Technical notes on using Analog Devices DSPs, processors and development tools
Visit our Web resources http://www.analog.com/ee-notes and http://www.analog.com/processors or e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

PGO Linker - A Code Layout Tool for Blackfin Processors

Contributed by Kaushal Sanghai

Rev 1 – December 4, 2006

Introduction

Blackfin® processors provide on-chip Level 1 (L1) memory that can be configured to take advantage of application-specific workload characteristics. L1 memory can be configured as SRAM, or it can be split into a mix of SRAM and cache memory. L1 SRAM memory improves performance by statically mapping the most frequently executed code sections and critical real-time code.

To achieve the best system performance, an application developer can utilize the execution profile of an application and domain knowledge to hand-tune the code memory layout. However, this can be extremely time-consuming for large applications.

To assist in automating this tedious process and decrease development time, a link-time profile-guided optimization (PGO) tool, *PGO Linker*, has been developed to produce an efficient code map for L1 SRAM instruction memory. It uses the execution profile of the application and program optimization techniques to yield the most efficient code layout.

This document describes the functional behavior of the PGO Linker tool, as well as its usage, performance benefits, features, and limitations.

Motivation

Figure 1 shows the Blackfin processor memory hierarchy. L1 memory can be used entirely as SRAM memory, or it can be split between SRAM and cache memory.

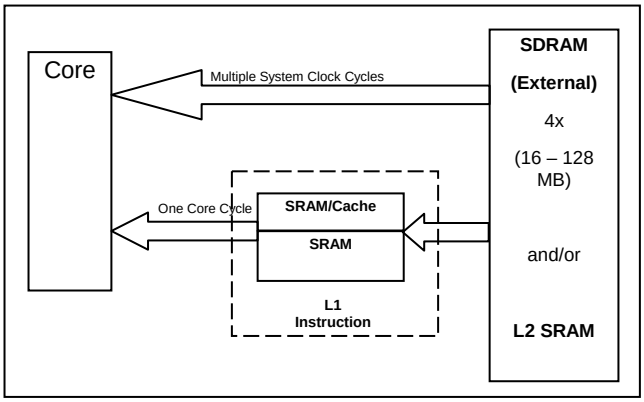

Figure 1. Blackfin Memory Hierarchy

Internal L2 memory shown in Figure 1 is available only on ADSP-BF561 dual-core Blackfin processors. It can be configured as SRAM only.

All addresses that do not resolve to L1 SRAM are accessed directly by the core in external memory via the External Bus Interface Unit (EBIU), resulting in costly system clock (SCLK) domain accesses. However, when L1 cache memory is used, external memory is accessed only after a cache miss, which improves average memory access time. Performance can be further improved by utilizing L1 SRAM. Code mapped to L1 SRAM is not subject to cache misses, thereby completely eliminating any external memory accesses and providing guaranteed fast access to all requests by the core. Unfortunately, all code cannot be mapped to L1 SRAM, due to the limited amount of internal memory available.

For better performance, only critical real-time code and the most frequently executed code are mapped to L1 SRAM space. This ensures minimum memory access latency for most of the program's execution. The less frequently accessed code can be mapped to external memory, which is accessed directly by the core or through L1 cache (if cache is enabled).

The PGO Linker automates the process of mapping the most frequently executed code to L1 SRAM memory by using program optimization techniques to produce an efficient code layout. This eliminates the need to hand-tune the code layout and saves the developer from having to make tedious changes to source code.

How Does the PGO Linker Work?

This section describes the functional behavior of the PGO Linker utility. Figure 2 shows the steps performed by the tool.

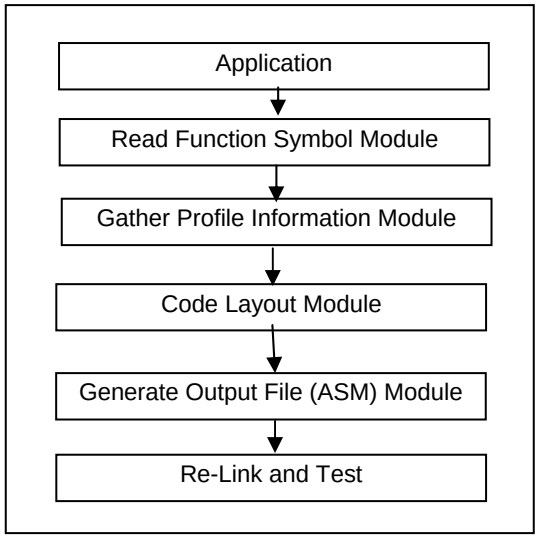

Figure 2. Functional Flow of PGO Linker

First, the PGO Linker gathers the functional symbol information and the profile data output from the VisualDSP++® IDDE. To obtain the profile data, the tool uses a profiling mechanism based on the type of session being run.

Emulator sessions use the statistical profiler, and simulator sessions use the linear profiler. The PGO Linker's optimization command line switches (`O1` and `O2`) specify the optimization technique to be employed during the build to produce an output `.asm` file.

The output `.asm` file contains an ordered priority list for all functions in the application. Once this file is included in the project, the linker uses the priority list of functions specified in the output file (`.asm`) and the linker description file (`.ldf`) to produce the optimal code layout.

How Do I Use the PGO Linker?

The PGO Linker is a command-line utility installed under the VisualDSP++ directory. This tool requires a Blackfin processor executable (`.dxe`) file and input profile data to produce the code layout.

The output of the PGO Linker is only as good as the profile data set it gathers from the VisualDSP++ IDDE.

The input profile data should be independent of the initial memory placement of the code by the linker. By default, the linker places functions in the order in which they appear in the source files listed in the project folder. Functions are first mapped to L1 SRAM memory, and the remaining functions (if any) are mapped to the higher levels of the memory hierarchy.

The statistical profiler accounts for memory access latencies to the different levels of memory. This implies that functions placed in external memory will have a higher execution percentage (in terms of real time) than if the same functions were to be placed in on-chip L1 SRAM. Thus, to avoid any bias to functions placed in external memory, place all code in one level of memory hierarchy (preferably in external memory) with cache enabled. This minimizes the

memory access latencies to the different levels of memory in the execution profile.

To completely exclude memory latencies in the execution profile, run the PGO Linker tool in a compiled simulation session. In a compiled simulation session, the linear profiler profiles only the instruction count; therefore, it does not account for any access latencies due to memory hierarchy.

The sample input data set should be representative and small for better and faster results.

To run the PGO Linker:

1. Load the program in VisualDSP++.
2. Choose `Tools->Profiler` to open the Profiler window.
3. Run the program with a sample input data set.
4. Wait until the program halts or is halted manually.
5. Open a Windows Command Prompt window.
6. Execute the PGO Linker utility with the appropriate command-line arguments (Listing 1). The tool produces an `.asm` file.
7. Include the generated `.asm` file in the project.
8. Rebuild the project.

```
PGOLinker <dxefile(.dxe)> <Output file (.asm)> -L1Min -L1Max -L1Step –algorithm

<dxefile(.dxe)> - Blackfin executable
<Output file(.asm)> - The file would contain all the linker directives for the
function symbols

Options
-------
-help:   To display all options
-L1Min:  Minimum size of L1 Instruction SRAM, default: 4KB
-L1Max:  Maximum size of L1 Instruction SRAM, default: 80KB
-L1Step: Steps of increase in L1 SRAM, default: 4KB
-algorithm: O1 | O2
```

Listing 1. PGO Linker Command Line Example

The `.asm` file produced by the PGO Linker is added to the project. Upon rebuilding the project, the linker places functions based on the priorities assigned in the generated `.asm` file. The performance can be evaluated with test inputs.

Output File Format and .priority Directive

The output has an `.asm` extension and the following format:

```
.extern function_symbol;

.priority function_symbol,
src_file_path, assigned_priority #;
```

where:

`.extern` notifies the linker that *`function_symbol`* is an externally defined symbol.

`function_symbol` is the mangled function name. The linker uses mangled function names for mapping object code.

`.priority` is the linker directive used for mapping functions based on the assigned priority level.

`src_file_path` is used to resolve static symbols.

`assigned_priority #` is a positive integer whose value specifies the order of mapping for the functions ¬ the higher the number, the higher the priority.

An example of an output file is:

```
.extern _foo;
.priority _foo,"..\..\src\", 900;
.extern _bar;
.priority _bar,"..\..\src\", 897;
```

In this case, the `foo( )` function has higher priority than the `bar( )` function.

Because of the `.priority` directives associated with function symbols in the example above, the linker will place the `foo( )` function before mapping the `bar( )` function in L1 memory since 900 is greater than 897.

In the generated file, function priorities are assigned with an interval of "3", providing the user with the flexibility to add functions based on application knowledge. Also, any number of functions can be given higher priority above the highest priority function. In the same way, any number of functions can be assigned lower priority below the lowest priority function.

Note that the `.priority` directive can also be used without the PGO Linker.

Besides the output file, the PGO Linker also provides information pertaining to L1 size versus execution percentage in order to help evaluate L1 memory configurations and size.

The dark blue text in Figure 3 indicates when the PGO Linker tool should be used in a typical embedded system design cycle. Note that for any changes to source code, the layout should be regenerated.

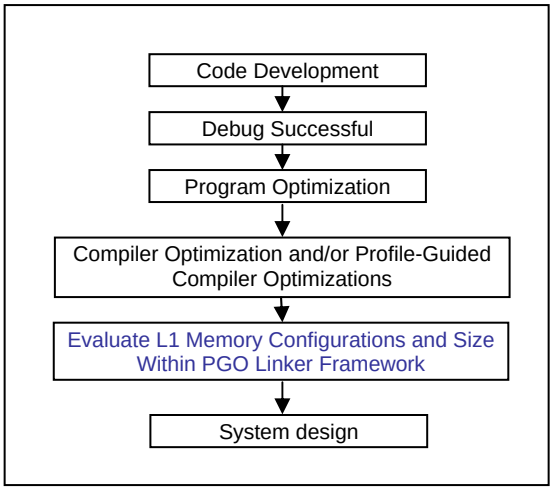

Figure 3. When To Use PGO Linker

Benefits, Features, Limitations, and Usage

Performance improvements are largely dependent on application characteristics in terms of code size and the number of functions. Experiments over a set of benchmark programs showed a 3% to 32% improvement in core cycles as compared to the default linker mapping. In most cases, the `O2` algorithm provided an additional 1% to 5% improvement compared to the `O1` algorithm. The tests were conducted with cache enabled for a 16K L1 SRAM.

As a general rule of thumb, the PGO Linker is most useful for applications with code size exceeding 150 Kbytes and with over 200 functions.

Benefits

In addition to performance benefits, the PGO Linker also:

- Reduces development time
- Performs code layout on a per-function basis without the need to modify source code
- Provides greater control over code ordering and reduces code development effort, as there is no need to modify the `.ldf` file
- Incorporates two optimization techniques to produce an efficient code layout

Features

- Completely automated ¬ user intervention is not required
- Used in all VisualDSP++ sessions (simulator, compiled simulator, and emulator)
- Efficient in runtime
- Output is transparent to the user

Limitations

- Profile dependent ¬ if the sample input data set is not representative, the results may vary for different test inputs. This phenomenon is less sensitive in the case of code execution profiles.
- It is restricted only to code layout. Data layout should still be managed by the developer.

Typical Usage Scenarios

- Applications with large code size and large number of functions
- Performance evaluations of out-of-the-box benchmark programs
- Porting of legacy code, when user knowledge of application behavior is minimal

To Get Started

Download the PGO Linker utility from the associated `.zip` file. Copy `PGOLinker.exe` into the VisualDSP++ installation folder (default):

Program Files\Analog Devices\VisualDSP++

The PGO Linker utility works for VisualDSP++ releases starting with version 4.0 (updated July, 2006).

Appendix A

Sample Command-Line Output

```
C:\ProgramFiles\visualdsp>PGOLinker.exe "C:\mpeg\ mpeg2dec\BF533\mp2decoddata2.dxe"
"mp2.asm"
The command line options are configured as follows:-
DSPExecutable-->C:\mpeg\mpeg2dec\BF533\mp2 decoddata2.dxe
Linker directive Map File --> mp2.asm
Minimum L1 size selected --> 4
Maximum L1 size selected --> 80
L1 memory incremented in steps of --> 4
Optimization switch --> -O2

Connecting to the IDDE and loading Program
Connection to the IDDE established
Obtaining function symbol information
Function symbol information obtained

Getting profile Information
Analyzing the profile information
Analysis Done

Total sample count collected is --> 12513

The total execution from L1 for 4KB of L1 is 97.9142%
Total functions in L1 17
The total execution from L1 for 8KB of L1 is 100%
Total functions in L1 30
-----------------------------------------------
Ready to generate the asm file
Linker directive .asm file generated
```

References

[1] *ADSP-BF533 Blackfin Processor Hardware Reference.* Rev 3.2, July 2006. Analog Devices Inc.

[2] *VisualDSP++ 4.5 Linker and Utilities Manual*, Rev 2.0, April 2006. Analog Devices Inc.

[3] *Guide to Blackfin Processor LDF Files (EE-237),* Rev 1, May 2004. Analog Devices Inc.

Document History

Revision	Description
Rev 1 – December 4, 2006 *by Kaushal Sanghai*	Initial Release