Dynamic Call Graph

{{#switchcategory:MSP430=<McuHitboxHeader/>|C2000=<McuHitboxHeader/>|Stellaris=<McuHitboxHeader/>|TMS570=<McuHitboxHeader/>|MCU=<McuHitboxHeader/>|MAVRK=<MAVRKHitboxHeader/>|<HitboxHeader/>}}

Contents

Dynamic Call Graph

Overview

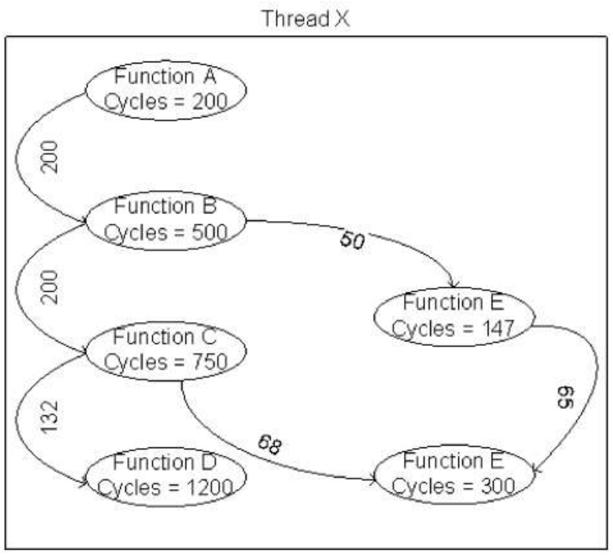
Dynamic Call Graph is a method of analyzing the call tree of an application using output from XDS560 Trace. The Goal is to display a thread based representation of actual function execution in an application. Dynamic Call Graph will take trace data capture while running an application in real time on real hardware, and convert the output to a GNU gprof like format. See an example of GNU gprof format here (http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html#SEC6) It supports generating profiling information on a thread by thread basis in multi-threaded applications as well.

Dynamic Call Graph can be implemented on any C64x+ device that supports trace.

Goal

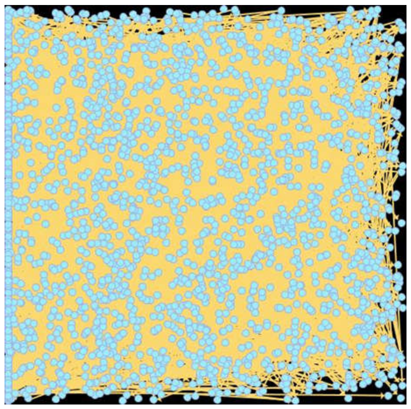
The goal of Dynamic call Graph is to create a cycle accurate thread based representation of actual function execution in an application. As shown in the image at right, the type of data that we are looking for on a thread level basis is the number of times a function was called, the various parent functions, the child functions, and the number of cycles spent in each.

We will accomplish this goal by using Trace to store a time stamp each time a function was entered and exited. We will also use Trace to capture the ID of the next thread in order to distinguish between different thread contexts. Once we have captured the information, we will display the result sin a GPROF like format for the user to analyze.



Desired Call Graph Output

Why GPROF? Why not graphics?



Graphical Call Graph

Many people wonder why use a text based GPROF format rather than a graphical display package that might show something similar to the Desired Call Graph output in the image above. The answer is really the size and complexity of an application. We just don't have enough space on a computer monitor or on a standard hard copy

We have analyzed a typical Communications Infrastructure application with the GUESS graphing package (http://graphexploration.cond.org) and have seen results similar to those in the image shown at left. As you should notice in the image, the background is actually black. The blue dots are function nodes, and the tan lines denote connections between parent and child functions. You'll notice on the left side of the image that the functions are neatly lined up along the side, but the connections to the children force the rest of the nodes to be placed haphazardly. To make this image remotely intuitive, we would likely need to print it on a ~3' x 3' (1 M x 1 M) poster. At that point, we might be able to read the values in the nodes, but would need significantly more area for the connectors to become clear. It becomes evident that this is into a feasible approach to display call graph data.

Why only 3 levels of Call tree

The 3 levels of call tree were chosen because this is the expected granularity that would be desired. With these three levels for each function, the user can completely reconstruct the entire call tree if he wanted to. The three levels is a good enough granularity to give a picture of the execution of any function of interest. It also lets the user focus on a single function at a time, by displaying only the function of interest's immediate parents and children.

Advantages of Dynamic Call graph over Traditional Methods of Profiling

There are a number of advantages to profiling in this manner over traditional profiling on the simulator.

- Real Time/Real Hardware/Real Results

Profiling with hardware trace allows you to profile on real hardware, at real speed, with real results. A 64MB buffer of trace data can be captured in just a few seconds. The same amount of data would take days to capture on a simulator. And on the simulator, we need to rely on simulation of peripherals and other real-world elements that will affect our application running on hardware. With hardware trace, we are capturing the real execution of the application, with real peripheral effects, without having to modify an application to run on the simulator.

- Function Call Context

With the standard TI profiler, we see how many times a function is called, and the average number of cycles the function takes. But this is all of the information that we get. In some cases, we might like to know that *function A* takes an average of 1000 cycles when called from *function B*, but takes an average of 5000 cycles when called from *function C'. The gProf like format shows additional detail that gives more information about the call context of each function.*

Limitations of Dynamic Call Graph

- Application rebuild required

In order to be able to capture call graph data, we must insert triggering instructions a the beginning and end of every function. Version 6.1 of the TI C6000 compiler supports the ability to do this. But you will be requured to add these ooks and then rebuild your application

- Additional Code Footprint

Because we're adding these hooks to the code, we're going to consume a bit of extra cycle footprint. For any function, the entry and exit hooks are 2 words each. Keep in mind though, that some functions have multiple exit points. The foorptint for any given function is 2 Words for the entry point plus two words x the number of exits.

- Additional Cycles

These instructions will also consume a few additional cycles. For any function, expect the hook functions to consume 12 cycles in every function (6 for the entry and 6 for the exit). Also, assume a few additional cycles for the thread switch function to write the id of the next thread to a global variable. The 12 cycles will be included in the trace for each function. However, the extra cycles for writing the global thread id will not be captured because we typically don't add function hooks the thread hook function.

Dynamic Call Graph Theory

Dynamic Call Graph capture function execution cycles by inserting a *function hook* at the entry and exit of every function. the function hook essentially tells trace to store PC and timing whenever that particular instruction is executed. Beginning with Code Generation Tools 6.1.0, the compiler allows switches to inline small functions in the entry and exits of each function. There is also a pragma to denote functions that should not be hooked.

<syntaxhighlight lang="c"> void entryHook(){

```
asm (" nop 5");
_mark(0);
```

}

void exitHook(){

```
asm (" nop 5");
_mark(1);
```

} </syntaxhighlight>

The _mark() instructions are C intrinsics that implement the lower level assembly MARK instructions. A MARK instruction essentially acts as a NOP, except the Advanced Event Triggering logic senses an event when it is executed. The nop 5 instruction immediately before the mark instruction ensures that when the mark instruction is detected, the application will not be in a delay slot of a branch (avoiding a trace limitation). Trace is programmed such that when a MARK instruction is executed, the PC and Time Stamp are captured. By iterating through the trace output, we can recreate the complete cycle accurate call graph.

Detecting Thread Changes

If we are trying to profile a multi threaded application, we need to know when the threads get switched and the ID of the currently executing thread. In order to capture this data,w e must implement a Thread Switch Function that will write the ID of the next task to a global variable, and then capture all writes to this location with data trace. The thread switch function is a function that is called by the operating system when it is switching thread contexts. See the documentation on your specific operating system for information on how to implement a thread switch function.

Interpreting The Output

A standard dynamic call graph looks like the output below. Note, the output is for a single thread in a multi threaded application.

```
=====================================================================
=====================================================================
Thread: _TSK2
=====================================================================
=====================================================================
index   excl_cycles    called         name
        4181834        1727/3455          p2 [3]
        2110276        1728/3455          p1 [2]
[0]     6292110        3455           child [0]
        668            4/7                prdfxn0 [4]
---------------------------------------------------------------------
                                          <spontaneous>
[1]     1159064        1              task [1]
        43187          1728/1728          p1 [2]
        43175          1727/1727          p2 [3]
        503            3/7                prdfxn0 [4]
---------------------------------------------------------------------
        43187          1728/1728          task [1]
[2]     43187          1728           p1 [2]
        2110276        1728/3455          child [0]
---------------------------------------------------------------------
        43175          1727/1727          task [1]
[3]     43175          1727           p2 [3]
        4181834        1727/3455          child [0]
---------------------------------------------------------------------
        668            4/7                child [0]
        503            3/7                task [1]
[4]     1171           7              prdfxn0 [4]
--------------------------------------------------------------------
```

Output Details

There are 5 functions called within the thread _TSK2. The functions are *child*, *task*, *p1*, *p2*, and *prdfxn0*. Each of the sections above details the specifics of a single function, which will be referred to as the function of interest. The function of interest is denoted by the index in the index column, and an outdented name in the name column. So for the above example, *child* is the function of interest in the first section, *task* in the second section, and *p1* in the 3rd section and so on.

Each function of interest is detailed with its immediate parents and immediate children. So, again, in the above example, child is called by both p1, and p2. And the only thing called by child is prdfxn0. (Note: prdfxn0 is actually a periodic function called by an interrupt. When it occurs between the entry and exit of child, it will appear as a function called by child.)

Focusing on the child function, we see that it is called a total of 3455 times. 1727 of those times it was called by p2. 1728 of those times it was called by p3. We also see that the total exclusive cycles for the child function is 6292110. Of these cycles, 4181834 were generated in executions of child called by p2. 2110276 were generated in executions of child called by p1. We also see that prdfxn0 was *called* a total of 7 times, but only 4 times by child. In those 4 times it was called by child, it consumed a total of 668 cycles.

All of the data is contained within this table to reconstruct the entire call graph.

Trace Script Download

The script package can be downloaded at the following location https://www-a.ti.com/downloads/sds_support/applications_packages/trace_csv_scripts/index.htm

Note that in order to use the stand alone trace decoder, you must have version 3.0.0 of the scripting package or later.

Trace Script Frequently Asked Questions

Generic Scripting FAQS

- **Q: Why do I sometimes see "UNKNOWN", "UNKNOWN" in the output for functions/filenames**
  - **A: The function/filename symbols are determined from the output of the ofd6x.exe (Object File Dump utility), which generates a list of functions and filenames from the .out file, along with their staring and ending addresses. If "UNKNOWN" values are showing in the output, it's because trace captured program execution that had a program address outside the ranges specified by the OFD utility. Common causes might be code that has been dynamically loaded/allocated which wouldn't have associated information in the .out file. You can usually determine the exact cause by looking in the file generated by ofd6x.exe and the .map file and determining why the offending program address is not defined.**

Dynamic Call Graph Specific FAQS

- **Q: How do interrupts affect the call graph?**
  - **A: As noted above, when an interrupt service routine gets called, it will show up as the child of whichever function it was executing when the ISR was called. In the example above, prdfxn0 is called by an interrupt. It shows up as a child of task and child, meaning it was taken whenever it was executing those functions.**

{{ 1. switchcategory:MultiCore=	Keystone= ▪ For technical support on	C2000=*For technical support on the C2000*	DaVinci=*For technical support on DaVincoplease*	MSP430=*For technical support on MSP430*	OMAP35x=*For technical support on OMAP please*	OMAPL1=*For technical support on OMAP please*	MAVRK=*For technical support on MAVRK*	*For technical s please post yo questions at http://e2e.ti.cor*

- 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 **Dynamic Call Graph** here.

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 **Dynamic Call Graph** here.

*please post your questions on The C2000 Forum. Please post only comments about the article **Dynamic Call Graph** here.*

*post your questions on The DaVinci Forum. Please post only comments about the article **Dynamic Call Graph** here.*

*please post your questions on The MSP430 Forum. Please post only comments about the article **Dynamic Call Graph** here.*

*post your questions on The OMAP Forum. Please post only comments about the article **Dynamic Call Graph** here.*

*post your questions on The OMAP Forum. Please post only comments about the article **Dynamic Call Graph** here.*

*please post your questions on The MAVRK Toolbox Forum. Please post only comments about the article **Dynamic Call Graph** here.*

*Please post on comments abo article **Dynami Graph** here.*

}}

Links

Amplifiers & Linear
Audio
Broadband RF/IF & Digital Radio
Clocks & Timers
Data Converters

DLP & MEMS
High-Reliability
Interface
Logic
Power Management

Processors

- ARM Processors
- Digital Signal Processors (DSP)
- Microcontrollers (MCU)
- OMAP Applications Processors

Switches & Multiplexers
Temperature Sensors & Control ICs
Wireless Connectivity

{{#switchcategory:MSP430=<McuHitboxFooter/>|C2000=<McuHitboxFooter/>|Stellaris=<McuHitboxFooter/>|TMS570=<McuHitboxFooter/>|MCU=<McuHitboxFooter/>|MAVRK=<MAVRKHitboxFooter/>|<HitboxFooter/>}}

Retrieved from "https://processors.wiki.ti.com/index.php?title=Dynamic_Call_Graph&oldid=181987"

This page was last edited on 21 July 2014, at 06:41.