ABSTRACT:
Embedded systems need strict timing and code size requirements. Designing a embedded system is hard, since it is application specific, it executes the program repeatedly. Also it is tightly constrained with low cost, low power consumption, small size and fast execution. The system is reactive and real-time, it continually reacts to changes in the system's environment and it must compute certain results in real-time without delay. One of the techniques used to optimize the execution time is Multidimensional Retiming technique. It is a strategy for performing large and complex tasks faster. A large task can either be performed serially or can be decomposed into smaller tasks and to be performed simultaneously, i.e., in parallel. Software pipelining is one of the most important optimization techniques to improve the execution time of loops by increasing the parallelism among successive loop iterations while multidimensional retiming can use the outer loop parallelism. It introduces large overheads in loop index generation and code size due to transformation. This project uses algorithms of software pipelining for nested loops with minimal overheads based on the fundamental understanding of the properties of software pipelining for nested loops. The Chained MD Retiming or SPINE FULL algorithm generates fully parallelized loops with computation time and code size overheads as small as possible. Application chosen for parallel processing is industrial fire security system. In this system a sequential process can be broken into parallel tasks and each can be executed in parallel. The tasks are auto telephone dialer, counter for employee entry, exit and motor water sprayer. When ever the sensed temperature is above the required level the real time kernel responds to the temperature sensor, dial to the executive fire officer, count the employee using IR transmitter and start the motor to spray the water.

1 INTRODUCTION
With the advance of the technology, embedded systems with multiple cores or VLIW-like architectures, such as TI’s TMS320C6x, Philips’ TriMedia, and IA64, etc., become necessary to achieve the required high performance for the applications with growing complexity. To exploit multiple functional units or processors in parallel embedded systems, software pipelining is widely used to explore the instruction-level parallelism in loops. Code size is one of the most critical concerns for many embedded processors because the capacity of on-chip memory modules is still very limited due to the chip size, cost and power considerations. It shows that the code size of software-pipelined loop nests is greatly affected by the execution sequence and the software pipelining degree chosen by an optimization technique. An example of software pipelining on a nested loop is shown in Figure 1. Figure 1(a) shows the original loop nest. The schedule length of the loop body is 3 control steps. Figure 1(b) and Figure1(c) are software pipelined loops with the same schedule length of loop body which is 1 control step. However, the code in Figure 1(c) is much more complicated than that in Figure 1(b). The optimization technique generating the code in Figure 1(b) uses a row-wise execution sequence, while the technique generating the code in Figure 1(c) uses a skewed execution sequence which introduces large overheads in code size and loop indexes computation. Because of the space limitation, we do not show the code sections of epilogues which are about the same size as the prologues in this case. Optimization techniques on nested loops need to consider both timing and code size requirements, therefore, becomes a great challenge for parallel compiler. While software pipelining of single loops has been extensively studied and implemented, there is very few work done for the software pipelining problem on nested loops. A few existing techniques that could be applied to nested loop optimization either cannot fully explore the parallelism in a nested loop or do not consider the overheads such as loop indexes and loop bounds computation, and code size expansion due to transformation. The standard software pipelining techniques for single loops...
focuses on one-dimensional problems. When applied to nested loops, it only optimizes the innermost loop. While nested loops usually exhibit dependencies across loop dimensions. Therefore, the performance improvement that can be obtained by the standard software pipelining techniques is very limited. However, this technique makes code generation extremely difficult, and introduces large overhead in loop index generation. The best effort existing in industry on nested loop pipelining is to overlap the executions of the prologue and epilogue of the innermost loop, called outer loop pipelining. In this method, the dependencies among the outer loop iterations are still not exploited. Hence, the potential parallelism that can be explored is very limited. The only existing method that can fully explore the potential parallelism in multi-dimensional problems is multi-dimensional (MD) retiming.

Limitations of existing Technique

There are several techniques can be applied to optimize nested loops. However, they all have limitations in terms of performance improvement, code size, or complexity of code generation. By analyzing the limitations of the existing techniques, the issue of nested loop optimization becomes clear. The standard software pipelining techniques for one-dimensional loops are well developed. However, when they are applied to optimize nested loops, the performance improvement is very limited. Consider the MDFG in Figure 2(a) with cycle period $\Phi(G)=3$. The best result can be achieved by the standard software pipelining techniques, such as Modulo scheduling, is $\Phi(G)=2$, as shown in Figure 2(b). Actually, the MDFG can be fully-parallelized. Figure 2(c) shows the retimed MDFG by using our SPINE technique which exploit the outer loop dependencies. The detailed SPINE algorithms will be presented later.
3 Basic Principles
This section gives an overview of basic concepts and principles related to software pipelining problem for nested loops. These include multi-dimensional data flow graph, multi-dimensional retiming, and software pipelining. And demonstrate that retiming and software pipelining are essentially the same concept. A discussion of the limitations of the existing techniques for optimizing nested loops is also provided in the below sections.

3.1 Multi-Dimensional Data Flow Graph
A multi-dimensional data flow graph (MDFG) $G = (V, E, d, t)$ is a node-weighted and edge weighted directed graph, where $V$ is the set of computation nodes, $E \subseteq V \times V$ the set of edges representing dependencies, $d$ is a function from $E$ to $\mathbb{Z}^n$ representing the multi-dimensional delays between two nodes, where $n$ is the number of dimensions, and $t$ is a function from $E$ to $\mathbb{Z}^n$ to a set of positive integers, representing the computation time of each node. Programs with nested loops can be represented by an MDFG with cycles as shown in Figure 3(a).

Figure 3: (a) An MDFG. (b) Code of the nested loop. (c) The static schedule.
An iteration is the execution of each node in $V$ exactly once. Iterations are identified by a vector index $i = (i_1, i_2, i_3, \ldots, i_n)$, starting from $(0,0,\ldots,0)$ where elements are ordered from the innermost loop to outermost loop. Inter-iteration dependencies are represented by edges with delays. In particular, an edge $e(u \rightarrow v)$ with delay $d(e)$ indicates that the computation of node $v$ at $j$th iteration requires data produced by node $u$ at $(j-d(e))$th iteration. An inter-iteration dependency with all zero elements but the first element is called inner-loop dependency, which is the same as the dependency of a single loop. Otherwise, it is called an outer-loop dependency. In the case of nested loops, The dependencies within the same iteration are represented by edges without delay $(d(e)=(0,0,0,\ldots,0))$. A static schedule must obey these intra-iteration dependencies.

The cycle period of a DFG is defined as the computation time of the longest zero-delay path, which corresponds to the minimum schedule length when there is no resource constraint. Thus, the cycle period of the DFG in Figure 3(a) is 3. The iteration space of the two-dimensional data flow graph show in Figure 3(a) is shown in Figure 4(a). The marked cell is iteration $(0,0)$. The inter-iteration dependencies $B \rightarrow D$ and $C \rightarrow D$ go across loop dimensions. They are shown clearly in the corresponding cell dependency graph (CDG) in Figure 4(b), where nodes represent iterations, and edges represent inter-iteration dependencies. A correct execution sequence of a nested loop must obey inter-iteration dependencies in a CDG. For the CDG in Figure 4(b), a row-wise execution can correctly execute the nested loop. We define a schedule vector $s$ as a normal vector for a set of parallel hyper-planes (indicated by dotted lines in a CDG) that define the execution sequence.

Figure 4: (a) Iteration space of the MDFG in Figure 3(a). (b) The cell dependency graph.
For example, the schedule vector of the CDG shown in Figure 4(b) is $s=(0,1)$. Note that most of the nested loops are programmed in row-wise execution sequence. We will show later that a skew of schedule vector $s=(0,1)$ results in huge computation and code size overheads for
software pipelining on nested loops. Based on the definition of schedule vector, a legal execution sequence has to satisfy the following conditions: First, \( S.d(e) \geq 0; e \in E_c \) second, there must not exist any cycle in \( G_c \).

### 3.2 Multi-Dimensional Retiming and Software Pipelining

The retiming technique can be applied on a data flow graph to minimize the cycle period in polynomial time by evenly distributing the delays in the graph. The delays are moved around in the graph in the following way: a delay unit is drawn from each of the incoming edges of \( V \), and then added to each of the outgoing edges of \( V \), or vice versa. Note that the retiming technique preserves data dependencies of the original DFG. The multi-dimensional retiming (MD retiming) function \( r: V \rightarrow \mathbb{Z}^n_+ \) represents the number of delay units moved through node \( v \in V \).

**Figure 5:** (a) The retimed MDFG \( G_r \) of MDFG in Figure 2(a) with \( r(D)=(1,0) \) (b) Code of the retimed MDFG.

Figure 5(a) shows the retimed MDFG of Figure 3(a) with MD retiming function \( r(D)=(1,0) \). Consider a retimed DFG \( G_r=(V,E,dr,t) \) computed by retiming \( r \). The number of delays of any edge \( e \) (\( u \rightarrow v \)) after retiming can be computed as \( d_r(e)=d(e)+r(u)-r(v) \). And the total number of delays remains constant for any cycle in the graph. When a delay is pushed through node \( D \) to its outgoing edge as shown in Figure 5(a), the actual effect on the schedule of the new MDFG is that the \( i^{th} \) copy of \( D \) is shifted up and is executed with \((i-0,1)\)th copy of nodes A, B, and C. Because there is no dependency between node \( D \) and \( A \) in the new loop body, these two nodes can be executed in parallel. The schedule length of the new loop body is then reduced from three control steps to two control steps. This transformation is illustrated in Figure 3(c) and Figure 5(c). Note that the original zero-delay edge \( D \rightarrow A \) in Figure 3(a) has a delay \((1,0)\) after retiming. The cycle period is then reduced from 3 to 2 time units.

In fact, every retiming operation corresponds to a software pipelining operation. When \( r(u) \) delay units are pushed through a node \( u \), every copy of this node is moved by \( r(u) \) iterations. Hence, a new iteration consists of nodes from different iterations. Some nodes are shifted out of the loop body to be the prologue to provide the necessary data for the iterative process. Correspondingly, some nodes will be executed after loop body to complete the process. These are called epilogue. With MD retiming function \( r \), we can trace the pipelined nodes and also measure the size of prologue and epilogue. When \( r(v)=(r_x,r_y) \) delays are pushed through node \( v \), there are \( r_x \) copies of node \( v \) appeared in the prologue in \( x \)-dimension, and \( r_y \) copies of node \( v \) in the prologue in \( y \)-dimension. The number of copies of a node in the epilogue can also be derived in a similar way. For example, Figure 3(a) shows the MDFG of the code in Figure 3(b). Figure 5(a) shows the retimed graph for the software-pipelined loop in Figure 5(b) with \( r(D)=(1,0) \).

We can see from the iteration space shown in Figure 6(a) that there is exactly one copy of node 8 in the prologue in \( x \)-dimension. The corresponding cell dependency graph is shown in Figure 6(b).

**Figure 6:** (a) The iteration space the retimed MDFG in Figure 5(a). (b) The cell dependency graph.
A MDFG can always be fully parallelized by using MD retiming, which means that all the nodes in the MDFG can be executed in parallel if there is no resource constraint. For example, Figure 7(a) and Figure 7(b) show the fully-parallelized MDFG and its static schedule, respectively, with retiming function $r(D) = (2,0)$ and $rA = (1,0)$.

Figure 7: (a) A fully-parallelized graph of the MDFG in Figure 3(a). (b) The static schedule.

However, an arbitrarily chosen retiming may cause cycles in CDG of a multi-dimensional data flow graph, which is not executable. An illegal retiming is shown in Figure 7(a). By simple inspection of the CDG in Figure 7(b), we notice that a cycle is created by the dependencies $(1,0)$ and $(-1,0)$. Another interesting example is shown in Figure 8(a) which has two orthogonal delay vectors $d(B \rightarrow C) = (1,0)$ and $d(D \rightarrow A) = (0,1)$. The cell dependency graph is shown Figure 8(b).

Figure 8: (a) An MDFG. (b) The cell dependency graph.

In this case, the standard software pipelining can do nothing to optimize the loop even with loop index interchange. However, it can still be fully parallelized by the standard MD retiming, such as the chained MD retiming, as shown in Figure 9(a).

Figure 9: (a) A fully-parallelized MDFG using chained MD retiming. (b) The cell dependency graph.

The SPINE-FULL Algorithm aims to achieve a fully parallelized loop. Therefore, the desired cycle period is set to be for unit-time MDFG. First, the algorithm tries to use row-wise execution sequence, i.e. schedule vector $s$, to fully parallelize the MDFG. If it is not achievable, the algorithm tries the schedule vector $s$ if it is a legal schedule vector. The schedule vector $s$ represents an column-wise execution and can be transformed to row-wise execution by loop index interchange. The schedule vectors $(1,0)$ and $(0,1)$ are the ones that produce the smallest code size expansion and the least loop index and bounds computation. If these two schedule vectors are not achievable, then the algorithm will use chained MD retiming. In this way, we can achieve full parallelism with minimal code size overhead. Using our SPINE-FULL, we can fully parallelize the MDFG shown in Figure 5(a) with standard row-wise execution ($s$), so the code-size overhead is minimal, which cannot be achieved by another methods.

4 Application in real time system

Application chosen for parallel processing is industrial fire security system. In this system a sequential process can be broken into parallel tasks and each can be executed in parallel. The tasks are auto telephone dialer, counter for employee entry, exit and motor water sprayer. When ever the sensed temperature is above the required level the real time kernel responds to the temperature sensor, dial to the executive fire officer, count the employee using IR transmitter.
and start the motor to spray the water. In this application the MD retiming technique is implemented and the execution time is reduced in the real time system.

5 CONCLUSION
The existing techniques cannot optimize nested loops effectively for many embedded systems with strict timing and code size requirements. The standard software pipelining techniques can not fully explore the parallelism in nested loops. Multi-dimensional retiming, on the other hand, can fully parallelize a nested loop, but does not consider timing and code size overheads. Therefore, it is not suitable for software pipelining on nested loops. In this paper, we present theory and algorithms of nest-loop software pipelining with minimal overheads in loop index generation and code size based on the fundamental understanding of the properties of software pipelining problem on nested loops. Our SPINE-FULL algorithm tries to fully parallelize nested loops with a row-wise execution sequence whenever it is possible. For the loops cannot be fully parallelized using row-wise execution sequence, the algorithm quickly identifies them and generates full-parallel nested loop with overheads as small as possible. Our SPINE-ROW-WISE algorithm generates the best parallelized result with the minimal overheads using a row-wise execution sequence. The experimental results show that SPINE technique outperforms the Modulo scheduling in nest-loop software pipelining for all our benchmarks. It is also superior to MD retiming for most of the cases in terms of code size and execution time of the final code.

6 REFERENCES