Complementing Software Pipelining with Software Thread Integration

LCTES ’05 - June 16, 2005

Won So and Alexander G. Dean
Center for Embedded System Research
Dept. of ECE, North Carolina State University
wso@ncsu.edu, alex_dean@ncsu.edu
Motivation

• Multimedia applications
  - Demand high performance to embedded systems
• DSPs with VLIW or EPIC architectures
  - Maximize processing bandwidth
  - Deliver predictable, repeatable performance
  - E.g. Philips Trimedia, TI VelociTI and StarCore etc.
• However, speed is limited
  - Difficult to find independent instructions.
  - Software pipelining (SWP) can suffer or fail.
    • Complex control flow
    • Excessive register pressure
    • Tight loop-carried dependence
**Software Thread Integration (STI)**

- Software technique which interleaves multiple threads into a single implicitly multithreaded one
- Idea: STI for high-ILP
  - Merge multiple procedures into one → Increase compiler’s scope
  - Code transformations to reconcile control flow diffs.
    → Enable arbitrary alignment of code regions
    → Move code to use available resources
- Method: Procedure cloning and integration
  - Integrate parallel procedure calls in the app.
  - Create procedures with better efficiency
  - Convert procedure level parallelism into ILP
STI for High-ILP

- Assumption: Parallel procs. identified a priori.
  - Manually identified or automatically extracted
  - Exploit stream programming languages (e.g. StreamIt)
- Goal: Code transformations to improve ILP
  - Apply transformations based on schedule analysis
  - Complement key optimization (i.e. SWP)
- Contribution: Improve DSP app. dev. process
  - Efficient compilation of C or C-like languages with additional dataflow information
  - Reduce the need for extensive manual C and assembly code optimization and tuning
Related Work

SWP for complex control flow
- Hierarchical reduction [Lam88]
- Enhanced MS [Water92]
- MS for multiple exits [Lavery92]
- Multiple-II MS [Water-Perez95]
- All-path pipelining [Stoodley96]

STI complements existing SWP methods.

STI jams whole procedures.

Loop optimizations
- Loop jamming
- Loop unrolling
- Unroll-and-jam [Carr96]

STI exploits coarse-grain parallelism.

Stream programming

Integrated procedures do the work of multiple ones.

Interprocedural optimizations

STI transforms control flow.

- Procedure inlining

- Procedure cloning [Cooper93]
Classification of Loops

- IPCs of Loops from TI Image/DSP library

**SWP-Good** Speedup=2, High-IPC

**SWP-Poor** Speedup<2, Low-IPC and dependence bounded

**SWP-Fail** Calls, conditionals, lack of registers, no valid schedule
STI Overview

• STI transformations
  - Reconcile control flow diffs: Enable arbitrary alignment
  - Use CDG: Apply hierarchically and repeatedly
  - Conditionals: Duplication
  - Loops: Jamming, peeling, unrolling and splitting

• Two levels of integration
  - Assembly
  - HLL \(\rightarrow\) We pursue this.

• Side effects
  - Code size increase
  - Increase of register pressure
  - Additional data memory traffic
STI for Loops

1) Loop jamming + splitting
2) Loop unrolling + jamming + splitting
3) Loop peeling + jamming + splitting
STI for Loops (contd.)

- Conditional
  - Duplicate code into all conditionals
  - Increase instructions in BBs
- Call
  - Treat them as regular statements
  - Find more instructions to fill delay slots
# STI Transformations

- STI transformations for code regions A and B

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>Loop</th>
<th>Acyclic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SWP-Good</td>
<td>SWP-Poor</td>
<td>SWP-Fail</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop</td>
<td>SWP-Good</td>
<td>Do not apply STI</td>
<td>Do not apply STI</td>
</tr>
<tr>
<td></td>
<td>SWP-Poor</td>
<td>STI: Jam (+Unroll A)</td>
<td>STI: Loop peeling</td>
</tr>
<tr>
<td></td>
<td>SWP-Fail</td>
<td>STI: Jam (+Unroll loop with smaller II)</td>
<td>STI: Code motion</td>
</tr>
<tr>
<td>Acyclic</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- STI: Code motion
- STI: Loop peeling
- Do not apply STI
Platform and Tools

- Target architecture: TI TMS320C64x
  - Fixed-point DSP, VelociTi.2
  - Clustered: (4FU + 32 Reg.) x 2
  - ISA: Predication, SIMD, 1~5 delay slots
  - On-chip: 16KB L1P/L1D$, 1024KB SRAM

- Compiler and tool
  - TI C6x compiler
    - Option -o2: all optimizations but interprocedural ones
    - Option -mt: aggressive memory anti-aliasing
  - C64x simulator in Code Composer Studio (CCS) 2.20
    - stall.Xpath: stalls due to cross-path communication
    - stall.mem: stalls due to memory bank conflict
    - stall.l1p: stalls due to program cache misses
    - stall.l1d: stalls due to data cache misses
    - exe.cycles: cycles other than stalls
Experiments

**SWP-Poor**
- iir
- fft
- hist
- errdif

**SWP-Good**
- fir
- fdct
- idct
- corr

**SWP-Fail**
- s1cond
- s1call
- s2cond
- s2call

Integration

- **SWP-Poor + SWP-Poor**
  - iir_sti2
  - fft_sti2
  - hist_sti2
  - errdif_sti2

- **SWP-Good + SWP-Good**
  - fir_sti2
  - fdct_sti2
  - idct_sti2

- **SWP-Poor + SWP-Good**
  - fir_iir
  - firu8_iir
  - corr_errdif
  - corru8_errdif

- **SWP-Fail + SWP-Fail**
  - s1cond_sti2
  - s1call_sti2
  - s1condcall
  - s2cond_sti2
  - s2call_sti2
  - s2condcall
SWP-Poor + SWP-Poor

(SWP-Poor + SWP-Poor) → speedup > 1

(SWP-Good + SWP-Good) → speedup < 1

Source of speedup → exe.cycles

Source of slowdown → stall.mem, stall1d, stall1p
**SWP-Poor + SWP-Good**

(SWP-Poor+SWP-Good) → speedup > 1

Increase unroll factors → increase speedup

Source of speedup → exe.cycles

Impact of stalls → not consistent
SWP-Fail + SWP-Fail

Increasing Number of Input Items

Source of speedup
⇒ exe.cycles

Source of slowdown
⇒ stall.mem, stall1d, stall1p

(SWP-Fail + SWP-Fail) ⇒ speedup > 1
Conclusions

• STI transformations for high-ILP
  - Determined by control structure and utilization

• STI complements SWP
  - SWP-Poor+SWP-Poor: 26% speedup (HM)
  - SWP-Poor+SWP-Good: 55% speedup (HM)
  - SWP-Fail+SWP-Fail: 16% speedup (HM)

• Future work: automatic integration
  - Algorithm for code transformation
    • Heuristics to match code regions (loop and acyclic)
    • Reconcile more complex control flow differences
    • Estimate the impact of dynamic events
  - Develop a tool chain for automatic integration
  - Support STI for StreamIt programs
Thanks.

• Any questions?
Code Size

Code size increase after STI

- Significant in s1cond, s2cond → b/c of conditionals
- (SWP-Fail+SWP-Fail) → b/c of conditionals
- (SWP-Poor+SWP-Good) → code size is less than sum → increase after unrolling
Back-up

• STI Overview
• Procedure Cloning and Integration
  - Method
  - Results
• Automatic Integration
  - Overview of Algorithm
  - Tool Chain
STI Overview

- Design spaces and steps for STI
Procedure Cloning and Integration

• STEP 1: Identify candidate procedures
  - Find procedures dominating performance by profiling or run-time estimation

• STEP 2: Examine parallelism
  - Find independent procedure-level data parallelism
    • Each procedure call handles its own data sets
    • Data sets are independent of each other

• STEP 3: Perform procedure integration
  - Design the control flow of integrated procedure
    • Simple techniques for identical procedures
      1) Loops with same loop counts: loop jamming
      2) Loops with conditionals: duplicate conditionals
PCI (contd.)

• STEP 4: Optimize the application
  – Dynamic approach
    • RTOS chooses the most efficient version at run-time
  – Static approach
    • Replace original procedure calls to integrated procedures
    • Select the combinations of most efficient versions

![Diagram of RTOS and Direct call flow]
Preliminary Results

• Application: JPEG
  - cjpeg and djpeg with lena.ppm (512x512x24bit)

• Target architecture: Itanium
  - EPIC + predication + speculation
  - 16KB L1 D$/I$, L2 96KB U$, L3 2048KB U$

• Software tools
  - Compilers: GCC, Pro64, ORCC and Intel compilers
  - OS: Linux for IA-64
  - pfmon: Performance Monitoring Unit (PMU) library and tool

• Integration method
  - Integrated 3 procedures
    • FDCT and Encode in cjpeg, IDCT in djpeg
    • Self-integrate 2 or 3 procedure calls
  - Manually integrated at C source level and statically executed
Performance of Procedures

STI speeds up best compiler (Intel-O2-u0) by 17%

13% speedup

Slowdown in many cases because of I$ misses

Sweet spot varies between one, two and three threads

Application speedup with the best compiler
cjpeg – 11%
djpeg – 4%
Overview of Algorithm

1. Form CDG for each procedure.
2. Annotate CDG based on analysis of ASM code.
   • Utilizations: # instructions, # cycles
   • SWP info: SWP-Good/Poor/Fail
   • Code size, Reg. use, memory traffic, working set size etc.
3. Rank order code regions.
   • In terms of idle resources and other factors
Overview of Algorithm (contd.)

4. Choose the best combination of code regions.
   - Align loop regions first then do rest of code
   - Avoid unbeneficial combinations (e.g. SWP-Fail + SWP-Good)

5. Try integration.
   - Overlap loop iterations by loop jamming and unrolling
   - Find opportunity for loop peeling and code motion

6. Generate C code for integrated procedure.

7. Compile it.

8. Analyze the performance.
   - Decide whether it is worth to try other transformations
Automation of Code Transformations

- Plan to build C to C translator

Now a Perl script

Can use open compilers e.g. gcc, orcc

Will be developed

TI C6x compiler