
CodeScape

ASE Mode Trace Profiler Guide

Legal Notice

IMPORTANT

The information contained in this publication is subject to change without notice. This publication is supplied "as is" without warranty of any kind, either express or implied, including but not limited to the implied warranties or conditions of merchantability or fitness for a particular purpose. In no event shall Cross Products be liable for errors contained herein or for incidental or consequential damages, including lost profits, in connection with the performance or use of this material whether based on warranty, contract, or other legal theory.

This publication contains proprietary information which is protected by copyright. No part of this publication may be reproduced in any form, or stored in a database or retrieval system, or transmitted or distributed in any form by any means, electronic, mechanical photocopying, recording, or otherwise, without the prior permission of Cross Products Limited.

Revision History

Release Version 2.3 January 2000

Release Version 2.2.0 August 1999

© 1996 - 2000 Cross Products Limited. All rights reserved.

Microsoft, MS-DOS, and Windows are registered trademarks, and Windows NT and JScript are trademarks of Microsoft Corporation in the United States and other countries. CodeScape and SNASM are registered trademarks of Cross Products Limited in the United Kingdom and other countries. Brief is a registered trademark of Borland International. CodeWright is a registered trademark of Premia Corporation. Multi-Edit is a trademark of American Cybernetics, Inc. All other trademarks or registered trademarks are the property of their respective owners.

Cross Products Ltd
23 The Calls, Leeds, West Yorkshire, LS2 7EH
telephone: +44 113 242 9814
facsimile: +44 113 242 6163
www.crossproducts.co.uk
email sales: enquiry@crossproducts.co.uk
email support: support@crossproducts.co.uk

Contents

About this guide	1
What is profiling?	3
Statistical profiling	3
Function trace profiling	3
ASE Mode Trace Profiler Tutorial	5
About this tutorial	5
Profiling example 1	5
Loading the example program file	5
Setting up the ASE Mode Trace Profiler counters	6
Selecting an interrupt lock filter	7
Shinobi	8
WinCE	8
Other	8
Manually Coding Profiler On/Off Bios Calls	8
Setting the Hits display threshold	9
Setting the Profiler breakpoints	10
Starting the Profiler and running the program	12
Analyzing the profile data	13
Hits	13
C1:(Cycles) - Counter one	13
C1:+Children (Cycles)	13
Values shown in red	13
C2(Counts) and C2:+Children (Counts)	14
Using the Instruction Cache Optimizer	15
Linking with an .LOR file	15
Analyzing your code before optimization	15
Optimizing your code	18
Appendix A: Profiler's shortcut menu	21
Appendix B: Profiler File Format	23
Profiler File ID Structure	24

Contents

File Header Structure 25

Overlay Area Data Structure 28

Function Data Structure 29

Trace Function Call Structure 32

Trace Function Return Structure 36

Counter Description Table 37

About this guide

This guide is divided into five sections:

What is profiling? provides a brief description of the principles and objectives behind profiling.

ASE Mode Trace Profiler Tutorial follows on from the Statistical Profiler tutorial and shows you the advanced profiling features which take advantage of Hitachi SuperH™ hardware functions to provide statistics on many different events.

Using the Instruction Cache Optimizer describes how to optimize your code by re-ordering functions to make better use of the instruction cache.

Appendix A explains the features on the Profilers's shortcut menu.

Appendix B describes in detail the Profiler file format for profile data which is saved in .prf format.

NOTE: *The example files referred to in the tutorial are included on the release CD in the
Tutorials\Profiler\Exam1 and ...\Exam2.*

What is profiling?

Statistical profiling

The Statistical Profiler included with CodeScape is an analysis tool for examining the run-time behavior of programs written for Hitachi SuperH™ microprocessors. It shows where your program spends its time so you can identify inefficient sections of code.

Statistical Profiling is described in detail in the CodeScape Statistical Profiler Guide

Function trace profiling

Function trace profiling provides detailed analysis of how functions call each other. The method employed in CodeScape to do this depends on the model of Hitachi SuperH™ microprocessor you are debugging.

ASE Mode Trace profiling described in this document uses Hitachi SuperH™ hardware functions to provide statistics on many different events such as cache misses and pipeline stalls. It is only available for SH7750 and the SH7091-EVA microprocessors as it relies on the use of the microprocessor's ASE Mode to gather performance data.

You can use the data from the function trace profile to relink your code with functions grouped together to make better use of the instruction cache.

What is profiling?

ASE Mode Trace Profiler Tutorial

About this tutorial

This tutorial assumes you are running CodeScape on a supported Hitachi SuperH™ target microprocessor (SH7750 and the SH7091-EVA). It also assumes that you know how to use CodeScape and are familiar with its regular functions and regions and that you have read the Statistical Profiler Tutorial.

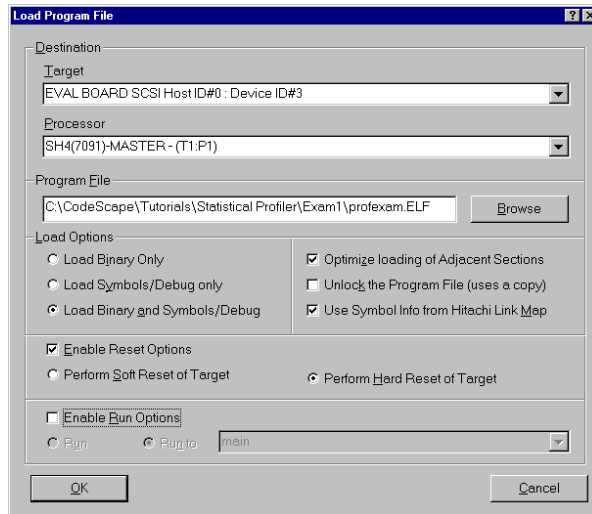
The example files used in this tutorial can be found in the CodeScape installation on your development PC under `Tutorials\Profiling\Exam1` and `Exam2`. If you did not select the Tutorials option during installation, the tutorial files can be copied from the CodeScape CD.

Profiling example 1

Loading the example program file

1. Start CodeScape.
2. In the Target window, right-click and select Load Program File...
3. Browse to the file `Tutorials\Profiler\Exam1\profexam.elf`.
4. Make the settings in the Load Program File dialog box as shown in Figure 1.

Figure 1: Load Program File options



5. Click OK.

A progress box is displayed briefly and profexam.elf is downloaded onto the target.

Setting up the ASE Mode Trace Profiler counters

1. From the Tools menu select Profiler.
The Profiler Setup dialog box appears in CodeScape as shown in Figure 2.

The steps below describe what settings to make in the Profiler Setup dialog box.

2. Select *ASE Mode Trace Profiling*.
3. In the Counter 1 drop down list select Elapsed Time.
4. In the Counter 2 drop down list select Instruction cache miss.
Each list includes all the Performance Measurement Counters that are available in the SH7750 and SH7091-EVA, a full list is given in Appendix B.

NOTE: *You can choose to ignore either one of the counters if you are using it for a specific purpose in your application.*

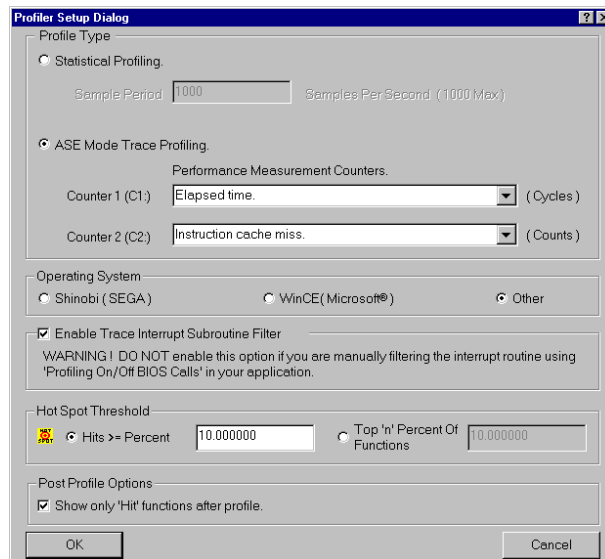
Selecting an interrupt lock filter

The Profiler has to continuously stop and start the processor to gain access to data. A side effect of this is that program execution slows down in real time. This does not effect processor time so the Profiler's results remain accurate in terms of processor clock cycles.

Due to this side effect the Profiler can sometimes take longer to profile an interrupt routine than the duration of interrupt timer so there is always an interrupt pending when exiting from the routine. This causes execution of the interrupt and nothing else so you get 100% of the Profiler's time spent in the interrupt.

To solve this problem there are some Trace Interrupt Subroutine Filters available in the Profiler's Setup. These filters turn off profiling when entering the Interrupt and turn it back on at RTE. This reduces the time spent in the interrupt and reduces the possibility of an interrupt lock.

Figure 2: Profiler Setup dialog



You select the appropriate filter for your operating system in the Profiler Setup dialog as shown above. For this example use the CodeScape filter which is not specific to any operating system.

1. For the Operating System select Other.
2. Check the Enable Trace Interrupt Subroutine Filter checkbox.

Specific details of the different filtering methods are given below.

Shinobi

If you are using the Shinobi Library, selecting the Shinobi operating system automatically enables the Shinobi Filter. On hitting the Profiler's Start Breakpoint this automatically patches the Shinobi Library with these Profiler Bios Calls:

```
PROFILE OFF BIOS CALL // turn profiling off
Routine
PROFILE ON BIOS CALL  // turn it back on again
RTE
```

When profiling is in progress a message is displayed in the Profiler's title bar stating whether the Shinobi Filter has enabled successfully or unsuccessfully.

NOTE: *You must add a Profiler Start Breakpoint for the Shinobi filter to work. You must place this breakpoint sometime after the initialisation of Shinobi, `sbInitSystem()`, so that the data that the filter depends on is in memory.*

WinCE

Currently this enables the CodeScape filter which automatically detects the interrupt/RTE and does the filtering for you. The CodeScape filter is not OS-specific and does not require any special library functions.

Other

This gives you the option of using the CodeScape Filter but also allows you to turn the filter off so that you can manually code Profiler On/Off BIOS Calls into your interrupt routine (see below).

Manually Coding Profiler On/Off Bios Calls

When using the CodeScape filter the automatic interrupt/RTE detection takes a finite amount of time. If your interrupt timer is even faster than the automatic filtering can deal with you must manually code Profiler On/Off Bios Calls into your interrupt routine. Full details and code examples are given in the *SH4 Debug Interface* manual in the section titled *Disable subroutine tracing (profiling)*.

NOTE: *If you are using Bios Calls in your code to filter the interrupt routine you must NOT enable automatic filtering.*

Setting the Hits display threshold

1. In the *Hot Spot Threshold* box click *Hits >= Percent* and in the text box enter 10%.
2. Select *Show only 'Hit' functions after profile*.

NOTE: By default you get the *Hits* and *Cache Line* columns when *Trace profiling*. These show *Hot Spots* in the same way as for *Statistical Profiling*.



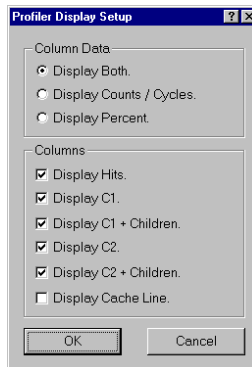
3. Click OK.
4. In the Profiler window click  and  to display Source and Disassembly regions.
5. Right-click in the Profiler window and select *Profiler Display Setup...*
Set the options as shown below in Figure 3.

Figure 3: Profiler Display Setup options



6. Click OK.

Setting the Profiler breakpoints

When ASE Mode Trace profiling it is important to set Profiler breakpoints to start at the Main loop or to isolate the section of code you want to profile. If you profile without breakpoints the Profiler will waste time generating unwanted data by profiling the boot code and library initialisation.

NOTE: *If you are using the Shinobi interrupt subroutine filter you must insert a Profiler Start Breakpoint after `sbInitSystem()` for the filter to work.*

There are two types of Profiler breakpoints, start and stop, which are different to normal breakpoints. When a start breakpoint occurs, an instruction is sent to the Profiler to start profiling. The Profiler runs until either a stop breakpoint occurs or until it is stopped manually. Unlike other breakpoints, on hitting a Profiler breakpoint, the execution of the code does not stop, instead the program continues to run. You can set normal breakpoints in CodeScape that trigger different actions during profiling.

- You can only set one Profiler start breakpoint, but you can set multiple stop breakpoints.
- If the start breakpoint is at the beginning of a loop then you should set at least one stop breakpoint at the end of the loop otherwise a recursion can occur in the Profiler function call tree.
- Program execution does not stop when a Profiler breakpoint occurs.
- It takes a finite amount of time to process a Profiler breakpoint, therefore a frequently hit breakpoint increases the time it takes to profile the program but does not effect the processor time (i.e. the number of clock cycles).

NOTE: *You can set up breakpoints in CodeScape that will trigger the Profiler to start. You insert the breakpoints in your program in the normal way using CodeScape and then configure the Trigger Actions dialog box to start or stop the Profiler upon hitting the breakpoints. If you use this method you must remember to enable the Profiler before running the program (stop/start button is green when the Profiler is enabled)*

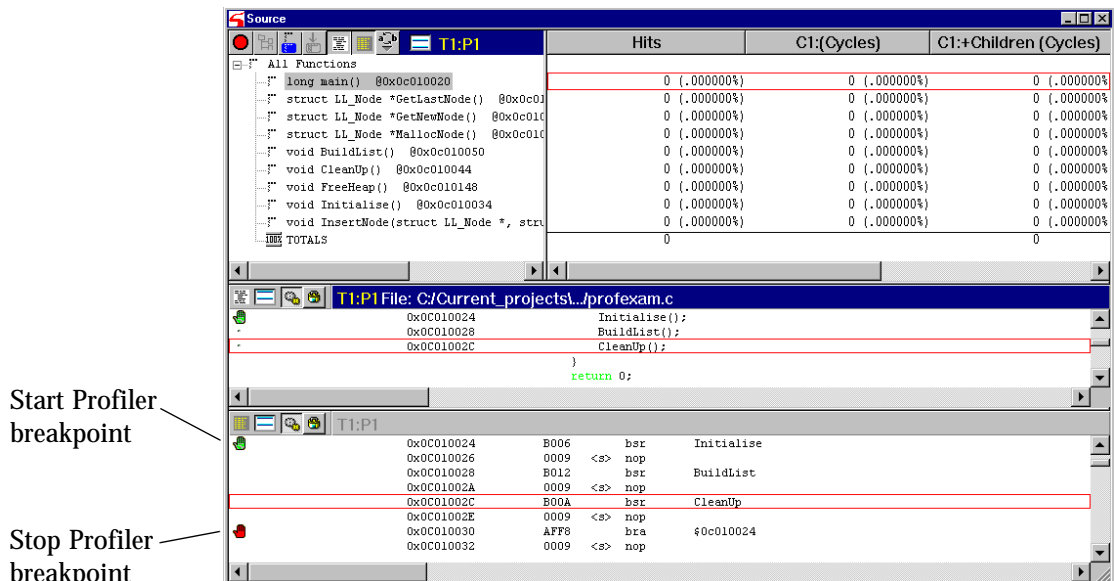
1. In the function list click on *main*.
The cursor in the Disassembly region stops on *main*.
2. Click on *main* in the Disassembly region.

3. Press the down arrow key twice until the cursor stops on *Initialise*.
4. Press F5 to insert a start breakpoint.
A green hand appears at the insertion point to indicate a start breakpoint.
5. Press the down arrow key six times to scroll down to the *bra* instruction at 0x0C010030 in the Disassembly region.
6. Press F5 to insert a stop breakpoint.
A red hand appears at the insertion point to indicate a stop breakpoint. The Profiler automatically decides whether the breakpoint is a start or a stop based on your previous action.
The Profiler window should now look like Figure 4.

You can see that only one breakpoint is visible in the Source region. This is because the loop that you have isolated to profile begins and ends on the same instruction in the source code.

In some circumstances the stop breakpoint can appear before the start breakpoint in the Source region. This is because the generated executable code can be different from the source code due to compiler optimizations. You can always see exactly where the breakpoints occur by looking at the Disassembly region.

Figure 4: The Profiler window before performing profile



Starting the Profiler and running the program




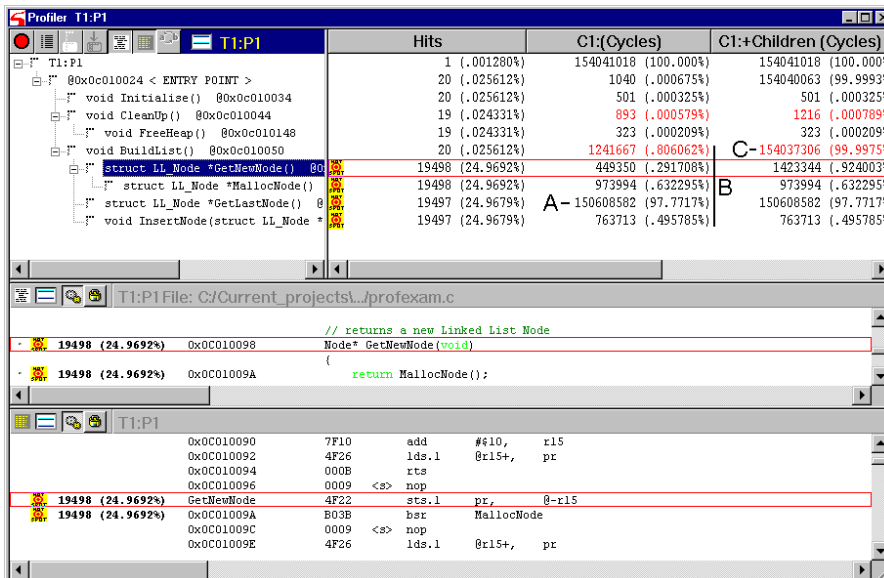
- Click  (red) to start the Profiler.
The button is green when the Profiler is running and red when the Profiler is stopped.
- Press F9 to run the program. Let the program run for about 30 seconds and press F9 again to stop it.
The Hits counter updates while the program is running.
The length of time that you profile the program for depends on the size and complexity of the program you are debugging. In the example below the Profiler ran for approximately 30 seconds and the program completed 19 full loops in that time. Depending on your requirements may need to run the Profiler for several hours or a just single pass.
- Click  (green) to stop the Profiler.
The Profiler takes a few seconds to resolve the data, after which you see the final results in the Profiler window.
- Click  to display the function list as a tree structure.
You might need to expand the nodes of the tree if it has not fully expanded as shown in Figure 5.

Figure 5: The Profiler results display showing C1 and C1 + Children



Analyzing the profile data

This program simply creates a single-ended linked list of ascending numbers from 1 to 1000, clears the list, and then starts again.

Hits

The Hits column shows the actual number of hits on each function during the profile. The Hits column after an ASE Mode Trace profile is different from the Hits column after a Statistical profile because the Statistical profile gives the number of hits detected by a periodic sample. Therefore the number of hits detected during Statistical profiling is an indication of the time spent in each function whereas in ASE Mode Trace profiling it is the *actual* number of hits on each function. You can see this by comparing the Hits results from a Statistical profile (see example 1 on page 10 of the Statistical Profiler Tutorial) with the Elapsed Time from an ASE Mode Trace profile (see the C1 column in Figure 5).

C1:(Cycles) - Counter one

The C1 column shows the Elapsed Time (the number of processor clock cycles) spent in each function in the section of code you profiled. You can see straight away that the program spent a large proportion of its time (over 97%) in the function GetLastNode (indicated by A in Figure 5).

In the function GetLastNode, the program locates the last entry in the list by searching from the start of the list until it reaches the last node. Rewriting GetLastNode so that it does not search from the start of the list each time will remove the inefficiency and speed up the program as demonstrated previously in example 2 of the Statistical Profiler Tutorial.

C1:+Children (Cycles) - Counter one plus the total of all the functions it called

From the tree view you can see that the function BuildList has the children GetNewNode (with child function MallocNode), GetLastNode and InsertNode. These are all shown as hot spots in the Hits column. The column C1:+Children shows the Elapsed Time in each function plus the Elapsed Time in all the functions it called. Hence the value at C is the sum of the values at B.

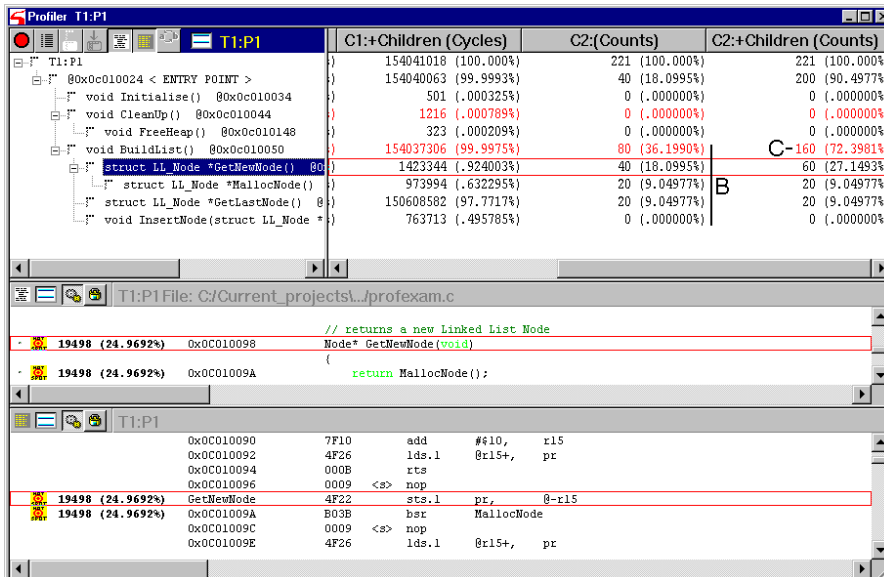
Values shown in red

Incorrect counter data, shown in red, can occur when there is more than one Profile Event (JSR, RTS, Interrupt, Exception, RTE, BSR, BSRF) in the SH4's pipeline. This means there is no way of knowing which event incremented which counter and by what amount. This is an anomaly of the SH4's tracing hardware and the effect on the Profiler's results is usually negligible.

C2(Counts) and C2:+Children (Counts)

Figure 6 shows the results for the Performance Counter you specified for Counter 2 in the Profiler setup. In this case it is Instruction Cache Misses and the numbers given are the actual number of Instruction Cache misses during the profile. As before, the value at C is the sum of the values at B.

Figure 6: The Profiler results display showing C2 and C2 + Children



Using the Instruction Cache Optimizer

The Instruction Cache Optimizer is a tool which analyzes the data gathered during a function trace profile and suggests ways in which individual functions and function groups in your code can be organized into 8K *cliques* for better use of the instruction cache.

The optimizer pairs functions and function groups together on the basis of how often they call each other. It then goes on to look at parents and children of those functions and so on, eventually forming a clique of functions which will benefit from being present sequentially in the instruction cache. The size of a clique is always less than the cache size (8K for the SH4) so that all its members can be loaded into the cache simultaneously.

This is a very important optimization on the SH series of processors because an instruction cache miss is usually 25 to 30 times slower than an instruction cache hit.

Linking with an .LOR file

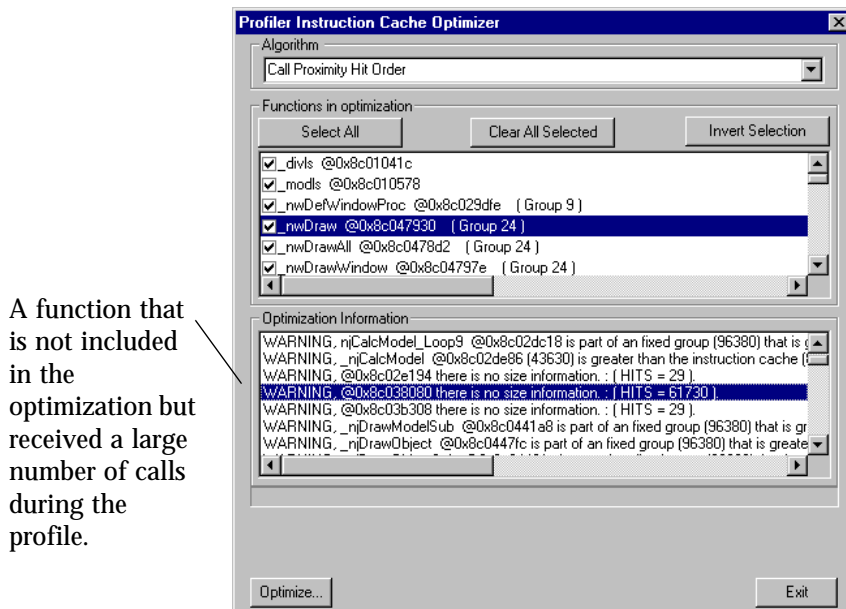
An .LOR file is the output from the Instruction Cache Optimizer to describe the optimum link order for your application. This file can be read by a third party linker to automatically relink your application and optimize your instruction cache.

The file is also readable in a text editor so it can be used as a guide to help identify functions that can be grouped or re-ordered when manually optimizing your code.

Analyzing your code before optimization

- Right-click in the Profiler window and select Optimize...
The optimizer takes a few seconds to read the data from the Profiler and then the Instruction Cache Optimizer dialog box appears as shown in Figure 7.

Figure 7: Profiler Instruction Cache Optimizer: Example of a warning



The top window, *Functions in optimization*, shows all the functions called during the profile that will be included in the optimization. If a function belongs to a fixed group of functions which cannot be split (such as a third party library file) the group number is shown after the function name. The group numbers are assigned arbitrarily by the optimizer and have no significance other than for identification purposes in the optimizer window and the .LOR file.

The lower window, *Optimization Information*, shows two sets of information, Warnings and Fixed Function Groups.

Warnings

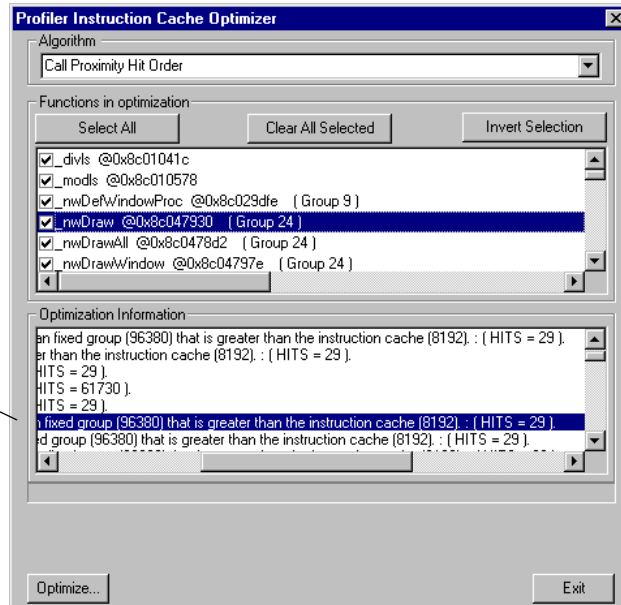
This is a list of functions which cannot be included in the optimization. The reason why each function cannot be included and the number of times it was called during the profile is also given.

In the example in Figure 7 a function is highlighted which cannot be included in the optimization because the optimizer has no size information for it. This function was called 61730 times during the profile and therefore requires further investigation because an advantage can probably be gained by optimizing a function that is called so often.

The function above it in the list was only called 29 times during the profile, and although it cannot be included in the optimization, it does not require further investigation because it was called so seldom that optimizing it will result in little or no performance enhancement for your code.

Figure 8: Profiler Instruction Cache Optimizer: Example of a warning

A function that is not included in the optimization and received very few calls during the profile.



In the example in Figure 8 a function is highlighted that cannot be included in the optimization because it belongs to a fixed function group that is bigger than the cache size. The size of the function group to which it belongs is shown in brackets (96380). Again, this function received so few calls that it probably does not matter that it is not included in the optimization.

Fixed Function Groups

Fixed Function Groups are listed in numerically in the Optimization Information window. All the functions that were called during the profile and the fixed function group to which they belong are included. Obviously functions that were not called during the profile cannot be included in the optimization because there is no data for them.

The example shown in Figure 9 over the page shows a function that belongs to Fixed Function Group 9. In the *Functions in optimization* window you can see that this is the only function from group 9 that is included in the optimization. An

advantage might be gained by separating this function from Fixed Function Group 9 so that it can be independently positioned in a clique close to the functions it interacts with.

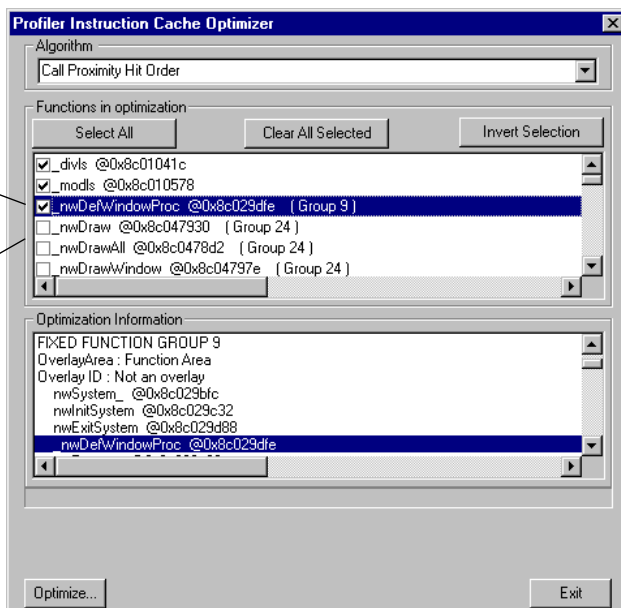
Optimizing your code

1. In the *Functions in optimization* list select the functions you want to specifically include or exclude from the optimization.
You cannot deselect functions individually if they belong to Fixed Function Group. The optimizer automatically deselects whole group as shown for Group 24 in Figure 9.

Figure 9: Profiler Instruction Cache Optimizer: Example of a Fixed Function Group

The only function from Fixed Function Group 9 that is included in the optimization.

By deselecting one function in group 24, the whole group is automatically deselected.



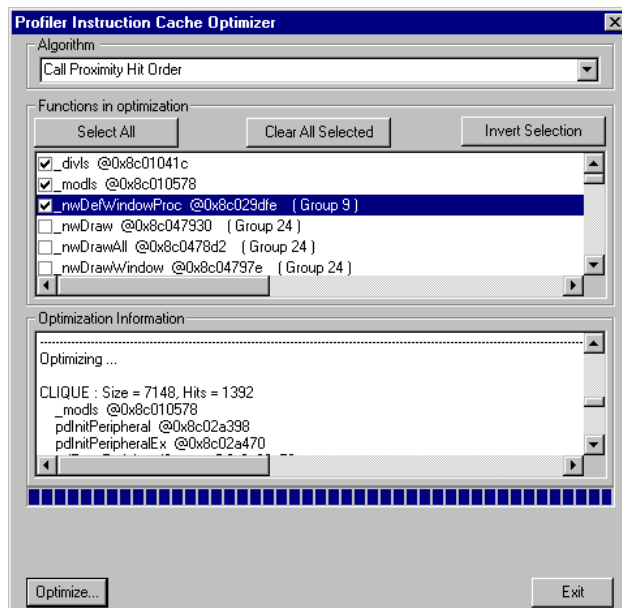
2. In the Algorithm drop down list select the algorithm by which you want to optimize your code.
Currently only Call Proximity Hit Order is available.
3. Click Optimize and select the location where you want to save the .LOR file.
A progress bar appears and a log shows the action of the optimizer as it writes the .LOR file to the specified location.

After optimization the Optimization Information window shows the cliques that the optimizer suggests for regrouping the functions in your code as shown in Figure 10.

For each clique the following information is given:

- Its size in bytes.
- The total number of function calls (hits) received by all its functions during the profile.
- The names of the functions it contains.

Figure 10: Profiler Instruction Cache Optimizer: Example of a Clique



More detailed information is given in the .LOR file. You can use this file to link with a third party linker or can use it in a text editor as a guide to help you manually re-order the functions in your code.

Appendix A: Profiler's shortcut menu

Table 1: The Profiler's shortcut menu

Select:	To:
File	Load or save profile information. Profiles are saved using the extension .prf.
Enable Profiler	Start or stop the profiler.
One Pass Between Breakpoints	Set one pass between a Profiler Start Breakpoint and a Profiler Stop Breakpoint.
Remove All Profiler Breakpoints	Remove all Profiler breakpoints.
Trace Tree Profile Display	Display a tree showing each function, the functions that called it, and the functions it called. Trace profiling only.
Function Profile Display	Display a simple list of the functions in your program in the Profiler window.
Function Profile Filter	Arrange the view to show one of the following: all functions, all tagged functions, or all untagged functions.
Untag All	Untag all currently tagged functions.
Sort	Arrange the column view of the active Function Profile Display.
Source Display	View your program's original source code.
Disassembly Display	View your program at instruction level (assembly code).
Rename Function...	Enter a new name for a specific function.

Select:	To:
Profiler Display Setup...	Specify the profile display options.
Optimize...	Instruction cache optimization. Trace profiling only.
Setup...	Specify options for Statistical Profiling, or ASE Mode Trace Profiling.

Appendix B: Profiler File Format

This appendix describes the Profiler File Format. The following tables of data structures that define the file format are in the order that they appear in the file.

The file format consists of six structures:

1. Profiler File ID
This is the label at the start of the file which initially identifies it as a profiler file.
2. File Header
This is the file's header which contains global information about the profile.
3. Overlay Area Data
This is a description of an overlay area.
4. Function Data
This contains data of a function that was called during the profile.
5. Trace Function Call
This describes a function call which has its own function data unique to that particular call path.
6. Trace Function Return
This describes a function return.

The position of the *Trace Function Call/Return* structures in the file defines the function's call/return relationship.

If the file represents a Statistical Profile as opposed to a Trace Profile there will be no *Trace Function Call* or *Trace Function Return* structures in the file.

The *File Header* indicates the type of profile. If it is a Statistical Profile the *Number Of Function Calls* and *Trace Information Offset* will be set to zero along with all the Counter data. In this case only the Function Hits data is valid. If the file represents a Trace Profile all the data is valid.

Incorrect counter data (see *Function Data Structure* and *Trace Function Call Structure*) can occur when there is more than one Profile Event (JSR, RTS, Interrupt, Exception, RTE, BSR, BSRF) in the SH4's pipeline. This means there is no way of knowing which event incremented which counter and by what amount. This is an anomaly of the SH4's tracing hardware and the effect on the Profiler's results is usually negligible.

The *Counter Description Table* describes all the counters available in the SH4's ASE mode debug facility.

Profiler File ID Structure

Byte	Type	Profiler File ID
0	Byte	0x50 'P'
1	Byte	0x52 'R'
2	Byte	0x4f 'O'
3	Byte	0x46 'F'
4	Byte	0x49 'I'
5	Byte	0x4c 'L'
6	Byte	0x45 'E'
7	Byte	0x52 'R'
8	Byte	0x20 ' '
9	Byte	0x46 'F'
10	Byte	0x49 'I'
11	Byte	0x4c 'L'
12	Byte	0x45 'E'
13	Byte	0x3a ':'

File Header Structure

Byte	Type	File Header
0 1 2 3	Unsigned Long	(LSB) Size of Header to follow. (0x00000055) (MSB)
4	Byte	File Version
5 6	2 bytes	Spare Bytes
7	Byte	Instruction Cache : 0x01 Enabled (8k) , 0x00 Disabled.
8	Byte	Day (0 - 31)
9	Byte	Month (0 - 12)
10	Byte	Year eg. 98 - 99 - 00 - 01
11	Byte	Hour (0 - 24)
12	Byte	Min (0 - 60)
13	Byte	Second (0 - 60)
14	Byte	Type of Profile 0x01 - Statistical Profile - in this mode only the Function Hits are valid. All the counter data in every structure will be zero. 0x02 - Trace Profile - all counters and hits are valid.
15 16 17 18 19 20 21 22	Unsigned Quadword	(LSB) Overall accumulative total number of Function Hits (MSB)

Byte	Type	File Header
23 24 25 26 27 28 29 30	Unsigned Quadword	(LSB) Overall accumulative total of Counter 1 (MSB)
31 32 34 35 36 37 38	Unsigned Quadword	(LSB) Overall accumulative total of Counter 1 minus the functions it called. (MSB)
39 40 41 42 43 45 46	Unsigned Quadword	(LSB) Overall accumulative total of Counter 2 (MSB)
47 48 49 50 51 52 53 54	Unsigned Quadword	(LSB) Overall accumulative total of Counter 2 minus the functions it called. (MSB)
55	Byte	What Counter 1 represents, see the Counter Description Table at the end of the document.
56	Byte	What Counter 2 represents, see the Counter Description Table at the end of the document.
57 58 59 60	Unsigned Long	(LSB) Number of Overlays (MSB)

Byte	Type	File Header
61 62 63 64	Unsigned Long	(LSB) Number of functions (MSB)
65 66 67 68	Unsigned Long	(LSB) Number of function calls. If Byte 14 is set to 0x01 (for Statistical Profile) then this will be set to zero. (MSB)
69 70 71 72	Unsigned Long	(LSB) Overlay Information Offset Offset into this file from the End of the Header to the Overlay Information. This is only valid if the Number Of Overlays field is greater than zero. (MSB)
73 74 75 76	Unsigned Long	(LSB) Function Information Offset Offset into this file from the End of the Header to the Function Information. (MSB)
77 78 79 80	Unsigned Long	(LSB) Trace Information Offset Offset into this file from the End of the Header to the Trace Information (function calls/returns). If Byte 14 is set to 0x01 (for Statistical Profile) then this will be set to zero. (MSB)
81 82 83 84 85 86 87 88	8 bytes	Spare.

Overlay Area Data Structure

The Overlay Data that follows is a list of Overlays that exist within the program. The data is repeated *Number Of Overlays* times. *Number Of Overlays* can be found in the File Header.

Byte	Type	Overlay Area Data
0 1 2 3	Unsigned Long	(LSB) Size of Overlay Area Data to follow. (MSB)
4 5 6 7	Unsigned Long	(LSB) Overlay Area Index ID. (MSB)
8 9 10 11	Unsigned Long	(LSB) Start Address of Overlay Area. (MSB)
12 13 14 15	Unsigned Long	(LSB) End Address of Overlay Area. (MSB)
16 17 18 19	Unsigned Long	(LSB) Current Overlay ID (at the time of saving). (MSB)

Function Data Structure

The Function Data that follows is a list of functions called during the profile with Accumulative Totals for that function regardless of call path. The data is repeated *Number Of Functions* times. *Number Of Functions* can be found in the File Header.

Byte	Type	Function Data
0 1 2 3	Unsigned Long	(LSB) Size Of Function Data to Follow. (MSB)
4	Byte	Function Description. 0x00 - Unknown. 0x01 - Function. 0x02 - Interrupt/Exception (unknown VBR + ?). 0x03 - Exception VBR + 0x100. 0x04 - Exception VBR + 0x400. 0x05 - Interrupt VBR + 0x600. 0x06 - Exception DBR. 0x07 - User Defined Block.
5 6	Unsigned Word	Attributes (Bit Field) 0x00000001 - This function has possible incorrect counter data. 0x00000002 - This function was Tagged (Selected) by the user. All other bit-fields will be 0.
7 8	Unsigned Word	(LSB) Cache Line Number. 0 - 256 (256 lines * 32 Bytes = 8k) , 0xFFFF Non Cacheable (MSB)
9 10 11 12	Unsigned Long	(LSB) Minimum Program Counter of Function Scope. (MSB)
13 14 15 16	Unsigned Long	(LSB) Max Program Counter of Function Scope. (MSB)
17 18 19 20	Unsigned Long	(LSB) Overlay Area Index ID ID 0xFFFFFFFF = Not part of an overlay. (MSB)

Byte	Type	Function Data
21 22 23 24	Unsigned Long	(LSB) Overlay ID ID 0xFFFFFFFF = Not an Overlay. (MSB)
25 26 27 28	Unsigned Long	(LSB) Function Group ID ID 0xFFFFFFFF = No Group. (MSB)
29 30 31 32 33 34 35 36	8 Bytes	Spare.
37 38 39 40 41 42 43 44	Unsigned Quadword	(LSB) Accumulative total number of Function Hits for this function. (MSB)
45 46 47 48 49 50 51 52	Unsigned Quadword	(LSB) Accumulative total of Counter 1 for this function. (MSB)
53 54 55 56 57 58 59 60	Unsigned Quadword	(LSB) Accumulative total of Counter 1 minus the functions it called for this function. (MSB)

Byte	Type	Function Data
61 62 63 64 65 67 68	Unsigned Quadword	(LSB) Accumulative total of Counter 2 for this function. (MSB)
69 70 71 72 73 74 75 76	Unsigned Quadword	(LSB) Accumulative total of Counter 2 minus the functions it called for this function. (MSB)
77 78	Word	(LSB) The size of the following function name. ('n1') (MSB)
79	String	Function Name. The string is of the size specified in the previous field.
79 + 'n1'	Word	(LSB) The size of the following Linker Label. ('n2') (MSB)
79 + 'n1' + 2	String	Linker Label. The string is of size specified in the previous field. The Linker Label is the label that was used to link this function.
79 + 'n1' + 2 + 'n2'	Word	(LSB) The size of the following Qualified Name. (MSB)
79 + 'n1' + 2 + 'n2' + 2	String	Qualified Name. The string is of size specified in the previous field. The Qualified Name is used to specify a function in greater detail (filename, class etc.) in the event of a duplicate function name.

Trace Function Call Structure

The Function Trace Data (function calls/returns) consists of two data structures *Trace Function Call* and *Trace Function Return*. To distinguish between them the first byte after the data size is 0xFF for a function call and 0x00 for a return. There is a file offset to this data that is stored in the File Header. Also stored in the File Header is the *Number of Function Calls*. The number of following Trace Function Calls is coupled with an equal amount of Trace Function Returns.

Byte	Type	Trace Function Call
0 1 2 3	Unsigned Long	(LSB) Size of Trace Function Call Data to follow. (MSB)
4	Byte	0xFF Function Call identifier.
5 6 7 8	Unsigned Long	(LSB) Unique Call Number. This number is present in the matching 'Return From Function' data. (MSB)
9	Byte	Function Description. 0x00 - Unknown. 0x01 - Function. 0x02 - Interrupt/Exception (unknown VBR +). 0x03 - Exception VBR + 0x100. 0x04 - Exception VBR + 0x400. 0x05 - Interrupt VBR + 0x600. 0x06 - Exception DBR. 0x07 - User Defined Block.
10 11	Unsigned Long	Attributes (Bit Field) 0x00000001 - This function has possible incorrect counter data. All other bit-fields will be 0.
12 13	Unsigned Word	(LSB) Cache Line Number. 0 - 256 (256 lines * 32 Bytes = 8k), 0xFFFF(-1) Non Cacheable (MSB)
14 15 16 17	Unsigned Long	(LSB) Minimum Program Counter of Function Scope. (MSB)

Byte	Type	Trace Function Call
18 19 20 21	Unsigned Long	(LSB) Max Program Counter of Function Scope. (MSB)
22 23 24 25	Unsigned Long	(LSB) Overlay Area Index ID ID 0xFFFFFFFF = Not part of any overlay. (MSB)
26 27 28 29	Unsigned Long	(LSB) Overlay ID ID 0xFFFFFFFF = Not an overlay. (MSB)
30 31 32 33	Unsigned Long	(LSB) Function Group ID ID 0xFFFFFFFF = No Group. (MSB)
34 35 36 37 38 39 40 41	8 Bytes	Spare.
42 43 44 45 46 47 48 49	Unsigned Quadword	(LSB) Accumulative total number of Function Hits for this function with this call path. (MSB)
50 51 52 53 54 55 56 57	Unsigned Quadword)	(LSB) Accumulative total of Counter 1 for this function with this call path. (MSB)

Byte	Type	Trace Function Call
58 59 60 61 62 63 64 65	Unsigned Quadword	(LSB) Accumulative total of Counter 1 minus the functions it called for this function with this call path. (MSB)
66 67 68 69 70 71 72 73	Unsigned Quadword	(LSB) Accumulative Total of Counter 2 for this function with this call path. (MSB)
74 75 76 77 78 79 80 81	Unsigned Quadword	(LSB) Accumulative Total of Counter 2 minus the functions it called for this function with this call path. (MSB)
82 83	Word	(LSB) The Size of the Following Function Name. ('n1') (MSB)
84	String	Function Name. The string is of size specified in the previous field.
84 + 'n1'	Word	(LSB) The Size of the following Linker Label. ('n2') (MSB)
84 + 'n1' + 2	String	Linker Label. The String is of size specified in the previous field. The Linker Label is the label that was used to link this function.
84 + 'n1' +2+'n2'	Word	(LSB) The Size of the following Qualified Name. (MSB)

Byte	Type	Trace Function Call
84 + 'n1' +2 + 'n2' +2	String	Qualified Name. The string is of size specified in the previous field. The Qualified Name is used to specify a function in greater detail (filename, class etc.) in the event of a duplicate function name.

Trace Function Return Structure

Byte	Type	Trace Function Return
0 1 2 3	Unsigned Long	(LSB) Size Of Trace Function Return Data to Follow. (0x00000005) (MSB)
4	Byte	0x00 Function Return Identifier.
5 6 7 8	Unsigned Long	(LSB) Unique Call Number. This number is to ensure that the return Unique Number matches the Call Unique Number. (MSB)

Counter Description Table

Counter Description Number	Counter Description	Count/Cycles
0x01	Operand access (read/with cache)	Count
0x02	Operand access (write/with cache)	Count
0x03	UTLB miss	Count
0x04	Operand cache read miss	Count
0x05	Operand cache write miss	Count
0x06	Instruction fetch (with cache) - 2 instructions fetched simultaneously	Count
0x07	Instruction TLB miss.	Count
0x08	Instruction cache miss	Count
0x09	All operand access	Count
0x0a	All instruction access - 2 instructions fetched simultaneously	Count
0x0b	On-chip RAM operand access	Count
0x0c	On-chip RAM instruction access	Count
0x0d	On-chip I/O space access	Count
0x0e	Operand access (read + write/with cache)	Count
0x0f	Operand cache read + write miss	Count
0x10	Branch instruction	Count
0x11	Branch taken	Count
0x12	BSR/BSRF/JSR	Count
0x13	Instruction execution	Count
0x14	2- instruction simultaneous execution	Count
0x15	FPU instruction execution	Count
0x16	Interrupt (normal)	Count

Counter Description Number	Counter Description	Count/Cycles
0x17	Interrupt (NMI)	Count
0x18	TRAPA instruction execution	Count
0x19	UBC-A match	Count
0x1a	UBC-B match	Cycles
0x21	Instruction cache fill	Cycles
0x22	Operand cache fill	Cycles
0x23	Elapsed time	Cycles
0x24	Pipeline freeze (by cache miss/instruction)	Cycles
0x25	Pipeline freeze (by cache miss/data)	Cycles
0x27	Pipeline freeze (by branch instruction)	Cycles
0x28	Pipeline freeze (by CPU register)	Cycles
0x29	Pipeline freeze (by FPU)	Cycles

Cross Products Ltd
23 The Calls, Leeds, West Yorkshire, LS2 7EH
telephone: +44 113 242 9814
facsimile: +44 113 242 6163
www.crossproducts.co.uk
email sales: enquiry@crossproducts.co.uk
email support: support@crossproducts.co.uk