Loop Fusion for Clustered VLIW Architectures

Yi Qian, Steve Carr, Philip Sweany - LCTES/SCOPES’02
Presented by Bertram Schmitt – 2011/05/16
Outline

- Motivation
- Clustered VLIW machines
- Overview of loop transformations
- Partition of functional units
- Implementation of loop fusion
- Experimental results
Motivation

- Systems used for **digital signal processing** (DSP) usually have a computation demand where a majority of execution time is typically spend in loops.

- Embedded systems used for DSP require maximum performance with **minimal power consumption and chip costs**.

- A high level of **instruction-level parallelism** (ILP) can be used to improve performance.

- **Software pipelining** can often exploit ILP.
**Motivation**

- **However:** Software pipelining (in some instances) hinders the goal of low power consumption and low chip costs as it may need more physical registers than available.

- **Register pressure problem:** less hardware registers are available than would have been optimal
  - more spills and reloads
  - performance might actually decrease

- **Clustering** is used to allow a large number of simultaneous reads/writes for registers
Clustered VLIW Machines

- Clustered VLIW machines use several small register banks with a low number of ports instead of one large register bank with many ports.

- Each register bank is grouped with one or more functional units that can access data directly only from the local register bank.

- These groups are called clusters.

- Additional overhead occurs when a functional unit needs a value that is currently located in another cluster.
Clustered VLIW Machines

- The efficiency of clustered architectures mainly depends on
  - partitioning instructions among clusters to **minimize** intercluster data movement
  - scheduling instructions to **maximize utilization** of function units
- **High-level loop transformations** are a good opportunity to improve the quality of the partitioned code
Goal

- **Goal**: increase the amount of data-independent parallelism in innermost loops, leading to a better partition

- In previous work, transformations like **loop unrolling** and **unroll-and-jam** have already been applied to increase parallelism on clustered VLIW machines

- **In this paper**: analysis of the effect of **loop fusion** (while still using the benefits from previously examined loop transformations)
Loop Unrolling

- **Unroll** the loop to reduce instructions for controlling the loop and to possibly increase parallelism

- Example:

```c
for(i=0; i<N; i++) {
    A[i] = B[i] + C[i];
}
```

```c
for(i=0; i<N; i+=4) {
    A[i] = B[i] + C[i];
    A[i+1] = B[i+1] + C[i+1];
    A[i+2] = B[i+2] + C[i+2];
    A[i+3] = B[i+3] + C[i+3];
}
```
Unroll-and-jam

- Also called **outer-loop unrolling**
- The transformation *unrolls* an outer loop and then *jams* the resulting inner loops back together
- Can be used to **parallelize innermost loop body**
Unroll-and-jam

Example:

```c
for(i=0; i<4; i++) {
    for(j=0; j<4; j++) {
        c[i][j] = 0;

        for(k=0; k<4; k++) {
            c[i][j] = a[i][k] * b[k][j] + c[i][j];
        }
    }
}

for(i=0; i<4; i++) {
    for(j=0; j<4; j+=2) {
        c[i][j] = 0;
        c[i][j+1] = 0;

        for(k=0; k<4; k++) {
            c[i][j] = a[i][k] * b[k][j] + c[i][j];
            c[i][j+1] = a[i][k] * b[k][j+1] + c[i][j+1];
        }
    }
}
```
Loop Fusion

- By combining two (or more) loop nests into a single nest, operations from the second loop can be scheduled in any empty slots that exist in the original loop.

- The resulting code may:
  - provide more parallelism than the two loops in isolation.
  - reduce the overhead due to loop code.
  - take less code space.
Partitioning

- In clustered VLIW architectures a distinct set of registers is associated with each cluster of functional units

- **Problem**: what happens if a register from another functional unit needs to be accessed?

- Copying a value from one register bank is expensive

- Compiler has to allocate registers to banks to reduce the number of copies while retaining a high degree of parallelism
Partitioning

- **Approach:**
  1. Build intermediate code with symbolic registers, assuming a single infinite register bank
  2. Build data dependence graphs (DDGs) and perform software pipelining still assuming an infinite register bank
  3. Partition the registers to register banks (and thus to the preferred functional unit(s))
  4. Apply value cloning for induction variables and loop invariants
5. Re-build DDGs and perform instruction scheduling attempting to assign operations to the “proper” (cheapest) functional unit based upon the location of the registers.

6. With functional units specified and registers allocated, perform “standard” Chaitin/Briggs graph coloring register assignment for each register bank.
Example code:

```c
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        c0 = 0.0;

        for (k = 0; k < n; k+=2) {
            c0 += a[i][k] * b[k][j];
            c0 += a[i][k+1] * b[k+1][j];
        }
    }
}
```

Ideal schedule for innermost loop (assuming a single-cycle latency for \texttt{add} and a two-cycle latency for \texttt{mult} and \texttt{load}): 

<table>
<thead>
<tr>
<th>LoadInstruction</th>
<th>AddInstruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>load r1, a[i][k]</td>
<td>add r4, r4, r5</td>
</tr>
<tr>
<td>load r2, b[k++] [j]</td>
<td>mult r5, r7, r6</td>
</tr>
<tr>
<td>load r7, a[i][k]</td>
<td>add r4, r4, r3</td>
</tr>
<tr>
<td>load r6, b[k++] [j]</td>
<td>mult r3, r1, r2</td>
</tr>
</tbody>
</table>
Partitioning: Example

- Partition could look like this (assuming \(i, j\) and \(k\) are cloned to have one copy in each of the clusters):

\[
P1 = \{r1, r2, r3, r4, i1, k1, j1\} \quad \text{and} \quad P2 = \{r5, r6, r7, i2, k2, j2\}
\]

With the following schedule:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>load (r1, a[i1][++k1])</td>
<td>copy (r55, r5)</td>
</tr>
<tr>
<td>load (r2, b[k1++][j1])</td>
<td>mult (r5, r7, r6)</td>
</tr>
<tr>
<td>add (r4, r4, r55)</td>
<td>nop</td>
</tr>
<tr>
<td>add (r4, r4, r3)</td>
<td>load (r7, a[i2][++k2])</td>
</tr>
<tr>
<td>mult (r3, r1, r2)</td>
<td>load (r6, b[k2++][j2])</td>
</tr>
</tbody>
</table>
Loop optimization strategy

Three steps:

1. Fuse loops to enhance intracluster parallelism and enhance the intercluster parallelism later exploited by unrolling
2. Unroll-and-jam loops to increase intracluster parallelism
3. Unroll-and-jam or unroll loops to enhance data-independent parallelism across multiple clusters

The transformation in step 3 is analogous to a parallel loop where different iterations run on different processors
Communication Cost

- Careless use of loop fusion may decrease overall performance due to high intercluster communication

- Example:

```c
for (i = 0; i < n; i++)
    a[i] = x[i] + q;
for (i = 0; i < n; i++)
    b[i] = a[i] * c[i];
for (i = 0; i < n; i++)
    d[i] = a[i-1] + b[i];
```

```c
for (i = 0; i < n; i++)
    a[i] = x[i] + q;
    b[i] = a[i] * c[i];
    d[i] = a[i-1] + b[i];
```
Communication Cost

- **Problem**: How to determine if loop fusing is profitable?

- **Definition**:
  - Communication cost $C_L(l)$:
    - the number of alignment conflicts at level $l$ of a loop $L$
  - Set $Min_L$ for a loop nest $L$:
    - a set of loop levels that have the minimal communication cost among communication costs at all levels
Communication Cost

- Algorithm to determine, if loop fusion is profitable:
  - Compute the communication cost for the original loops and the fused loop
  - Find common loop levels at which both loops have minimal communication cost
  - Loop fusion is profitable if the communication cost of the fused loop at one of these levels is equal to the sum of the communication costs of the original loops at the same level (i.e., no extra alignment conflicts are introduced into the fused loop at this level)
Implementing Loop Fusion

- Algorithm for loop fusion (based on an approach by Kennedy):
  - Construct a fusion graph, where each node $n$ represents a loop
  - An edge $(n_1, n_2)$ is added between two nodes if there exists a dependence between $n_1$ and $n_2$
  - Examine if loop fusion is possible
Implementing Loop Fusion

- An edge \((n_1, n_2)\) is marked **fusion-preventing** if:
  - Loop headers are not compatible
  - A forward loop-independent dependence is a backward loop-carried dependence after fusion
  - Two loops are in a *bad path*. A bad path is a path from \(n_1\) to \(n_2\) which includes a node that cannot be fused with either \(n_1\) or \(n_2\)

- Determine if loop fusion is **profitable**

- If it is profitable: fuse loops \(n_1\) and \(n_2\) and repeat process until no more fusible loop pairs can be found
Experimental Results

- Effectiveness of transformations was evaluated on a DSP benchmark suite by Texas Instruments, called *URM*
- *URM* includes two applications containing 119 loops
- Fusion was applicable to 12 sets of loops consisting of 25 loops
- Best results were obtained on an architecture with 16 functional units and 2 clusters
- Average speed-ups up to 2.15x compared to the original code
Experimental Results

- Loop fusion was also evaluated on a VLIW machine, the *Texas Instruments TMS320C64x* (or *C64x*)

- The C64x CPU is a **two-cluster** VLIW fixed-point processor with **eight functional units** that are divided equally between the clusters

- Average speed-up of 1.46x
Summary

- Evaluation of the effects of **loop fusion** in addition to other high-level loop transformations
- Introduction of algorithms to
  - partition functional units
  - determine fusion profitability
- Speed-ups up to **2.15x** are possible
Thank you for your attention!