Recursive Variable Expansion: A Loop Transformation for Reconfigurable Systems

Zubair Nawaz, Ozana Silvia Dragomir, Thomas Marconi, Elena Moscu Panainte,
Koen Bertels, Stamatis Vassiliadis
Computer Engineering Lab
Delft University of Technology, The Netherlands
{Z.Nawaz, O.S.Dragomir, T.M.Thomas, E.Moscu-Panainte, K.L.M.Bertels, S.Vassiliadis}@tudelft.nl

Abstract

Loops are an important source of performance improvement, for which there exists a large number of compiler based optimizations. Few optimizations assume that the loop will be fully mapped on hardware. In this paper, we discuss a loop transformation called Recursive Variable Expansion, which can be efficiently implemented in hardware. It removes all the data dependencies from the program and then the parallelism is only bounded by the amount of resources one has. To show the performance improvement and the utilization of resources, we have chosen four kernels from widely used applications (FIR, DCT, Sobel edge detection algorithm and matrix multiplication). The hardware implementation of these kernels proved to be 1.5 to 77 times faster (depending on application) than the code compiled and run on PowerPC.

1. Introduction

Loops are an important source of performance improvement, for which there exists a large number of compiler optimizations [11, 6, 3]. A major performance can be achieved through *loop parallelization*. A major obstacle to obtain parallelism is due to dependencies in the program, one has to reorganize the program to reduce or remove the dependencies in order to get maximum parallelism. In this paper, we have tried to remove the data dependencies and our primary goal is to see the maximum parallelism we can achieve if we somehow defy the data dependencies, assuming we have unlimited resources on Reconfigurable Computing.

The contributions of this paper are (a) a loop transformation technique for reconfigurable systems called **Recursive Variable Expansion** (RVE), which removes all the data dependencies from the program; then, the parallelism is only bounded by the amount of resources one has. (b) We obtained speedups of up to 77 times on Virtex II Pro platform FPGA against the software only implementation for the considered kernels running on PowerPC processor.

The Delft Workbench [1] is a semi-automatic platform for integrated hardware-software co-design targeting heterogeneous computing systems containing reconfigurable components. The Delft Workbench targets the MOLEN [9] machine organization. This paper focuses on the loop parallelization in compiler optimizations part of the Delft Workbench.

Many projects have focused on semi-automatic translation of high level language like C to HDL with minimum intervention from the programmer. Few to name are GARP [5], ROCCC [3], DEFACTO [6] and SPARK [4]. In all these frameworks, researchers have tried to optimize the compiled code using techniques developed for Multiprocessor computers. Prefix computation is also used for loop parallelization [10]. This technique is quite useful for loop-carried dependencies.

The rest of the paper is organized as follows. Next section presents a motivational example. Section 3 formally describes the RVE along with its benefits and limitations. Section 4 presents the experimental setup, kernels used and results obtained which compares the performance of RVE with software only implementation. Finally we conclude the paper in Section 5.

2. Motivational Example

We will use Example 1 to understand the proposed loop transformation. Figure 1 shows the iteration space [11] with dependencies among the various iterations of the loop nest. As seen from the dependency graph in Figure 1, none of the loops can be parallelized. Statement reordering is also not possible due to $S_1 \delta S_2$.

Applying Loop Skewing Transformation. As shown in Figure 1, if the sequence to access the array A is changed

Example 1 Motivational Example

```
for i = 2 to n
  for j = 2 to m
S1  A[i, j]= A[i-1, j] + c
S2  B[i, j]= A[i, j]+ A[i, j-1]
  end for
end for
```

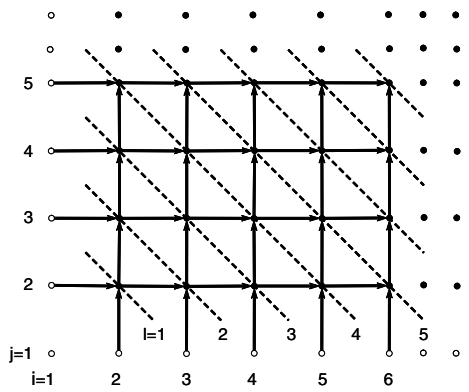

Figure 1. Iteration Space showing dependencies among the iterations. White points are inputs and black points are computed.

to a sequence followed by slanted dashed lines with trailing values of *l*, then it does not violate the dependencies between the iterations and it also keeps the same program behavior. This particular type of transformation is called loop skewing[11]. When it is applied we get Example 2. Now inner loop is parallelizable. It is assumed that each addition takes one cycle and the area required on FPGA is directly proportional to number of terms to be added.

Time estimate for Loop Skewing. The inner loop is expanded fully on to the FPGA and is executed in parallel. The outer loop is iterated *m+n-1* times and the time for each iteration of the outer *I* loop is *2 cycles,* therefore the total time to execute the transformed code is *2(m+n-1) cycles*.

Area estimate for Loop Skewing. The maximum number of terms to be added is *4*maximum number of iterations of the inner loop* for some outer loop. The lower bound for

Example 2 After Loop Skewing

```
for I = 2 to m+n
  for J = max(2,I-n) to min(m,I)
S1  A[I-J, J]= A[I-J-1, J] + c
S2  B[I-J, J]= A[I-J, J]+ A[I-J, J-1]
  end for
end for
```

Example 3 Recursively substituting the values

```
A[4, 3]= A[3, 3] +c
       = A[2, 3] +c +c
       = A[1, 3] +c +c +c
```

the inner loop remains *2* until *I* becomes *n+2* and the upper bound for the same *I* becomes *min(m, n+2)*. Using this upper and lower bound, the number of iterations for the inner loop is *min(m, n+2)-1*. When the lower bound is increased with the increase of *I*, the upper bound is increased with the same amount if $I < m$, or it remains *m*, if $m < I$, which shows that the maximum number of iterations for the inner loop is *min(m, n+2)-1*. So the maximum terms that need to be added are *4*min(m, n+2)-1*.

Applying RVE Transformation. Let us take the same example and fully unroll both loops. If we look at Example 1 with *i=4* and *j=3* (see Example 3), A[3, 3] is needed to compute A[4, 3], however if we replace A[3, 3] with its computation, then A[4, 3] is no more dependent on A[3, 3], but rather on A[2, 3], and if we repeat this procedure until A[4, 3] is only the function of input variables, then it can be computed without waiting for other results, if provided enough resources. The same can be done for every statement.

Area estimate for RVE. The number of terms to be added in Example 1 can be represented by a recurrence equation. Given that the statement S_1 is only dependent on *i*, suppose the number of addition terms in S_1 is denoted by $T(i)$ and is given by $T(i) = T(i-1) + 1$. The solution to this equation is $T(i) = i$. Similarly, suppose the number of addition terms in S_2, denoted by $S(i)$ is given by $S(i) = 2T(i)$, which solves to $S(i) = 2i$, so the number of addition terms for both the statements is $\tau(i) = S(i) + T(i) = 3i$. If we expand both the statements for all iterations until they cannot be expanded further, then the total number of addition terms τ is given by

$$\tau = \sum_{i=2}^{n}\sum_{j=2}^{m}\tau(i) = \frac{3}{2}(m-2)(n^2+n-2) = O(n^2m) \quad (1)$$

Suppose the variable which is used later in the program is only B[n, m], then there is no need to expand all the terms in all iterations on to FPGA, because after expansion, B[n, m] is a function of only inputs, which can be computed readily. Then the total number of addition terms is $\tau = S(n) = 2n$.

Time estimate for RVE. The expressions expanded in width contain only addition operators and there are no dependencies between them. The addition operator is associative, therefore the most efficient way to add them efficiently is adding in parallel in a complete binary tree fashion, where each level takes one cycle. Then $O(\log n)$ cycles is required to add *n* terms. Since all the terms are expanded

and executed in parallel then maximum time taken by any statement will be the one which has maximum number of addition terms, which in this example is *2n*. So the overall time will be $O(\log n)$ cycles.

It shows that the time to compute the same result is $O(\log n)$ cycles in RVE as compared to linear time in *m* or *n* for loop skewing. This speed up is at the expense of larger area.

3. Recursive Variable Expansion

The basic idea of the transformation is that for any two statements S_i and T_j for some iterations *i* and *j*, such that $S_i \delta T_j$ then, instead of T_j waiting for the computation done in S_i, that computations could be replicated in T_j, which would make it independent of S_i. Recursively repeat this procedure for every statement until there is no dependency on any statement. Rather every statement is a function of inputs only, which can be easily computed in parallel. So at the cost of redundant computations we can remove the dependencies among all statements making it suitable for unbounded parallelism. The transformation is named *Recursive Variable Expansion*, because all variables that create dependencies are recursively substituted with their values. Even though an algorithm was developed independently, the credit goes to [8], as it was initially published in 1972.

The suggested loop transformation is applied in the following steps. (1) Profile the application to identify CPU intensive loops. (2) The identified loops are fully unrolled. (3) The values for every assignment statement are Recursively substituted. (4) Constant folding is applied and then any possible computation during the compile time is done to reduce the computation at runtime.

Tree height reduction algorithm is applied [8]. The resulted tree will provide the sequence to parallelize the statements with least number of levels.

The success of RVE depends on the statements having associative binary operators. Because once such statements are expanded in width and are functions of known and independent variables, then these associative operators allow us to largely parallelize the operations by applying the tree height reduction. Even if there are some non associative operators, they can be handled in a special way [8]. An expression of *n* terms and depth *d* of parenthesis can be computed in $O(1 + 2d + \lceil \log_2 n \rceil)$ levels using at most $\lceil \frac{n-2d}{2} \rceil$ processing elements [7].

Like many other loop optimizations, a number of restrictions apply: (a) For some statements the terms expand so fast that the tree height reduction produces speedup equal to single processor despite the usage of large area. (b) The bounds for the loops must be known during the compilation. (c) This technique is suitable only for loop nest having limited control dependency or no control dependency.

If these restrictions are satisfied, the following benefits hold: (a) It removes all the data dependencies from the part of the program on which it is applied, then the parallelism is only bounded by the amount of resources one has. (b) It minimizes the memory accesses as initially all the input variables are read once and then all the computations are done using those values and finally the output is written back to memory. (c) This single technique exploits more parallelism with out making wide selection and scheduling other loop transformations. (d) This transformation can also work with non perfectly nested and unnormalized loops.

4. Experimental Results

Integer only implementations for the following four representative kernels are taken from three different application domains. (1) SOBEL is a 3×3 convolution mask over an integer image. (2) MM is a 16-bit integer matrix multiplication of a 12×6 matrix by a 6×4 matrix. (3) FIR is a finite impulse response filter with 16 tap over 32 consecutive 8-bit elements. (4) DCT is an integer implementation of 2D Discrete Cosine Transform for 8×8 block.

4.1. Software and Hardware Implementation

For software only implementation, the kernels written in C are compiled using GCC 4.2.0 with level 3 optimization. The compiled codes are simulated for IBM PowerPC 405 processor immersed into the FPGA fabric, which runs at 250MHz. To estimate the time taken by the software only implementation, the machine instructions are counted and segregated into computation and memory access instructions using PSIM simulator [2].

For hardware implementation, we applied RVE as mentioned in Section 3, which outputs only those statements that are used later in the program. VHDL code was then written for these expressions. To take the advantage of the parallelism in the VHDL code, all the input variables required to compute the statements are read from memory and stored in registers on the FPGA. Then all possible computations are performed in parallel according to sequence provided by tree height reduced expression. Finally the results are written back to memory. We have used two memory models: on chip memory and the DDR on the board.

The FCCM used in our experiment is Molen prototype implemented on the Virtex II Pro platform FPGA of Xilinx as described in [9]. Xilinx ISE version 8.2.022 , XST and ModelSIM SE are used to generate, synthesis and simulate the VHDL respectively.

	Pure Software Execution					Hardware Execution on Virtex II Pro platform								
		On Chip		DDR				On Chip			DDR			
	Computation (cyc)	Mem (cyc)	Total (cyc)	Mem (cyc)	Total (cyc)	Frequency (MHz)	Computation (cyc)	Mem (cyc)	Total (cyc)	Speed up	Mem (cyc)	Total (cyc)	Speed up	Area (Slices)
Sobel	59	84	143	560	619	127.14	4	43	92.4	1.53	215	430.64	1.44	541
MM	797	1224	2021	8160	8957	118.67	2	102	219.1	9.22	966	2039.3	4.39	64792
FIR	1025	1536	2561	10240	11265	129.60	2	32	65.6	39.05	286	555.55	20.28	13850
DCT	6439	9147	15586	60980	67419	110.81	3	86	200.8	77.63	966	2186	30.84	1067552

Table 1. Performance and Area Utilization for Software only and Virtex II platform

4.2. Results

Each kernel is executed on the PowerPC, we refer to this as the pure software execution. It is also executed on the FPGA referred to as hardware execution (see Table 1). The time for the execution on the Virtex II Pro platform is normalized according the clock speed of PowerPC 405 to make a fair comparison. The speedup for the hardware implementation compared to the software one for the given kernels is between 1.5 and 77 times. A large gain is achieved in the computation time, but also the memory access is improved as all the inputs are read only once as a whole block then all subsequent computations are done and finally results are written back as a block. Secondly, the variables are stored in the registers in FPGA, from where data can be accessed efficiently. Currently, the bottleneck is memory transfer, so the speedup can be enhanced further if the time to access the memory is improved. The area occupied by the first three kernels is well within the area capacity of current FPGAs (like Virtex 4-XC4VLX200 contains 89,088 slices). However, the area covered by the DCT is around 10 times more than the currently available FPGAs. Even though current FPGA technology cannot satisfy such area requirements, future technology (following Moore's law) will be able to do so. The area estimates for DCT are therefore obtained through extrapolation. The time estimates for DCT are made by mapping only one out of 64 elements on the FPGA and taking that time for the whole block of 64 elements.

5. Conclusions and Future Research

In this paper, we have presented Recursive Variable Expansion, a loop transformation for reconfigurable systems. This transformation is applied to the most frequently executed code and the resulted code is translated to VHDL to be implemented efficiently on FPGA. This transformation removes all the dependencies in the applied region, therefore is highly parallelizable. This loop transformation is applied on four kernels from various applications and the generated code is implemented on Molen prototype and it shows speedup ranging from 1.5 to 77 times. This performance gain is at the cost of more area on the FPGA. Three out of four kernels can be mapped on the current FPGAs. The main objective of the paper is to minimize the computation time, therefore trade-off is not made to balance the computation and memory access time. Future work will involve relaxing the restrictions imposed by the algorithm by including loops with extensive control structure and restricting area as on the current FPGAs.

References

[1] Delft workbench. online: http://ce.et.tudelft.nl/dwb.

[2] Psim. online: http://sourceware.org/psim/.

[3] Roccc. online: `http://www.cs.ucr.edu/~roccc/`.

[4] Spark: A parallelizing approach to the high level synthesis of digital circuits. online: http://mesl.ucsd.edu/spark/.

[5] T. Callahan, J. Hauser, and J. Wawrzynek. The garp architecture and c compiler. *IEEE Computer*, (4):62–69, April 2000.

[6] P. Diniz, M. Hall, B. Park, J. So, and H. Ziegler. Bridging the gap between compilation and synthesis in the defacto system. *LECTURE NOTES IN COMPUTER SCIENCE*, (2624):52–70, April 2003.

[7] M. Y. Kuck, D.J. Bounds on the parallel evaluation of arithmetic expressions using associativity and commutativity. *Acta Informatica*, 3:203–216, September 1974.

[8] M. Y. S.-C. C. Kuck, D.J. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *Transactions on Computers*, C-21:1293–1310, 1972.

[9] S. Vassiliadis, S. Wong, and S. D. Cotofana. The molen $\rho\mu$-coded processor. In *11th International Conference on FPL*, pages 275–285, August 2001.

[10] H. Wang, A. Nicolau, and K.-Y. S. Siu. The strict time lower bound and optimal schedules for parallel prefix with resource constraints. *IEEE Transactions on Computers*, 45(11):1257–1271, 1996.

[11] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.