Expert Techniques for Improving DSP Application Performance Using Advanced Code Tuning Tools

Jackie Brenner
Senior Member Technical Staff
Texas Instruments
j-brenner@ti.com

SPRE868

Real-Time Application Software Design Flow

- Most developers go through this software development process
- The time spent in each stage varies per application and per developer
- Today we will focus on the Analyze and Tune stage

Analyze and Tune

Ability to optimize code for best code size/performance

Ability to find and understand system bottlenecks

Ability to monitor and track the real time execution of tasks in HW

Ability in HW to collect lots of data in real-time

Rich Compiler optimization options

Fast Simulators for rich profiling

Profile Based Compiler for code size/cycle count trade-off

Cache Visualization and Analysis

Current Limitations on Tuning Tools

- Amount of profile data can be overwhelming
- Accuracy and speed of profiling on a simulator may be less than desired
- Compiler options and feedback exist but guidance is limited
- System issues like cache memory are not taken into account

Advanced Code Tuning Tools

- **The entire premise for the Advanced Code Tuning tools is to provide pro-active advice for assisting developers in tuning applications to meet your performance requirements**
- **This is achieved through the following:**
  - Allows you to set code size and cycle count goals based on your application needs
  - Provides fast and accurate measurements to attainment of these goals
  - Provides advice to help attain goals if they are not met
  - Allows tracking changes from optimization run to optimization run
- **Today's new tuning tools are simulation based**

Advanced Code Tuning Tools Benefits

Improving the Tuning and Optimization Process

- Interactively guides the developer
- Not only measures performance but gives specific advice on how to make improvements

Build Expertise into the Tools

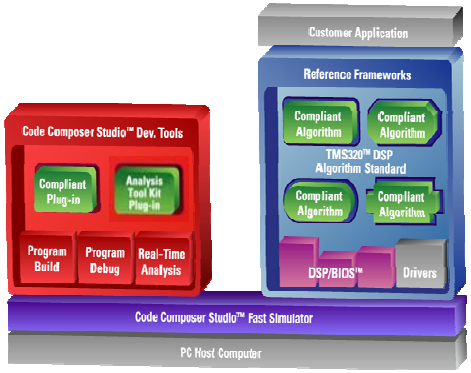
- Shortens learning curve
- Greatly speeds the tuning process
  - Time-to-market

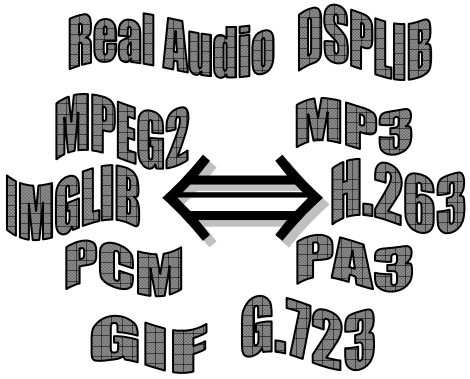
Tuning Code Components

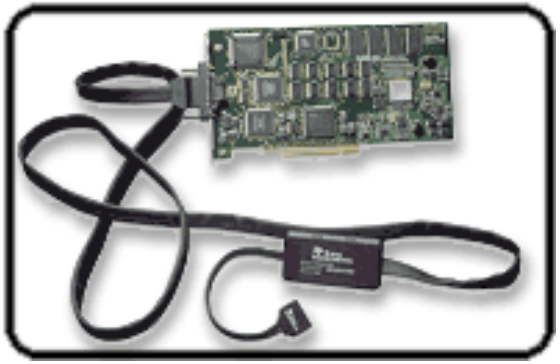
- Dashboard provides a general advice window, a goals window, a set-up profile window to select the activities that you want to perform and a profile viewer that displays the data collected from your chosen activities
- CodeSizeTune (known today as PBC) allows you to make the trade-off between code size and cycle count
- CacheTune allows you to pinpoint the place and time where the application is performing sub-optimally due to cache issues
- Compiler Consultant provides advice on how to improve performance in your C source

Simulator Cycle Accuracy

- 100% Accurate at CPU level
- Accuracy is within 5% for on-chip applications (CPU + Internal Memory) as compared to HW
- Accuracy is within 10% for system level applications (CPU + Memory + DMA + External Memory) as compared to HW

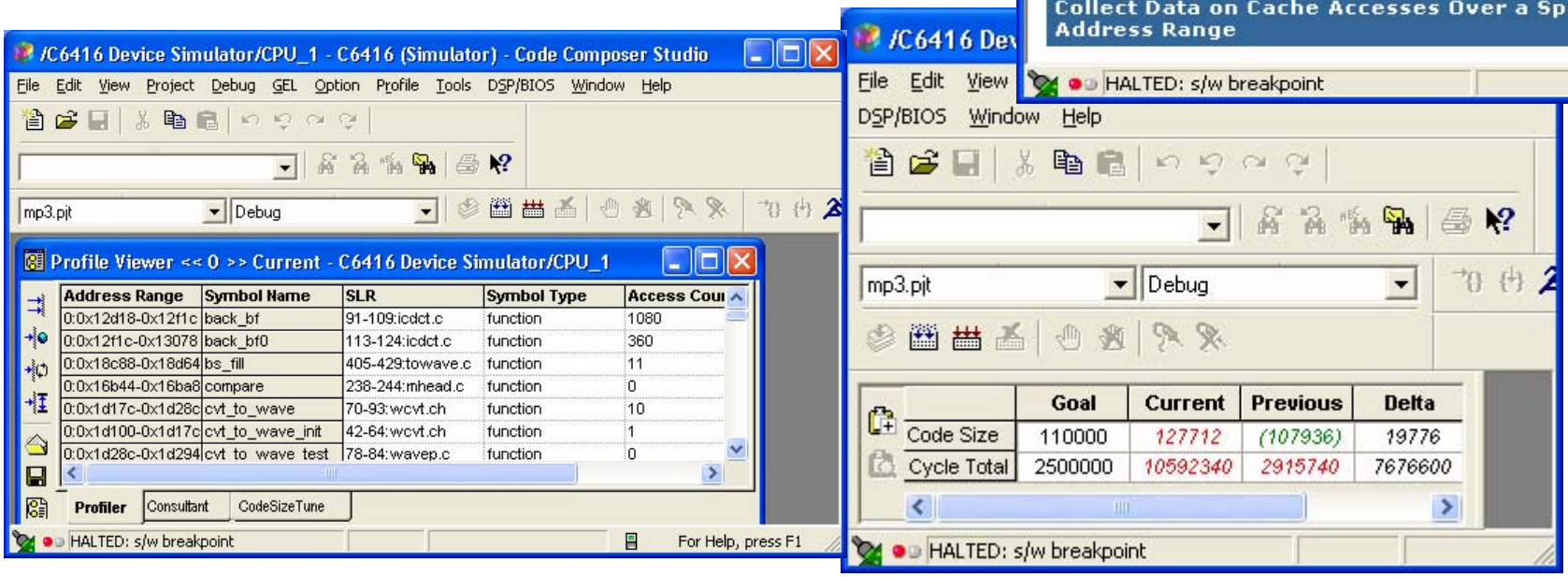






Tuning Dashboard

- Framework for individual tuning tools
- Abstracts details
- Track progress towards goals
- Display collected data
- Pro-active advice on how to obtain goals (specific to your application)

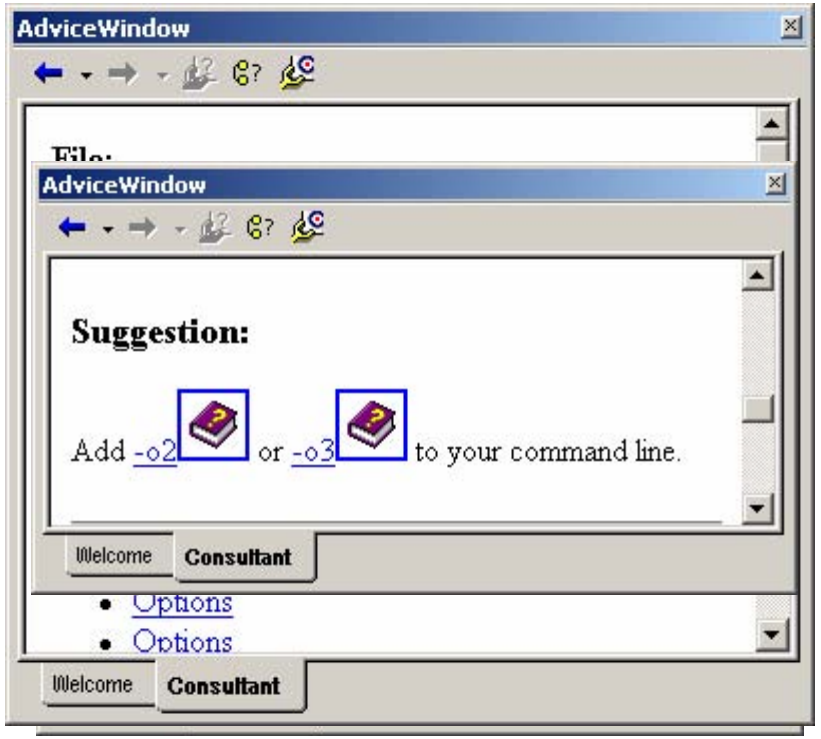


Dashboard Labs

- **Lab 1 - Profile Setup:**
  - Launch Profile Setup
  - Gather Application Level Profile Data
- **Lab 2 - Goals Window:**
  - Launch the Goals window
  - Set goals based on the needs of the application
  - Track tuning progress through a goals log
  - Interpret data in the Goals window
- **Lab 3 - Advice Window:**
  - Navigate in the Advice Window
  - Get advice on specific tuning tools

Compiler Consultant

- Help the compiler do a better job
- Detects areas where optimizations could not be applied
- Provide details on the problem
- Suggest how to fix
  - Often involves providing the compiler with more information
  - Build option change
  - Directive use
  - Pragmas
  - Trip counts
  - Inlining
- Provide links to examples



Compiler Consultant Labs

- **Lab 1 – Options Advice**
  - **Using compiler optimization at –o2**
- **Lab 2 – Alias Advice**
  - **Minimize aliasing with the restrict keyword**
- **Lab 3 – Alignment Advice**
  - **Increasing data load/store capability**
- **Lab 4 – Turning off debug**
  - **Turning off debug information except for profiling**

Pointer Aliasing Information

```
void example1_c (short *xptr, short *yptr, short *zptr,
                 short *w_sum, int N) {
    int i, w_vec1, w_vec2;
    short w1,w2;

    w1 = zptr[0];
    w2 = zptr[1];
    for (i = 0; i < N; i++){
        w_vec1 = xptr[i] * w1;
        w_vec2 = yptr[i] * w2;
        w_sum[i] = (w_vec1 + w_vec2) >> 15;
        }
    }
```

Inner Loop Requires:

2 LDs from mem
2 MPYs
1 ADD
1 SHR
1 ST to mem

Compiler creates a 10 cycle loop because it does not know if *xptr/*yptr alias *w_sum.

Loop carry dependency is 10.

Adding Pointer Aliasing Information

```
void example1_c(short restrict *xptr, short restrict *yptr, short *zptr,
                short *w_sum, int N)
/* Adding restrict removes dependency between xptr/yptr and w_sum;
they don't point to the same memory location */
 {
      int i, w_vec1, w_vec2;
      short w1,w2;

      w1 = zptr[0];
      w2 = zptr[1];
      for (i = 0; i < N; i++){
          w_vec1 = xptr[i] * w1;
          w_vec2 = yptr[i] * w2;
          w_sum[i] = (w_vec1 + w_vec2) >> 15;
      }
 }
```

Loop carry dependency is 0

Pointer Alignment Information

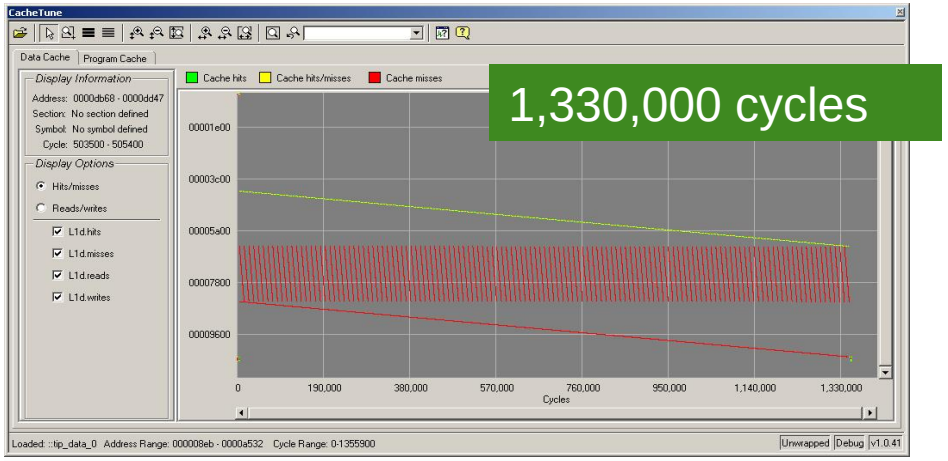
```
void example2_c(short restrict *xptr, short restrict *yptr, short *zptr,
                short *w_sum, int N) {
    int i, w_vec1, w_vec2;
    short w1,w2;

_nassert((int)(xptr) % 8) == 0);
_nassert((int)(yptr) % 8) == 0);
/* _nassert is used to tell the compiler that xptr and yptr are double word
aligned; Compiler can now use LDDW to load 4 xptr and 4 yptr values at a
time */

    w1 = zptr[0];
    w2 = zptr[1];
    for (i = 0; i < N; i++){
        w_vec1 = xptr[i] * w1;
        w_vec2 = yptr[i] * w2;
        w_sum[i] = (w_vec1 + w_vec2) >> 15;
        }
    }
```

Tuning Progress Chart

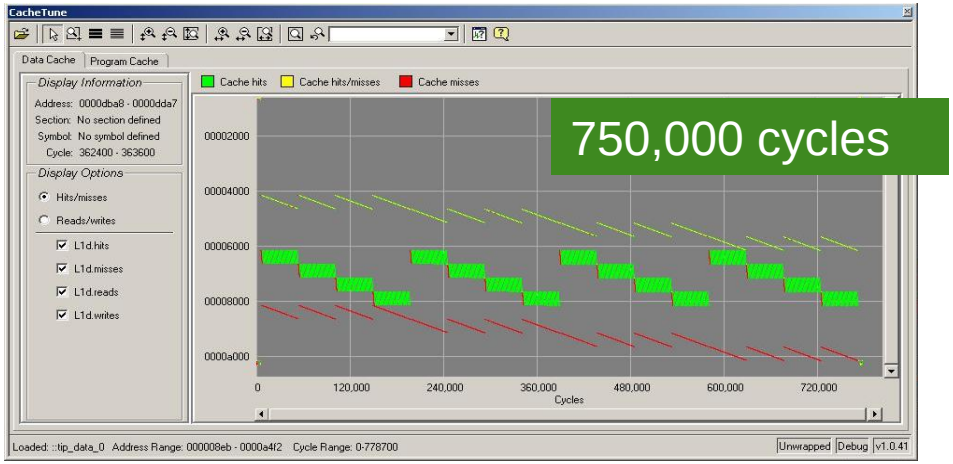
	Code Size	Cycle Count
Dashboard Lab 2 - Initial		
Consultant Lab 1 - Options		
Consultant Lab 2 - Alias		
Consultant Lab 3 - Alignment		
Consultant Lab 4 – Debug Off		

CacheTune

- Optimize cache memory system
  - Reduce cycle count due to cache misses
- Advice on how to make changes
  - Function relocation
  - Code/data partitioning
  - Data reordering
  - Data pre-fetching



1,330,000 cycles



750,000 cycles

CacheTune Labs

- Lab 1 - Cache Overhead
  - Prepare MP3 application to collect cache data
  - Assess the cache overhead for the MP3 application
- Lab 2 - Collecting Cache Access Data over Time
  - Visualize the memory accesses in cache
- Lab 3 - Finding and Removing Conflicts in Data Cache
  - Use the zooming features to view cache information
  - Use Find Conflicting Symbols tool to identify some object conflicts and eliminate them in data cache
- Lab 4 – Move .far into on-chip L2 memory
  - Further eliminate conflict misses in the data cache
  - Measure the improvement in cache stalls/cache overhead

CodeSizeTune

- Formerly known as PBC
- Balance CPU cycles & code size
  - Adjust build options at the function level
- Advice guides you through usage

Final Tuning Progress Chart

	Code Size	Cycle Count	L1D stalls	L1P Stalls
Dashboard Lab 2 - Initial			NA	NA
Consultant Lab 1 - Options			NA	NA
Consultant Lab 2 - Alias			NA	NA
Consultant Lab 3 - Alignment			NA	NA
Consultant Lab 4 – Debug Off			NA	NA
CacheTune Lab 1 – Initial Cache Data				
CacheTune Lab 3 – Remove Data Conflict				
CacheTune Lab 4 – Move .far into L2				

Advanced Code Tuning Tools Summary

Improving the Tuning and Optimization Process

- Interactively guides the developer
- Not only measures performance but gives specific advice on how to make improvements

Build Expertise into the Tools

- Shortens learning curve
- Greatly speeds the tuning process
  - Time-to-market

Experienced User Track

Expert Techniques for Improving DSP Application Performance Using Advanced Code Tuning Tools

Jackie Brenner
Senior Member Technical Staff
Texas Instruments
j-brenner@ti.com