Nested Loops Optimization for Multiprocessor Architecture Design

Andrea Leonardi    Nelson L. Passos
Dept. of Computer Science
Midwestern State University
Wichita Falls, TX 76308

Edwin H-M. Sha
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, 46556

Abstract

Multi-dimensional systems, including image processing, geophysical signal processing, and fluid dynamics, are becoming one of the most important targets of computational improvement studies. Most of the optimized solutions to those problems point to the use of Application Specific Integrated Circuits (ASICs). From the analysis of the multi-dimensional programming code, one can observe that nested loop like structures are often the most time consuming part. Designing ASICs with multiple processing units is usually the appropriate solution to achieve the required computational performance. In this paper, a new loop transformation algorithm, which allows an efficient utilization of the multiprocessor system is presented. Uniform nested loops are modeled as multi-dimensional data flow graphs. New loop structures are generated so that an arbitrary number of processors available in the system can run in parallel. An example demonstrates the effectiveness of the algorithm.

1. Introduction

Multi-dimensional systems, including image processing, geophysical signal processing, and fluid dynamics, are becoming one of the most important targets of computational improvement studies. Most of the optimized solutions to those problems point to the use of Application Specific Integrated Circuits (ASICs). In order to obtain a high degree of computational speed, parallel architectures are commonly utilized in the ASIC design. In such cases, the partitioning of the problem becomes a significant part of the design process. However, most of the existing synthesis tools are not able to deal with complex control structures such as nested loops. Loop constructs coded in VHDL usually represent how many times a section of the circuit was replicated in the final synthesis [15]. If multiple software units (microprocessors) were available as possible components of the new design, then a refinement of the loop specification would allow a better synthesis of the problem. This paper introduces a new approach on partitioning nested loops in order to obtain parallelism across distinct iterations and improve the schedule of those software units.

Most of the previous studies on VHDL synthesis have focused on standards [2], formal verification [3], extensions to support analog components [16], or target technologies, such as FPGA, CMOS, etc. [7]. The development of application specific hardware description languages, such as Silage, HIFI, and others [5, 6, 10], aimed to the synthesis of multi-dimensional signal processing applications do not consider the possible parallelism across loop iterations. Others techniques have been developed to be applied when writing the VHDL code in order to overcome possible synthesis constraints [17]. Since multi-dimensional applications are highly dependent on the execution of nested loops, in this paper we present a technique, which analyzes such loops, splitting them in sections that can run in parallel.

Loop transformations have also been applied in other areas, such as scheduling and parallel processing to speed up the execution time of nested loops [8, 9, 14, 18]. Most of these techniques target the fine-grain parallelism that can be found in the loop body and should be considered complementary to our method. Aiken and Nicolau developed a loop parallelization technique that examines a partial execution history of a loop and schedules the statements of those iterations with a greedy algorithm [1]. However, the search for a repeating pattern in a single loop has the disadvantages of unpredictability of the size of such pattern and the gain of performance. Manjikian and Abdelrahman suggested a code transformation called shift-and-peel to fuse loops and allow parallel execution [11]. However, when the number of peeled iterations exceeds the number of iterations per processor, this method is not efficient.

Callahan proposed loop alignment to allow synchronization-free parallel execution of a loop nest, and uses code replication to resolve conflicts in alignment requirements [4]. This technique may result in an exponential growth in the size of loop bodies to address such alignment conflicts, and the execution of replicated code is a significant source of overhead.

Passos, Sha and Bass proposed a partitioning and retiming method for multi-dimensional systems [13]. However, the dependencies within a node, and the
This paper presents a new loop transformation method based in the multi-dimensional retiming technique and its applicability to the design of multiprocessor systems. It begins by showing the basic concepts in Section 2, including an overview on the multi-dimensional retiming technique. Section 3 shows the theoretical aspects on the processor allocation technique. Section 4 describes the final results in a simple example, while a final section summarizes the concepts introduced in this paper.

2. Basic Principles

A multi-dimensional data flow graph (MDFG) \( G = \langle V, E, d, t \rangle \) is a node-weighted and edge weighted directed graph, where \( V \) is the set of computation nodes, \( E \subseteq V \times V \) is the set of dependence edges, \( d \) is a function that returns the multi-dimensional delay between two nodes, and \( t \) is a function that returns the computational time of each node.

Iterations are described by a vector \( i \), equivalent to a multi-dimensional index, starting at \((0,0,...,0)\). Iteration dependencies are represented by vector-weighted edges. An edge \( e \) from \( u \) to \( v \) with delay vector \( d(e) \) means that the computation of the node \( v \) at iteration \( k \) depends on the execution of node \( u \) at iteration \( k - d(e) \). An edge with delay \((0,0,...,0)\) represents a data dependence within the same iteration. We will consider loops that present the characteristic of constant dependence vectors, i.e., their data dependencies are at a constant distance in the iteration space; those loops are called uniform loops.

A simple example consists of a filter defined by the function:

\[
H(z_1, z_2) = \frac{1}{(1 - C_1 z_1^{-1} - C_2 z_2^{-1})}
\]

which could be described by:

\[
\begin{align*}
&\text{for } (i = 1; i \leq 1000; i++) \\
&\quad \text{for } (j = 1; j \leq 1000; j++) \\
&\quad \quad y(i, j) = c1 * y(i-1, j) + c2 * y(i, j-1) + x(i, j);
\end{align*}
\]

This problem could be also represented by a multi-dimensional data flow graph as shown in Fig. 1. Nodes in the graph represent operations and edges represent data dependencies. Specifically, node B is equivalent to \( c1y(i-1, j) \), C represents \( c2y(i, j-1) \), D is the sum of B and C, while A is D added to \( x(i,j) \). The labels on the edges indicate the multi-dimensional distance between iterations. Notice that dependence vectors are represented by pairs \((d_x, d_y)\) where \( d_x \) corresponds to the dependence distance in the Cartesian axis representing the outermost loop, while \( d_y \) corresponds to the innermost loop. This initial representation is the same as the lexicographic order proposed by Lamport in [8]. In this example the MDFG consists of \( V = \{A, B, C, D\} \), \( E = \{e1 : (A, B), e2 : (A, C), e3 : (C, D), e4 : (D, A), e5 : (B, D)\} \), for \( d(e1) = (0, 0) \), \( d(e2) = (0, 0) \), \( d(e3) = (0, 1) \), \( d(e4) = (0, 0) \), \( d(e5) = (1, 0) \). For simplicity, each function is assumed to be executed in one time unit; therefore, \( t(\text{node}) = 1 \). In our approach, considering a target design consisting of four processors, the code is divided in four loops responsible for the parallel execution of the iterations. At the end of each iteration, a shared memory system or a message passing mechanism is used to satisfy future data dependencies.

![Fig. 1 MDFG representing a simple filter](image)

Multi-Dimensional Retiming

A multi-dimensional retiming \( r \) is a function from \( V \) to \( Z^n \) that redistributes the nodes in an iteration space created by the replication of an MDFG. A new MDFG \( G_r \) is produced, such that each iteration still has one execution of each node in \( G \). The retiming vector \( r(u) \) of a node \( u \in G \) represents the offset between the original iteration containing \( u \), and the one after retiming. The delay vectors change accordingly to preserve data dependencies, i.e., \( r(u) \) represents delay components pushed into the edge \( u \to v \), and subtracted from edges \( w \to u \), where \( u, v, w \in G \). Therefore, we have \( dr(e) = d(e) + r(u) - r(v) \). As a consequence of the definition of
the retiming function, the following property can be observed:

Property 2.1: Given an MDFG \( G = (V, E, d, t) \) and \( k \) processors, a PDG \( P = (\Pi, \delta) \) is a node weighted and edge weighted directed graph, where \( \Pi \) is the set of processors, with \( |\Pi| = k \). \( \delta \) is the set of dependence edges between processors and \( \delta \) is the function representing the MD delay between two processors.

An MDFG is transformed to a processor dependency graph (PDG) according to the number of available processors established by the end-user of this technique. The PDG shows the dependency edges among the various processors. Fig. 2a shows a PDG with four processors aligned parallel to the x-axis for the MDFG shown in Fig. 1. In order to develop the transformation algorithm, we assume that the memory access time is uniform, and that the loop structures under consideration are uniform loops, i.e., their data dependencies are at a constant distance in the iteration space. The problem represented by the MDFG is translated into a PDG according to the property described below:

Property 3.1: Given an MDFG \( G = (V, E, d, t) \), and \( k \) processors, then the relative processor dependency graph PDG \( P = (\Pi, \delta) \), is such that \( \forall e \in E \) connecting nodes \( u, v \in V \) with \( d(e) = (x, y) \); there exists \( e' \in \delta \) connecting node \( u \) to node \( v \) of processor \( m \), where \( m, n \in \Pi, 1 \leq m, n \leq k \), such that:

\[
\begin{align*}
  & m = (n - 1 + x) \mod k + 1, \text{ and} \\
  & \delta(e') = (x', y') = (\text{int}( (n - 1 + x)/k), y)
\end{align*}
\]

A multi-dimensional retiming function \( r(u) = (0, r_\gamma) \) is applied to the PDG to change the dependence edges eliminating sequential processing among different processors. Full parallelism is achieved when all edges between any two processors become non-zero delay edges [12]. The retimed PDG for the example is shown in Fig. 2b. The loop bodies are modified according to the retiming function applied to the PDG. Fig. 3 shows the transformed code for this example, where the function SYNC does the synchronization of all processors when they reach the point where the prologues are done and the parallel execution can start.

![Fig. 2 (a) PDG (b) PDG after retiming](image)

Processor # n
\( k = 4; \) // number of processors
for ( \( i = n; i \leq 1000 ; i+k \))
  for ( \( j = 1; j \leq 1000 ; j+j \))
    if( (i==1) & & (j==r_\gamma +1) )
      SYNC
      \( y (i,j) = c1 * y(i-1,j)+c2*y(i,j-1)+x(i,j) \);

Fig. 3 Modified code for processor k.

After constructing the PDG representing the problem, there is a possibility that zero delay or negative delays (such as \((0,-1)\)) will appear in the PDG. If for all edges between any two processors the delay is a positive delay, then parallelism across loop iterations without retiming has already been achieved. Otherwise the multi-dimensional retiming technique must be applied to change the zero and negative delay edges into positive delays. Again, there is no concern about the edges within the same processor block since they will be executed sequentially by that processor. In order to produce the final code for the \( k \) processors, the same retiming function that is applied to the SYNC instruction in the body of the loop. The multiprocessor retiming algorithm can then be stated as follows:
Multiprocessor Retiming Algorithm MRA:
Input: MDFG G = (V, E, d, t), number of processors k;
PDG P = (Π, ε, δ) ← transform (G, k);

while ∃ p, u, v ∈ Π, u ≠ v | p → e1 u → e2 v, δ(e1) ≠ (0, 0), δ(e2) ≠ (0, 0),
     apply r(u) = r(v);
End MRA

4. Example

Considering again the filter example introduced in section 2, the edges between processors 1 and 2, 2 and 3, 3 and 4, have δ(e) = (0, 0). The multi-dimensional retiming function used in this example is r(processor #1) = (0,3), r(processor #2) = (0,2), and r(processor #3) = (0,1). To produce the final code for the processors, the same retiming function was applied to the SYNC instructions in the body of the loop.

This example demonstrates the effectiveness of this method in breaking down the original nested loops defining the multi-dimensional computations. This mechanism will allow a more efficient design of multiprocessor architectures using VLSI technology. Such algorithm can be easily implemented in high-level synthesis tools and used to achieve the target performance requirements.

5. Summary

This paper presented a new multi-dimensional retiming algorithm to exploit parallelism among loop iterations in a multiprocessor design environment. The loop is modeled by a multi-dimensional loop dependence graph. This new technique restructures the loop, splitting the original code in sections targeted to each of the processors available. Full parallelism of the loop body across loop iterations, i.e. the loop iterations executed in parallel, substantially decrease the overall computation time. The theory and the algorithm were presented in details.

6. Acknowledgements

This work was supported by the National Science Foundation under Grant No. MIP 9704276.

References