A SCALABLE LOOP OPTIMIZATION APPROACH
FOR SCALABLE DSP PROCESSORS

Jian Wang
Wireless Speech and Data Processing
Nortel Networks Montreal
Montreal, QC, Canada, H3E 1H6
email: jiwang@nortelnetworks.com

Bogong Su        Erh-Wen Hu
Dept. of Computer Science
The William Paterson University of New Jersey
Wayne, NJ 07470, USA
email: bsuwpc@frontier.wilpaterson.edu

ABSTRACT
This paper proposes the possibility of reuse of the existing optimized DSP code on a scalable high-performance VLIW DSP processor. Since loops are the critical paths in most DSP applications, we focus on issues related to loop optimization. In our approach, we first perform a loop alignment transformation on the source level; we then reuse the existing optimized loop code on the assembly level. The approach is highly portable because it is independent of DSP hardware details. It can be used directly by a DSP programmer on source level and/or by a DSP compiler designer to implement independent optimization modules.

1. INTRODUCTION
Recently, several high-performance DSP processors based on VLIW architecture such as TI's C6x [5], ADI’s TigerSHARC [3], and Starcore’s SC140 [2] have been announced. Although today’s silicon is dense enough to allow us to build cost-effective single-chip VLIW-based DSP processors, there are still some technical limitations. For example, the area of register file grows approximately with the square of the number of ports [1], it limits the number of function units connected to the register file. Clustered register file architectures seem to be a solution: each cluster consists of one register file connected to one data path which has approximately four function units. Through integrating multiple clusters controlled by VLIW instructions, the performance of these DSP processors is scalable. However, by increasing the number of data paths and the length of the VLIW instructions in order to enhance performance, the existing optimized DSP software needs to be rewritten. It impacts not only the time-to-market of a new DSP product but also the various kinds of custom-made DSP applications. In particular, some critical hand-written assembly code must be re-written. As a result, it presents a big challenge to applications programmers and DSP compiler designers.

This paper presents a scalable loop optimization approach, which promotes the reuse of the existing optimized code on the scalable high-performance VLIW DSP processors. It can be applied at source code level as well as implemented in DSP compilers.

2. A MOTIVATING EXAMPLE
To help illustrate the idea behind our loop optimization approach, we use the following motivating example. Consider first the execution of the loop

\[ \text{for} \ (i = 1, 100) \ \{ \ c[i] = x + y * a[i] \}. \]

(a) One-Data-Path DSP
(4 operations per cycle: 1 addition, 1 multiplication, 1 load, 1 store)

(b) The sequential code

Do #100, End

MOV (r1)+, X1; // load a_i to X1
MPY X1, Y0, A; // A = y * a_i
ADD X0, A, B;   // B = x + y * a_i
MOV B, (r2)+ ;   // store c_i

End

(b) The sequential code

Do #100, End

MOV (r1)+, X1
MPY X1, Y0, A || MOV (r1)+, X1
ADD X0, A, B || MPY X1, Y0, A || MOV (r1)+, X1
Do #100, End

MOV B, (r2)+ || ADD X0, A, B || MPY X1, Y0, A
|| MOV (r1)+, X1

End

(c) The optimized code

(S1 || S2 means S1 and S2 will be issued for execution at the same cycle)

Figure 1. A Motivating Example
which is targeted to a DSP with one data path as shown in Figure 1(a). The loop contains typical MAC operations. Figure 1(b) shows the sequential code of the loop. By applying software pipelining technique [4,6] to the sequential code, we obtain the optimized loop code as shown in Figure 1(c).

![Memory Diagram](image-url)

(a) The upgraded Two Data Path DSP
(8 operations per cycle: 2 additions, 2 multiplications, 2 loads, and 2 stores)

MOV1 (r 11 )+, X 11 || MOV2 (r 21 )+, X 21  
MPY1 X 11 , Y 10 , A 1 || MOV1 (r 11 )+, X 11  
|| MPY2 X 21 , Y 20 , A 2 || MOV2 (r 21 )+, X 21  
ADD1 X 10 , A 1 , B 1 || MPY1 X 11 , Y 10 , A 1  
|| MOV1 (r 11 )+, X 11 || ADD2 X 20 , A 2 , B 2  
|| MPY2 X 21 , Y 20 , A 2 || MOV2 (r 21 )+, X 21  
Do #100, End  
MOV1 B1, (r 12 )+ || ADD1 X 10 , A 1 , B 1  
|| MPY1 X 11 , Y 10 , A 1 || MOV1 (r 11 )+, X 11  
|| MOV2 B2, (r 22 )+ || ADD2 X 20 , A 2 , B 2  
|| MPY2 X 21 , Y 20 , A 2 || MOV2 (r 21 )+, X 21  
End

(b) The upgraded optimized code
(Si || Si means Si and Si will be issued for execution in the same cycle; OP denotes the operation from path i, where i = 1, 2)

Do #100, End  
MOV1 B1, (r 12 )+ || ADD1 X 10 , A 1 , B 1  
|| MPY1 X 11 , Y 10 , A 1 || MOV1 (r 11 )+, X 11  
|| MOV2 B2, (r 22 )+ || ADD2 X 20 , A 2 , B 2  
|| MPY2 X 21 , Y 20 , A 2 || MOV2 (r 21 )+, X 21  
End

The two divided loops will execute successfully in parallel on the two-data-path DSP as long as there is no data dependence between them. The optimized code can be used for each of these two loops and the final optimized code is shown in Figure 2(b).

However, many loops in DSP applications have data dependences between iterations. Hence, the above method cannot be directly used. In the following section, we will introduce a pre-process, which can handle those loops with inter-iteration data dependences and allows code reuse.

### 3. CONVERTING A DO-ACROSS LOOP INTO A DO-ALL LOOP

Before discussing our proposed pre-process, we first present some basic concepts.

**Data Dependence**: If a variable is written by statement A and its value will be used by statement B, we say there is a data dependence from A to B. For example, in Figure 3(a), there are data dependences from s1 to s2, s3 to s4 and s2 to s3.

**Dependence Distance**: If a data dependence is from a statement of the i-th iteration to a statement of the (i+k)th iteration, we say this data dependence has a dependence distance of k, where k is a non-negative integer.

**Loop-Carried Dependence**: If dependence distance k is non-zero, the data dependence is called a loop-carried dependence; if k is zero, the data dependence is called a loop-independent dependence.

**DDG (Data Dependence Graph)**: A DDG is defined in terms of (V, E, w), where V denotes all statements in the loop, E denotes all data dependences and w is the dependence distance attached to each edge.

**Do-All and Do-Across Loops**: If the DDG of a loop has no loop-carried dependence, this loop is called a do-all loop; otherwise, it is called a do-across loop.

Figure 3(a) shows an example of a do-across loop. Figure 3(b) shows the DDG of Figure 3(a). From the DDG, it is clear that there is a loop-carried dependence from s2 to s3, since s3 of the (i+1)th iteration uses the data created in s2 of the i-th iteration.

A do-across loop can be converted to a do-all loop by a simple loop transformation (also known as loop alignment [7]) as long as there is no data dependence circle in its DDG. In the following, we will discuss how to convert a do-across to a do-all using the loop example in Figure 3(a).

The conversion is a process of re-constructing the loop body. We first define a loop index k to each statement in the loop body. s1(k) means the instance of s1 of the k-th iteration. In the original loop body, all statements have the same loop index which is actually the loop index variable, and the loop will be executed in the following order:
Without violating data dependences, the loop can be executed in another order:
\[
\begin{align*}
&\text{s}_1(1)\text{s}_2(1)\text{s}_3(1)\text{s}_4(1) \\
&\text{s}_1(2)\text{s}_2(2)\text{s}_3(2)\text{s}_4(2) \\
&\quad \vdots \\
&\text{s}_1(101)\text{s}_2(101)\text{s}_3(101)\text{s}_4(101).
\end{align*}
\]

In other words, we must observe the following three rules in the conversion process:
1. All data dependences in the loop must not be broken.
2. For any two statements, si and sj with a loop-independent dependence where dependence distance is zero, the loop indices for the statements in the reconstructed loop body must be the same.
3. For any two statements si and sj with a loop-carried dependence where dependence distance is \( w \), then sj's loop index must be equal to \( w \) in order to change this dependence to a loop-independent dependence in the re-constructed loop body.

We now conclude the above discussion by formalizing the conversion problem as follows: Given a do-across loop and its DDG= \((V,E,w)\), find a loop index \( \text{lpid}(s) \) for each statement \( s \), such that for any \( e=(s_1,s_2) \) in \( E \), \( w(s_1,s_2) = \text{lpid}(s_2) - \text{lpid}(s_1) \).

4. OUR SCALABLE LOOP OPTIMIZATION APPROACH

Based on the discussion in the previous section, we present an algorithm to convert a do-across loop to a do-all loop. The algorithm, named DOACROSS2DOALL in this paper, contains the following steps:

begin
1. Construct the DDG = \((V,E,w)\) for the given loop.
2. If there is a data dependence circle in the DDG, then return UNDO.
3. If there are any two paths, \( L_1 \) and \( L_2 \), from \( s_1 \) to \( s_2 \), such that \( \sum_{e \in L_1} w(e) \neq \sum_{e \in L_2} w(e) \) then return UNDO;
4. Add a new node, src, to the DDG and create a new edge with weight 0 from src to each node of the DDG.
5. Compute maximum distance, \( \text{maxd}(s) \), from src to each node \( s \).
6. Define the loop index, \( i - M + \text{maxd}(s) \), to each statement (node) \( s \), where \( i \) is the loop index and \( M = \max(\text{maxd}(s)) \) for all \( s \).
7. Re-write the loop in terms of the new loop index of each statement. A prelude and a postlude are also constructed.
end

The proof of the correctness of the above algorithm is not included in this paper due to the length limitation of the paper.

There are two cases where the above algorithm returns UNDO and the given do-across loop cannot be converted to a do-all. In the first case, the DDG contains at least one circle so the loop cannot be converted to a do-all in nature. Our future work is to present a method to handle this type of loop programs. In the second case, there are two paths from the same source node \( s_1 \) to the same destination node.
s2 and the lengths of these two paths are different. For this type of loops, there is a need to first split/duplicate the destination node before applying the above algorithm.

In the following, we present our scalable loop optimization approach.

\textbf{begin}
1. If the given loop is a do-all, go to step 3.
2. Call algorithm \texttt{DOACROSS2DOALL}, if it returns \texttt{UNDO}, then return \texttt{FAIL}.
3. Let the number of data paths be \( N \) in the upgraded DSP, re-write the do-all loop as a two-level nested loop where the outer level has \( N \) iterations.
4. If the optimized code of the loop for one data path exists, go to step 5; otherwise, optimize the innermost loop under the resource constraints of one data path.
5. Map the outer loop onto the \( N \) data paths, one-to-one.
6. Return \texttt{SUCCESS}.
\textbf{End}

A complete working example is presented in the next section.

\section{A Working Example}

We will use the loop in Figure 3(a) as a working example to demonstrate our new approach. We use TI's C6x as our DSP architecture model so C6x's instruction format \cite{ref6} is used in this example. The pre-process was illustrated in section 3. This section will focus on the assembly code generation.

Some C6x's instructions have a long latency (e.g. \texttt{LDH} is a five cycles operation) which may make the optimized code very complicated and not readable. For simplicity, we assume that the latency for all instructions is one machine cycle. In Figures 4(a), we software pipeline the do-all loop in Figure 3(c) under the resource constraints of one data path. Then, this optimized one-data-path code is

\begin{verbatim}
STH .S1  A6, *A13++  |  ADD  .L1  A4, A5, A6
  |  LDH  .S1 *A11++, A4  |  STH  .S1 A3, *A11
  |  MPY  .M1 A2,A1,A3  |  LDH  .S1 *A10++, A2
\end{verbatim}

(a) The optimized loop body for one data path

\begin{verbatim}
  |  ADD  .L1 A4, A5, A6  |  ADD  .L2 B4, B5, B6
  |  LDH .S1 *A11++, A4  |  LDH .S2 *B11++, B4
  |  MPY  .M1 A2,A1,A3  |  MPY  .M2 B2,B1,B3
  |  LDH .S1 *A10++, A2  |  LDH .S2 *B10++, B2
\end{verbatim}

(b) The optimized loop body for two data paths

Figure 4  Assembly code of the do-all loop body

"\textit{duplicated}" and "\textit{scaled-up}" to obtain C6x's two-data-path code as shown in Figure 4(b). Obviously, the assembly code of two-data-path DSP is very similar to that of the one-data-path DSP. However, the value of the loop count with two data paths is half of that of the original loop, so the speedup is 2.

From this example we have demonstrated that it is easy to obtain the code for the two-data-path DSP from that of one-data-path by using our scalable loop optimization approach and the execution time of the two-data-path DSP will be reduced by half as the value of the loop count is reduced.

\section{Conclusion}

DSP processors based on the VLIW architecture are scalable. It is a challenge for DSP programmers and DSP compiler designers to study the scalability problem of the optimized DSP code and the advanced DSP compilation techniques.

This paper presents a simple and novel loop optimization method which promotes the reuse of existing optimized code when the processor is scaled up. It thus helps reduce the time-to-market for real-time DSP software development.

\section*{Acknowledgment}

We would like to thank Mr. Rafi Rabipour for his helpful suggestions to improve this paper, and to thank Mr. Yves Helaudais for his assistant in generating the working example.

\section*{References}

\begin{enumerate}
  \item T. Halfhill, StarCore Reveals Its First DSP, Microprocessor Report, May 10, 1999.
  \item E. Stotzer and Ernst Lesis, Modulo Scheduling for the TMS320C6x Vliw DSP, Proc. of LCTES'99, May 1999
\end{enumerate}