

Program Cache Layout

The version 7.0.0 release of the C6000 Code Generation Tools includes new capability to help you develop better program cache efficiency into your applications. Program cache layout is the process of placing code in memory to minimize the occurrence of conflict misses in the program cache.

Contents

Background and Motivation

- Problem Description
- Approach
- Goal

What Level of Performance Improvements Can You Expect to See?

- Evaluating L1P Cache Usage
- Further Resources

Program Cache Layout Related Features and Capabilities

- Path Profiler
 - Analysis Options
 - Environment Variables
- Cache Layout Tool, clt6x
- Linker
 - preferred_order Option
 - unordered() Linker Command File (LCF) Operator

Program Cache Layout Development Flow

- Gather Dynamic Profile Information
- Generate Preferred Function Order from Dynamic Profile Information
- Utilize Preferred Function Order in Re-Build of Application

Comma-Separated Values (CSV) Files with WCG Information

Linker Command File Operator - unordered()

- Basics
- About DOT Expressions in the Presence of unordered()
- GROUPs and UNIONs

Cache Layout Tool Tutorial

Things To Be Aware Of

Addendum: Using 7.x.x CLT Capability with 6.1.x CGT

- General Approach
- Constraints / Assumptions
- Development Flow
- An Example

Background and Motivation

Problem Description

A program cache miss happens when an instruction fetch fails to read an instruction from the program cache and the processor is required to access the instruction from the next level of memory. A request to L2 or external memory has a much higher latency than an access from the first level instruction cache.

- Effective utilization of the L1P instruction cache is an important part of getting the best performance from your processor.
- In the C6x family of processors, L1P instruction cache misses can cause significant overhead because the L1P cache is direct-mapped.
- Some applications (like h264, for example) can spend 30+% of the processor's time in L1P stall cycles due to L1P cache misses.

Approach

- Many L1P cache misses are conflict misses.
- Conflict misses occur when the cache has recently evicted a block of code that is now needed again. In a program cache this often occurs when two frequently executed blocks of code (usually from different functions) interleave their execution and are mapped to the same cache line.

For example, suppose there is a call to function B from inside a loop in function A. Suppose also that the code for function A's loop is mapped to the same cache line as a block of code from function B that is executed every time that B is called. Each time B is called from within this loop, the loop code in function A will be evicted from the cache by the code in B that is mapped to the same cache line. Even worse, when B returns to A, the loop code in A will evict the code from function B that is mapped to the same cache line.

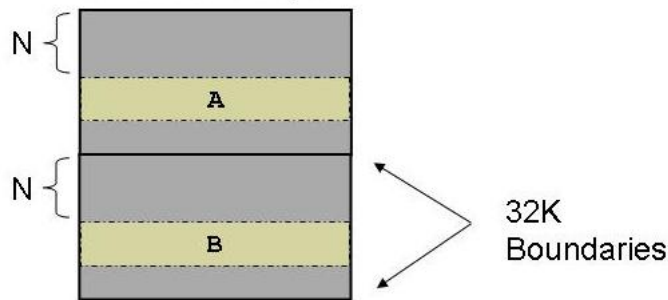
- Most C6000 devices have 32K level 1 program cache (L1P)
- Direct mapped
 - Memory locations 32K apart correspond to the same location in cache

Code

```
A() {
    for (...)
        B();
}

B() { ... }
```

Worst Case Layout



Every iteration through the loop will cause 2 program cache conflict misses. If the loop is heavily traversed, then the number of processor cycles lost to program cache stalls can become quite large.

- Many program cache conflict misses can be avoided with more intelligent placement of functions that are active at the same time. Program cache efficiency can be significantly improved using code placement strategies that utilize dynamic profile information that is gathered during the run of an instrumented application.

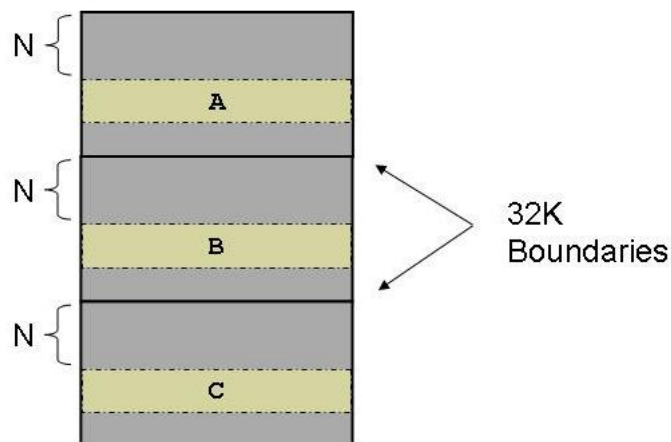
Consider another example, where function A calls function B, and function B calls function C. If these functions are all placed on the same cache line, then B will evict A every time that A calls it, and C will evict B every time that B calls it.

Code

```
top() {
    for (...)
        A();
}

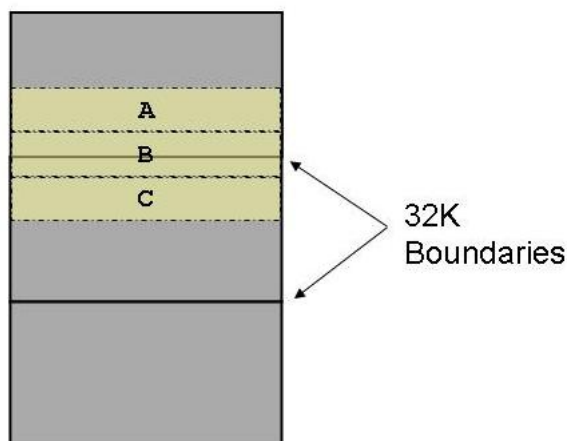
A() { B(); ... }
B() { C(); ... }
C() { ... }
```

Worst Case Layout



With intelligent placement, placing A, B, and C next to each other, we can avoid the cache conflict misses between them.

Improved Layout



- In the version 7.0 release of the C6000 code generation tools, a new cache layout tool, clt6x, is included. clt6x will take dynamic profile information in the form of a weighted call graph (WCG) and create a preferred function order command file that can be input into the linker to guide the placement of function subsections.

Goal

- Use cache layout tool to help improve your program locality and reduce the number of L1P cache conflict misses that occur during the run of your application, thereby improving your application's performance.

What Level of Performance Improvements Can You Expect to See?

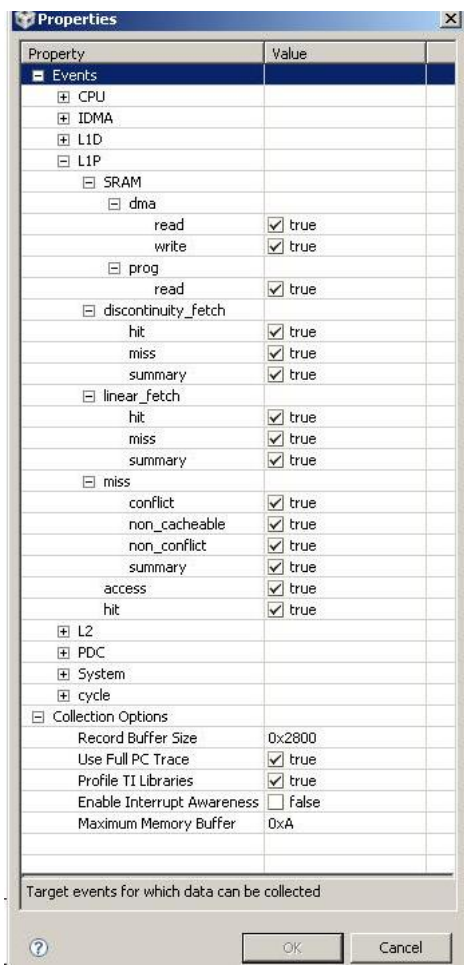
If your application does not suffer from inefficient usage of the L1P cache, then the new program cache layout capability will not have any affect on the performance of your application. Before you invest development time into applying the program cache layout tooling to your application, the usage of the L1P cache in your application should be analyzed.

Evaluating L1P Cache Usage

Spending some time evaluating the L1P cache usage efficiency of your application will not only help you determine whether or not your application might benefit from using program cache layout, but it will also provide a rough estimate as to how much performance improvement you can reasonably expect from applying program cache layout.

There are several resources available to help you evaluate L1P cache usage in your application. One way of doing this is to use the Function Profiling capability in Code Composer Studio (available in CCS version 4 or later).

For example, if CCS is configured to use the "C64+ Megamodule Cycle Accurate Simulator" target configuration, you can set up the profiling capability to gather L1P cache information while your application is running. You can find the basic information about enabling the function profiling capability [here](#). To turn on collection of L1P cache information on a function by function basis, you will want to expand the L1P button while you have the function profiling "Properties" window open:



Then click on the buttons under the L1P expansion that you are interested in. In the above image, the profiler is set up to collect all of the available information about the L1P cache.

The number of CPU stall cycles that occur due to L1P cache misses gives you a reasonable upper bound estimate of the number of CPU cycles that you may be able to recover with the use of the program cache layout tooling in your application. Please be aware that the performance impact due to program cache layout will tend to vary for the different data sets that are run through your application.

Further Resources

- Information about profiling with Debug Server Scripting (DSS) can be found [here](#).
- Information about mining information from the XML output of the ofd6x utility can be found in the [Code Generation Tools XML Processing Scripts](#) wiki article.
- This is a link to a presentation that was given at the 2008 TI Developer's Conference about C64+ cache visibility and application-level techniques to improve memory performance (http://wiki.sanb.design.ti.com/twiki/pub/SDSApps/ConferenceCommitments/TIDC08_SystemLevelCacheVisibilityandOptimization.ppt)

Program Cache Layout Related Features and Capabilities

The C6000 CGT v7.0 introduces some new features and capabilities that can be used in conjunction with the cache layout tool, clt6x. The following is a summary:

Path Profiler

The C6000 CGT includes a path profiling utility, pprof6x, that is run from the compiler, cl6x. The pprof6x utility is invoked by the compiler when the **--gen_profile** or the **--use_profile** command is used from the compiler command line:

```
cl6x --gen_profile ... file.c
cl6x --use_profile ... file.c
```

For further information about Profile Based Optimization and a more detailed description of the profiling infrastructure within the C6000 CGT, please see the latest *TMS320C6x Optimizing C Compiler User's Guide*.

Analysis Options

"--analyze=callgraph"	- Instructs the compiler to generate weighted call graph (WCG) analysis information.
"--analyze=codecov"	- Instructs the compiler to generate code coverage analysis information. This option replaces the previous --codecov option.
"--analyze_only"	- Halt compilation after generation of analysis information is completed.

- Behavior
 1. pprof6x will append code coverage/WCG analysis information to existing CSV files that contain the same type of analysis information.
 2. pprof6x will check to make sure that an existing CSV file contains analysis information that is consistent with the type of analysis information it is being asked to generate (whether it be code coverage or WCG analysis). Attempts to mix code coverage and WCG analysis information in the same output CSV file will be detected and pprof6x will emit a fatal error and abort.

Environment Variables

To assist with the management of output CSV analysis files, pprof6x supports two new environment variables:

TI_WCGDATA	- Allows user to specify single output CSV file for all WCG analysis information. New information will be appended to the CSV file identified by this environment variable, if the file already exists.
TI_ANALYSIS_DIR	- Specifies the directory in which the output analysis file will be generated. The same environment variable can be used for both code coverage information and weighted call graph information (all analysis files generated by pprof6x will be written to the directory specified by the TI_ANALYSIS_DIR environment variable).

- NOTE: The existing TI_COVDIR environment variable is still supported when generating code coverage analysis, but is overridden in the presence of a defined TI_ANALYSIS_DIR environment variable.

Cache Layout Tool, clt6x

```
usage: "clt6x <CSV files with WCG info> -o folder.cmd"
```

- Create a preferred function order command file from input WCG information.

Linker

--preferred_order Option

```
"--preferred_order=<function specification>"
```

- Prioritize the placement of a function relative to others based on the order in which --preferred_order options are encountered during the linker invocation.

unordered() Linker Command File (LCF) Operator

```
"unordered()"
```

- This operator will relax placement constraints placed on an output section whose specification includes an explicit list of which input sections are contained in the output section.

Program Cache Layout Development Flow

Once you have determined that your application is experiencing some inefficiencies in its usage of the program cache, you may decide to include the program cache layout tooling in your development to attempt to recover some of the CPU cycles that are being lost to stalls due to program cache conflict misses.

This section presents a development flow that incorporates the use of the program cache layout tooling. To get started using the program cache layout capability, it is recommended that you read this section and then proceed to the simple cache layout tool tutorial below.

Gather Dynamic Profile Information

The cache layout tool, clt6x, relies on the availability of dynamic profile information in the form of a weighted call graph (WCG) in order to produce a preferred function order command file that can be used to guide function placement at link-time when your application is re-built.

There are several ways in which this dynamic profile information can be collected. For example, if you are running your application on hardware, you may have the capability to collect a PC discontinuity trace. The discontinuity trace can then be post-processed to construct WCG input information for the clt6x.

The method for collecting dynamic profile information that is presented here relies on the path profiling capabilities in the C6000 code generation tools. Here is how it works:

1. Build an instrumented application

We are going to build an instrumented application using the **--gen_profile_info** option ...

```
Compile:
%> cl6x <options> --gen_profile_info <src_file(s)>
```

```
Compile and link:
%> cl6x <options> --gen_profile_info <src_file(s)> -z -l<lnk.cmd>
```

Use of **--gen_profile_info** instructs the compiler to embed counters into the code along the execution paths of each function.

2. Run instrumented application to generate .ppd file

When the application runs, the counters embedded into the application by **--gen_profile_info** keep track of how many times a particular execution path through a function is traversed. The data collected in these counters is written out to a profile data file named pprofout.ppd.

The profile data file is automatically generated. For example, if you are using the C64+ simulator under CCS, you can load and run your instrumented program, and you will see that a new pprofout.ppd file is created in your working directory (where the instrumented application is loaded from).

3. Decode profile data file

Once you have a profile data file, the file is decoded by the profile data decoder tool, pdd6x, as follows:

```
%> pdd6x -e<instrumented app out file> -o=pprofout.prf pprofout.ppd
```

pdd6x produces a .prf file is then fed into the re-compile of the application that uses the profile information to generate WCG input data.

4. Use decoded profile information to generate WCG input

The compiler now supports a new option, **--analyze**, which is used to tell the compiler to generate WCG or code coverage analysis information. Its syntax is as follows:

```
--analyze=callgraph -- Instructs the compiler to generate WCG information.
```

```
--analyze=codecov -- Instructs the compiler to generate code coverage
                    information. This option replaces the previous
                    --codecov option.
```

The compiler also supports a new **--analyze_only** option which instructs the compiler to halt compilation after the generation of analysis information has been completed. This option replaces the previous **--onlycodecov** option.

To make use of the dynamic profile information that you gathered, re-compile the source code for your application using the **--analyze=callgraph** option in combination with the **--use_profile_info** option:

```
%> cl6x <options> -mo --analyze=callgraph --use_profile_info=pprofout.prf <src_file(s)>
```

Use of **-mo** instructs the compiler to generate code for each function into its own subsection. This option provides the linker with the means to directly control the placement of the code for a given function.

The compiler generates a CSV file containing WCG information for each source file that is specified on the command line. If such a CSV file already exists, then new call graph analysis information will be appended to the existing CSV file. These CSV files are then input to the cache layout tool (clt6x) to produce a preferred function order command file for your application.

For more details on the content of the CSV files (containing WCG information) generated by the compiler, please see *Comma Separated Values (CSV) Files with WCG Information* section below.

Generate Preferred Function Order from Dynamic Profile Information

At this point, the compiler has generated a CSV file for each C/C++ source file specified on the command line of the re-compile of the application. Each CSV file contains weighted call graph information about all of the call sites in each function defined in the C/C++ source file.

The new cache layout tool, clt6x, collects all of the WCG information in these CSV files into a single, merged WCG. The WCG is processed to produce a preferred function order command file that is fed into the linker to guide the placement of the functions defined in your application source files. This is how to use clt6x:

```
%> clt6x *.csv -o forder.cmd
```

The output of clt6x is a text file containing a sequence of **--preferred_order=<function specification>** options. By default, the name of the output file is "forder.cmd", but you can specify your own file name with the -o option. The order in which functions appear in this file is their preferred function order as determined by the clt6x.

In general, the proximity of one function to another in the preferred function order list is a reflection of how often the two functions call each other. If two functions are very close to each other in the list, then the linker interprets this as a suggestion that the two functions should be placed very near to one another. Functions that are placed close together are less likely to create a cache conflict miss at runtime when both functions are active at the same time. The overall effect should be an improvement in program cache efficiency and performance.

Utilize Preferred Function Order in Re-Build of Application

Finally, the preferred function order command file that is produced by the clt6x is fed into the linker during the re-build of the application, as follows:

```
%> cl6x <options> -z *.obj forder.cmd -l<lnk.cmd>
```

The preferred function order command file, forder.cmd, contains a list of **--preferred_order=<function specification>** options. The linker prioritizes the placement of functions relative to each other in the order that the **--preferred_order** options are encountered during the linker invocation.

Each **--preferred_order** option contains a function specification. A function specification can describe simply the name of the function for a global function, or it may provide the path name and source file name where the function is defined. A function specification that contains path and file name information is used to distinguish one static function from another that has the same function name.

As mentioned earlier, the **--preferred_order** options are interpreted by the linker as suggestions to guide the placement of functions relative to each other. They are *not* explicit placement instructions. If an object file or input section is explicitly mentioned in a linker command file SECTIONS directive, then the placement instruction specified in the linker command file takes precedence over any suggestion from a **--preferred_order** option that is associated with a function that is defined in that object file or input section.

This precedence can be relaxed by applying the **unordered()** operator to an output specification as described in the *Linker Command File Operator - unordered()* section below.

Comma-Separated Values (CSV) Files with WCG Information

The format of the CSV files generated by the compiler under the "--analyze=callgraph --use_profile_info" option combination is as follows:

```
"caller","callee","weight" [CR][LF]
<caller spec>,<callee spec>,<call frequency> [CR][LF]
<caller spec>,<callee spec>,<call frequency> [CR][LF]
<caller spec>,<callee spec>,<call frequency> [CR][LF]
...
```

- Line 1 of the CSV file is the header line. It specifies the meaning of each field in each line of the remainder of the CSV file. In the case of CSV files that contain weighted call graph information, each line will have a caller function specification, followed by a callee function specification, followed by an unsigned integer that provides the number of times a call was executed during run time at a given call site.
- There may be instances where the caller and callee function specifications are identical on multiple lines in the CSV file. This will happen when a caller function has multiple call sites to the callee function. In the merged WCG that is created by the clt6x, the weights of each line that has the same caller and callee function specifications will be added together.
- The CSV file that is generated by the compiler using the path profiling instrumentation will not include information about indirect function calls or calls to runtime support helper functions (like _remi or _divi). However, you may be able to gather information about such calls with another method (like the PC discontinuity trace mentioned earlier).
- The format of these CSV files is in compliance with the RFC-4180 specification of Comma-Separated Values (CSV) files. For more details on this specification, please see the following URL:

```
http://tools.ietf.org/html/rfc4180
```

Linker Command File Operator - unordered()

A new **unordered()** operator is now available for use in a linker command file. The effect of this operator is to relax the placement constraints placed on an output section specification in which the content of the output section is explicitly stated.

Basics

Consider an example output section specification:

```
SECTIONS
{
    .text:
    {
        file.obj(.text:func_a)
        file.obj(.text:func_b)
        file.obj(.text:func_c)
        file.obj(.text:func_d)
        file.obj(.text:func_e)
        file.obj(.text:func_f)
        file.obj(.text:func_g)
        file.obj(.text:func_h)

        *(.text)
    } > PMEM

    ...
}
```

In the above SECTIONS directive, the specification of '.text' explicitly dictates the order in which functions are laid out in the output section. That is, by default, the linker will layout func_a through func_h in exactly the order that they are specified, regardless of any other placement priority criteria (such as a preferred function order list that is enumerated by **--preferred_order** options).

The **unordered()** operator can be used to relax this constraint on the placement of the functions in the '.text' output section so that placement can be guided by other placement priority criteria.

The **unordered()** operator can be applied to an output section as follows:

```
SECTIONS
{
  .text: unordered()
  {
    file.obj(.text:func_a)
    file.obj(.text:func_b)
    file.obj(.text:func_c)
    file.obj(.text:func_d)
    file.obj(.text:func_e)
    file.obj(.text:func_f)
    file.obj(.text:func_g)
    file.obj(.text:func_h)

    *(.text)
  } > PMEM

  ...
}
```

So that, given a list of **--preferred_order** options as follows:

```
--preferred_order="func_g"
--preferred_order="func_b"
--preferred_order="func_d"
--preferred_order="func_a"
--preferred_order="func_c"
--preferred_order="func_f"
--preferred_order="func_h"
--preferred_order="func_e"
```

The placement of the functions in the '.text' output section will then be guided by this preferred function order list. This placement will be reflected in a linker generated map file, as follows:

SECTION ALLOCATION MAP				
output section	page	origin	length	attributes/ input sections
.text	0	00000020	00000120	
		00000020	00000020	file.obj (.text:func_g:func_g)
		00000040	00000020	file.obj (.text:func_b:func_b)
		00000060	00000020	file.obj (.text:func_d:func_d)
		00000080	00000020	file.obj (.text:func_a:func_a)
		000000a0	00000020	file.obj (.text:func_c:func_c)
		000000c0	00000020	file.obj (.text:func_f:func_f)
		000000e0	00000020	file.obj (.text:func_h:func_h)
		00000100	00000020	file.obj (.text:func_e:func_e)
		...		

About DOT Expressions in the Presence of unordered()

Another aspect of the **unordered()** operator that should be taken into consideration is that even though the operator causes the linker to relax constraints imposed by the explicit specification of an output section's contents, the **unordered()** operator will still respect the position of a DOT expression within such a specification.

Consider the following output section specification:

```
SECTIONS
{
  .text: unordered()
  {
    file.obj(.text:func_a)
    file.obj(.text:func_b)
    file.obj(.text:func_c)
    file.obj(.text:func_d)

    . += 0x100;

    file.obj(.text:func_e)
    file.obj(.text:func_f)
    file.obj(.text:func_g)
    file.obj(.text:func_h)

    *(.text)
  } > PMEM

  ...
}
```

In the above specification of '.text', a DOT expression, ". += 0x100;", separates the explicit specification of two groups of functions in the output section. In this case, the linker will honor the specified position of the DOT expression with respect to the functions on either side of the expression. That is, the **unordered()** operator will allow the preferred function order list to guide the placement of func_a through func_d relative to each other, but none of those functions will be placed after the hole that is created by the DOT expression. Likewise, the **unordered()** operator allows the preferred function order list to influence the placement of func_e through func_h relative to each other, but none of those functions will be placed before the hole that is created by the DOT expression.

GROUPs and UNIONs

The **unordered()** operator can only be applied to an output section. This includes members of a GROUP or UNION directive. For example,

```
SECTIONS
{
```

```
GROUP
{
    .grp1:
    {
        file.obj(.grp1:func_a)
        file.obj(.grp1:func_b)
        file.obj(.grp1:func_c)
        file.obj(.grp1:func_d)
    } unordered()

    .grp2:
    {
        file.obj(.grp2:func_e)
        file.obj(.grp2:func_f)
        file.obj(.grp2:func_g)
        file.obj(.grp2:func_h)
    }

    .text: { *(.text) }
} > PMEM

...
}
```

The above SECTIONS directive applies the **unordered()** operator to the first member of the GROUP. The '.grp1' output section layout can then be influenced by other placement priority criteria (like the preferred function order list), whereas the '.grp2' output section will be laid out as explicitly specified.

The **unordered()** operator cannot be applied to an entire GROUP or UNION. Attempts to do so will result in a linker command file syntax error and the link will be aborted.

Cache Layout Tool Tutorial

As a means of familiarizing yourself with the cache layout tool development flow, you can walk through this guided tour of the development of a simple application. Included in the release distribution, you will find a sub-directory, `clt_tutorial`. This sub-directory contains the following files:

```
clt_tutor.txt
main.c
lots.c
rare.c
```

You can also get the C source files from [here](#).

To begin the tutorial, change your location to the `clt_tutorial` sub-directory or copy the source files to your own working directory. Then consider the following ...

1. Introduction to the Source Files

- `main.c`:
 - defines `main()`
 - `main()` calls `rare()` once
 - `main()` calls `main.c:local()` 4 times
 - defines static `local()`
 - `main.c:local()` calls `lots()` 80 times
- `lots.c`:
 - defines `lots()`; globally visible
 - `lots()` calls `lots.c:local()` 100+ times
 - defines `lots.c:local()`
- `rare.c`:
 - defines `rare()`; globally visible

2. Build an Instrumented Application

```
%> rm -f *.ppd
%> cl6x -mv64+ --gen_profile_info main.c lots.c rare.c -z -lnk.cmd -o app.out -m app.map
```

3. Gather Dynamic Profile Information

- Load and run `app.out` with CCS C64+ Simulator
- You should have a `pprofout.ppd` file in your working directory after completing this step.

4. Decode Profile Data File

```
%> rm -f *.prf
%> pdd6x pprofout.ppd -eapp.out -o=pprofout.prf
```

- You should have a `pprofout.prf` file in your working directory after completing this step.

5. Use Profile Information in Re-Compile of Application

```
%> cl6x -mv64+ --use_profile_info=pprofout.prf --analyze=callgraph -mo main.c lots.c rare.c
```

- You should have 3 CSV files in your working directory after completing this step: `main.csv`, `lots.csv`, `rare.csv`. Their contents should be as follows:

```
main.csv:
"caller","callee","weight"
main,rare,1
main,main.c:local,4
main,printf,1
main,fflush,1
main.c:local,printf,4
main.c:local,lots,80
```

```
lots.csv:
"caller", "callee", "weight"
lots,lots.c:local,80
lots,lots.c:local,28
lots.c:local,printf,108
```

```
rare.csv:
"caller", "callee", "weight"
```

6. Generate Preferred Function Order Command File

```
%> rm app_pfo.cmd
%> clt6x main.csv lots.csv rare.csv -o app_pfo.cmd
```

- You should have an app_pfo.cmd in your working directory containing:

```
--preferred_order="printf"
--preferred_order="lots.c:local"
--preferred_order="lots"
--preferred_order="main.c:local"
--preferred_order="main"
--preferred_order="rare"
--preferred_order="fflush"
```

- This is the preferred function order list. Note that the two versions of local() are distinguished by their source file names:

```
"main.c:local"
"lots.c:local"
```

7. Re-Link Application Incorporating Preferred Function Order

```
%> cl6x -mv64+ -z main.obj lots.obj rare.obj app_pfo.cmd -lnk.cmd -o app_opt.out -o app_opt.map
```

- You should have an app_opt.map file in your working directory. If you open it and look at the contents of the .text output section, you should see that the placement of the functions specified in the app_pfo.cmd file should match their actual order in that file.

Things To Be Aware Of

There are some behavioral characteristics and limitations of the program cache layout development flow that you should bear in mind:

- Generation of Path Profiling Data File (.ppd)

When running an application that has been instrumented to collect path-profiling data (using **--gen_profile_info** compiler option during build), the application will use functions in the runtime support library to write out information to the path profiling data file (pprofout.ppd in the above tutorial). If there is a path profiling data file already in existence when the application starts to run, then any new path profiling data generated will be appended to the existing file.

To prevent combining path profiling data from separate runs of an application, you will need to either rename the path profiling data file from the previous run of the application or remove it before running the application again.

- Indirect Calls Not Recognized by Path Profiling Mechanisms

When using available path profiling mechanisms to collect weighted call graph information from the path profiling data, pprof6x does not recognize indirect calls. An indirect call site will not be represented in the CSV output file that is generated by pprof6x.

You can workaround this limitation by introducing your own information about indirect call sites into the relevant CSV file(s). If you take this approach, please be sure to follow the format of the callgraph analysis CSV file ("caller", "callee", "call frequency").

If you are able to get weighted call graph information from a PC trace into a callgraph analysis CSV, this limitation will no longer apply (as the PC trace can always identify the callee of an indirect call).

- Multiple **--preferred_order** Options Associated with Single Function

There may be cases in which you might want to input more than one preferred function order command file to the linker during the link of an application. For example, you may have developed or received a separate preferred function order command file for one or more of the object libraries that are used by your application.

In such cases, it is possible that one function may be specified in multiple preferred function order command files. If this happens, the linker will honor only the first instance of the **--preferred_order** option in which that function is specified.

Addendum: Using 7.x.x CLT Capability with 6.1.x CGT

It is possible to apply the program cache layout capability to applications that use older versions of the compiler to generate the object code that is to be linked together to form the final application. For instance, you may be using the version 6.1.x C6x compiler to generate object code for your application, but you suspect that your application suffers from inefficient usage of the L1P cache.

If your application spends a significant amount of time in L1P stall cycles due to excessive L1P cache misses, then you may be able to realize some benefit by creating a preferred order linker command file using the version 7.x.x compiler and linking your 6.1.x object files together using the version 7.x.x linker.

General Approach

- Build a profiling version of your application using the version 6.1.x code generation tools.
- Run the profiling version of your application over a representative example set of input data in order to collect path profiling statistics.
- Build a preferred order linker command file with the version 7.x.x code generation tools. The compiler makes use of the profiling data that was generated in the previous step and uses it to help analyze call-graph information that can be fed into and processed by the program cache layout utility to produce a preferred order linker command file (LCF).

4. Compile the source files of your application using the version 6.1.x compiler, but this time be sure to generate code for functions into their own subsections (use the `-mo` option to enable this).
5. Link your 6.1.x object files together using the version 7.x.x linker and incorporating the preferred order linker command file into the final link. The preferred order linker command file will help to guide the placement of function subsections relative to one another to reduce the number of L1P cache conflict misses that occur when the application is run.

Constraints / Assumptions

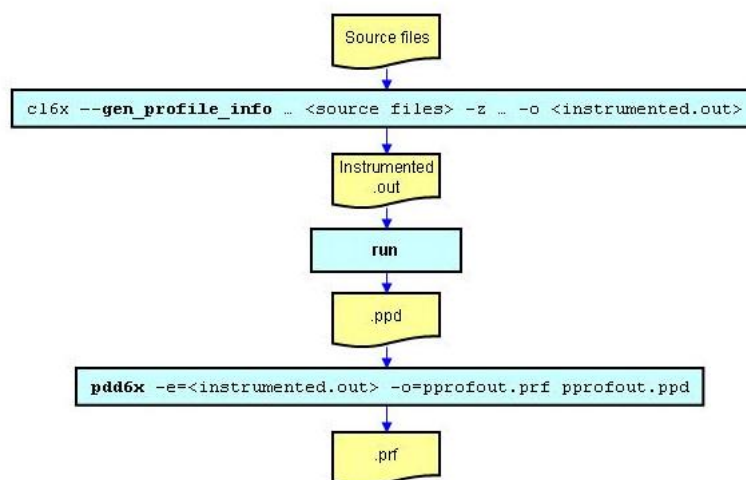
Note: Use of the program cache layout tooling in version 7.x.x of the tools makes use of the path profiling capabilities that were developed for version 6.1.x of the tools. We do not recommend trying to apply the program cache layout tooling to objects that are built with any versions of the compiler prior to 6.1.x.

- You must have access to both the version 6.1.x code generation tools and the version 7.x.x code generation tools.
- You must be able to switch from one toolset to another while proceeding through the development flow described below.
- You must have some way of gathering profiling data into an output file. This is enabled when running the CCS debugger in simulation mode. If you do not have access to a CCS simulator, then you may need to add some low-level file-I/O functions to the profiling version of your application, namely `fwrite()`. Please see the RTS source files if you would like to see an example implementation.
- You must use the `-mo` option to compile the 6.1.x object files that will be linked together in the final link step to create the updated version of your application.

Development Flow

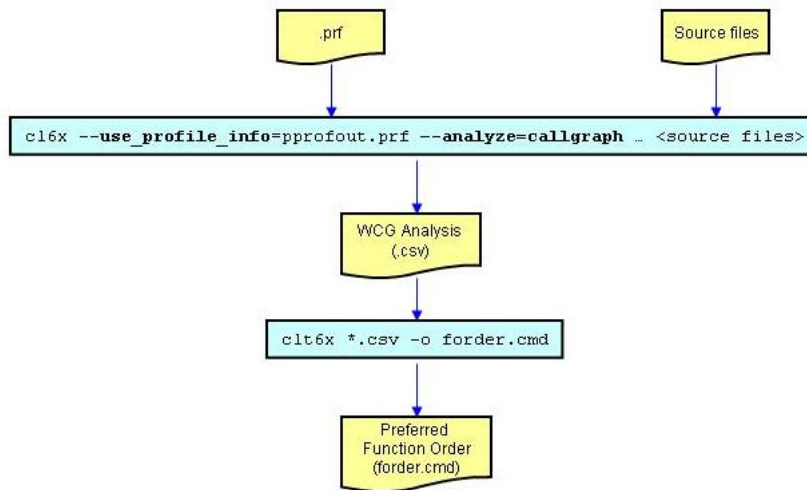
It is assumed that you already have a baseline version of your application that was built using the version 6.1.x code generation tools. This should have been built using your normal compiler and linker options. If possible, run this application and gather statistics about the L1P cache usage efficiency of your baseline application (number of total cycles executed, number of cycles spent in L1P stall due to L1P cache misses, number of L1P cache misses, etc.). This information may be useful in measuring what impact the use of the cache layout tooling has on your application.

1. **Profile Application** - Compile and link your application using the version 6.1.x code generation tools with the `--gen_profile_info` option to create a version of your application that will generate profile information when it is run with an example set of input data. Run this version of the application using an example set of input data to generate path profiling data into a `.pdd` file.
2. **Decode Path Profiling Data** - Use the profiling data decoder utility to parse the path profiling data file (`.pdd`) and create a compiler usable profile information file (`.prf`).



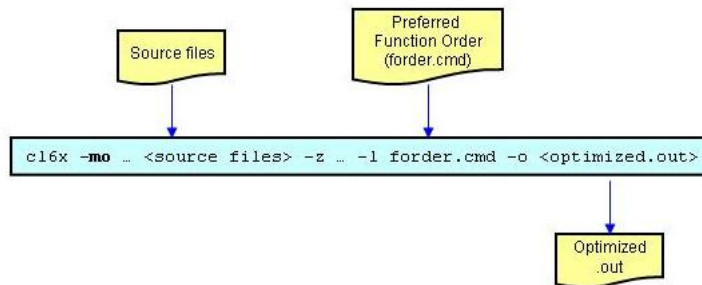
Generate/Collect Profiling Information

3. **Create Weighted Call Graph Information Files** - Compile your application source files using the version 7.x.x compiler; adding the `-mo`, `--use_profile_info`, and `--analyze=callgraph` options to your normal compile options. This compile step will create a series of comma-separated value files (`.csv`) containing call-graph information that can be fed into the program cache layout tool, `clt6x`, to create a preferred order linker command file.
4. **Create Preferred Order Linker Command File** - Using the cache layout tool, `clt6x`, available in your version 7.x.x code generation tools, process the weighted call graph information that was generated in the previous step (`.csv` files) to build a preferred order linker command file (LCF).



Generate Preferred Function Order LCF

5. **Compile Application Using -mo Option** - Compile your application source files using the version 6.1.x compiler adding the -mo option to your original compile options. This will enable the subsequent final link step to easily manipulate the placement of functions. The -mo option causes the compiler to generate code for a given function into its own subsection. The linker can then control the placement of the code for the function via the subsection name.
6. **Link Application** - Link the object files that you generated in the previous step (6.1.x object files) using the version 7.x.x linker. The preferred order linker command file that was generated by the cache layout tool in step 5 should be specified as input to this final link. The preferred order options in this linker command file will help to guide the placement of functions relative to each other in order to reduce the number of L1P cache conflict misses that are likely to occur.



Compile/Link Optimized Application

An Example

To prove this development flow, I have built and run an example tcpip benchmark application.

Here are the step-by-step major details:

- Set up path to 6.1.x toolset

```
%> set path=<your path to 6.1.x tools>
```

- Build a profiling version of the tcpip application using the --gen_profile_info compiler option

```
%> set C6X_C_DIR=<your include and library search path>
%> mkdir -p c64p
%> cl6x -mv6400+ --abi=coffabi -qq -D... -I... -mh -mi -mt -k -fsc64p -fr64p -ftc64p -eoo -o3 --gen_profile_info socket.c
%> etc.
%> cl6x -mv6400+ --abi=coffabi -q -D... -I... -mh ... c64p/socket.o ... -z -l./C64X.cmd -lrts64plus.lib -x -o tcpip_c64p.out -m tcpip_c64p.map
```

- Run the profiling version of the tcpip application using the benchmark input data set. Profiling data is automatically collected into a path profiling data file (.ppd) by the CCS simulator that was used to run the benchmark.

- Can use CCS simulator for this

1. Bring up CCS
2. Configure CCS to use C6400+ simulator
3. Load and run tcpip_c64p.out

- Note general performance and L1P usage efficiency statistics (to be used in later comparison to measure impact of using CLT)
- Decode the path profiling data into a compiler-readable profile information file (.prf)

```
%> pdd6x -e tcpip_c64p.out -o tcpip_c64p_prof.prf tcpip_c64p_prof.ppd
```

- Set up path to 7.x.x toolset

- Compile with v7.x.x tools and `--use_profile_info` option (use path profiling information) to generate call-graph analysis information

- Process call-graph analysis information with cache layout tool to generate a preferred order linker command file, **tcpip.cmd**

- Set up path to 6.1.x toolset

- Compile application source files using **-mo** option to generate code for functions into their own subsection. This enables the subsequent link to manipulate the placement of functions via their subsections.

- Set up path to 7.x.x toolset

- Link 6.1.x object files together using 7.x.x linker and incorporate preferred order linker command file, **tcpip.cmd**, to guide the placement of functions relative to each other in order to reduce the occurrence of L1P cache conflict misses.

- Run the updated tcpip application using the same benchmark input data set as before.

- Again using CCS simulator for this

1. Bring up CCS
2. Configure CCS to use C6400+ simulator
3. Load and run tcipip_c64p_opt.out

- Note general performance and L1P usage efficiency statistics and compare numbers vs. earlier run of application. Difference represents performance impact of using the cache layout tool in the development flow.

Application	Total Cycles (baseline)	L1P Stall Cycles (baseline)	Total Cycles (clt6x)	L1P Stall Cycles (clt6x)	(baseline/clt6x)*100 - 100
tcpip	43057291	16247019	36029377	9230267	19.506%

{{ 1. switchcategory:MultiCore= - For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum - For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum Please post only comments related to the article Program Cache Layout here.	Keystone= - For technical support on MultiCore devices, please post your questions in the <u>C6000 MultiCore Forum</u> - For questions related to the BIOS MultiCore SDK (MCSDK), please use the <u>BIOS Forum</u> Please post only comments related to the article Program Cache Layout here.	C2000=For technical support on the C2000 please post your questions on <u>The C2000 Forum</u>. Please post only comments about the article Program Cache Layout here.	DaVinci=For technical support on DaVincoplease post your questions on <u>The DaVinci Forum</u>. Please post only comments about the article Program Cache Layout here.	MSP430=For technical support on MSP430 please post your questions on <u>The MSP430 Forum</u>. Please post only comments about the article Program Cache Layout here.	OMAP35x=For technical support on OMAP please post your questions on <u>The OMAP Forum</u>. Please post only comments about the article Program Cache Layout here.	OMAPL1=For technical support on OMAP please post your questions on <u>The OMAP Forum</u>. Please post only comments about the article Program Cache Layout here.	MAVRK=For technical support on MAVRK please post your questions on <u>The MAVRK Toolbox Forum</u>. Please post only comments about the article Program Cache Layout here. }} For techni please po: questions http://e2e.ti.com/forums/processors/program-cache-layout Please po comments article Program Cache Layout
---	--	---	--	--	---	--	--

Wireless Connectivity

- [OMAP Applications Processors](#)

Retrieved from "https://processors.wiki.ti.com/index.php?title=Program_Cache_Layout&oldid=49072"

This page was last edited on 29 December 2010, at 13:57.

Content is available under [Creative Commons Attribution-ShareAlike](#) unless otherwise noted.