IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI’s standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of that third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:
Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2002, Texas Instruments Incorporated
Read This First

About This Manual

The Analysis Tool Kit (ATK) is a set of tools that help enhance robustness and efficiency analyses of embedded DSP applications. These tools assist in developing correct and efficient implementations of application software. The ATK is available as an update over Code Composer Studio™ (CCS) IDE 2.2. The tool kit is comprised of two tools, based on the simulation platform:

- Cache Analysis
- Code Coverage and Multi–event profiler

The **Cache Analysis tool** provides a graphical visualization of cache accesses over time. This tool is highly effective at highlighting non–optimal cache usage (due to factors such as conflicting code placement, inefficient data access patterns etc). Using this tool, developers can significantly optimize cache efficiency thereby reducing the cycles consumed in the memory subsystem.

The **Code Coverage and Multi–event Profiler tool** provides two distinct capabilities:

- Code Coverage: provides visualization of source line coverage. This facilitates developers to construct tests to ensure adequate coverage of their code.

- Multi–event Profiler: provides function profile data collected over multiple events of interest – all in a single simulation run of the application. Events include CPU cycles, instructions executed, pipeline stalls, cache hits, misses and so on. This tool helps identify hotspots, and possible contributing factors affecting performance.

This user’s guide describes the usage of the ATK in detail: how to setup the ATK, how to use the ATK for data collection, and how to visualize data.
Notational Conventions

This document uses the following conventions.

- Program listings, program examples, and interactive displays are shown in a special typeface similar to a typewriter’s. Examples use a bold version of the special typeface for emphasis; interactive displays use a bold version of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

Here is a sample program listing:

0011  0005  0001         .field    1, 2
0012  0005  0003         .field    3, 4
0013  0005  0006         .field    6, 3
0014  0006         .even

Here is an example of a system prompt and a command that you might enter:

C:  csr -a /user/ti/simuboard/utilities

Trademarks

Trademarks are the property of their respective owners.

Related Documentation from Texas Instruments

Code Coverage And Multi–event Profiler User’s Guide (SPRU624)

Using the Code Coverage And Multi–event Profiler for Creating Robust and Efficient Applications Application Report (SPRA868)

Cache Analysis User’s Guide (SPRU575A)

Using the Cache Analysis tool for improving Cache Utilization Application Report (SPRA863)

Choosing an Appropriate Simulator Configuration in Code Composer Studio Application Report (SPRA864)
1 Introduction ................................................................. 1-1
Provides an introduction to the Analysis Toolkit.
1.1 Introduction .......................................................... 1-2

2 Analysis Toolkit in the Application Software Development Life Cycle .......... 2-1
Describes the capabilities of the analysis tools, and how they help application development.
2.1 Algorithms and Application Development Workflows ........................... 2-2
2.2 Using the Code Coverage Capability to Create Robust Applications .......... 2-4
2.3 Using the Multi-event Profiler Capability for Efficiency Analysis .............. 2-6
2.4 Using the Cache Analysis Tool for Efficiency Analysis .......................... 2-8

3 Installation .................................................................... 3-1
This chapter provides information on installing and uninstalling the ATK, along with the supporting software, examples, and documentation that come with the ATK.
3.1 Getting Started with the Analysis Toolkit ....................................... 3-2
3.2 What You Will Find in Your ATK Installation ................................. 3-3
  3.2.1 Visualization Tools and Scripts .............................................. 3-3
  3.2.2 Advanced ATK Control Panel .............................................. 3-3
  3.2.3 Documentation ............................................................... 3-4
  3.2.4 Examples ................................................................. 3-5
3.3 Uninstalling the ATK ...................................................... 3-6

4 Analysis Toolkit Usage .................................................... 4-1
This chapter reviews the steps needed to use the different features of the Analysis toolkit.
4.1 Select the Appropriate Simulator Configuration ................................. 4-2
4.2 Set Up the Simulator Configuration File for Data Collection ................. 4-3
4.3 Basic Usage Mode ................................................................ 4-7
  Iterative data collection in the current Code Composer Studio IDE session .... 4-7
4.4 Advanced Usage Mode ......................................................... 4-9
  4.4.1 Invoking the Advanced ATK Control Panel ............................ 4-9
  4.4.2 Starting a New Data Collection Session .................................. 4-11
  4.4.3 Closing the Ongoing Data Collection Session ......................... 4-12
  4.4.4 Saving the Settings specified in the Advanced ATK Control Panel .... 4-12
5 Batch-mode Data Collection and Visualization ........................................... 5-1
Provides an overview of automatically collecting trace data.
5.1 Generated Trace Files ............................................................................. 5-2
5.2 Batch-mode Data Collection in Code Composer Studio IDE ...................... 5-3
   5.2.1 Stopping Data Collection ................................................................. 5-4
   5.2.2 Post-processing Collected Data ....................................................... 5-4
   5.2.3 Launching Visualization Tools ......................................................... 5-5
   5.2.4 Controlling Data Collection ........................................................... 5-5
6 Guidelines ..................................................................................................... A-1
7 Quick Reference ............................................................................................. B-1
Figures

2–1 Using the Code Coverage and Multi-event Profiler, and Cache Analysis Tools for Algorithm Development ......................................................... 2-2
2–2 Using the Cache Analysis Tool for Application Development .................. 2-3
2–3 Sample Visualization of the Code Coverage Summary by Functions ........ 2-4
2–4 Sample visualization of Detailed Source Code Coverage Information ....... 2-5
2–5 Sample Visualization of Multi-event Profile Summary Over Functions ........ 2-6
2–6 Sample Visualization of Detailed Multi-event Profiler Information (Annotated C55x Source Code) .......................................................... 2-7
2–7 Sample Visualization of Cache Analysis Tool Showing Hits and Misses Over Time .................. 2-9
2–8 Sample Cache Visualization Showing the Interference Grid .................. 2-10
2–9 Data Access Pattern Before Optimization ......................................... 2-11
2–10 Data-Access Pattern After Optimization ........................................... 2-12
3–1 Startup Program Menu Highlighting ATK Shortcuts ............................... 3-2
4–1 Import Configuration Window ............................................................ 4-2
4–2 Analysis toolkit menu ........................................................................ 4-7
4–3 ATK Control Panel menu option ....................................................... 4-9
4–4 Advanced ATK Control Panel ........................................................... 4-10
4–5 ATK Control Panel Overview ............................................................ 4-11
5–1 Sample Perl Script Snippet Showing Data Collection Steps ................... 5-3
The Analysis Tool Kit provides powerful tools for application software development. These tools address two key aspects of development – application validation and efficiency analysis.

The cache analysis capability facilitates significant cache improvements to be obtained, by graphically highlighting inefficiencies in cache usage.

The multi–event profiler capability facilitates thorough performance analysis of applications. It helps identify where the performance losses are and why. By providing information regarding the sources of the inefficiency, it enables quick and effective optimizations to be performed.

The code coverage capability helps perform adequate validation of algorithms. By identifying uncovered source lines, it enables directed creation of new tests.

The ATK is available as an update over Code Composer Studio IDE 2.2 and is hosted over the simulation platform. The ATK release includes extensive documentation and examples to enable developers to derive maximum benefits of the tool kit.

All ATK capabilities are accessible from the Code Composer Studio IDE. The Advanced ATK control panel provides flexibility over data collection as well as helps automate it. Thus, the Analysis Tool Kit provides capabilities that bring significant value to the developer for the creation of correct and efficient applications.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Introduction</td>
<td>1-2</td>
</tr>
</tbody>
</table>
1.1 Introduction

Digital Signal Processing (DSP) solutions are becoming increasingly prevalent in many areas of computing – communication systems, imaging and audio systems, and other emerging application domains. Innovative and performance-hungry applications are being created to provide differentiated products to end-users. From a DSP application developer standpoint, one of the biggest challenges is to ensure that their implementations meet set stringent performance bounds.

Achieving maximum performance levels has significant advantages. Consider a communication system employing a digital signal processor for signal processing functions such as noise removal, echo cancellation etc. An initial implementation of the application software may consume processing bandwidth allowing the signal processor to support upto N data samples per second. By optimizing the software to consume less processor bandwidth per sample, we could facilitate greater than N data samples to be processed per second. The increased efficiency has far-reaching benefits. It allows more subscribers to be supported by the communication system with the same hardware infrastructure. This also helps drive costs down. In addition, the available headroom in performance may be utilized for deploying more sophisticated algorithms, or even for adding more features.

Creating such high performance applications requires a thorough understanding of and insight into the applications – how are different pieces of the software performing from an efficiency standpoint and why. There are many factors that affect performance. One key to high performance is obviously to boost the performance of the algorithms on the Central Processing Unit (CPU). Application developers typically look into CPU performance bottlenecks early in the life cycle of software development. While CPU bottlenecks are measured in terms of cycles, the causes are many: inefficient utilization of CPU parallelism, inefficient scheduling of instructions causing resource conflicts and stalls, inefficient memory references and so on. Knowing exact causes or origins of bottlenecks greatly aids efficiency analysis and guides the improvements to be made to the algorithm. Such an analysis requires extensive visibility into CPU behavior when applications are run on it.

The Analysis Tool Kit (ATK) is a set of tools that are created to specifically enrich robustness and efficiency analysis capabilities for application software development. The tool kit is available over Code Composer Studio (CCS) IDE v2.2 via the update advisor.
The Multi–event Profiler capability of the ATK is designed to facilitate CPU performance analysis. It provides extensive visibility into CPU behavior addressing the two key concerns – where the bottlenecks are, and why. Available on both the C55x and C6000 platforms, it provides function profile data over many potential performance bottlenecks – CPU cycles, executed instructions, pipeline stalls, cache hits and misses. Further, all profile data is visualized following a single run of the application, thereby presenting a complete visual summary of the performance. The single–run collection of data decreases the development time as well as avoids manual bookkeeping of multiple sets of data. In addition to the function performance summary, the Multi–event Profiler allows drilling down into specific problem areas. It visualizes application performance at the granularity of individual source lines. This is especially helpful for analyzing modules written in assembly language. Using this tool, you will be able to quickly identify and rectify bottlenecks in your C55x and C6000 applications.

Another key aspect to achieving high performance is the memory subsystem of the DSP platform. Due to its slower speed, the memory subsystem often becomes a critical bottleneck to the overall throughput of the application. If the memory does not “keep pace” with the CPU, it would starve the CPU of data causing under utilization of CPU bandwidth. In order to alleviate this, many processors have been designed with on–chip cache systems. Caches are small, fast memories that provide matching access speeds to the CPU. Caches are based on the principles of spatial and temporal locality of accesses to ensure that the CPU is not stalled for data frequently. A CPU access (program or data) to a location that is already present in the cache is termed a "cache hit." If the location was not present in the cache, it is termed a "cache miss".

Reducing cache misses is key to overcoming memory subsystem bottlenecks. To achieve this, developers need to have a thorough understanding of where cache misses occur during execution and why. Developers need to know which functions and/or data accesses suffer high cache miss counts. Further, they need to know why high miss counts are occurring. Such analysis facilitates significant cache optimizations to be performed. It enables achieving efficient placement of functions and data in memory to avoid mutual cache access conflicts (accesses to locations mutually replace one another in the cache causing repeated cache misses for these locations). It also enables identifying and improving patterns of memory accesses that are causing non–optimal cache utilization, so that the developer can easily spot the areas needing improvement. For example, modifying the stride of array accesses or changing a column–major access to a row–major access can significantly reduce cache
misses. Such analysis requires extensive insight into the run–time behavior of the application with regard to the caches.

The Cache Analysis tool in the ATK is designed to support this analysis. It graphically visualizes program and data cache accesses over time. This enables quick and effective identification of poor cache utilization thereby guiding the optimizations. Available on both the C55x and C6000 platforms, the tool significantly simplifies analyses such as:

- Identifying which functions or data are mutually conflicting one another,
- and
- Which data memory accesses are inefficiently utilizing the cache

Many C55x and C6000 applications have had their memory subsystem cycle counts significantly cut down with the aid of this tool. The application note "Using the Cache Analysis Tool to Improve Cache Utilization" presents examples where over 50% reduction in CPU cycles was obtained using this tool.

While achieving high performance levels remains a very important objective of embedded application software development, software validation has also acquired critical importance. DSP applications are increasingly deployed into many areas including healthcare, automotive, robotics and control. In all cases, it is important that the application behaves correctly. Thus, thorough validation of all aspects of the software is essential. An important aspect of validation is to know what parts of code have been tested and what aspects need testing. In today’s complex applications, the problem of determining sufficiency of tests is a particularly hard problem. The ATK provides the Code Coverage capability to enable this analysis. By means of this capability, developers may easily obtain quantified measures of which lines of code have been executed and which functions still require further validation or examination.

The Code Coverage tool shows source code in a color–coded visualization thereby aiding quick identification of sources that need further validation. It also provides a function summary of coverage information for quick navigation. The application note "Using the Code Coverage and Multi–event Profiler for Robustness and Efficiency Analyses" describes an example of how 100% code coverage was attained on sample functions by means of this tool.

The ATK is utilized on the simulation platforms. All the above capabilities are provided on both the C55x and C6000 platforms to support the application developer. These tools help tackle the real development problems of ensuring correctness and efficiency.
Analysis Toolkit in the Application Software Development Life Cycle

The analysis tool kit supports the complete application software development life cycle. Typically, development starts with the creation of algorithms. At this stage, the code coverage tool may be used for thorough validation of algorithms. Subsequently, algorithms are tuned for high performance (to meet performance budgets). At this stage, the multi-event profiler and cache analysis tools may be used to identify where the performance losses are and what the contributing factors are. Next, optimized algorithms are integrated to form the complete application. This stage may also include: use of peripherals and DMAs for data I/O, efficient buffer management, use of BIOS for multiple algorithm integration. Once the application is created, it is also tuned for performance. At this stage, the cache analysis tool may be used for improving program and data cache performance by optimizing the program and data layout in the system.

See Figure 2–1 and Figure 2–2 for a quick summary of how the tools fit into the development cycle.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Algorithm and Application Development Workflows</td>
<td>2-2</td>
</tr>
<tr>
<td>2.2 Using the Code Coverage Capability to Create Robust Applications</td>
<td>2-4</td>
</tr>
<tr>
<td>2.3 Using the Multi-event Profiler Capability for Efficiency Analysis</td>
<td>2-6</td>
</tr>
<tr>
<td>2.4 Using the Cache Analysis Tool for Efficiency Analysis</td>
<td>2-8</td>
</tr>
</tbody>
</table>
2.1 Algorithms and Application Development Workflows

Figure 2–1. Using the Code Coverage and Multi-event Profiler, and Cache Analysis Tools for Algorithm Development
Figure 2–2. Using the Cache Analysis Tool for Application Development
2.2 Using the Code Coverage Capability to Create Robust Applications

The code coverage tool helps algorithm validation. It highlights the source code coverage obtained on the application run over a set of test vectors. Developers can thus quickly identify what aspects of functionality have not been verified, and construct further test vectors to cover these.

A sample visualization of the code coverage summary information is shown in Figure 2–3.

Figure 2–3. Sample Visualization of the Code Coverage Summary by Functions

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>I</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>File</td>
<td>Line no.</td>
<td>Size(byte)</td>
<td>Start address(hex)</td>
<td>#times called</td>
<td>%coverage</td>
<td>Total Instructions</td>
</tr>
<tr>
<td>2</td>
<td>Select_func</td>
<td>select_func.c</td>
<td>3</td>
<td>188</td>
<td>0x0000001C</td>
<td>2</td>
<td>75</td>
</tr>
<tr>
<td>3</td>
<td>main</td>
<td>main.c</td>
<td>5</td>
<td>436</td>
<td>0x000000A660</td>
<td>1</td>
<td>85</td>
</tr>
<tr>
<td>4</td>
<td>Others</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>27425</td>
<td></td>
</tr>
</tbody>
</table>

In this summary, the tool shows the percentage coverage that has been achieved for each function on a run of the application. This helps quickly identify functions that have not been covered adequately. Once a function has been identified, detailed coverage information over the source code of the selected function may be obtained. A sample visualization of this detailed source code coverage information is shown in Figure 2–4.
Using the Code Coverage Capability to Create Robust Applications

Figure 2–4. Sample visualization of Detailed Source Code Coverage Information

By coloring the uncovered source lines in “red”, and the covered lines in “green”, the tool enables easy visual identification of source lines that have not been covered. This in turn facilitates construction of appropriate tests.

The application report “Using the Code Coverage and Multi–event Profiler Tool for Robustness and Efficiency Analyses” describes in detail how the tool facilitates code coverage analysis. It provides a step–by–step example of achieving 100% code coverage on a sample function. It also shows how the tool helps identify unreachable code.
Using the Multi-event Profiler Capability for Efficiency Analysis

One of the most important objectives of DSP application developers is to satisfy the stipulated performance budget imposed on the application. Thus, a significant amount of effort is spent in identifying bottlenecks in algorithms and resolving them.

The Multi–event profiler tool presents extensive profile data over functions. The profile data includes counting many events of interest such as CPU cycles, executed instructions, pipeline stalls, cache hits and misses. Further, all profile data is accumulated in a single run of the application with little overhead in simulation performance (for example, the overhead is less than 5% on the C55x simulator platform). Based on the profile data, developers can quickly identify performance losses as well as possible sources of the losses, and make necessary source code changes.

A sample visualization of the function profile summary information is shown in Figure 2–5.

![Sample Visualization of Multi-event Profile Summary Over Functions](image)

The summary shows multiple event counter values for each function that was executed. Knowing CPU cycle counts helps identify which functions are performing inefficiently. In addition, knowing the sources of the inefficiency – high CPU stalls, high cache misses and so on – helps identify optimizations. This capability significantly benefits efficiency analysis.

For assembly code, it is also possible to obtain detailed profile data at the granularity of individual source lines. This enables effective identification of sources of inefficiency – instructions that are stalling, resources involved in causing stalls and so on. A sample visualization of this detailed profile information is shown in Figure 2–6.
The application report “Using the Code Coverage and Multi–event Profiler Tool to Increase Application Robustness and Efficiency” describes in detail how the tool facilitates efficiency analysis. It provides detailed examples illustrating the utility of the multi–event nature of the profiler. On the C6000 example, we demonstrate over 50% reduction in CPU cycles by removing memory bank conflicts; on the C55x example, we demonstrate over 30% reduction in CPU cycles by removing address phase stalls and prefetch stalls.
2.4 Using the Cache Analysis Tool for Efficiency Analysis

Optimizing the application to make efficient use of the memory subsystem is key to meeting performance goals. Efficient use of on–chip caches is of importance as it improves the CPU throughput by reducing CPU stall cycles due to memory activity.

The cache analysis tool is effective at highlighting inefficient cache usage. It graphically shows access patterns over time, color–coding the accesses to distinguish cache hits from misses (see Figure 2–7 for a sample visualization of the cache analysis tool). Cache hits are shown as “green” pixels while cache misses are shown in “red”. The accessed addresses are plotted along the Y–axis while access times are plotted along the X–axis. The tool also provides symbolic information display for addresses. In addition, the tool provides various filters, panning and zoom features to navigate and drill down into problem areas.

This tool visualizes both program and data cache accesses. Using this visualization, it is possible to quickly identify functions or data structures that are conflicting one another due to their placement in memory. The “interference grid” feature in the tool allows for visualizing all conflicting addresses with respect to a chosen address. Figure 2–8 below shows a sample visualization of this feature on the program cache. By observing patterns of cache misses at two or more grid lines, we can clearly identify functions that are mapped to the same cache sets causing them to thrash one another.

The tool also enables restructuring of access patterns as to improve cache performance. Figure 2–9 and Figure 2–10 show an example of an optimization that was guided by this visualization.
Figure 2–7. Sample Visualization of Cache Analysis Tool Showing Hits and Misses Over Time
Figure 2–8. Sample Cache Visualization Showing the Interference Grid
Using the Cache Analysis Tool for Efficiency Analysis

Figure 2-9. Data Access Pattern Before Optimization
The application report “Using the Cache Analysis Tool to Improve Cache Utilization” describes in detail how the tool facilitates cache efficiency analysis. It provides a step–by–step example highlighting the use of the tool for both program and data cache optimizations. The example shows how data cache misses were reduced by over 90% and program cache misses were reduced by over 99% resulting in 50% improvement in the total cycles consumed by the application.
The ATK is available as an update over Code Composer Studio IDE 2.2 via the update advisor.

The ATK is available on both the C6000 and C55x simulator platforms. Separate update patches are released for the two platforms.

### Topic | Page
--- | ---
3.1 Getting Started with the Analysis Toolkit | 3-2
3.2 What You Will Find in Your ATK Installation | 3-3
3.3 Uninstalling the ATK | 3-6
3.1 Getting Started with the Analysis Toolkit

The ATK is available as an update over Code Composer Studio IDE 2.2 via the update advisor.

The ATK is available on both the C6000 and C55x simulator platforms. Separate update patches are released for the two platforms. Users may install the ATK as an update over Code Composer Studio v2.2.

When installation is complete, a shortcut in the startup program menu is created under respective Code Composer Studio IDE installation. For example, when the C55x ATK update is installed, the shortcut “Analysis Tool Kit” appears in the startup program menu under C55x menu. It has these submenus:

- The ATK user guide (this document)
- The cache analysis tool
- Uninstall

Figure 3–1. Startup Program Menu Highlighting ATK Shortcuts
3.2 What You Will Find in Your ATK Installation

The following components are installed into the Code Composer Studio IDE 2.2 installation as part of the ATK

- Visualization tools and scripts
- ATK control panel
- Documentation
- Examples

3.2.1 Visualization Tools and Scripts

The visualization tools and scripts are installed under `<CCS 2.2 Installation Root>\bin\utilities` directory.

For example, if C6000 Code Composer Studio IDE 2.2 is installed at C:\ti directory, then you will find the tools under C:\ti\bin\utilities directory.

Note that the cache analysis tool is also accessible from the ATK shortcut installed under the start-up program menu (see Figure 3–1).

The code coverage and multi–event profiler data is visualized using Microsoft Excel. It is assumed that Microsoft Excel tool is available on the user machine.

3.2.2 Advanced ATK Control Panel

The ATK control panel is a Code Composer Studio IDE tool that facilitates control over data collection. The control panel allows developers to manually control the starting and stopping of data collection. In addition, the control panel allows specification of other settings such as directory path for trace file generation, additional source file search path and so on. This tool is meant for advanced usage wherein developers have explicit control over all aspects of data collection.
3.2.3 Documentation

The following documentation is available as part of the ATK to facilitate both easy access to information as well as enable best use of the tools:

**The ATK user guide (this document):**

This document introduces the ATK, its capabilities and usage. It provides a step–by–step procedure for running the tool kit.

**Code Coverage and Multi–event Profiler User’s Guide (SPRU624):**

This document is a user guide to the code coverage and multi–event profiler capability. It describes the tool, its set–up, invocation, and use.

**Using Code Coverage and Multi–event Profiler for Robustness and Efficiency Analyses Application Report (SPRA868):**

This application report illustrates, via simple examples, how to use the tool for performing coverage analysis and efficiency analysis.

**Cache Analysis User’s Guide (SPRU575A):**

This document is a user guide to the cache analysis tool. It describes the tool, its set–up, invocation, and use.

**Using the Cache Analysis Tool to Improve Cache Utilization Application Report (SPRA863):**

This application report demonstrates, via simple examples, how to use the tool for performing cache optimizations.

Readers are urged to consult relevant documentation for usage steps as well as to make effective use of the tools.

All documentation is installed into the `<CCS 2.2 installation>/docs/pdf directory`:
### 3.2.4 Examples

Example executables and source code are provided to illustrate the ATK tools' capabilities. They are provided on both the C6000 and C55x simulator platforms:

1) **Cache Analysis Examples:**

   - **Matrix Multiply:**
     This example is available on the C6000 platform. It has two versions of the matrix multiplication algorithm – the basic, three–loop version and the block–based version. It shows how temporal locality is improved in the block–based version improving data cache performance dramatically.

   - **Conflict:**
     This example is available on the C6000 platform. It highlights how function placement conflicts may be identified using this tool.

   - **Jpeg:**
     This example is available on the C6000 platform. It highlights program cache misses. It provides a good example of the use of the tool on a real–world application.

   - **Matrix Transpose Operation:**
     This example is available on both the C55x and C6000 platforms. It has two versions of a matrix algorithm – a basic, two–loop version and a block–based version. The basic version suffers high data cache and program cache misses. The block–based version improves temporal locality of the data significantly thereby reducing data cache misses. Further, it highlights how function placement conflicts may be identified using this tool.

2) **Code Coverage Examples:**

   - **Tree:**
     This example is available on both the C55x and C6000 platforms. It reads sequences of numbers from files, inserts them into a binary search tree and then traverses this tree to print out contents. It provides sample input sequences to demonstrate the code coverage utility.

   - **Interpreter:**
     This example is available on both the C55x and C6000 platforms. It simulates a simple assembly–like language. It reads “programs” written in this language, simulates them and outputs “results”. It highlights how new test vectors may be identified by means of the source code coverage information.
3.3 Uninstalling the ATK

To uninstall the ATK, do one of the following:

- Select “Uninstall ATK” from the respective Code Composer Studio IDE start menu

  OR

- Use the “Add/Remove Programs” tool in the Windows control panel.

The uninstall process removes all files & registry settings specific to the ATK.
Analysis Toolkit Usage

This chapter provides you with step–by–step instructions for using the Analysis toolkit, as well as advanced features that can be utilized when needed.

Once installed, the analysis tool kit is accessible from the Code Composer Studio IDE. The following steps need to be performed to start using the tool kit:

**Step 1:** Select the appropriate simulator configuration

**Step 2:** Set up the simulator configuration file for data collection

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Select the Appropriate Simulator Configuration</td>
<td>4-2</td>
</tr>
<tr>
<td>4.2 Set Up the Simulator Configuration File for Data Collection</td>
<td>4-3</td>
</tr>
<tr>
<td>4.3 Basic Usage Mode</td>
<td>4-7</td>
</tr>
<tr>
<td>4.4 Advanced Usage Mode</td>
<td>4-9</td>
</tr>
<tr>
<td>4.4.1 Invoking the Advanced ATK Control Panel</td>
<td>4-9</td>
</tr>
<tr>
<td>4.4.2 Starting a New Data Collection Session</td>
<td>4-11</td>
</tr>
<tr>
<td>4.4.3 Closing the Ongoing Data Collection Session</td>
<td>4-12</td>
</tr>
</tbody>
</table>
4.1 Select the Appropriate Simulator Configuration

As described in Section 2.1, on page 2-2, application development goes through various stages requiring appropriate simulator support. The application report “Using the Right Simulator Configuration” describes TI simulation offerings and explains how developers may pick appropriate configurations to suit their needs. Readers are urged to understand the suite of simulator configurations offered and make appropriate selections.

For the example, let us use the C6713 functional sim Little Endian configuration. First, cc_setup is invoked and this configuration file is imported:

*Figure 4–1. Import Configuration Window*
4.2 Set Up the Simulator Configuration File for Data Collection

Each simulator configuration is associated with a simulator configuration file. This file may be accessed from cc_setup as shown in the following steps:

**Step 1:** Right–click on the selected board (in this case, the C6xxx Simulator (Texas Instruments) board) displayed in the System Configuration window.

**Step 2:** Select Properties from the drop-down menu to display the Board Properties window.

**Step 3:** Select the Board Properties tab.

The Value field entry is the name of the simulator configuration file.
Step 4: Select the simulator configuration file supplied with the Analysis toolkit.

The Analysis toolkit requires profile data collection related entries to be enabled in the simulator configuration file. To provide a convenient way of enabling data collection, the toolkit provides configuration files that already have these entries appropriately set.

These configuration files have the names with suffix "_profile" and are available in the drivers directory inside the Code Composer Studio IDE installation. Thus, for example, if the imported simulator configuration filename is "sim6713_functional_simulator.cfg," the toolkit–supplied configuration file will be named "sim6713_functional_simulator_profile.cfg."

By clicking the file browse button, you can browse to and select the appropriate toolkit–supplied configuration file.

Step 5: If desired, bring up this configuration file in a text editor and look for the profile module.

This section has entries that enable collection for the Code Coverage and Multi–event Profiler tool as well as the Cache Analysis tool.

By default, this configuration file may be used as is. However, it is necessary to be aware of the data collection entries in it in order to turn off data collection or to modify the cache time interval value.

See the next image.
Set Up the Simulator Configuration File for Data Collection

Step 6: Set up data collection for the Code Coverage and Multi–event Profiler tool.

The following entry needs to exist in the profile module to set up data collection for the Code Coverage and Multi–event Profiler tool.

```
MULTI_EVENT_PROFILE_AND_COVERAGE  ON;
```

Step 7: Set up data collection for the Cache Analysis tool.

The following entry needs to exist in the profile module to set up data collection for the Cache Analysis tool.

```
CACHE_PROFILE  <collection time interval>;
```

The collection time interval is a parameter that controls the cycle duration of cache accesses that get recorded as a single record in...
the trace file. See the *Cache Analysis User’s Guide* (SPRU575A) for information on selecting the appropriate collection time interval values for your applications.

For example, if the desired time interval is 1000, set the entry to:

```
CACHE_PROFILE 1000;
```

**Note:**

- Different simulator configurations support different sets of analysis capabilities. Consult the respective ATK tool user’s guides to make sure that your chosen configuration supports the analysis capability you need.

- If data collection is turned on, it impacts the simulation performance. This overhead depends on what data collection is turned on. Hence, it is advised that only the relevant data collection be turned on to achieve highest possible simulation performance.

- If the configuration file has been edited, save this file and save the system configuration in preparation for launch Code Composer Studio using the new configuration.

- It is important that the simulator configuration file is correctly edited. It is recommend that, before any changes to the supplied configuration files are made, you make a copy of the configuration file. The saved copy can then be edited and selected via CCStudio Setup. The configuration can also be exported via CCStudio Setup for future use.

Having set up the simulator configuration file to enable appropriate data collection, the analysis toolkit software is now ready for use.

The toolkit provides three primary modes of usage:

- Basic
- Advanced
- Batch
4.3 Basic Usage Mode

Having set up the simulator for profile data collection as mentioned in the previous section, the analysis tool kit is ready for use. In basic usage mode, the analysis tool kit is used in the following sequence:

1) Launch Code Composer Studio IDE.
2) Load application program.
3) Run the application to completion.
4) Launch visualization tools.

Data collection is started as soon as the application program is loaded. While the program runs, the simulator accumulates the profile data. The data collection is stopped when the user launches a visualization tool.

Visualization tools can be launched from the Analysis toolkit menu off of the Tools menu.

Figure 4–2. Analysis toolkit menu

Note:
The trace files generated by the simulator are output to the directory from which the application program was loaded.

Iterative data collection in the current Code Composer Studio IDE session

Having visualized and analyzed the profile data, users can restart collection in one of the following ways:
Basic Usage Mode

- Reset, restart the application
- Modify application, rebuild and reload the application
4.4 Advanced Usage Mode

This usage mode supports mechanisms to accomplish the following:

1) Explicit control over data collection
   a) Starting a new data collection session
   b) Closing the ongoing data collection session

2) Controlling the directory at which trace files will be generated

3) Other settings
   a) Additional search paths for source files
   b) Log file path

All these mechanisms are accessible from the advanced ATK control panel.

4.4.1 Invoking the Advanced ATK Control Panel

The control panel is located under the Tools Menu (see the following figure).

Figure 4–3. ATK Control Panel menu option

You should see the control panel window as shown in the next figure:
Invoking the Advanced ATK Control Panel

Figure 4–4. Advanced ATK Control Panel
About the ATK Control panel

The control panel allows control over duration of data collection.

1) When user clicks on button "Start New Session", data collection is started; and the button toggles to "Close Session".

2) When user clicks on button "Close Session", data collection is stopped; and the button toggles to "Start New Session".

The control panel allows user configuration of following settings:

1) Output directory for trace files dump: Trace data files are generated in the directory mentioned in this field; by default, data files are generated into the directory from which the program was loaded.

2) Search path for source files: The code coverage and multi–event profiler tool requires access to source files. The path entries here (separated by semi–colons) along with the paths specified in the currently active project are searched for source files.

3) Log file name with complete path: The analysis tool kit logs all error messages and warnings to a log file. The path of the log file is specified in this entry.

The user may access online help on any of these features by clicking on the "Help" button.

4.4.2 Starting a New Data Collection Session

Data collection for analysis is started manually by clicking on the Start New Session button in the control panel.

Manual mode of starting a data collection session is useful when only a part of the complete run of the application is to be analyzed. In such a case, user sets a breakpoint in code to detect when the starting point for data collection is reached. When the breakpoint is reached, the user manually starts the data collection session. As an example, this mode is useful in analyzing the data collected over one specific frame of execution in a multi–frame simulation run.

Note:

In manual mode, before starting data collection, first you need to close any ongoing session. Since program load would have implicitly started collection, you need to click on the button two times:

- First click on "Close Session" to stop any on–going collection
- Next click on "Start New Session" to start your collection
4.4.3 Closing the Ongoing Data Collection Session

Data collection session is closed (stopped) manually by clicking on the Close Session button in the control panel.

Manual mode of stopping data collection is useful when only a part of the complete run of the application is to be analyzed. In such a case, user sets a breakpoint in code to detect when the stopping point for data collection is reached. When the breakpoint is reached, user manually stops collection.

4.4.4 Saving the Settings specified in the Advanced ATK Control Panel

It is possible to save the specified settings in the Advanced ATK Control panel by saving the workspace. The workspace needs to be saved using the CCS File→Workspace→Save Workspace As menu item.

Thus, users could specify the appropriate trace file dump folder, paths of source files and log file path via the Advanced ATK Control panel only once and have these settings saved. Subsequent Code Composer Studio IDE invocations could load this saved workspace to ensure that the specified settings in the control panel are automatically used.
Batch-mode Data Collection and Visualization

It is often desirable to be able to collect trace data in an automated way. Typically, if trace data is to be collected over many runs of the application or over many applications, then it is preferable to use a “script” that runs the set of applications in a batch, and collects & stores trace data from every run. All collected data may then be visualized offline (without invoking Code Composer Studio IDE). Batch mode is also useful for applications than run for large periods of time and collect big trace files.

Batch mode collection of trace data is facilitated via Code Composer Studio IDE scripting. Code Composer Studio IDE scripting uses Perl and is available as an update over Code Composer Studio IDE 2.2.

Code Composer Studio IDE scripting provides access to all the basic functions of Code Composer Studio IDE via Perl functions. Using Code Composer Studio IDE scripting, it is possible to launch Code Composer Studio IDE, open projects, load and run programs, and collect trace data.

Typically, Code Composer Studio IDE scripting is installed under the folder:

<CCS Installation Path>/utilities/ccs_scripting

In batch mode collection, the simulator creates raw trace files containing collected data. In some cases, these trace files may require post-processing in order to visualize them.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Generated Trace Files</td>
<td>5-2</td>
</tr>
<tr>
<td>5.2 Batch-mode Data Collection in Code Composer Studio IDE</td>
<td>5-3</td>
</tr>
</tbody>
</table>
5.1 Generated Trace Files

All trace files are named after the name of the corresponding program that generated them. For instance, suppose the name of the program that was loaded is “hello.out”. Then, all trace files are named “hello.<extension>” where <extension> denotes the type of trace.

Table below documents the trace files that are generated.

<table>
<thead>
<tr>
<th>Trace Files</th>
<th>Tool(s) to be run</th>
<th>Remarks</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;coff file&gt;.xls</td>
<td>Microsoft Excel</td>
<td>Used for code coverage and multi–event profiler visualizations</td>
<td>hello.xls</td>
</tr>
<tr>
<td>&lt;coff_file&gt;.tprof</td>
<td>Human–readable format</td>
<td>This trace file contains raw data required for code coverage and multi–event profiler visualizations. This file needs to be post–processed in order to create corresponding MS Excel file. This file is published in a human–readable format; This allows custom post–processing utilities to be built.</td>
<td>hello.tprof</td>
</tr>
<tr>
<td>&lt;coff_file&gt;.i.tip</td>
<td>Cacheanalysis.exe</td>
<td>Used for visualizing program cache trace using cache analysis tool.</td>
<td>hello.i.tip</td>
</tr>
<tr>
<td>&lt;coff_file&gt;.d.tip</td>
<td>Cacheanalysis.exe</td>
<td>Used for visualizing data cache trace using cache analysis tool.</td>
<td>hello.d.tip</td>
</tr>
<tr>
<td>&lt;coff_file&gt;.x.tip</td>
<td>Cacheanalysis.exe</td>
<td>Used for visualizing cross reference data cache trace using cache analysis tool.</td>
<td>hello.x.tip</td>
</tr>
</tbody>
</table>
5.2 Batch-mode Data Collection in Code Composer Studio IDE

The sequence of steps to be used for batch mode data collection and visualization are:

Step 1: Launch Code Composer Studio IDE.
Step 2: Load the program.
Step 3: Run the program.
Step 4: Stop data collection.
Step 5: Post process the collected data.
Step 6: Launch the visualization tool(s).

The figure below provides an illustrative example of such a Code Composer Studio IDE script.

All the data collection and post-processing functions are provided by the Perl module `ATK_functions.pm` located in the Code Composer Studio IDE scripting directory.

The functions provided by this module are described below.

Figure 5–1. Sample Perl Script Snippet Showing Data Collection Steps

```perl
my $MyCCScripting = new CCS_SCRIPTING_PERL::CCS_Scripting();
# Open CCS
$MyCCScripting –> CCSOpen($CCS_SCRIPTING_PERL::ISA_C64,
    0, 0, $CCS_SCRIPTING_PERL::PLATFORM_SIMULATOR, 1);
# Open the project
#$MyCCScripting –> ProjectOpen($MyProject);
# Build a project
#$MyCCScripting –> ProjectBuild();
# Load program
$MyCCScripting –> ProgramLoad($MyProgram);
# Run program
$MyCCScripting –> TargetRun;
# Stop Data collection
ATKStopDataCollection $MyCCScripting;
# Post–process collected data - required only for Code Coverage and Multi–event Profiler tools

# step 1 - merge raw coverage outputs from multiple runs (if required)
ATKMergeCoverage $CCS_DIR,, $curr_tprof, $merged_tprof;
# step 2 - post process raw output to MS Excel format
ATKprof2xls $CCS_DIR, $MyProgram, $merged_tprof, $ProjDir;
# Launch visualization tools
```
5.2.1 Stopping Data Collection

Data collection should be stopped before collected data can be post-processed for visualization. This is done by the perl function:

\texttt{ATKStopDataCollection}

This function dumps out collected trace data files to the directory from which the program executable was loaded. The only argument required by this function is the handle to the Code Composer Studio IDE Scripting object.

5.2.2 Post-processing Collected Data

Post-processing of collected data is required only for visualizing Code Coverage or the Multi-event Profiler.

5.2.2.1 Merging Trace Data Collected Across Multiple Runs

A common requirement while using Code Coverage is the ability to be able to merge different coverage data from multiple runs of the application code. Typically, users may run their applications many times, each time with a different set of test vectors. At the end of a few runs, they need to visualize cumulative coverage obtained over on the application over all the application runs. This is achieved by using the perl function:

\texttt{ATKMergeCoverage}

This function merges the tprof trace data collected in the current run with those already collected from previous runs.

This function takes three arguments:

- The Code Composer Studio IDE installation path
- The tprof file obtained from the current run of the application
- The merge destination tprof file (this file holds the merged trace data from all previous runs)

Note that it is up to the user to identify the merge destination tprof file and to make sure that this file can be written to.

It may also be noted that this function need not be used if the user is not interested in viewing merged code coverage data.
5.2.2.2 **Converting the .tprof File**

The following perl function is provided to convert a tprof trace file to a MS Excel (.xls) file:

```
ATKTprof2Xls
```

This function takes as input a .tprof file and corresponding source file path information to output a MS Excel (.xls) file.

This function takes four arguments:

- The Code Composer Studio IDE installation path
- The name of the loaded program
- The tprof file to be converted
- Semi–colon (`;`) separated string of source file paths. The function searches all paths in this string to locate and read associated source files. These paths are necessary to construct source code coverage information on a source line–by–line basis.

5.2.3 **Launching Visualization Tools**

Once post-processing is completed, the generated MS Excel file may be launched directly by the user by double–clicking on the file in Windows Explorer.

As indicated earlier, the cache trace files (*.tip files) need no post–processing steps. They may be visualized by launching the Cache Analysis tool. The cache analysis tool is installed as:

```
<CCS installation>\bin\utilities\cacheanalysis\cacheanalysis.exe
```

5.2.4 **Controlling Data Collection**

In the above steps, there was no need to explicitly start data collection. Data was implicitly collected for the entire run of the application. However, if users wanted to collect data only over a part of the entire application run, then they need to explicitly:

- Start data collection, and
- Stop data collection

Typically, the workflow would involve the use of breakpoints to determine when to start or stop data collection:
**Batch-mode Data Collection in Code Composer Studio IDE**

**Step 1:** Launch Code Composer Studio IDE.

**Step 2:** Load the program.

**Step 3:** Stop data collection (to avoid collecting data that is not needed).

**Step 4:** Set up breakpoint in code (to determine the start of data collection).

**Step 5:** Run the program till breakpoint.

**Step 6:** Start data collection.

**Step 7:** Set up the breakpoint in code (to determine the stop of data collection).

**Step 8:** Run the program till breakpoint.

**Step 9:** Stop data collection.

**Step 10:** Post process the collected data.

**Step 11:** Launch the visualization tool(s).

Among the steps above, the step of starting data collection requires the use of the ATK function:

`ATKStartDataCollection`

This function starts data collection. The only argument required by this function is the handle to the Code Composer Studio IDE Scripting object.
Appendix A

Guidelines

1) Trace files can grow to large sizes if analysis data is collected over long periods of simulation. Be sure to remove unwanted trace files. The analysis tool kit does not remove trace files.

2) If you do not need analysis data collection, comment out the appropriate entries in the simulator configuration file. This ensures that only the relevant trace files get created. Further, since collection comes with a performance overhead on simulation, commenting out unwanted entries helps improve simulation performance. Alternately, you could also use the standard simulator configuration (without the suffix “_profile”) file.

3) As mentioned earlier, whenever data collection is started, the associated trace files are re-opened for writing; thus any old contents are lost. To prevent loss of existing trace data,
   a) Either rename old trace files before program load, or
   b) Rename the program to be loaded

4) Typically the visualization tools are used as part of an analyze, modify, rebuild and rerun sequence for iterative analysis. This involves generating trace files with the same names as those already generated. Thus, before starting data collection for a subsequent run, visualization tools that are visualizing the trace output of the current run need to be quit. Alternatively, the trace files from the current run may be “saved as” copies and the copies visualized. Saving a copy also enables comparison of data from different runs.
Appendix A
Quick Reference

This appendix provides a quick reference guide while working with the analysis toolkit.

**Basic Usage steps:**

**Step 1:** Launch Code Composer Studio IDE and load your application program.

**Step 2:** Run the program to completion.

**Step 3:** Launch visualization tools.

**Advanced Usage steps:**

**Step 1:** Launch Code Composer Studio IDE and load your application program.

**Step 2:** Launch the Advanced ATK Control Panel.

**Step 3:** Edit the settings in the control panel.

**Step 4:** Set up a breakpoint in the code at the point where you want to start data collection.

**Step 5:** Run the program till it hits the breakpoint.

**Step 6:** Start data collection by clicking on the Start New Session button in the Advanced ATK Control Panel.

**Step 7:** Set up a breakpoint in the code at the point where you want to stop data collection.

**Step 8:** Run the program till it hits the breakpoint.

**Step 9:** Stop data collection by clicking on the Close Session button.

**Step 10:** Launch the visualization tools.
Quick Reference

**Batch Usage steps (with Code Composer Studio Perl Scripting):**

**Step 1:** Launch Code Composer Studio IDE and load your application program.

**Step 2:** Run the program to completion.

**Step 3:** Stop data collection by calling "ATKStopDataCollection."

**Step 4:** Optionally merge different tprof files by calling "ATKMergeCoverage."

**Step 5:** Post process the tprof file to obtain an MS Excel file by calling "ATKTprof2Xls."