. Reconfigurable architectures vary from systems which have FPGAs and glue logic attached to a host computer to systems which include configurable logic on the same die as a microprocessor.

Application development onto such configurable hardware still necessitates expertise in low level hardware details. The developer has to be aware of the intricacies of the specific reconfigurable architecture to achieve high performance. Automatic mapping tools have also evolved from high level synthesis tools. Most tools try to generate hardware configurations from user provided descriptions of circuits in various input formats such as VHDL, OCCAM, variants of C, among others.

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

Automatic compilation of applications involves not only configuration generation, but also configuration management. CoDe-X [8] is one environment which aims to provide an end-to-end operating system for applications using the Xputer paradigm. General techniques are being developed to exploit the characteristics of devices such as partial and dynamic reconfiguration by using the concepts of Dynamic Circuit Switching [11], Virtual Pipelines [10] etc. But there is no framework which abstracts all the characteristics of configurable hardware and there is no unified methodology for mapping applications to configurable hardware.

In this paper we address some of the issues in the development of techniques for automatic compilation of applications. We develop algorithmic techniques for mapping applications in a platform independent fashion. First, we develop an abstract model of reconfigurable architectures. This parameterized abstract model is general enough to capture a wide range of configurable systems. These include board level systems which have FPGAs as configurable computing logic to systems on a chip which have configurable logic arrays on the same die as the microprocessor.

Configurable logic is very effective in speeding up regular, repetitive computations. Loop constructs in general purpose programs are one such class of computations. We address the problem of mapping a loop construct onto configurable architectures. We define problems based on the model which address the issue of minimizing reconfiguration overheads by scheduling the configurations. A polynomial time solution for generating the optimal configuration sequence for one important variant of the mapping problem is presented.

Our mapping techniques can be utilized to analyze application tasks and develop the choice of configurations and the schedule of reconfigurations. Given the parameters of an architecture and the applications tasks the techniques can be used statically at compile time to determine the optimal mapping. The techniques can also be utilized for runtime mapping by making static compile time analysis. This analysis can be used at runtime to make a decision based on the parameters which are only known at runtime.

Section 2 describes our Hybrid System Architecture Model(HySAM) in detail. Several loop mapping problems are defined and the optimal solution for one important variant is presented in Section 3. We show an example mapping in Section 4 and discuss future work and conclusions in Section 5.

1.1 Related Work

The question of mapping structured computation onto reconfigurable architectures has been addressed by several researchers. We very briefly describe some related work and how our research is different from their work. The previous work which addresses the related issues is Pipeline Generation for Loops [17], CoDe-X Framework [8], Dynamic Circuit Simulation [11], Virtual Pipelines [10], TMFPGA [14]. Though most of the projects address similar issues, the framework of developing an abstract model for solving general mapping problems is not fully addressed by any specific work.

2 Model

We present a parameterized model of a configurable computing system, which consists of configurable logic attached to a traditional microprocessor. This model can be utilized for analyzing application tasks, as regards to their suitability for execution on configurable logic and also for developing the actual mapping and scheduling of these tasks onto the configurable system.

We first describe our model of configurable architectures and then discuss the components of the model and how they abstract the actual features of configurable architectures.

2.1 Hybrid System Architecture Model(HySAM)

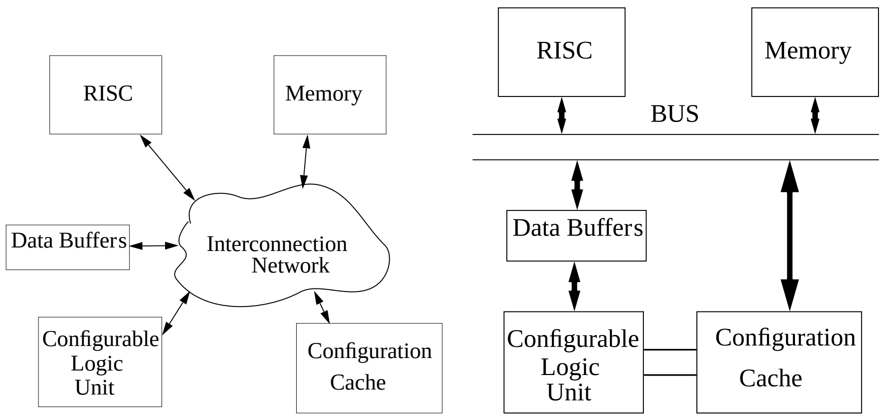

Fig. 1. Hybrid System Architecture and an example architecture

The *Hybrid System Architecture* is a general architecture consisting of a traditional RISC microprocessor with additional Configurable Logic Unit(CLU). Figure 1 shows the architecture of the HySAM model and an example of an actual architecture. The architecture consists of a traditional RISC microprocessor, standard memory, configurable logic, configuration memory and data buffers communicating through an interconnection network.

We outline the parameters of the Hybrid System Architecture Model(HySAM) below.

F	: Set of functions $F_1 \dots F_n$ which can be performed on configurable logic.
C	: Set of possible configurations $C_1 \dots C_m$ of the Configurable Logic Unit.
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.
N_c	: The number of configuration contexts which can be stored in the configuration cache.
k, K	: The reconfiguration time spent in configuring from the cache and external memory respectively.

W, D : The Width and Depth of the configurable logic which describe the amount of configurable logic available.

w : The granularity of the configurable logic which is the width of an individual functional unit.

S : The schedule of configurations which execute the input tasks.

E : Execution time of a sequence of tasks, which is the sum of execution time of tasks in the various configurations and the reconfiguration time.

The parameterized HySAM which is outlined above can model a wide range of systems from board level architectures to systems on a chip. Such systems include SPLASH [3], DEC PeRLE [16], Oxford HARP [9], Berkeley Garp [7], NSC NAPA1000 [15] 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 would have smaller size configurable logic(lower W and D) than an board level architecture but would have potentially faster reconfiguration times(lower k and K).

The model does not encompass the memory access component of the computation in terms of the memory access delays and communication bandwidth supported. Currently, it is only assumed that the interconnection network has enough bandwidth to support all the required data and configuration access. For a detailed description of the model and its parameters see [2].

3 Loop Synthesis

It is a well known rule of thumb that 90% of the execution time of a program is spent in 10% of the code. This code usually consists of repeated executions of the same set of instructions. The typical constructs used for specifying iterative computations in various programming languages are DO, FOR and WHILE, among others. These are generally classified as LOOP constructs.

Computations which operate on a large set of data using the same set of operations are most likely to benefit from configurable computing. Hence, loop structures will be the most likely candidates for performance improvement using configurable logic. Configurations which execute each task can be generated for the operations in a loop. Since each operation is executed on a dedicated hardware configuration, the execution time for the task is expected to lower than that in software. Each of the operations in the loop statement might be a simple operation such as an addition of two integers or can be a more complex operation such as a square root of a floating point number. The problems and solutions that we present are independent of the complexity of the operation.

3.1 Linear Loop Synthesis

The problem of mapping operations(tasks) of a loop to a configurable system involves not only generating the configurations for each of the operations, but also reducing the overheads incurred. The sequence of tasks to be executed have to be mapped onto a sequence of configurations that are used to execute these tasks. The objective is to reduce the total execution time.

Scheduling a general sequence of tasks with a set of dependencies to minimize the total execution time is known to be an NP-complete problem. We consider the problem of generating this sequence of configurations for loop constructs which have a sequence of statements to be executed in linear order. There is a linear data or control dependency between the tasks. Most loop constructs, including those which are mapped onto high performance pipelined configurations, fall into such a class.

The total execution time includes the time taken to execute the tasks in the chosen configurations and the time spent in reconfiguring the logic between successive configurations. We have to not only choose configurations which execute the given tasks fast, but also have to reduce the reconfiguration time. It is possible to choose one of many possible configurations for each task execution. Also, the reconfiguration time depends on the choice of configurations that we make. Since reconfiguration times are significant compared to the task execution times, our goal is to minimize this overhead.

Problem : Given a sequence of tasks of a loop, T_1 through T_p to be executed in linear order(T_1 T_2 ... T_p), where $T_i \in F$, for N number of iterations, find an optimal sequence of configurations S (=C_1 C_2 ... C_q), where $S_i \in C$ (=$\{C_1,C_2,\ldots,C_m\}$) which minimizes the execution time cost E. E is defined as

$$E = \sum_{i=1}^{q}(t_{S_i} + \Delta_{ii+1})$$

where t_{S_i} is execution time in configuration S_i and Δ_{ii+1} is the reconfiguration cost which is given by R_{ii+1}.

3.2 Optimal Solution for Loop Synthesis

The input consists of a sequence of statements $T_1 \ldots T_p$, where each $T_i \in F$ and the number of iterations N. We can compute the execution times t_{ij} for executing each of the tasks T_i in configuration C_j. The reconfiguration costs R_{ij} can be pre-computed since the configurations are known beforehand. In addition there is a loop setup cost which is the cost for loading the initial configuration, memory access costs for accessing the required data and the costs for the system to initiate computation by the Configurable Logic Unit. Though, the memory access costs are not modeled in this work, it is possible to statically determine the loop setup cost.

A simple greedy approach of choosing the best configuration for each task will not work since the reconfiguration costs for later tasks are affected by the choice of configuration for the current task. We have to search the whole solution space by considering all possible configurations in which each task can be executed. Once an optimal solution for executing up to task T_i is computed the cost for executing up to task T_{i+1} can be incrementally computed.

Lemma 1. *Given a sequence of tasks $T_1'T_2'\ldots T_r'$, an optimal sequence of configurations for executing these tasks* **once** *can be computed in $O(rm^2)$ time.*

Proof: Using the execution cost definition we define the optimal cost of executing up to task T_i' ending in a configuration C_j as E_{ij}. We initialize the E values as $E_{0j} = 0,\ \forall\ j:\ 1 \le j \le m$.

Now for each of the possible configurations in which we can execute T_{i+1}' we have to compute an optimal sequence of configurations ending in that configuration. We compute this by the recursive equation:

$$E_{i+1j} = t_{i+1j} + min_k(E_{ik} + R_{kj}) \quad \forall\ j:\ 1 \le j \le m$$

We have examined all possible ways to execute the task T_{i+1}' once we have finished executing T_i'. If each of the values E_{ik} is optimal then the value E_{i+1j} is optimal. Hence we can compute an optimal sequence of configurations by computing the E_{ij} values. The minimum cost for the complete task sequence($T_1'T_2'\ldots T_r'$) is given by $min_j[E_{rj}]$. The corresponding optimal configuration sequence can be computed by using the E matrix.

We can use dynamic programming to compute the E_{ij} values. Computation of each value takes $O(m)$ time as there are m configurations. Since there are $O(rm)$ values to be computed, the total time complexity is $O(rm^2)$. $\odot$

Lemma 1 provides a solution for an optimal sequence of configurations to compute one iteration of the loop statement. But repeating this sequence of configurations is not guaranteed to give an optimal execution for N iterations. Figure 2 shows the configuration space for two tasks T_1 and T_2 and four possible configurations C_1, C_2, C_3, C_4. T_1 can be executed in C_1 or C_3 and task T_2 can be executed in C_2 or C_4. The edges are labeled with the reconfiguration costs and cost for the edges and configurations not shown is very high. We can see that an optimal sequence of execution for more than two iterations will be the sequence $C_1\ C_4\ C_3\ C_2$ repeated $N/2$ times. The repeated sequence of $C_1\ C_4$ which is an optimal solution for one iteration does not give an optimal solution for N iterations.

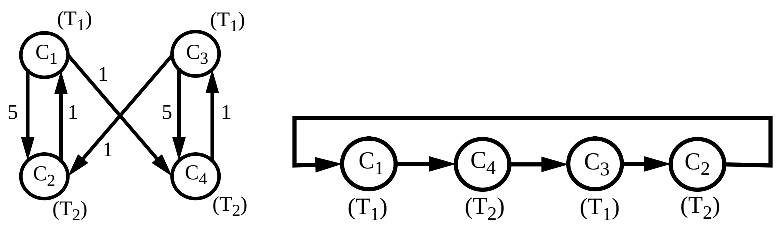

Fig. 2. Example reconfiguration cost graph and optimal configuration sequence

One simple solution is to fully unroll the loop and compute an optimal sequence of configurations for all the tasks. But the complexity of algorithm will be $O(Npm^2)$, where N is the number of iterations. Typically the value of N would be very high(which is desirable since higher value of N gives higher speedup compared to software execution). We assume $N \gg m$ and $N \gg p$. We show that an optimal configuration sequence can be computed in $O(pm^3)$ time.

Lemma 2. *An optimal configuration sequence can be computed by unrolling the loop only m times.*

Proof: Let us denote the optimal sequence of configurations for the fully unrolled loop by $C_1 C_2 \ldots C_x$. Since there are p tasks and N iterations $x = N * p$. Configuration C_1 executes T_1, C_2 executes T_2 and so on. Now after one iteration execution, configuration C_{p+1} executes task T_1 again. Therefore, task T_1 is executed in each of configurations C_1, C_{p+1}, C_{2*p+1}, $\ldots$, C_{x-p+1}. Since there are at most m configurations for each task, if the number of configurations in C_1, C_{p+1}, C_{2*p+1}, $\ldots$, C_{x-p+1} is more than m then some configuration will repeat. Therefore, $\exists\ y\ s.t.\ C_{y*p+1} = C_1$.

Let the next occurrence of configuration C_1 for task T_1 after C_{y*p+1} be C_{z*p+1}. The subsequence $C_1\ C_2\ C_3 \ldots\ C_{y*p+1}$ should be identical to C_{y*p+1} $C_{y*p+2}\ \ldots\ C_{z*p+1}$. Otherwise, we can replace the subsequence with higher per iteration cost by the subsequence with lower per iteration cost yielding a better solution. But this contradicts our initial assumption that the configuration sequence is optimal. Hence the two subsequences should be identical. This does not violate the correctness of execution since both subsequences are executing a fixed number of iterations of the same sequence of input tasks. Applying the same argument to the complete sequence $C_1 C_2 \ldots C_x$, it can be proved that all subsequences should be identical.

The longest possible length of such a subsequence is $m * p$(p possible tasks each with m possible configurations). This subsequence of $m * p$ configurations is repeated to give the optimal configuration sequence for $N * p$ tasks. Hence, we need to unroll the loop only m times. $\odot$

Theorem 3. *The optimal sequence of configurations for N iterations of a loop statement with p tasks, when each task can be executed in one of m possible configurations, can be computed in $O(pm^3)$ time.*

Proof: From Lemma 2 we know that we need to unroll the loop only m times to compute the required sequence of configurations. The solution for the unrolled sequence of $m * p$ tasks can be computed in $O(pm^3)$ by using Lemma 1. This sequence can then be repeated to give the required sequence of configurations for all the iterations. Hence, the total complexity is $O(pm^3)$. $\odot$

The complexity of the algorithm is $O(pm^3)$ which is better than fully unrolling ($O(Npm^2)$) by a factor of $O(N/m)$. This solution can also be used when the number of iterations N is not known at compile time and is determined at runtime. The decision to use this sequence of configurations to execute the loop can be taken at runtime from the statically known loop setup and single iteration execution costs and the runtime determined N.

4 Illustrative Example

The techniques that we have developed in this paper can be evaluated by using our model. The evaluation would take as input the model parameter values and the applications tasks and can solve the mapping problem and output the sequence of configurations. We are currently building such a tool and show results obtained by manual evaluation in this section.

The Discrete Fourier Transform(DFT) is a very important component of many signal processing systems. Typical implementations use the Fast Fourier Transform(FFT) to compute the DFT in $O(N \log N)$ time. The basic computation unit is the butterfly unit which has 2 inputs and 2 outputs. It involves one complex multiplication, one complex addition and one complex subtraction.

There have been several implementations of FFT in FPGAs [12, 13]. The computation can be optimized in various ways to suit the technology and achieve high performance. We describe here an analysis of the implementation to highlight the key features of our mapping technique and model. The aim is to highlight the technique of mapping a sequence of operations onto a sequence of configurations. This technique can be utilized to map onto any configurable architecture. We use the timing and area information from Garp [7] architecture as representative values.

For the given architecture we first determine the model parameters. We calculated the model parameters from published values and have tabulated them in Table 1 below. The set of functions(F) and the configurations(C) are outlined in Table 1 below. The values of n and m are 4 and 5 respectively. The Configuration Time column gives the reconfiguration values R. We assume the reconfiguration values are same for same target configuration irrespective of the initial configuration. The Execution Time column gives the t_{ij} values for our model.

Function	Operation	Configuration	Configuration Time	Execution Time
F_1	Multiplication(Fast)	C_1	14.4 μs	37.5 ns
	Multiplication(Slow)	C_2	6.4 μs	52.5 ns
F_2	Addition	C_3	1.6 μs	7.5 ns
F_3	Subtraction	C_4	1.6 μs	7.5 ns
F_4	Shift	C_5	3.2 μs	7.5 ns

Table 1. Representative Model Parameters for Garp Reconfigurable Architecture

The input sequence of tasks to be executed is is the FFT butterfly operation. The butterfly operation consists of one complex multiply, one complex addition and one complex subtraction. First, the loop statements were decomposed into functions which can be executed on the CLU, given the list of functions in Table 1. One complex multiplication consists of four multiplications, one addition and one subtraction. Each complex addition and subtraction consist of two additions and subtractions respectively. The statements in the loop were mapped to multiplications, additions and subtractions which resulted in the task sequence T_m, T_m, T_m, T_m, T_a, T_s, T_a, T_a, T_s, T_s. Here, T_m is the multiplication task mapped to function F_1, T_a is the addition task mapped to function F_2 and T_s is the subtraction task mapped to function F_3.

The optimal sequence of configurations for this task sequence, using our algorithm, was C_1,C_3,C_4,C_3,C_4 repeated for all the iterations. The most important aspect of the solution is that the multiplier configuration in the solution is actually the slower configuration. The reconfiguration overhead is lower for C_2 and hence the higher execution cost is amortized over all the iterations of the loop. The total execution time is given by $N * 13.055$ μs where N is the number of iterations.

5 Conclusions

Mapping of applications in an architecture independent fashion can provide a framework for automatic compilation of applications. Loop structures with regular repetitive computations can be speeded-up by using configurable hardware. In this paper, we have developed techniques to map loops from application programs onto configurable hardware. We have developed a general Hybrid System Architecture Model(HySAM). HySAM is a parameterized abstract model which captures a wide range of configurable systems. The model also facilitates the formulation of mapping problems and we defined some important problems in mapping of traditional loop structures onto configurable hardware. We demonstrated a polynomial time solution for one important variant of the problem. We also showed an example mapping of the FFT loop using our techniques. The model can be extended to solve other general mapping problems. The application development phase itself can be enhanced by using the model to develop solutions using algorithm synthesis rather than logic synthesis.

The work reported here is part of the USC MAARC project. 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. In addition, partitioning and mapping issues in compiling onto reconfigurable hardware are also addressed. Some related results can be found in [1], [4], [5], [6].

References

1. K. Bondalapati and V.K. Prasanna. Reconfigurable Meshes: Theory and Practice. In *Reconfigurable Architectures Workshop, RAW'97*, Apr 1997.
2. Kiran Bondalapati and Viktor K. Prasanna. The Hybrid System Architecture Model (HySAM) of Reconfigurable Architectures. Technical report, Department of Electrical Engineering-Systems, University of Southern California, 1998.

3. D. A. Buell, J. M. Arnold, and W. J. Kleinfelder. *Splash 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, 1996.
4. S. Choi and V.K. Prasanna. Configurable Hardware for Symbolic Search Operations. In *International Conference on Parallel and Distributed Systems*, Dec 1997.
5. Y. Chung and V.K. Prasanna. Parallel Object Recognition on an FPGA-based Configurable Computing Platform. In *International Workshop on Computer Architectures for Machine Perception*, Oct 1997.
6. A. Dandalis and V.K. Prasanna. Fast Parallel Implementation of DFT using Configurable Devices. In *7th International Workshop on Field-Programmable Logic and Applications*, Sept 1997.
7. 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.
8. R. Kress, R.W. Hartenstein, and U. Nageldinger. An Operating System for Custom Computing Machines based on the Xputer Paradigm. In *7th International Workshop on Field-Programmable Logic and Applications*, pages 304–313, Sept 1997.
9. 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.
10. W. Luk, N. Shirazi, S.R. Guo, and P.Y.K. Cheung. Pipeline Morphing and Virtual Pipelines. In *7th International Workshop on Field-Programmable Logic and Applications*, Sept 1997.
11. P. Lysaght and J. Stockwood. A Simulation Tool for Dynamically Reconfigurable FPGAs. *IEEE Transactions on VLSI Systems*, Sept 1996.
12. Xilinx DSP Application Notes. The Fastest FFT in the West, http://www.xilinx.com/apps/displit.htm.
13. 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.
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. NSC NAPA 1000 URL. http://www.national.com/appinfo/milaero/napa1000/.
16. 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.
17. M. Weinhardt. Compilation and Pipeline Synthesis for Reconfigurable Architectures. In *Reconfigurable Architectures Workshop RAW'97*. ITpress Verlag, Apr 1997.

This article was processed using the LaTeX macro package with LLNCS style