Linear Analysis and Optimization of Stream Programs

Andrew A. Lamb
William Thies
Saman Amarasinghe

The New Laboratory for Computer Science and Artificial Intelligence
Massachusetts Institute of Technology
Streaming Application Domain

- Based on audio, video, or data stream
- Increasingly prevalent and important
  - Embedded systems
    - Cell phones, handheld computers
  - Desktop applications
    - Streaming media
    - Software radio
  - High-performance servers
    - Software routers (Example: Click)
    - Cell phone base stations
    - HDTV editing consoles
  - Real-time encryption
  - Graphics packages
Properties of Stream Programs

- A large (possibly infinite) amount of data
  - Limited lifetime of each data item
  - Little processing of each data item
- Computation: apply multiple filters to data
  - Each filter takes an input stream, does some processing, and produces an output stream
  - Filters are independent and self-contained
- A regular, static computation pattern
  - Filter graph is relatively constant
  - A lot of opportunities for compiler optimizations
The StreamIt Language

- Goals:
  - Provide a High-Level Programming Paradigm
  - Improve Programmer Productivity
  - Match Performance of Hand-Hacked Assembly

- Contributions
  - Language Design, Structured Streams, Buffer Management (*CC 2002*)
  - Exploiting Wire-Exposed Architectures (*ASPLOS 2002*)
  - Scheduling of Static Dataflow Graphs (*LCTES 2003*)
  - Domain Specific Optimizations (*PLDI 2003*)
Example: Freq band detection

- Used in...
  - metal detector
  - garage door opener
  - spectrum analyzer

Source:
Application Report SPRA414
Texas Instruments, 1999
void->void pipeline FrequencyBand {
  float sFreq = 4000;
  float cFreq = 500/(sFreq*2*pi);
  float wFreq = 100/(sFreq*2*pi);

  add D2ASource(sFreq);

  add BandPassFilter(1, cFreq-wFreq, cFreq+wFreq, 100);

  add splitjoin {
    split duplicate;
    for (int i=0; i<4; i++) {
      add Detector(i/4);
      add LEDOutput(i);
    }
    join roundrobin(0);
  }
}
Freq band detection in StreamIt

```c
void->void pipeline FrequencyBand {
    float sFreq = 4000;
    float cFreq = 500/(sFreq*2*pi);
    float wFreq = 100/(sFreq*2*pi);
    add D2ASource(sFreq);

    float->float pipeline BandPassFilter(float gain, float ws, float wp, int num) {
        add LowPassFilter(1, wp, num);
        add HighPassFilter(gain, ws, num);
    }

    add splitjoin {
        split duplicate;
        for (int i=0; i<4; i++) {
            add Detector(i/4);
            add LEDOutput(i);
        }
        join roundrobin(0);
    }
}
```
void->void pipeline FrequencyBand {
    float sFreq = 4000;
    float cFreq = 500/(sFreq*2*pi);
    float wFreq = 100/(sFreq*2*pi);
    add D2ASource(sFreq);
    add BandPassFilter(1, cFreq-wFreq, cFreq+wFreq, 100);
    add splitjoin {
        split duplicate;
        for (int i=0; i<4; i++) {
            add Detector(i/4);
            add LEDOutput(i);
        }
        join roundrobin(0);
    }
}

float->float pipeline BandPassFilter(float gain, float ws, float wp, int num) {
    add LowPassFilter(1, wp, num);
    add HighPassFilter(gain, ws, num);
}

float->float filter LowPassFilter(float g, float cFreq, int N) {
    float[N] h;
    init {
        int OFF = N/2;
        for (int i=0; i<N; i++) {
            h[i] = g*sin(…);
        }
    }
    work peek N pop 1 push 1 {
        float sum = 0;
        for (int i=0; i<N; i++) {
            sum += h[i]*peek(i);
        }
        push(sum);
        pop();
    }
}
Freq band detection in StreamIt

void->void pipeline FrequencyBand {
  float sFreq = 4000;
  float cFreq = 500/(sFreq*2*pi);
  float wFreq = 100/(sFreq*2*pi);
  add D2ASource(sFreq);
  add BandPassFilter(1, cFreq-wFreq, cFreq+wFreq, 100);
  add splitjoin {
    split duplicate;
    for (int i=0; i<4; i++) {
      add Detector(i/4);
      add LEDOutput(i);
    }
    join roundrobin(0);
  }
}

float->float pipeline BandPassFilter(float gain, float ws, float wp, int num) {
  add LowPassFilter(1, wp, num);
  add HighPassFilter(gain, ws, num);
}

float->float filter LowPassFilter(float g, float cFreq, int N) {
  float[N] h;
  init {
    int OFF = N/2;
    for (int i=0; i<N; i++) {
      h[i] = g*sin(...);
    }
  }
  work peek N pop 1 push 1 { 
    float sum = 0;
    for (int i=0; i<N; i++) {
      sum += h[i]*peek(i);
    }
    push(sum);
    pop();
  }
}
Freq band detection on a TI DSP
Event Manager Module Reset

MAINT CODE - starts here

This is necessary for silicon revision 1.1, however, for
NOP START: SETC INTM ; Disable interrupts
SPLK #0002h, IMR ; Mask all core interrupts except INT2
LACC IFR ; Read Interrupt flags
SACL IFR ; Clear all interrupt flags
CLRC SXM ; Clear Sign Extension Mode
SPLK #0000h, GPTCON ; Clear General Purpose Timer Control
CLRC OVM ; Reset Overflow Mode
SPLK #0000h, T1CON ; Clear GP Timer 1 Control
SPLK #0000h, T2CON ; Clear GP Timer 2 Control
SPLK #0000h, T3CON ; Clear GP Timer 3 Control

Set up PLL Module

SPLK #0000h, COMCON ; Clear Compare Control

Set up Event Manager Module

T1COMPARE .set 2500
SPLK #0000000011000011b, CKCR0
T1PERIOD .set 5000 ; Sets up period for 4kHz frequency

Bit 14 (0) T2STAT - GP Timer 2 Status. READ ONLY
Bit 13 (0) T1STAT - GP Timer 1 Status. READ ONLY

DSP Implementation

Source: Application Report SPRA414, Texas Instruments, 1999
Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.

Cont.
Cont. Cont.

The New Laboratory for Computer Science and Artificial Intelligence
Massachusetts Institute of Technology

Source: Application Report SPRA414, Texas Instruments, 1999
Output is below THRESHOLD4, THRESHOLD2, & THRESHOLD1. Turn off LEDS

BELOW1 SPLK #0000h, LEDSOUT

Output is below THRESHOLD4, but above THRESHOLD1. Turn on DS1

ABOVE1 SPLK #0001h, LEDSOUT

Output is below THRESHOLD4, but above THRESHOLD2. Check if above

THRESHOLD3

ABOVE2 LT DIFFIN

TH3 MPY THRESHOLD3

PAC

SACH TEMP,1

LACC TEMP

SUB DIFFOUT

BCND ABOVE3, LT

Output is below THRESHOLD4 and THRESHOLD3, but above THRESHOLD2.

Turn on DS1-DS2

BELOW3 SPLK #0003h, LEDSOUT

Output is below THRESHOLD4, but above THRESHOLD3 and THRESHOLD2.

Turn on DS1-DS3

ABOVE3 SPLK #0007h, LEDSOUT

Output is above THRESHOLD4. Check if above THRESHOLD6

ABOVE4 LT DIFFIN

TH6 MPY THRESHOLD6

PAC

SACH TEMP,1

LACC TEMP

SUB DIFFOUT

BCND ABOVE6, LT

Output is above THRESHOLD4, but below THRESHOLD6. Check if above

THRESHOLD5.

BELOW6 LT DIFFIN

TH5 MPY THRESHOLD5

PAC

SACH TEMP,1

LACC TEMP

SUB DIFFOUT

BCND ABOVE5, LT

Output is above THRESHOLD4, but below THRESHOLD6 & THRESHOLD5. Turn

on DS1-DS4

BELOW5 SPLK #000Fh, LEDSOUT

Output is above THRESHOLD4 & THRESHOLD5, but below THRESHOLD6.

Turn on DS1-DS5

ABOVE5 SPLK #001Fh, LEDSOUT

Output is above THRESHOLD4 & THRESHOLD6. Check if above THRESHOLD8.

ABOVE6 LT DIFFIN

TH8 MPY THRESHOLD8

PAC

SACH TEMP,1

LACC TEMP

SUB DIFFOUT

BCND ABOVE7, LT

Output is above THRESHOLD4 & THRESHOLD6, but below THRESHOLD8 &

THRESHOLD7. Turn on DS1-DS6

BELOW7 SPLK #003Fh, LEDSOUT

Output is above THRESHOLD4, THRESHOLD6, & THRESHOLD7, but below

THRESHOLD8. Turn on DS1-DS7

ABOVE7 SPLK #007Fh, LEDSOUT

Output is above THRESHOLD4, THRESHOLD6, & THRESHOLD8. Turn on

DS1-DS8

ABOVE8 SPLK #00FFh, LEDSOUT

OUTLEDS OUT LEDSOUT, LEDS

Turn on the LEDS

RESTART_ADC MAR *, AR2 ; ARP = AR2

LACC * ; ACC = ADCTRL1

ADD #1h ; Set bit to restart the ADC

SACL * ; Start converting next value

LDP #232

LACC EVIFRA ; Clear the flag register of Event Manager

SACL EVIFRA

CLRC INTM ; Enable interrupts

RET ; Return to main line

===================================================================
I S R - PHANTOM

Description: Dummy ISR, used to trap spurious interrupts.

Withdraw

Last Update: 16-06-95

 PHANTOM & PHANTOM

Source: Application Report SPRA414, Texas Instruments, 1999
Conventional DSP Design Flow

Spec. (data-flow diagram)

Design the Datapaths (no control flow)

DSP Optimizations

Coefficient Tables

Rewrite the program

Architecture-specific Optimizations (performance, power, code size)

C/Assembly Code

Signal Processing Expert in Matlab

Software Engineer in C and Assembly
Any Design Modifications?

- Center frequency from 500 Hz to 1200 Hz?
  - According to TI, in the conventional design-flow:
    - Redesign filter in MATLAB
    - Cut-and-paste values to EXCEL
    - Recalculate the coefficients
    - Update assembly
  - If using StreamIt
    - Change one constant
    - Recompile

![Diagram showing A/D, Band pass, Duplicate, Detect, LED connections]
Design Flow with StreamIt

Application-Level Design

StreamIt Program (dataflow + control)

DSP Optimizations

Architecture-Specific Optimizations

C/Assembly Code

Application Programmer

StreamIt compiler
Design Flow with StreamIt

- Benefits of programming in a single, high-level abstraction
  - Modular
  - Composable
  - Portable
  - Malleable

- The Challenge: Maintaining Performance
  - Replacing Expert DSP Engineer
  - Replacing Expert Assembly Hacker
Our Focus: Linear Filters

- Most common target of DSP optimizations
  - FIR filters
  - Compressors
  - Expanders
  - DFT/DCT
  \[ \text{Output is weighted sum of inputs} \]

- Example optimizations:
  - Combining Adjacent Nodes
  - Translating to Frequency Domain
Representing Linear Filters

- A linear filter is a tuple \( <A, b, o> \)
  - \( A \): matrix of coefficients
  - \( b \): vector of constants
  - \( o \): number of items popped

- Example

\[
x \rightarrow A, b, o \rightarrow y = xA + b
\]
Representing Linear Filters

- A linear filter is a tuple \(<A, b, o>\)
  - \(A\): matrix of coefficients
  - \(b\): vector of constants
  - \(o\): number of items popped

- Example

\[
A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad o = 1
\]
Extracting Linear Representation

Resembles constant propagation
Maintains linear form \( <v, b> \) for each variable
  - Peek expression: generate fresh \( ^1v \)
  - Push expression: copy \( ^1v \) into \( A \)
  - Pop expression: increment \( o \)

Linear Dataflow Analysis

\(<A, b, o>\)
Optimizations using Linear Analysis

1) Combining adjacent linear structures

2) Shifting from time to the frequency domain

3) Selection of ‘optimal’ set of transformations
1) Combining Linear Filters

- Pipelines and splitjoins can be collapsed
- Example: pipeline

\[ y = x A \]
\[ z = x A B \]
\[ z = x C \]
Combination Example

Filter 1

Filter 2

\[ A = \begin{bmatrix} 4 & 5 & 6 \end{bmatrix} \]

\[ B = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \]

\[ C = [32] \]
AB for any A and B??

- Linear Expansion

Original

Expanded

\[ \sigma \]

\[ \text{pop} = \sigma \]
Floating-Point Operations Reduction

FIR
RateConvert
TargetDetect
FMRadio
Radar
FilterBank
Vocoder
Oversample
DTOA

Flops Removed (%)

Benchmarks:
- FIR
- RateConvert
- TargetDetect
- FMRadio
- Radar
- FilterBank
- Vocoder
- Oversample
- DTOA

0.3%
2) From Time to Frequency Domain

- Convolutions can be done cheaply in the Frequency Domain

- Painful to do by hand
  - Blocking
  - Coefficient calculations
  - Startup etc.
Floating-Point Operations Reduction

Flops Removed (%)

Benchmark

- FIR
- RateConvert
- TargetDetect
- FMRadio
- Radar
- FilterBank
- Vocoder
- Oversample
- DTOA

The New Laboratory for Computer Science and Artificial Intelligence

Massachusetts Institute of Technology
3) Transformation Selection

- When to apply what transformations?
  - Linear filter combination can increase the computation cost
  - Shifting to the Frequency domain is expensive for filters with pop > 1
    - Compute all outputs, then decimate by pop rate
  - Some expensive transformations may later enable other transformations, reducing the overall cost
Selection Algorithm

- Estimate minimal cost for each structure:
  - Linear combination
  - Frequency translation
  - No transformation
    - If hierarchical, consider all possible groupings of children

- Overlapping sub-problems allows efficient dynamic programming search

Cost function based on profiler feedback
Radar (Transformation Selection)

First compute cost of individual filters:
Radar (Transformation Selection)

First compute cost of individual filters:

- Linear Combination
- Frequency
- No Transform

Cost range: low to high
Radar (Transformation Selection)

First compute cost of individual filters:

- Linear Combination
- Frequency
- No Transform

1x1
Then, compute cost of 1x2 nodes:

- Linear Combination
- Frequency
- No Transform
Then, compute cost of 1x2 nodes:

![Diagram showing transformation selection for 1x2 nodes with different criteria: Linear Combination, Frequency, No Transform, and a 1x1 node with min and min≠ comparisons.]}
Radar (Transformation Selection)

Then, compute cost of 1x2 nodes:
Radar (Transformation Selection)

Continue with 1x3 2x1 3x1 4x1
1x4 2x2 3x2 4x2
2x3 3x3 4x3
2x4 3x4 4x4

Overall solution
Radar (Transformation Selection)
Radar

Maximal Combination and Shifting to Frequency Domain

Using Transformation Selection

2.4 times as many FLOPS

half as many FLOPS
Floating-Point Operations Reduction

The graph shows the floating-point operations reduction in different benchmarks. The x-axis represents the benchmarks and the y-axis represents the percentage of floating-point operations removed. The graph includes three categories: linear, freq, and autosel. The benchmarks are ordered as follows:

- FIR
- Rate Convert
- TargetDetect
- FMRadio
- Radar
- FileBank
- Vocoder
- Oversample
- DTOA

The percentage reductions are as follows:

- FIR: 0.3%
- Rate Convert: -140%
- TargetDetect: -40%
- FMRadio: -20%
- Radar: 0%
- FileBank: -140%
- Vocoder: 0%
- Oversample: 0%
- DTOA: 0%
Experimental Results

- Fully automatic implementation
  - StreamIt compiler

- StreamIt to C compilation
  - FFTW for shifting to the frequency domain

- Benchmarks all written in StreamIt

- Measurements
  - Dynamic floating-point instruction counting
  - Speedups on a general purpose processor
Execution Speedup

On a Pentium IV

The New Laboratory for Computer Science and Artificial Intelligence
Related Work

- SPIRAL/SPL (Püschel et. al)
  - Automatic derivation of DSP transforms
- FFTW (Friego et. al)
  - Wicked fast FFT
- Affine Analysis (Karr, Acta Informatica, 1976)
  - Affine relationships among variables of a program
- Linear Analysis (Cousot, Halbwatches, POPL, 1978)
  - Automatic discovery of linear restraints among variables of a program
Conclusions

- A DSP Program Representation: *Linear Filters*
  - A dataflow analysis that recognizes linear filters

- Three Optimizations using Linear Information
  - Adjacent Linear Structure Combination
  - Time Domain to Frequency Domain Transformation
  - Automatic Transformation Selection

- Effective in Replacing the DSP Engineer from the Design Flow
  - On the average 90% of the FLOPs eliminated
  - Average performance speedup of 450%

- **StreamIt**: A Unified High-level Abstraction for DSP Programming
  - Increased abstraction does not have to sacrifice performance

http://cag.lcs.mit.edu/linear/