

Using Code Coverage and Multi-event Profiler in Code Composer Studio v2.3 for Robustness and Efficiency Analyses

Amit Rangari, N.Pradeep

Software Development Systems

ABSTRACT

Given the complexity of DSP application software development, powerful tools are required for debugging, ensuring robustness, and analyzing the performance of applications.

In this application report, a new tool, Code Coverage and Multi-event Profiler, is presented. This tool provides the following capabilities:

- Source code coverage information to increase robustness
- Function profiling over multiple events of interest that affect code performance

This application report describes how these capabilities can be used during software development. It also illustrates the benefits of the tool by providing simple examples.

Contents

1	Introduction	3
2	Using the Code Coverage/Multi-event Profiler Tool in the Software Application Development Flow	5
3	Tool Overview	5
4	Using the Tool for Code Coverage Analysis	7
4.1	Example	7
4.2	Recommended Use	11
5	Using the Tool for Efficiency Analysis	11
5.1	C55x Example	12
5.2	Recommended Use	17
6	Conclusions	18
7	References	18
Appendix A	Source Code Listing for Code Coverage Example	19
Appendix B	Source Code Listing for C5500 Multi-event Profiler Example	20

List of Figures

Figure 1. Code Coverage	3
Figure 2. Multi-event Profiler	4
Figure 3. Using the Tool in the Software Development Flow	5
Figure 4. Summary View	6
Figure 5. Annotated Source Code	6
Figure 6. Sample Source Code for Code Coverage	7
Figure 7. Applying Test Vectors	8
Figure 8. Coverage Summary After First Run	8
Figure 9. Source Code Coverage After First Run	9
Figure 10. Coverage Summary After Second Run	9
Figure 11. Source Code Coverage After Second Run	10
Figure 12. Coverage Summary After Dead Code Elimination	10
Figure 13. A Typical Flow to Follow	11
Figure 14. Code Snippet to Demonstrate Use of the Multi-event Profiler for Identifying and Removing Pipeline Stalls	12
Figure 15. Multi-event Profile Output After First Run	13
Figure 16. Profile Data Sorted By CPU Cycles	14
Figure 17. Source Code for vectAdd.asm	15
Figure 18. Snippet of Code After Removing Address Phase Stalls	16
Figure 19. Multi-event Profile Output After Removing Address Phase Stalls	17

1 Introduction

DSP application developers face many challenges. These include:

- Reduced time-to-market for end products
- Increased application size and complexity
- Aggressive application performance budgets
- Increasingly complex DSP architecture platforms for implementation

In such a context, developers need powerful software tools to quickly create both correct and efficient applications.

The code coverage and multi-event profiler tool, available in the Analysis Toolkit (ATK), provides two important capabilities:

- Code coverage information identifying source code not exercised in a run of the application, and facilitating the construction of test vectors to ensure adequate code coverage. See Figure 1 for an overview.
- Profile data for functions over multiple events of interest in a single run of the application. Events include common CPU events (cycles, executed instructions, performance stalls) and memory events (cache hits/misses) that can highlight hotspots in the code. See Figure 2 for an overview.

This application report discusses the capability of this tool to provide information used in the application software development life cycle. The rest of the document is organized as follows:

- Section 2 discusses where the tool fits in the application software development life cycle.
- Section 3 provides an overview of the tool and how it is accessed as part of the ATK.
- Section 4 discusses the use of the tool for analyzing source code coverage, and illustrates the sequence of steps to be followed by means of a small example.
- Section 5 discusses the use of the multi-event profiler capability for performance analysis, and illustrates the sequence of steps to be followed with the help of an example.

What is Code Coverage?

Developers test the software program functionality by creating test vectors. When the program is run on a given test vector, it exercises only a subset of the source code. This is said to be the code covered by the test vector. Code coverage is the ratio of the number of source lines executed and the total number of source lines.

Why is Code Coverage important?

Knowing what parts of code have not been exercised helps construct further test vectors. Rather than construct a number of tests in an ad-hoc manner, developers can judiciously build new tests to attain better coverage over their code. Code coverage provides a meaningful, quantifiable mechanism towards improving product quality.

Full code coverage does not ensure all paths of execution have been exercised. Code coverage only highlights if a given source line was executed or not. It does not indicate if a given source line was executed via all possible paths of execution.

Figure 1. Code Coverage

What is Multi-event profiler?

The multi-event profiler gathers exclusive profile data for functions over multiple events in a single run of the application. Events include CPU events, such as clock cycles, instructions executed, and pipeline stalls, and memory events such as cache read hits, read misses, write hits, and write misses. The event counts help identify the reason why some functions are not performing as expected. By collecting profile data for many events in a single run, this profiler reduces the need for multiple iterations of simulation. This single-run technique can save significant amounts of time, especially in cases of long simulation runs.

Since this data is collected by running the application on a simulator, there is an associated overhead in simulation performance. However, this overhead is kept to a minimum (for example, it is about 5% on the C55x Cycle Accurate simulator).

For a full list of events available on the C6000™ and C55x™ simulator platforms, see the appropriate Code Composer Studio online help.

The simulator collects profile data for every program location that was decoded during the application run. Exclusive profile of a function refers to this data aggregated over the address range of the function. Since data is available at a per program location granularity, it allows detailed analysis at assembly level.

Figure 2. Multi-event Profiler

2 Using the Code Coverage/Multi-event Profiler Tool in the Software Application Development Flow

Figure 3 describes the roles of the tool in the application development flow. This tool is valuable in two distinct stages of development:

- For algorithm validation
- For algorithm efficiency analysis

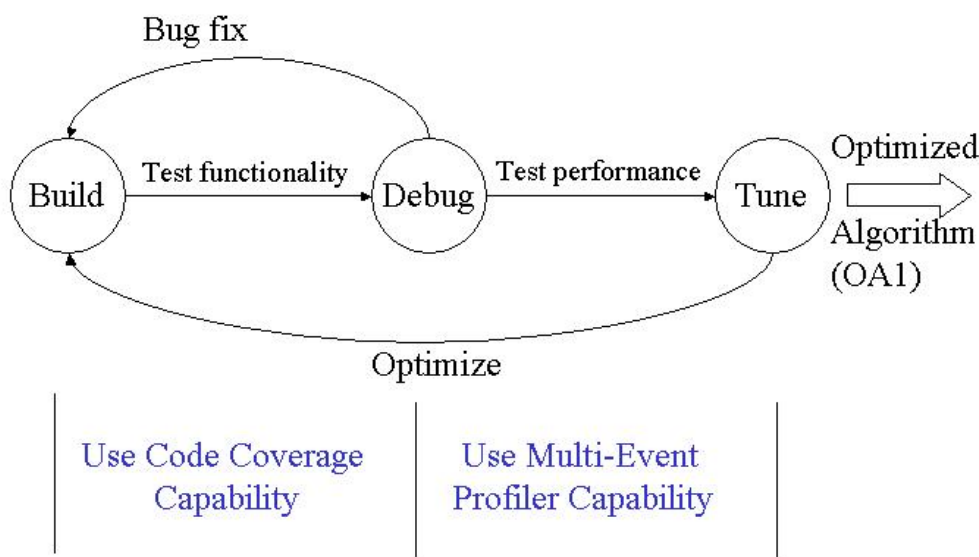


Figure 3. Using the Tool in the Software Development Flow

3 Tool Overview

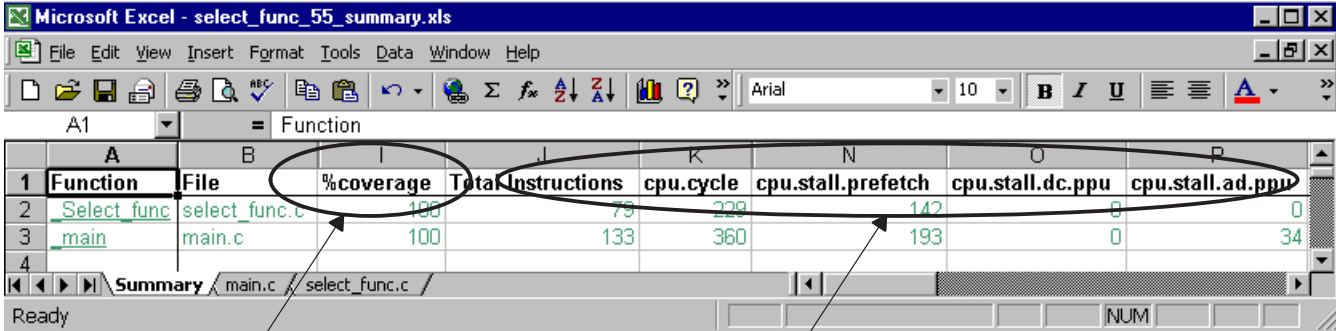
The code coverage and multi-event profiler tool is available as part of the ATK, which is an update advisor release that relies on the simulators provided in Code Composer Studio™ IDE. This capability is supported on the C6000™ and C55x™ simulators.

The steps to be followed to set up and use this tool are:

1. Set up the appropriate simulator configuration file to enable data collection.
2. Invoke Code Composer Studio and load the project.
3. Load and run the application.
4. Launch the code coverage and multi-event profiler tool.

NOTE: The visualization of coverage and profile data uses Microsoft Excel™. The *Code Coverage and Multi-event Profiler User's Guide* (SPRU624) provides detailed information on the tool usage.

Figure 4 shows a sample function summary view of the coverage and multi-event profile output.



Function	File	%coverage	Total Instructions	cpu.cycle	cpu.stall.prefetch	cpu.stall.dc.ppu	cpu.stall.ad.ppu
Select_func	select_func.c	100	79	228	142	8	0
main	main.c	100	133	360	193	0	34

Coverage information

Multi-event profile data

Figure 4. Summary View

Figure 5 shows a sample view of the annotated source code.

	A	B	C	D
	Line Count: Min	Line Count: Max	Line Number	Source Line
1			1	#include <stdio.h>
2			2	
3			3	void Select_func(int switch_value_parameter)
4			4	{
5	2834	2834	5	int switch_value;
6			6	
7	2834	2834	7	switch_value = abs(switch_value_parameter);
8			8	
9	2832	2834	9	switch(switch_value)
10			10	{
11			11	case 10:
12	1	1	12	printf("this is the case 10 \n");
13			13	break;
14			14	case 20:
15	1	1	15	printf("this is the case 20 \n");
16			16	break;
17			17	case 30:
18	0	0	18	printf("this is the case 30 \n");
19			19	break;
20			20	case -10: //Legacy Switch
21	0	0	21	printf("this is the case -10 \n");
22			22	break;
23			23	}
24	2834	2834	24	}
25			25	
26			26	
27				

Unvisited section of code

Figure 5. Annotated Source Code

4 Using the Tool for Code Coverage Analysis

The use of the code coverage capability is illustrated with a detailed example which shows how the tool helps identify new test vectors and dead code.

The C6416 functional simulator configuration is used for this example.

4.1 Example

Consider the following code snippet (Figure 6) to illustrate the use of the code coverage capability. The code snippet consists of a single function *Select_func*. The purpose of this example is to demonstrate how to achieve 100% code coverage on *Select_func* by developing suitable test vectors.

Figure 7 shows the setup to apply test vectors to the sample application. The test setup consists of a test vectors listing file named *testvectors.txt* that has a list of input data files. Each input data file has a test sequence, which is a series of numbers. For each number in the file, *Select_func* is called with that number as an argument.

This setup allows us to apply multiple test vectors in a single run of the application to obtain aggregated coverage data. It also enables easy creation and insertion of new vectors into the test flow.

```
void Select_func(int switch_value_parameter)
{
    int switch_value;

    switch_value = abs(switch_value_parameter);

    switch(switch_value)
    {
        case 10:
            printf("this is the case 10 \n");
            break;
        case 20:
            printf("this is the case 20 \n");
            break;
        case 30:
            printf("this is the case 30 \n");
            break;
        case -10: //Legacy Switch
            printf("this is the case -10 \n");
            break;
    }
}
```

Figure 6. Sample Source Code for Code Coverage

```

main()
{
    FILE *fp_testvector;
    FILE *fp_test;
    char test_fname[80];

    int switch_value_parameter;

    fp_testvector = fopen("testvectors.txt", "r");
    while ((fscanf(fp_testvector, "%s", test_fname) != 0 ))
    {

        fprintf(stdout, "fname = %s \n", test_fname);
        fp_test = fopen(test_fname, "r");
        while(!feof(fp_test))
        {
            fscanf(fp_test, "%d", &switch_value_parameter);
            fprintf(stdout, "value read = %d \n", switch_value_parameter);
            Select_func(switch_value_parameter);
        }
    }
}

```

Figure 7. Applying Test Vectors

The steps to run this example are:

1. Start the coverage analysis by creating the input data file input1.dat that has the contents:
10 20.
2. Add input1.dat to testvectors.txt.
3. Set up the appropriate simulator configuration file to enable coverage data collection.
For this example, select the sim6416_functional_simulator_profile.cfg file. See the *Code Coverage and Multi-event Profiler User's Guide* (SPRU624) for more details.
4. Bring up Code Composer Studio™ IDE and load the application project.
The project is located in the following directory:
<CCS_INSTALL_DIR>/examples/sim64xx/code_coverage/select_func
5. Compile the application with full debug (-g).
The tool requires full debug information in the COFF file (the executable) to generate coverage information. To enable this, the code must be compiled with the debugging option **-g**.
6. Load the application.
7. Run the application to completion.
input1.dat is the test vector.
8. Launch the code coverage tool by clicking on Tools → Analysis Toolkit → Code Coverage and Multi-event profiler. This launches Microsoft Excel. See Figure 8.

	A	B	C	D	E	F	G	H	I	J
	Function	File	Line no.	Start address (hex)	End address (hex)	#time s called	#lines of code	#lines executed	% coverage	Total Instructions
1	Select_func	select_func.c	3	0x254d8	0x25538	2	8	8	75	42
3	_main	main.c	5	0x24d84	0x24e26	1	12	12	100	70

Summary | main.c | select_func.c

Figure 8. Coverage Summary After First Run

9. Observe that the function `Select_func` has 75 percent coverage.
10. Click on the function name to get coverage information for the function. See Figure 9.

	A	B	C	D	E
	Line Count: Min	Line Count: Max	Line Number	Source Line	Total Instructions
1			1	#include <stdio.h>	
2			2		
3			3	void Select_func(int switch_value_parameter)	
4	2	2	4	{	4
5			5	int switch_value;	
6			6		
7	2	2	7	switch_value = abs(switch_value_parameter);	6
8			8		
9	2	2	9	switch(switch_value)	2
10			10	{	
11			11	case 10:	
12	1	1	12	printf("this is the case 10 \n");	3
13			13	break;	
14			14	case 20:	
15	1	1	15	printf("this is the case 20 \n");	3
16			16	break;	
17			17	case 30:	
18	0	0	18	printf("this is the case 30 \n");	0
19			19	break;	
20			20	case -10: //Legacy Switch	
21	0	0	21	printf("this is the case -10 \n");	0
22			22	break;	
23			23	}	
24	2	2	24	}	4
25			25		
26			26		
27			27		
28			28		

Summary / main.c / select_func.

Figure 9. Source Code Coverage After First Run

11. Observe that the cases with labels 10 and 20 (shown in green) are visited. Cases with labels 30 and -10 (shown in red) are not visited.
12. Create file `input2.dat` with the contents 30, -10 to cover the lines in red above.
13. Add `input2.dat` to `testvectors.txt`. It now has:
 - `input1.dat`
 - `input2.dat`
14. Reload and rerun the application to completion.
15. Launch the tool. See Figure 10.

	A	B	C	D	E	F	G	H	I	J
	Function	File	Line no.	Start address (hex)	End address (hex)	#times called	#lines of code	#lines executed	% coverage	Total Instructions
1	Select_func	select_func.c	3	0x254d8	0x25538	5	8	7	87.5	107
2	_main	main.c	5	0x24d84	0x24e26	1	12	12	100	138

Summary / main.c / select_func.c

Figure 10. Coverage Summary After Second Run

16. Observe that the function `Select_func` has moved from 75% to 87.5% coverage.

17. Click on the function name to get coverage information for the function. See Figure 11.

	A	B	C	D	E
	Line Count: Min	Line Count: Max	Line Number	Source Line	Total Instructions
1					
2			1	#include <stdio.h>	
3			2		
4			3	void Select_func(int switch_value_parameter)	
5	5	5	4	{	10
6			5	int switch_value;	
7			6		
8	5	5	7	switch_value = abs(switch_value_parameter);	15
9			8		
10	5	5	9	switch(switch_value)	5
11			10		
12			11	case 10:	
13	3	3	12	printf("this is the case 10 \n");	9
14			13	break;	
15			14	case 20:	
16	1	1	15	printf("this is the case 20 \n");	3
17			16	break;	
18			17	case 30:	
19	1	1	18	printf("this is the case 30 \n");	3
20			19	break;	
21			20	case -10: //Legacy Switch	
22	0	0	21	printf("this is the case -10 \n");	0
23			22	break;	
24			23	}	
25	5	5	24	}	10
26			25		

Summary / main.c / select_func.c

Figure 11. Source Code Coverage After Second Run

18. Note that the coverage is less than 100% even though all case labels have been specified in one of the input data files.
19. Study the body of the function to identify that the variable switch_value can not assume a negative value (see source line number: 7)
20. Edit this function to remove **dead code** – case label with value –10.
21. Rebuild, reload and rerun the application.
22. Launch the tool. See Figure 12. The function Select_func now has 100% coverage.

	A	B	C	D	E	F	G	H	I	J
	Function	File	Line no.	Start address (hex)	End address (hex)	#times called	#lines of code	#lines executed	% coverage	Total Instructions
1										
2	Select_func	select_func.c	3	0x255e4	0x25630	5	7	7	100	87
3	main	main.c	5	0x24d84	0x24e26	1	12	12	100	138

Summary / main.c / select_func.c

Figure 12. Coverage Summary After Dead Code Elimination

4.2 Recommended Use

The flowchart in Figure 13 depicts typical usage of the tool.

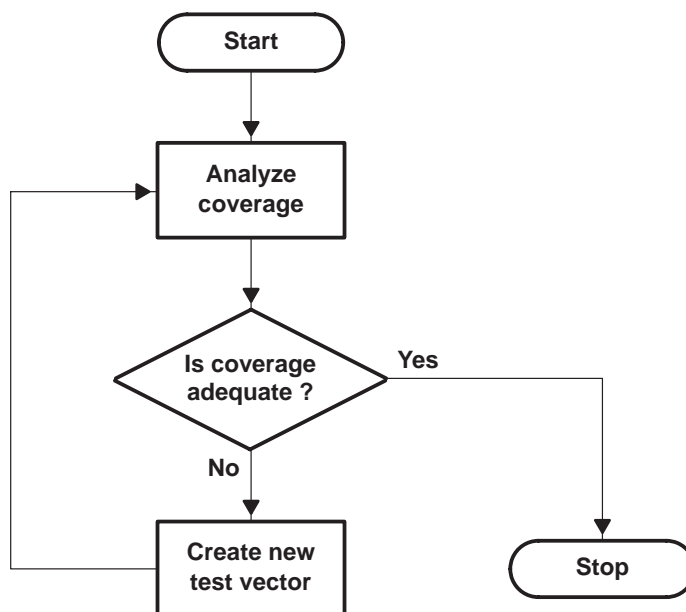


Figure 13. A Typical Flow to Follow

5 Using the Tool for Efficiency Analysis

We illustrate the use of the multi-event profiler capability with an example on the C55x cycle accurate simulator. The multi-event nature of the profiler helps identify functions with high cycle count and the contributing factors to that cycle count in a single run of simulation. We also highlight the detailed analysis capability by visualizing profile data at the granularity of assembly source code.

5.1 C55x Example

For this example, we use the C55x cycle accurate CPU simulator configuration. This example shows how the profiler helps identify and remove pipeline stalls. Figure 14 shows an assembly function, vectAdd, which the optimizations are demonstrated.

```
#include <stdio.h>
int mul( int x, int y);
int add( int x, int y);
main()
{
    int result[4][100];
    int a, b;
    int i ;

    a = 20 ;
    b = 15 ;

    for(i=0; i<100 ; i++)
    {
        result[0][i] = mul(a,b);
        result[1][i] = add(a,b);
        result[2][i] = vectAdd();
    }
}

.global _vectAdd
.sect "init"

x .int 10,5,5,6,7,8,9,4,78,56,89,45
y .int 10,5,5,6,7,8,9,4,78,56,89,45
.align

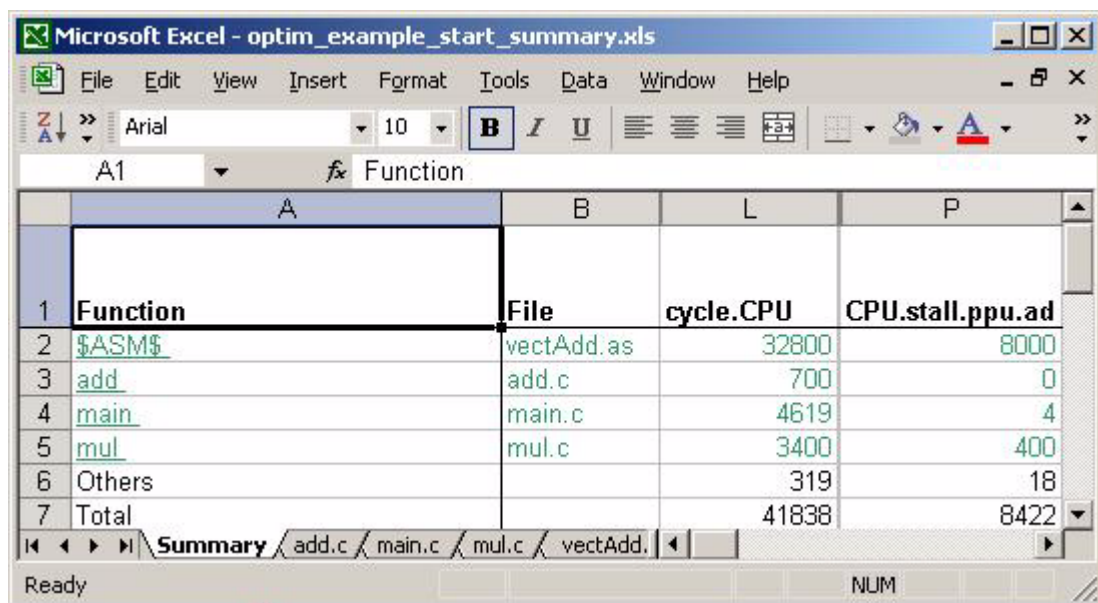
_vectAdd:
    AMOV #x,AR2      ;load pointers
    MOV *AR2, AR1
    MOV #0, AR4      ; loop counter
    MOV #0, AC0
    AMOV #y,AR3      ; pointer to B

L1:    CMP AR4 >= AR1, TC1
        nop
        nop
        nop
        nop
        BCC L2,TC1
        ADD #1, AR2
        ADD *AR2, AC0, AC0
        ADD #1, AR3
        ADD *AR3, AC0, AC0
        ADD #1, AR4
        B L1
L2:    MOV AC0, T0
        nop
        nop
        return
```

Figure 14. Code Snippet to Demonstrate Use of the Multi-event Profiler for Identifying and Removing Pipeline Stalls

The steps to run the C55x example are:

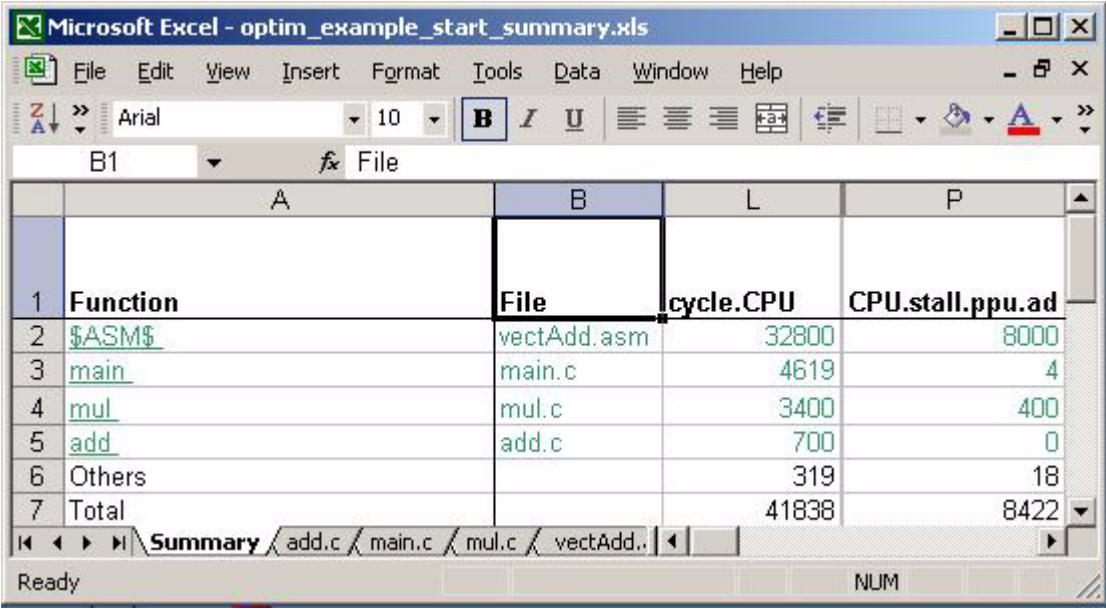
1. Set up the appropriate simulator configuration file to enable profile data collection.
For this example, select the SIM55xx_profile.cfg file. See the *Code Coverage and Multi-event Profiler User's Guide* (SPRU624) for more details.
2. Invoke Code Composer Studio IDE and load the project.
The project is located in the following directory:
<CCS_INSTALL_DIR>/examples/sim55xx/multi_event_profiler/pipeline_stall_optimization/optim_example_start
3. Set the project build options to:
 - a. build C files with **-gp** option. This option enables function profile information to be gathered.
 - b. build assembly files with **-g** option. This option enables assembly source line profile information to be gathered.
4. Load and run the application to completion.
5. Launch the multi-event profiler tool, which also launches Microsoft Excel. See Figure 15.



	A	B	L	P
	Function	File	cycle.CPU	CPU.stall.ppu.ad
1	\$ASM\$	vectAdd.as	32800	8000
2	add	add.c	700	0
3	main	main.c	4619	4
4	mul	mul.c	3400	400
5	Others		319	18
6	Total		41838	8422

Figure 15. Multi-event Profile Output After First Run

6. Sort profile data according to decreasing order of CPU cycle counts. See Figure 16.



	A	B	L	P
	Function	File	cycle.CPU	CPU.stall.ppu.ad
1	\$ASM\$	vectAdd.asm	32800	8000
2	main	main.c	4619	4
3	mul	mul.c	3400	400
4	add	add.c	700	0
5	Others		319	18
6	Total		41838	8422

Figure 16. Profile Data Sorted By CPU Cycles

7. The assembly module in the source file, vectAdd.asm, has the highest cycle count – 32800.
8. Examine the other event columns to identify the possible reason(s) for this high cycle count. We find that this module has **8000 cpu address phase stalls**. **The multi-event profiler has facilitated identification of both the most costly function (highest cycle count) as well as the sources of the high cycle count.**

In this example, we are interested in reducing address phase stalls.

9. Click on the assembly function name (“\$ASM\$”) to view annotated source line profile data. See Figure 17.

	A	B	C	D	F	J	C
	Line Count: Min	Line Count: Max	Line Number	Source Line	cycle.CPU	CPU.stall.ppu.ad a	
1							
2			1				
3			2	.global _vectAdd			
4			3	.sect "init"			
5			4				
6			5	x .int 10,5,5,6,7,8,9,4,78,56,89,45			
7			6				
8			7	y .int 10,5,5,6,7,8,9,4,78,56,89,45			
9			8				
10			9	.align			
11			10				
12			11	_vectAdd:			
13	100	100	12	AMOV #x,AR2 ;load pointers	600	0	
14	100	100	13	MOV *AR2, AR1	100	0	
15	100	100	14	MOV #0, AR4 ; loop counter	100	0	
16	100	100	15	MOV #0, AC0	100	0	
17	100	100	16	AMOV #y,AR3 ; pointer to B	100	0	
18			17				
19	1100	1100	18	L1: CMP AR4 >= AR1, TC1	7100	0	
20	1100	1100	19	nop	1100	0	
21	1100	1100	20	nop	1100	0	
22	1100	1100	21	nop	1100	0	
23	1100	1100	22	nop	1100	0	
24	1100	1100	23	BCC L2,TC1	1100	0	
25	1000	1000	24	ADD #1, AR2	5000	0	
26	1000	1000	25	ADD *AR2, AC0, AC0	1000	4000	
27	1000	1000	26	ADD #1, AR3	1000	0	
28	1000	1000	27	ADD *AR3, AC0, AC0	5000	4000	
29	1000	1000	28	ADD #1, AR4	1000	0	
30	1000	1000	29	B L1	5000	0	
31			30				
32	100	100	31	L2: MOV AC0, T0	600	0	
33	100	100	32	nop	400	0	
34	100	100	33	nop	100	0	
35	100	100	34	return	100	0	
36			35				

Figure 17. Source Code for vectAdd.asm

10. The source lines 25 and 27 show high CPU address phase stalls. This is caused by resources AR2 and AR3, respectively. The resource that causes the stall in line 25 is AR2 (modified by the instruction in line 24 and accessed for reading in the instruction at line 25). Similarly, AR3 causes the observed stall in line 27.

The annotated source code display helps perform detailed analysis at an assembly line level. By associating the stall events to exact program locations that have these stalls, it becomes possible to quickly identify the exact nature and cause of the stalls.

These stalls can be removed by replacing the conflicting instruction pairs with a single instruction. Replace the instructions in lines 24 and 25 with this single instruction:

ADD *AR2+, AC0, AC0

Also replace the instructions in lines 26 and 27 with the following single instruction:

ADD *AR3+, AC0, AC0

11. Figure 18 shows the vectAdd function after these modifications are made.

```

.global _vectAdd
.sect ".init"

x      .int 10,5,5,6,7,8,9,4,78,56,89,45
y      .int 10,5,5,6,7,8,9,4,78,56,89,45
.align

_vectAdd:
    AMOV #x,AR2      ;load pointers
    MOV *AR2, AR1
    ADD #1, AR2
    MOV #0, AR4      ; loop counter
    MOV #0, AC0
    AMOV #y,AR3      ; pointer to B
    ADD #1, AR3

L1:    CMP AR4 >= AR1, TC1
        nop
        nop
        nop
        nop
        BCC L2,TC1
    ADD *AR2+, AC0, AC0
    ADD *AR3+, AC0, AC0
    ADD #1, AR4
    B L1
L2:    MOV AC0, T0
        nop
        nop
        return

```

Figure 18. Snippet of Code After Removing Address Phase Stalls

12. Rebuild, reload and rerun the application.

13. Launch the multi-event profiler tool.

14. Sort by CPU cycle counts and review the total CPU cycles for the assembly module in the file vectAdd.asm (shown in Figure 19). The **address phase stalls have been reduced to zero** for the ASM module. **We have obtained 31 percent improvement in cycles for the assembly module.**

	A	B	L	P
	Function	File	cycle.CPU	CPU.stall.ppu.ad
1	Function			
2	\$ASM\$	vectAdd.asm	22700	0
3	add	add.c	700	0
4	main	main.c	4519	4
5	mul	mul.c	3400	400
6	Others		319	18
7	Total		31638	422

Figure 19. Multi-event Profile Output After Removing Address Phase Stalls

Key observations:

In this example, we illustrated the utility of the multi-event profiler on the C55x platform. The multi-event profiler identified the high cycle count functions in the assembly module as well as identified the sources of the performance loss (address phase stalls). Further, it enabled a detailed analysis by visualizing the events on an assembly instruction level. This allowed easy identification of instructions that were experiencing stalls as well as the resources causing those stalls (AR2, AR3 resources).

5.2 Recommended Use

As illustrated above, the following project build options make best use of the profiler:

- For C source files, use the '**-gp**' build option.
This option enables the gathering of profile data over functions in the application. Availability of source line profile information is tied to use of the '**-g**' build option. However, use of this option may hinder full optimization.
- For assembly files, use the '**-g**' option.
This option enables the gathering of profile data over both functions and source lines. Since linear assembly sources are assembled without optimizations, use of the '**-g**' option is recommended.

For further details on build options, see the *Code Coverage and Multi-event Profiler User's Guide* (SPRU624).

To make best use of the profile data over all the events, see the respective optimization guides.

6 Conclusions

In this application report, we described how the code coverage and multi-event profiler tool facilitates robustness and efficiency analyses of applications. The code coverage capability clearly highlights source code that was not executed over a run of the application. This facilitates directed creation of test vectors that can help ensure adequate coverage. We demonstrated this capability with a step-by-step example using the C6416 simulator platform. This example clearly shows how the tool aids in test vector creation as well as dead code identification.

The multi-event profiler capability visualizes function profile data for multiple events of interest, using a single simulation run for all data collection. This allows identification of high cycle count functions as well as the identification of the sources of the performance losses. Further, the tool allows detailed analysis by visualizing profile data at the granularity of assembly source code (annotated source code visualization).

We illustrated this capability via an example on the C55x simulator platform. The multi-event profiler not only helped identify high cycle count functions, but also helped isolate the causes. This helped identify significant optimizations very quickly. In the C55x example, we achieved over 30% improvement in cycles by removing address phase stalls. In this example, we also highlighted the utility of the annotated source code visualization. By observing profile data over individual assembly source statements, we could quickly identify the resources causing pipeline stalls.

These capabilities significantly benefit application software robustness and efficiency analyses.

7 References

1. *Analysis Toolkit for Code Composer Studio v2.3 User Guide* (SPRU623)
2. *Code Coverage and Multi-event Profiler User's Guide* (SPRU624)
3. *TMS320C6000 Optimizing Compiler User's Guide* (SPRU187)
4. *TMS320C55x Optimizing C/C++ Compiler User's Guide* (SPRU281)
5. *TMS320C6000 Peripherals Reference Guide* (SPRU190)
6. *TMS320C55x DSP Programmer's Guide* (SPRU376)

Appendix A Source Code Listing for Code Coverage Example

```

/***** file - select_func.c *****/
void Select_func(int switch_value_parameter)
{
    int switch_value;

    switch_value = abs(switch_value_parameter);

    switch(switch_value)
    {
        case 10:
            printf("this is the case 10 \n");
            break;

        case 20:
            printf("this is the case 20 \n");
            break;

        case 30:
            printf("this is the case 30 \n");
            break;

        case -10:
            //Legacy Switch
            printf("this is the case -10 \n");
            break;
    }
}

```

```

/***** file - main.c *****/
main()
{
    FILE *fp_testvector;
    FILE *fp_test;
    char test_fname[80];

    int switch_value_parameter;

    fp_testvector = fopen("testvectors.txt", "r");
    while ((fscanf(fp_testvector, "%s", test_fname) != 0 ))
    {
        fprintf(stdout, "fname = %s \n", test_fname);
        fp_test = fopen(test_fname, "r");
        while(!feof(fp_test))
        {
            fscanf(fp_test, "%d", &switch_value_parameter);
            fprintf(stdout, "value read = %d \n", switch_value_parameter);
            Select_func(switch_value_parameter);
        }
    }
}

```

Appendix B Source Code Listing for C5500 Multi-event Profiler Example

```

/***** File - main.c *****/

#include <stdio.h>
int mul( int x, int y);
int add( int x, int y);

main()
{
    int result[4][100];
    int a, b;
    int i ;

    a = 20 ;
    b = 15 ;

    for(i=0; i<100 ; i++)
    {
        result[0][i] = mul(a,b);
        result[1][i] = add(a,b);
        result[2][i] = vectAdd();
    }
}

/***** File - add.c *****/
int add( int x, int y)
{
    return (x+y);
}

/***** File - mul.c *****/
int mul( int x, int y)
{
    int mulResult = 0 ;
    int i;
    for(i = 0; i<y ; i++)
    {
        mulResult += x ;
    }

    return mulResult;
}

;***** File - vectAdd.asm - Before optimization
;*****

.global _vectAdd
.sect "init"

x      .int 10,5,5,6,7,8,9,4,78,56,89,45
y      .int 10,5,5,6,7,8,9,4,78,56,89,45

.align

_vectAdd:
    AMOV #x,AR2      ;load pointers
        MOV *AR2, AR1
    MOV #0, AR4      ; loop counter
        MOV #0, AC0
        AMOV #y,AR3      ; pointer to B

```

```

L1:      CMP AR4 >= AR1, TC1
        nop
        nop
        nop
        nop
        BCC L2,TC1
            ADD #1, AR2
            ADD *AR2, AC0, AC0
            ADD #1, AR3
        ADD *AR3, AC0, AC0
        ADD #1, AR4
        B L1

L2:      MOV AC0, T0
        nop
        nop
        return
;***** File - vectAdd.asm - After address phase stall optimization
****;

        .global _vectAdd
        .sect "init"

x        .int 10,5,5,6,7,8,9,4,78,56,89,45
y        .int 10,5,5,6,7,8,9,4,78,56,89,45

        .align

_vectAdd:
        AMOV #x,AR2      ;load pointers
            MOV *AR2, AR1
            ADD #1, AR2
        MOV #0, AR4      ; loop counter
            MOV #0, AC0
            AMOV #y,AR3    ; pointer to B
            ADD #1, AR3

L1:      CMP AR4 >= AR1, TC1
        nop
        nop
        nop
        nop
        BCC L2,TC1
            ADD *AR2+, AC0, AC0
            ADD *AR3+, AC0, AC0
        ADD #1, AR4
        B L1

L2:      MOV AC0, T0
        nop
        nop
        return
;***** File - vectAdd.asm - After prefetch stall optimization ****;

        .global _vectAdd
        .sect "init"

x        .int 10,5,5,6,7,8,9,4,78,56,89,45
y        .int 10,5,5,6,7,8,9,4,78,56,89,45

        .align

vectAdd:

```

```
    AMOV #x,AR2      ;load pointers
        MOV *AR2, AR1
        ADD #1, AR2
    MOV #0, AR4      ; loop counter
        MOV #0, AC0
        AMOV #y,AR3   ; pointer to B
        ADD #1, AR3
        nop ; added to make branch target L1 double-word aligned
        nop ; added to make branch target L1 double-word aligned

L1:    CMP AR4 >= AR1, TC1
        nop
        nop
        nop
        nop
        BCC L2,TC1
            ADD *AR2+, AC0, AC0
            ADD *AR3+, AC0, AC0
        ADD #1, AR4
        B L1

L2:    MOV AC0, T0
        nop
        nop
        return
```

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 the 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.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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

Copyright © 2004, Texas Instruments Incorporated