Runtime CPU Spike Detection using Manual and Compiler-Automated Instrumentation

Adisak Pochanayon
Principal Software Engineer
Netherrealm Studios
adisak@wbgames.com

Topics to Cover

- This talk is about runtime instrumentation-based profiling.
- Code Instrumentation Methods
- MK’s PET Profiler (Spike Detector)

Profilers

- Common types of profilers
  - Hardware Trace
  - Hardware Event-based
  - Software Event-based
  - Sampling
  - Instrumented

Manual Instrumentation

- Explicit Instrumentation / Code Markup
- Wrapper Functions
- Detours and Trampolines

Explicit Instrumentation

- Requires code markup (source modification)
  - StartMarker(INFO) / StopMarker(INFO)
- Scoped - ScopedMarker(INFO)

```
class CScopedMarker {
   CScopedMarker(ProfDesc &info) { StartMarker(info); }
   ~CScopedMarker(ProfDesc &info) { StopMarker(info); }
};
#define ScopedMarker(INFO) CScopedMarker(INFO) \
        ProfInfo##__LINE__
```

Wrapper Functions

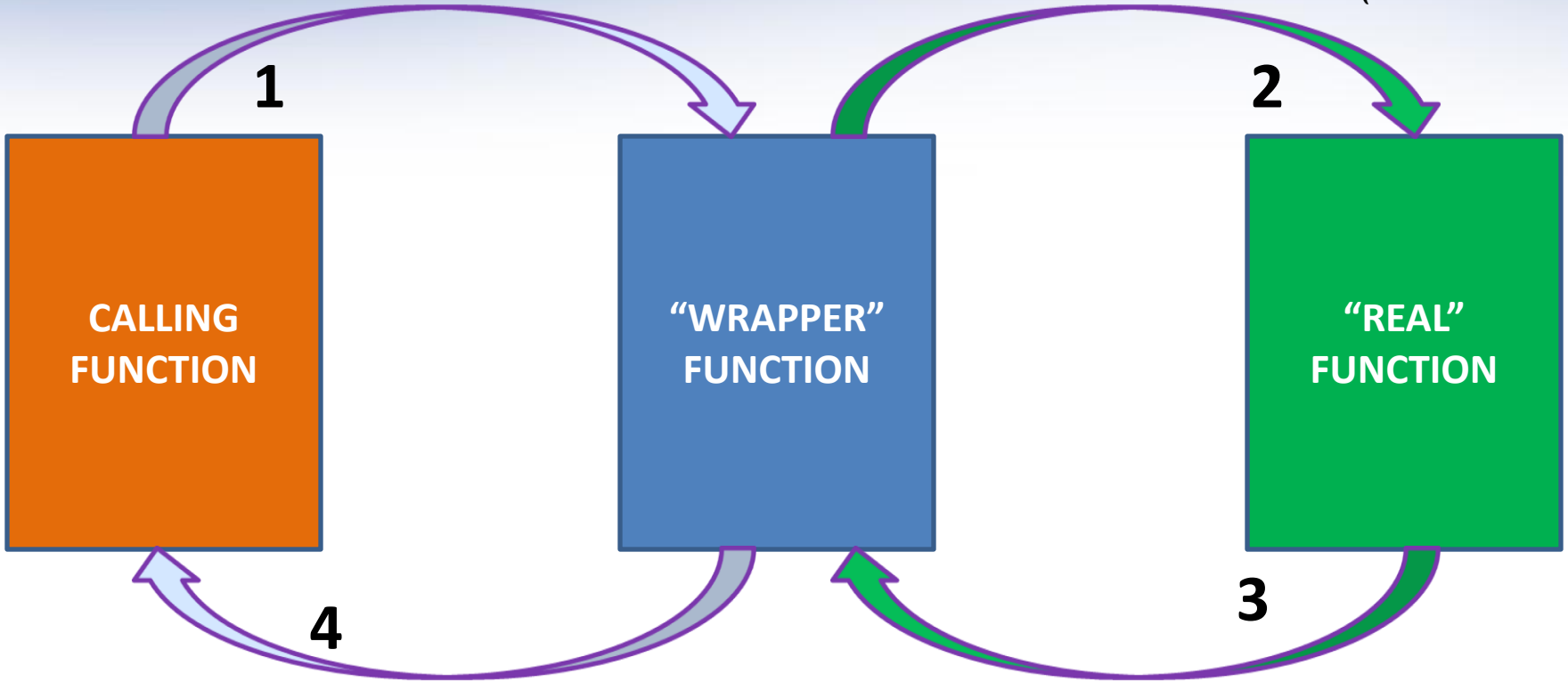
- Compile-time
  - #define function(…) wrapper_function
    - Plus – compiler independent
    - Drawback – only works if you have source code
- Link-Time Replacement / Wrapping
  - GCC option: -Wl,--wrap,function_name

    __wrap_function_name()

    __real_function_name()

Wrapper Functions

Wrapper Functions



Wrapper Functions

- Sample of wrapping malloc() with GCC / SNC
  - Add linker flags: -Wl,--wrap,malloc

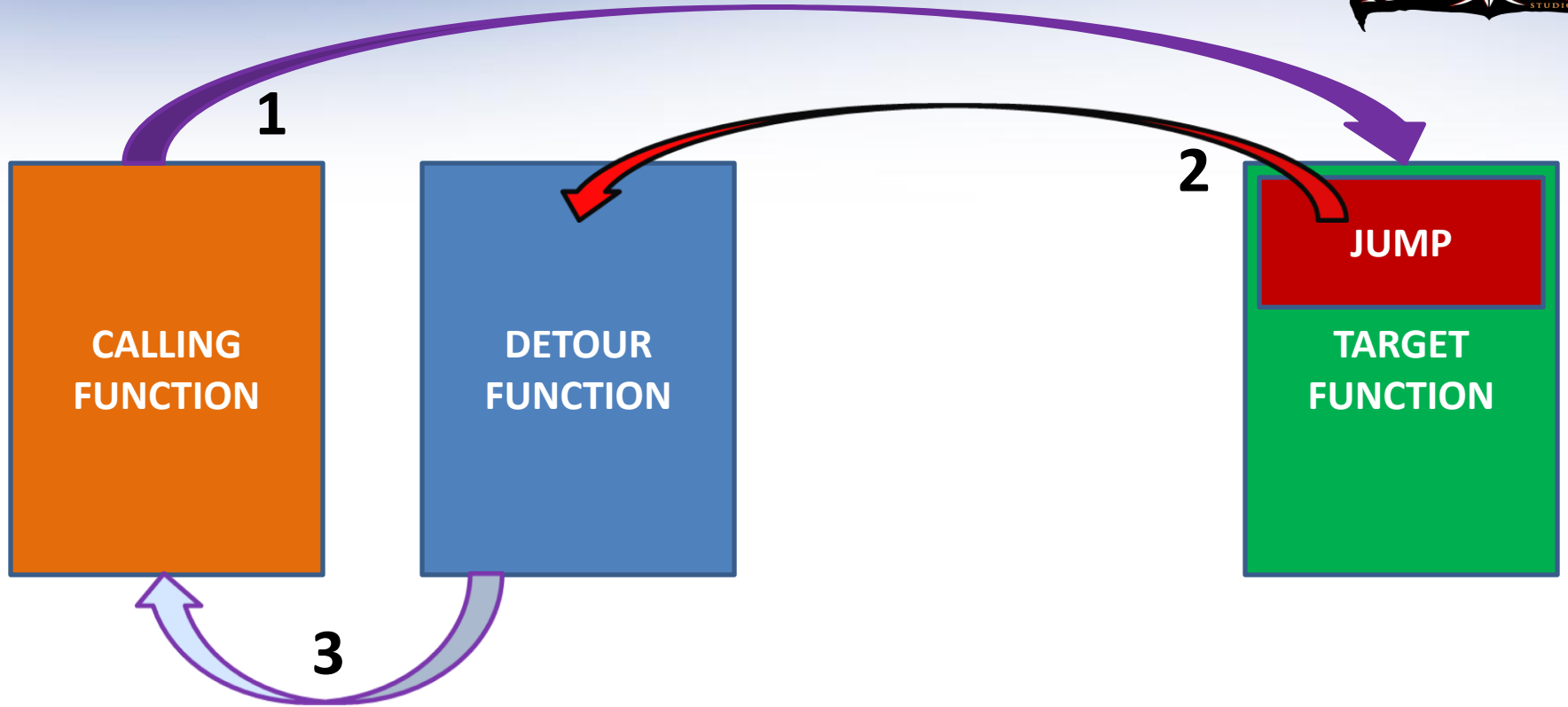
```
extern "C" void* __real_malloc(size_t);
extern "C" void* __wrap_malloc(size_t Size)
{
   // Call the original malloc() function
   return __real_malloc(Size);
}
```

Detours and Trampolines

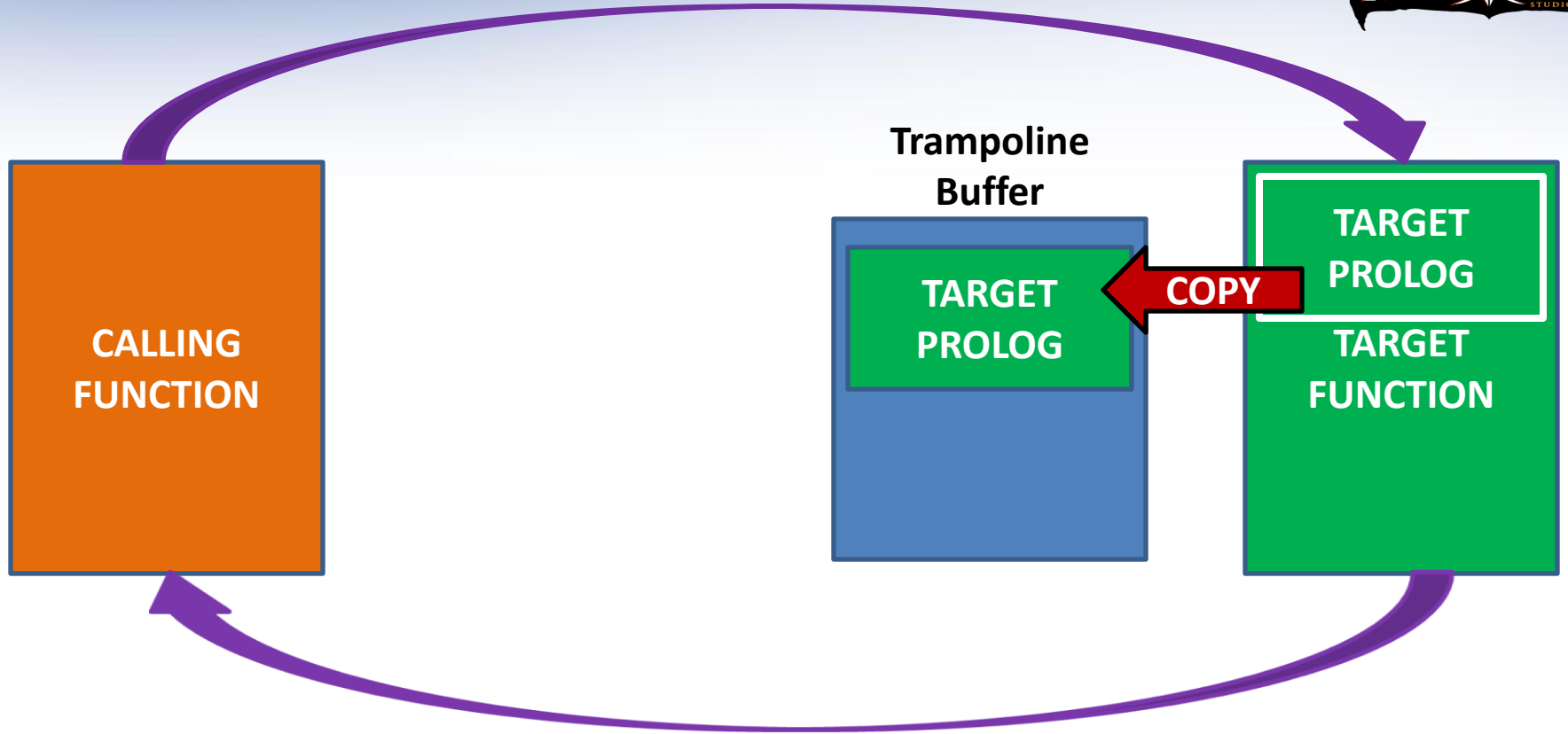
- A method of code modification for Instrumentation
  - Can be done on object code / binaries by Profilers
  - Run-time with instrumentation library calls
    - See Microsoft Detours
    - MIPS Example Code (Handout)
    - This is another form of manual instrumentation but does not require source markup of the target function.

Detours and Trampolines

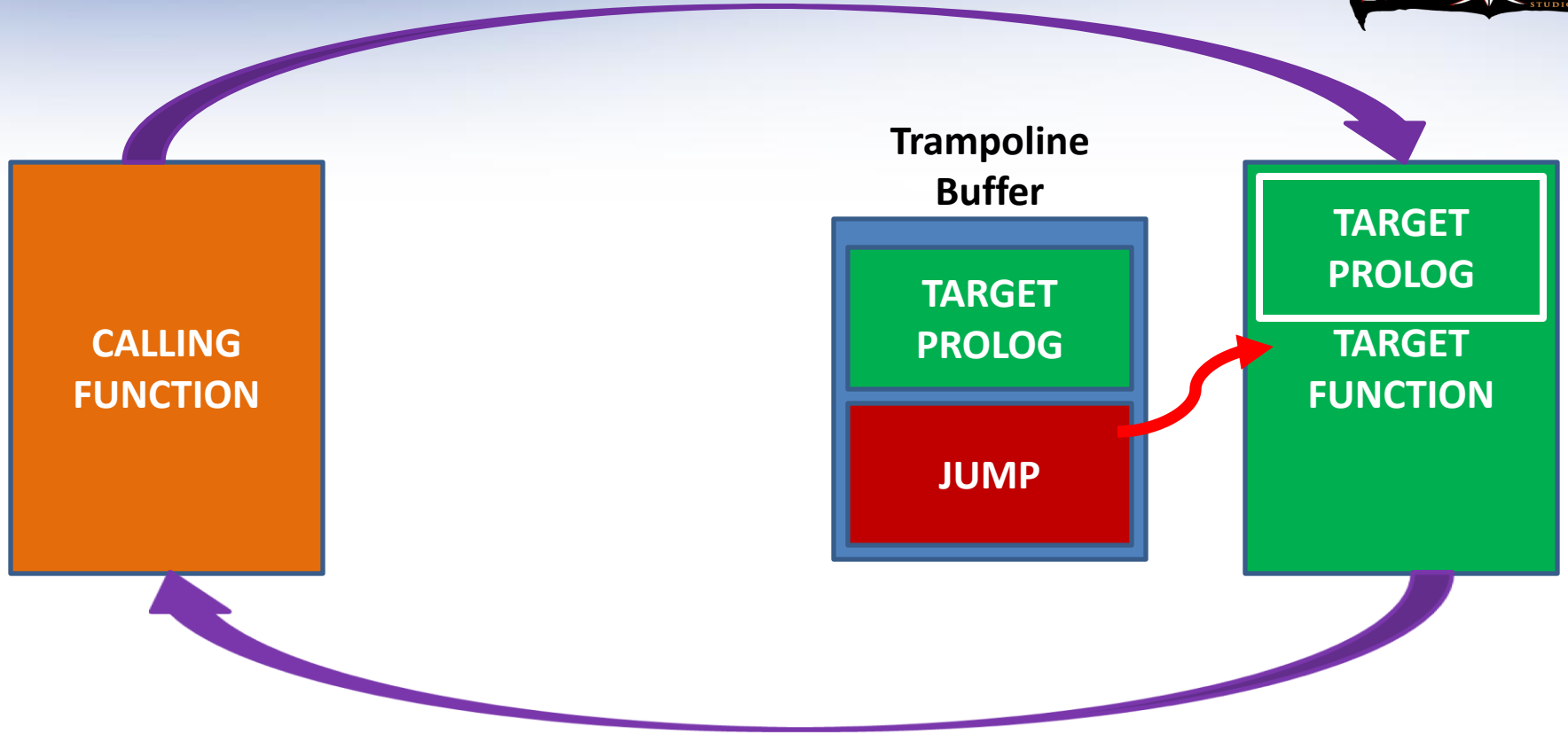
Detours


1
2
3
CALLING FUNCTION
DETOUR FUNCTION
JUMP
TARGET FUNCTION

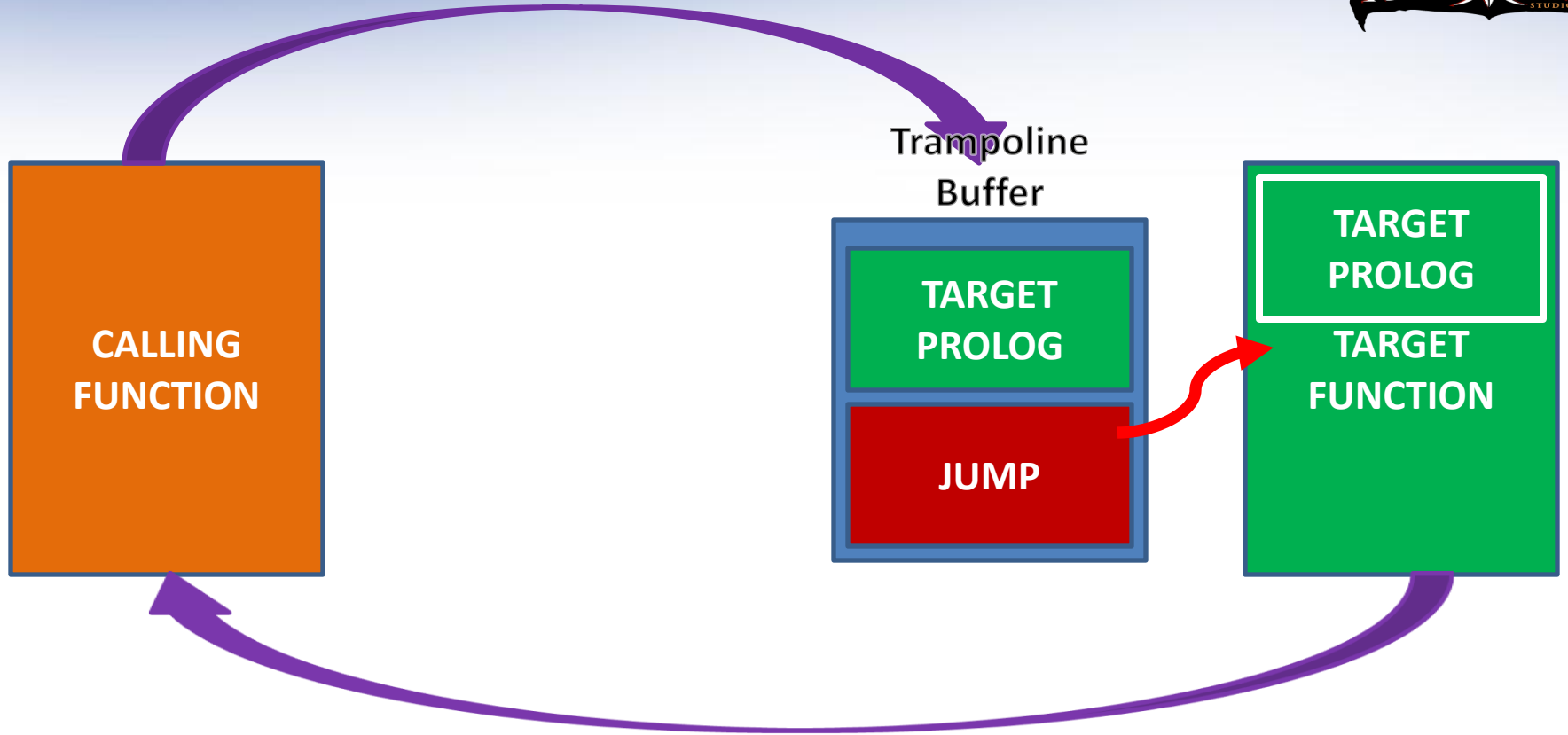
Trampolines



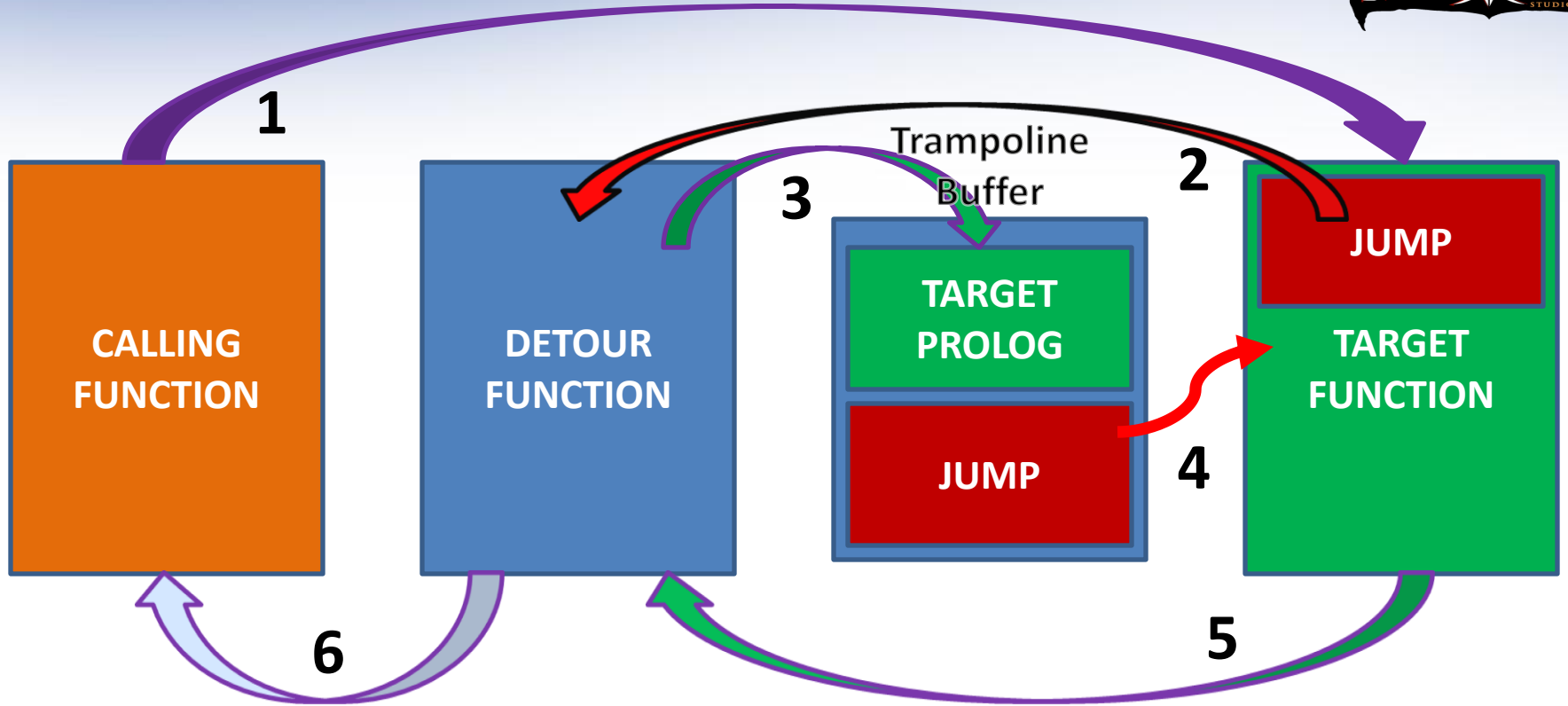
Trampolines



Trampolines



Detours and Trampolines



Detours and Trampolines

- Summary (Drawbacks)
  - Roll your own: Trivial on RISC / harder on CISC
  - Dealing with Page Protection / NX (No Execute)
  - Commercial implementations are $$$
    - Microsoft Detours Software
      - http://research.microsoft.com/en-us/projects/detours/
    - Microsoft 1999 Paper on Detours and Trampolines:
      - http://research.microsoft.com/pubs/68568/huntusenixnt99.pdf

Manual Instrumentation

- Summary of Manual Inst. methods
  - Explicit Markup
  - Wrapper Functions
  - Detours and Trampolines

- All require work identifying functions and user intervention (code markup, library calls, or linker parameters).

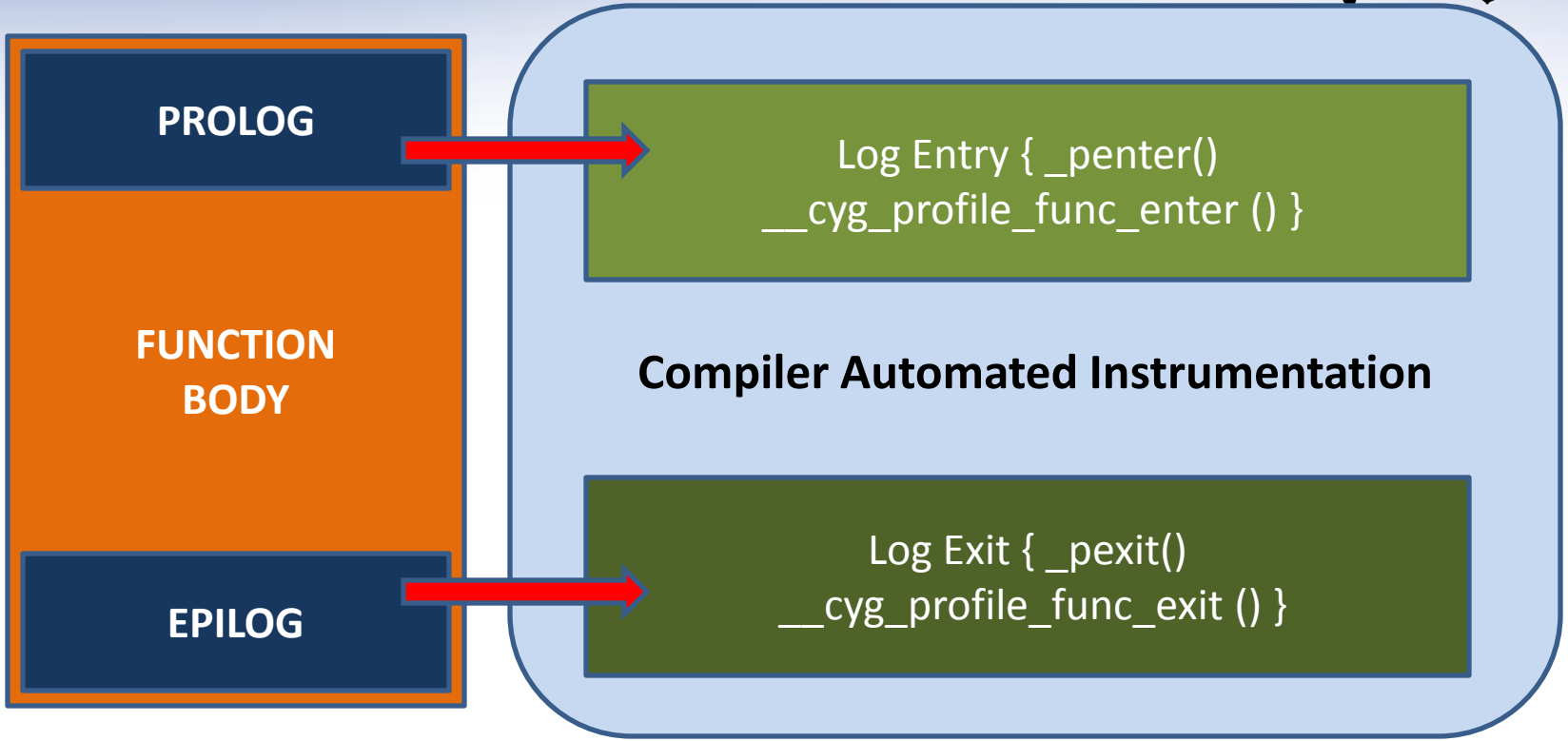
Automated Instrumentation

- Chances are you are already using it
  - Many profilers support it
    - Metrowerks CATS, VTune Call Graph, Visual Studio Profiler & Visual C++ /callcap and /fastcap, GNU gprof (w/ gcc –pg)
- Compiler-Assisted Instrumentation (CAI)
  - Allow for User Implementation of profiler but compiler does markup for you
    - GCC: -finstrument-functions / SNC -Xhooktrace
    - Visual C++: _penter() & _pexit() using /Gh and /GH

Automated Instrumentation

When a compiler generates machine code for a function, it generates a prolog (save registers, stack frame, etc.) and epilog (restore saved registers and states prior to return) in addition to the function body.

Automated Instrumentation



GCC & SNC CAI

- Compiler Option: -finstrument-functions
  - Generate instrumentation calls for entry and exit to functions. Just after function entry and just before function exit, the following profiling functions will be called with the address of the current function and its call site.

```
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit  (void *this_fn,void *call_site);
```

SNC CAI (PS3 / VITA)


```
void __cyg_profile_func_enter(void *this_fn, void *call_site)
{
    if(0==tls_PET_bIsInProcessing)
    {
        tls_PET_bIsInProcessing=true;
        _internal_PET_LogEntry(0);
        tls_PET_bIsInProcessing=false;
    }
}
```

Visual C++ CAI

- Using _penter() & _pexit() on Visual C++ is a bit more difficult.
  - Visual C++ inserts calls on function entry and exit but does not save any registers
  - At a very minimum, requires writing assembler to save the registers according to the platform ABI
  - Requires additional checks to be “useable”

Visual C++ CAI – X86


```
extern "C" void __declspec(naked) _cdecl
    _penter( void )
{
    _asm
    {
        push eax
        push ebx
        push ecx
        push edx
        push ebp
        push edi
        push esi
    }

    if(0==tls_PET_bIsInProcessing)
    {
        tls_PET_bIsInProcessing=true;
        // Call C Work Function
        _internal_PET_LogEntry(0);
        tls_PET_bIsInProcessing=false;
    }

    _asm
    {
        pop esi
        pop edi
        pop ebp
        pop edx
        pop ecx
        pop ebx
        pop eax
        ret
    }
}
```

Visual C++ CAI – XBOX 360

- CAI Supported on XBOX 360 (PowerPC)
- Almost same as PC

1)Save Registers

2)NEW STEP - Check for DPC (deferred procedure call)

3)TLS – re-entrant check

4)Call actual work function

5)Restore Registers

Visual C++ CAI – XBOX 360

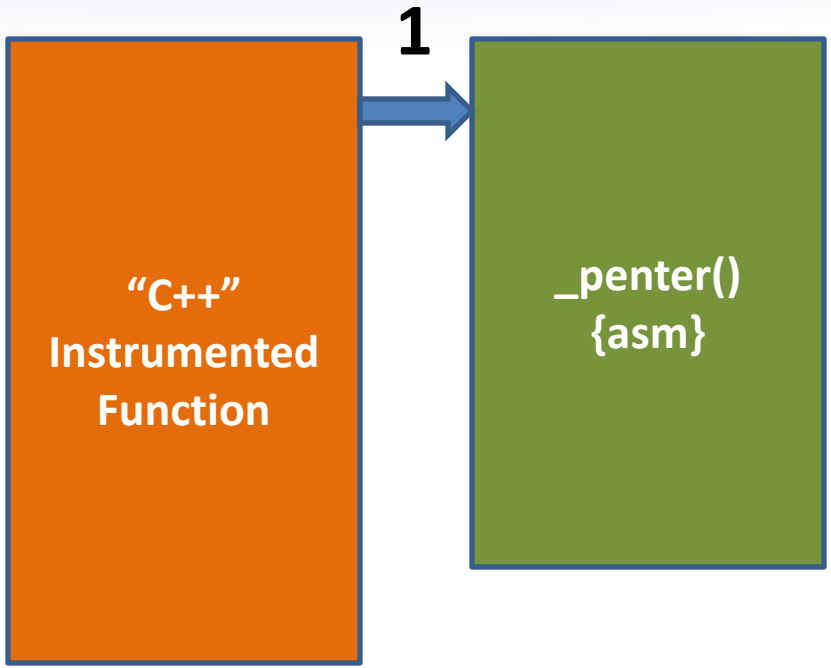
- PowerPC version is more complicated
  - More ABI Registers to Save and Restore
    - Have to save and restore FP regs if doing FP
  - Optimizations and Early Outs
  - TLS access must be done in ASM
    - Mixed naked asm / C does not work well like X86
  - See handout to follow along…

Visual C++ CAI – XBOX 360

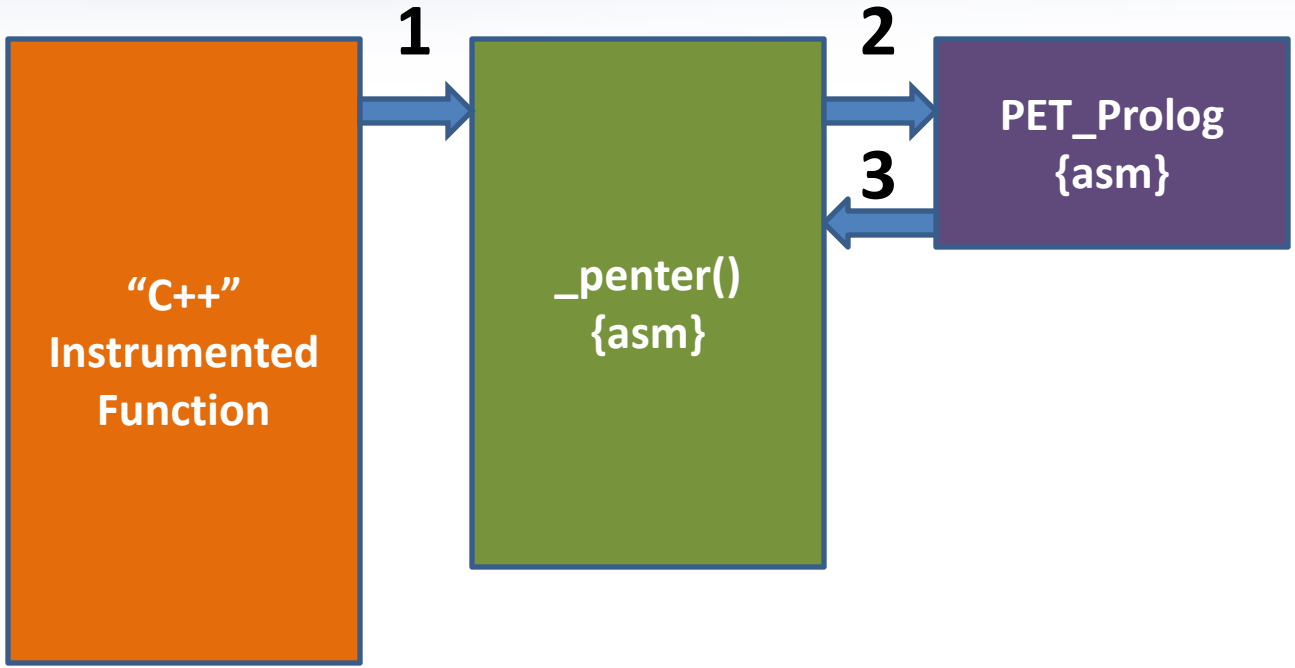

```
void __declspec(naked) _cdecl __penter( void )
{
    __asm
    {
        // Tiny Prolog
        //  - Set link register (r12) & return address (two steps)
        std             r12,-20h(r1)        // Saving LR here is extra step !
        mflr            r12
        stw             r12,-8h(r1)         // Return Address

        bl              PET_prolog
        bl              _internal_PET_LogEntry
        b               PET_epilog
    }
}
```

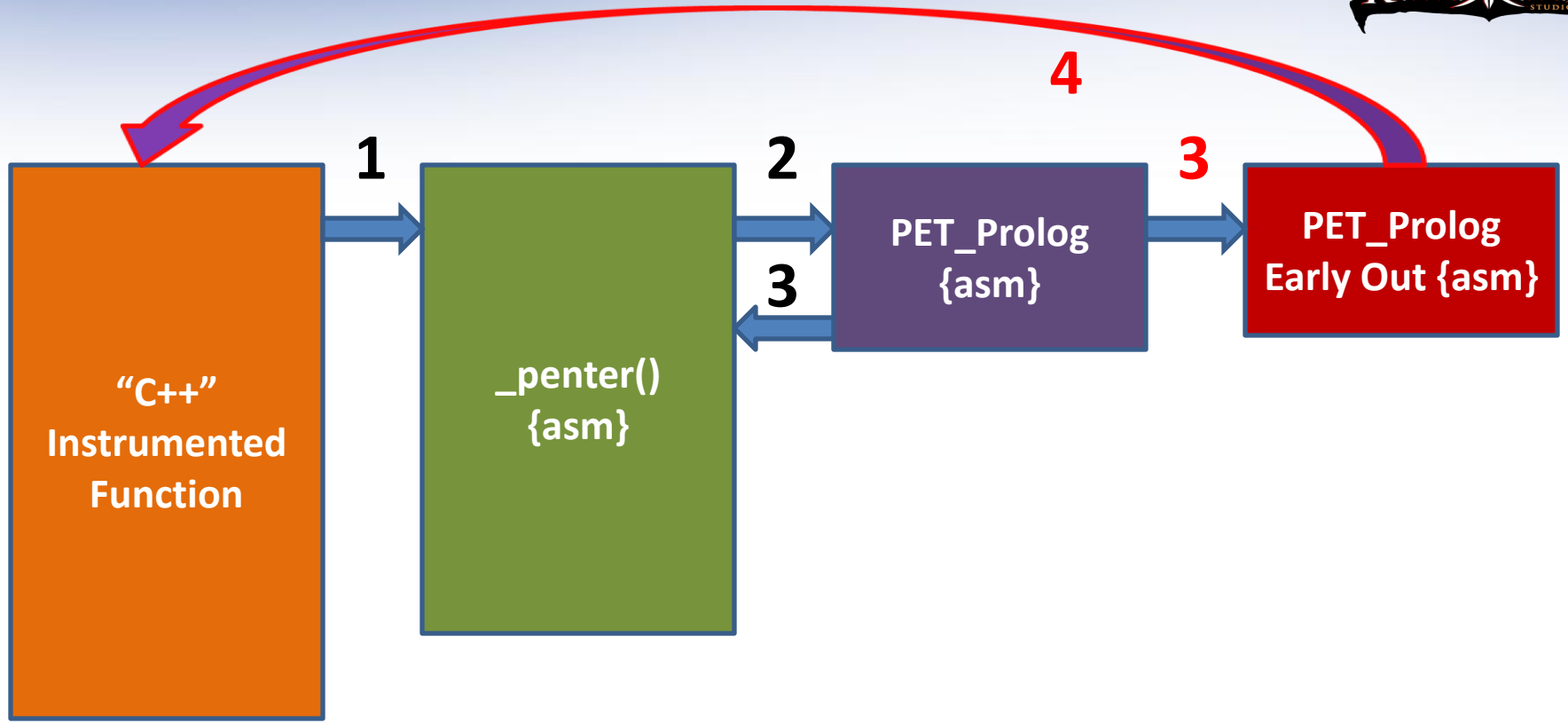
XBOX 360 CAI Flow: _penter()



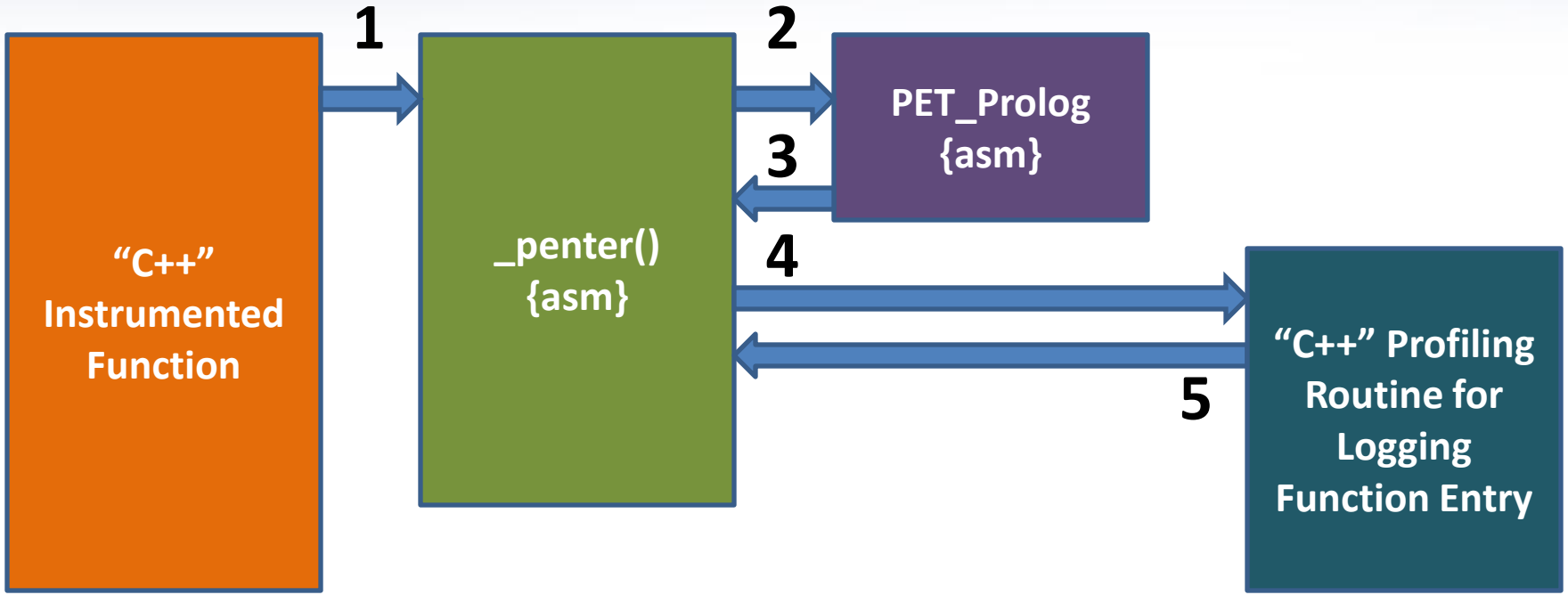
XBOX 360 CAI Flow: _penter()



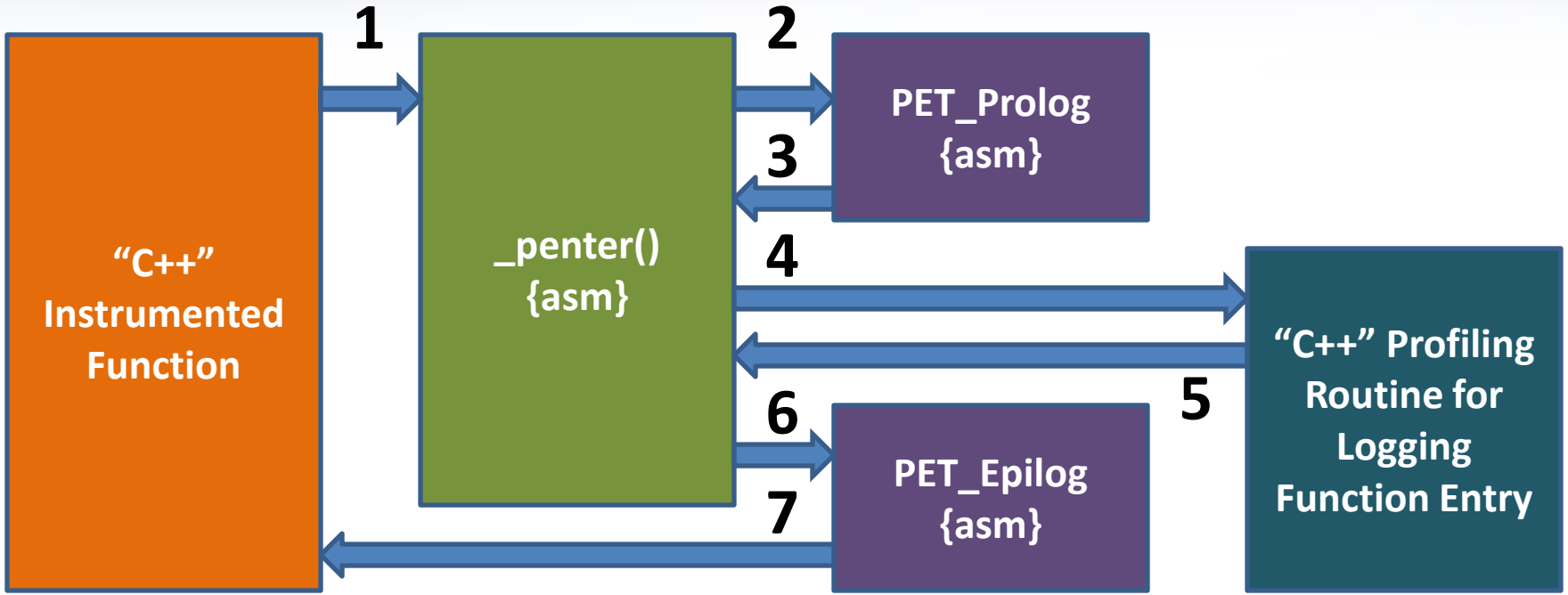
XBOX 360 CAI Flow: _penter()



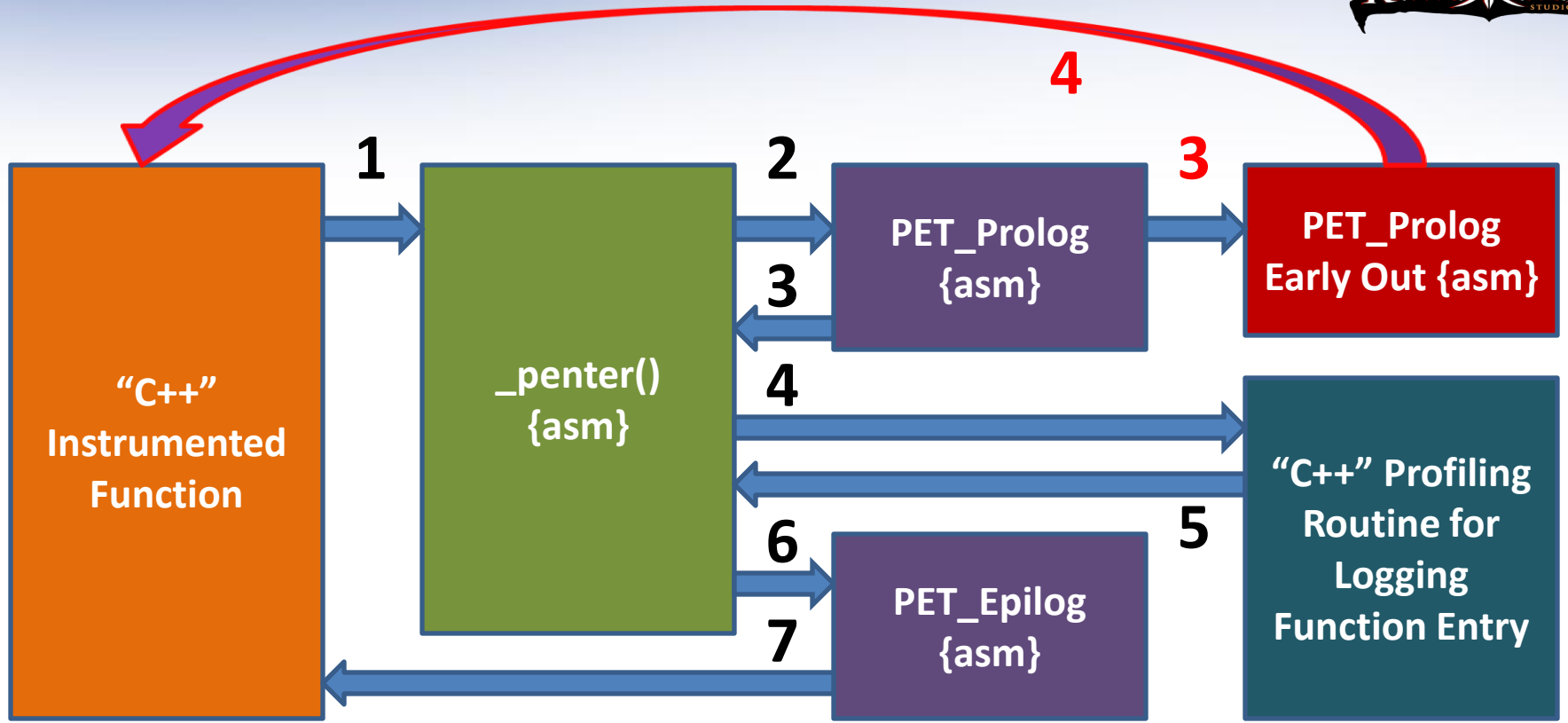
XBOX 360 CAI Flow: _penter()



XBOX 360 CAI Flow: _penter()



XBOX 360 CAI Flow: _penter()



Visual C++ CAI – XBOX 360

- PET_Prolog has five sections
  - Tiny Prolog to save minimal registers
  - Check for DPC & possible early out
  - Check for recursion (TLS var) & possible early out
  - Save temporaries (including r2) and return to parent
  - Early out returns all the way to grandparent function

Visual C++ CAI – XBOX 360

- Tiny Prolog to save minimal registers

```
// Tiny Prolog
//  - Save extra registers (r11,r14)
//  - Set stack frame (r1)
std             r11,-30h(r1)
std             r14,-28h(r1)

// Old Stack Pointer (r1) is at 0(r1) after this instruction
stwu            r1,-100h(r1)
```

Visual C++ CAI – XBOX 360

- Check for DPC & possible early out

```
// Get the TLS thread-specific base
lwz          r11,0(r13)
// Do not try to run in DPC!
// In DPC { 0(r13) == 0 }
cmplwi       cr6,r11,0
beq          cr6,label__early_exit_prolog
```

Visual C++ CAI – XBOX 360

- Check for recursion (TLS var) & possible early out

```
lau         r14,_tls_start          // Get the TLS global base
lau         r12,tls_PET_bIsInProcessing
lal         r14,r14,_tls_start
lal         r12,r12,tls_PET_bIsInProcessing
sub         r11,r11,r14             // TLS Base Offset (r11)
add         r14,r11,r12             // r14 == &tls_PET_bIsInProcessing
// Avoid recursion using thread variable tls_PET_bIsInProcessing
lwzx                    r12,r11,r12
cmplwi                  cr6,r12,0
bne                     cr6,label__early_exit_prolog
li                      r12,1
stw                     r12,0(r14)          // Set tls_PET_bIsInProcessing
```

Visual C++ CAI – XBOX 360

- Check for recursion (TLS var) & possible early out

That may have looked complicated but it's actually pretty simple. Here is the C++ equivalent for the last slide:

```
if(tls_PET_bIsInProcessing)
        goto label__early_exit_prolog;
tls_PET_bIsInProcessing=true;
```

Visual C++ CAI – XBOX 360

- Save temporaries (including r2) and return to parent

```
// Save r0/r2-r10 (temporaries)
std         r0,8h(r1)
std         r2,10h(r1)          // (r2 is reserved on XBOX 360)
std         r3,18h(r1)
std         r4,20h(r1)
std         r5,28h(r1)
std         r6,30h(r1)
std         r7,38h(r1)
std         r8,40h(r1)
std         r9,48h(r1)
std         r10,50h(r1)
blr         // Return To Caller
```

Visual C++ CAI – XBOX 360

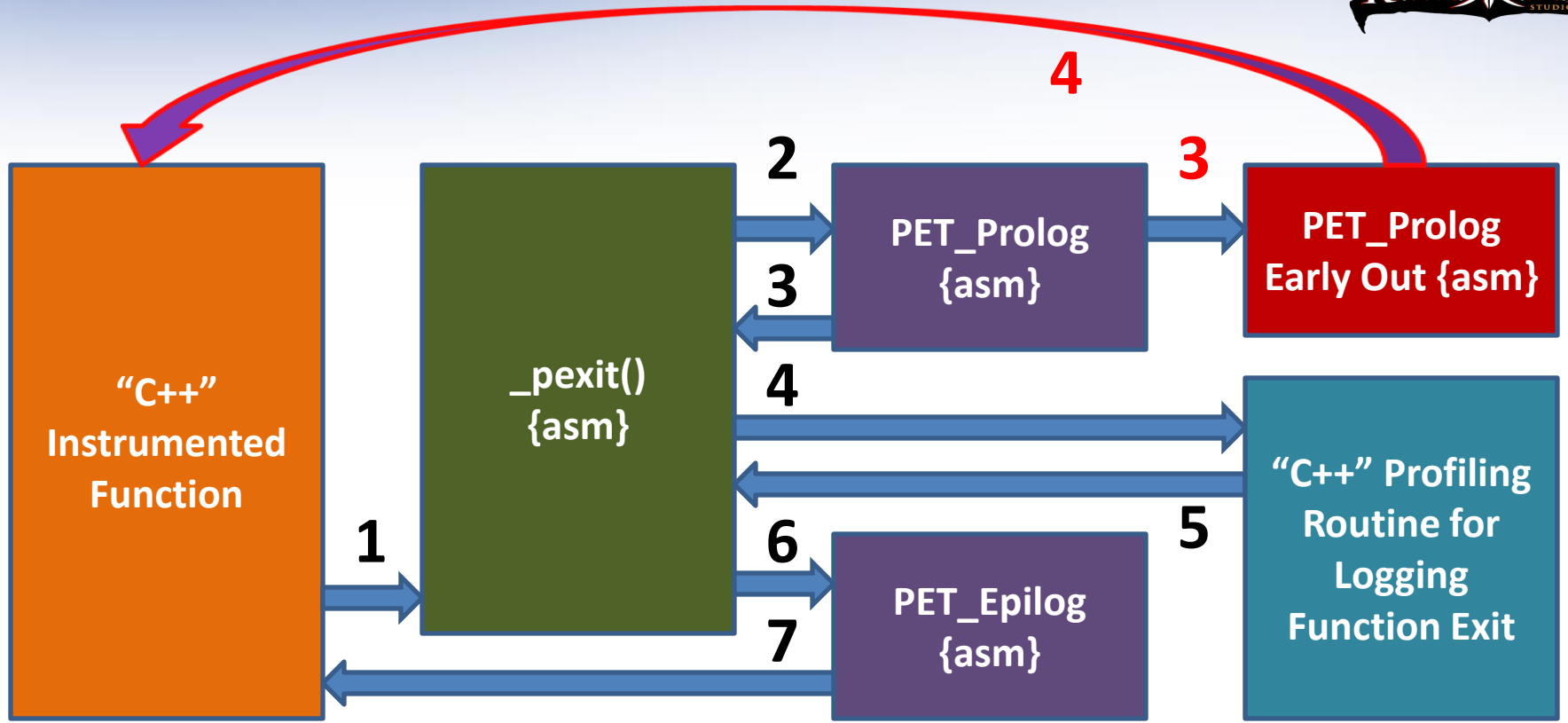
- Early out returns all the way to grandparent function

```
label__early_exit_prolog:
        // Tiny Epilog -- adjust stack (r1) & restore r12/r14/r11
        addi        r1,r1,100h
        lwz         r12,-8h(r1)
        mtlr        r12
        ld          r12,-20h(r1)
        ld          r14,-28h(r1)
        ld          r11,-30h(r1)
        blr
```

Visual C++ CAI – XBOX 360

- PET_Epilog is simpler (see handout)
  - Clear TLS recursion prevention variable
  - Restore temporaries
  - Restore registers used in “Tiny Prolog”
- Final note: Worker function must save / restore FP registers (fr0-fr13) if performing any floating point work.

XBOX 360 CAI Flow: _pexit()



How to use Instrumentation?

- So now that we have all these methods to instrument our code what do we do with these techniques?
- Hook it up to your profiling code ☺
- On the MK team, one of the uses is for an internally developed Runtime Spike Detector called the PET Profiler.

PET Profiler

- PET = Prolog Epilog Timing
  - Times Entry and Exit into functions and detects spikes
  - Can set a global threshold and any instrumented function that exceeds that threshold gets logged
    - With a stack and potentially extra info from markup
  - Works with the Compiler-Automated Instrumentation
    - Spike detection for every compiled function in game
    - Cost is about 15-30% performance overhead with CAI

PET Profiler

- NO CODE MARKUP required to detect spikes
  - Turn CAI on, and it will find any function that take longer to execute than the global threshold you set
- PET Tags / Code Markup is still useful
  - Simple PET_FUNCTION() (scoped marker)
  - Provides extra information to PET Profiler
  - Allows spike detection on platforms without CAI
  - Alternate Implementions = Enhances other profilers

PET Implementaion

- PET is implemented using TLS info stacks
  - Stack is incremented by CAI entry
  - Stack is decremented by CAI exit
  - When CAI is not present, PET Stack is inc'd and dec'd by markup (the scoped markers)
  - Scope Markers and API Markup provide additional information and functionality at a global level, function level, sub-system level, or thread level

PET Implementaion

- DATA in stack
  - Can be as small as 4 words depending on #define options selected
    1. time entered
    2. optional threshhold (to override global threshhold)
    3. optional threshhold for children
    4. description (most of these are optional)
       - function address
       - function name (pointer to static)
       - line number
       - source file name (pointer to static)
       - user generated description (dynamic string)

PET API / Basic Markup

PET_FUNCTION()

PET_SECTION_MANUAL(name)

PET_SetGlobalThresholdMSec(msecs)

PET_SetFunctionThresholdMSec(msecs)

PET_SetChildrenThresholdMSec(msecs)

PET_SetAllParentsThresholdMSec(msecs)

PET API / Thresholds

Why threshold functions? Lets say you set the global threshold to 1ms to find any function that takes over 1ms. Your Game::Tick() function for a game at 60 fps will take 16.66ms so to avoid kicking a spike warning, you can insert this markup into Game::Tick() :

```
PET_FUNCTION();
PET_SetFunctionThresholdMSec(1000.0/MIN_FPS);
```

Then the PET Profiler will not log a spike in your Game::Tick().

PET Example Thresholds

Script Functions (0.1ms)

Global Threshhold for Most Game Functions (1ms)

Tick Functions (1 Frame = 16 ms)

File I/O Functions (2 s)

PET API / Messages

PET_SetFunctionDescf(FMT,...)

- Allocates a temporary (shared) string from a pool and formats it with a description.
- If a spike occurs, emits that description in the log.

PET_OTMessagef(FMT,...)

- Logs an additional message if the function is Over the time Threshold (OT).
- Less overhead than PET_SetFunctionDescf() because output string is only generate if needed for message.

PET API / Messages

- Example:

```
void ExecuteScript(ScriptContext *sc)
{
        PET_FUNCTION();
        sc->ExecuteStep();
        PET_OTMessagef("Script function: %s",
                sc->GetCurrentFunctionName());
}
```

PET API / Messages

- Example:

```
void ExecuteScript(ScriptContext *sc)
{
        PET_FUNCTION();
        PET_SetFunctionThresholdMSec(0.1);
        PET_SetFunctionDescf("Script function: %s",
                sc->GetCurrentFunctionName());
        sc->ExecuteStep();
}
```

PET API / Conditional Control

Control if Profiling is Active
- PET_SetThreadActive()
- PET_FUNCTION_IGNORE()
- PET_FUNCTION_IGNORE_CHILDREN()
- PET_Pause()
- PET_Unpause()

Conditional Profiling (can allow "channels")
- PET_FUNCTION_CONDITIONAL(cond)
- PET_FUNCTION_CONDITIONAL_PAUSED(cond)
- PET_FUNCTION_CONDITIONAL_IGNORE_CHILDREN(cond)
- PET_FUNCTION_CONDITIONAL_IGNORE(cond)

PET Sample Log Output


```
PET: Function took too long: 11.302 MSec
   Thread: MainThread
   PEAK CHILD @ frame 2323
  Function Name Trace
    0) 0x82449eb4 - main
    1) 0x824499c8 - GuardedMain
    2) 0x82444c80 - FEngineLoop::Tick
    3) 0x82440ddc - UMK9GameEngine::Tick
    4) 0x82d6d39c - MKScriptVM::Tick
    5) 0x82d6f2dc - MKListNoDestroy::ForEach
    6) 0x82d6cd54 - MKScriptVM::Step
    7) 0x82d690ac - _call_c_function
    8) 0x82440ddc - CreateEnduranceOpponentPhase1
    9) 0x82440ddc - CreatePlayerPhase1
   10) 0x82440ddc - SpawnMKFGCharacterObj
   11) 0x82440ddc – CreateMeshes

PET: Thread (MainThread) Function took too long: 14.499 MSec
   10) 0x82440ddc - SpawnMKFGCharacterObj
PET: Thread (MainThread) Function took too long: 14.599 MSec
    9) 0x82440ddc - CreatePlayerPhase1
PET: Thread (MainThread) Function took too long: 15.097 MSec
    8) 0x82440ddc - CreateEnduranceOpponentPhase1

!!!!!----------------------------------------------------------!!!!!
Slow Script->C Call: _CreateEnduranceOpponentPhase1
         takes 16.576 ms (0.995 frames) to execute
!!!!!----------------------------------------------------------!!!!!

PET: Thread (MainThread) Function took too long: 15.377 MSec
    7) 0x82d690ac - _call_c_function
PET: Thread (MainThread) Function took too long: 15.401 MSec
    6) 0x82d6cd54 - MKScriptVM::Step
PET: Thread (MainThread) Function took too long: 15.490 MSec
    5) 0x82d6f2dc - MKListNoDestroy::ForEach
PET: Thread (MainThread) Function took too long: 15.578 MSec
    4) 0x82d6d39c - MKScriptVM::Tick
```

PET Sample Log Output


```
PET: Function took too long: 8.586 MSec
   Thread: MainThread
   PEAK CHILD @ frame 2422
  Function Name Trace
    0) Tick (LaunchEngineLoop.cpp - LINE: 1981)
    1) Tick (MK9Game.cpp - LINE: 499)
    2) Tick (ScriptCore.cpp - LINE: 908)
    3) Step (ScriptCore.cpp - LINE: 440)
    4) _call_c_function (ScriptCore.cpp - LINE: 4194)
    5) CreateEnduranceOpponentPhase1 (FGPlayer.cpp - LINE: 881)
    6) CreatePlayerPhase1 (FGPlayer.cpp - LINE: 527)
    7) SpawnMKFGCharacterObj (FGPlayer.cpp - LINE: 243)
    8) CreateMeshes (FGPlayer.cpp - LINE: 1088)
    9) SetSkeletalMesh (UnSkeletalComponent.cpp - LINE: 3465)

PET: Thread (MainThread) Function took too long: 12.851 MSec
    8) CreateMeshes (FGPlayer.cpp - LINE: 1088)
PET: Thread (MainThread) Function took too long: 15.429 MSec
    7) SpawnMKFGCharacterObj (FGPlayer.cpp - LINE: 243)
PET: Thread (MainThread) Function took too long: 15.522 MSec
    6) CreatePlayerPhase1 (FGPlayer.cpp - LINE: 527)
PET: Thread (MainThread) Function took too long: 15.869 MSec
    5) CreateEnduranceOpponentPhase1 (FGPlayer.cpp - LINE: 881)

!!!!!----------------------------------------------------------!!!!!
Slow Script->C Call: _CreateEnduranceOpponentPhase1
        takes 16.446 ms (0.987 frames) to execute
!!!!!----------------------------------------------------------!!!!!

PET: Thread (MainThread) Function took too long: 16.015 MSec
    4) _call_c_function (ScriptCore.cpp - LINE: 4194)
PET: Thread (MainThread) Function took too long: 16.035 MSec
    3) Step (ScriptCore.cpp - LINE: 440)
PET: Thread (MainThread) Function took too long: 16.200 MSec
    2) Tick (ScriptCore.cpp - LINE: 908)
```

PET Profiler Notes

- API is optional (CAI works without API)
- API compiles out to “nops” in Release Builds
  - Zero runtime code overhead when not used
- Manual Instrumentation / Alternate Implementation Modes are very low overhead
  - Depends on amount of instrumentation but typically almost unnoticeable (vs 15-30% for CAI)

PET Profiler Notes

- Markup is not necessary with PET but can give you finer grain control… especially over specific systems.
- Typical game markup involves adding only 30-50 markup lines total for CAI to get best results.
  - PET Profiler will detect spikes and “suggest” lines for you to markup in CAI mode
- More Markup gives better detail in Manual Mode

Real World Examples

- Code Optimization Pass on MK9
  - Just turn on in Game and Log
    - Created a large list of spiking functions
    - Narrowed down to “real” and “false” spikes
    - Instrumented small number of “false” spikes with PET_Pause, PET_Ignore, or appropriate thresholds
    - Took list of “real” spikes and distributed to various programmers for optimization

Real World Examples

- Sometimes running found unusual and unexpected spikes.
  - Load routines were not previously heavily profiled compared to in-game steady 60 FPS state (MK has load screen or decoupled movie so underlying load is not 60FPS)
  - Example: Selecting characters for ladder could take nearly a second due to poorly written code !!!

Real World Examples

- Tracking down specific issues
  - Batman Arkham City has streamed loading. In a couple places, we were looking for stalls in loading. One of my coworkers spent a couple days looking for a stall without success. I went to his office and turned on the PET Profiler and it found the exact code stall in a few minutes on the first attempt.

Real World Examples

- …Tracking down specific issues
  - Fighting moves that cause “bog” on VITA
    - First tried Manual Instrumentation
      - Slow but useful results which were verified in RAZOR
      - Divide and Conquer / Instrument Children of Slow functions
    - Then Automated Instrumentation
      - Turn on CAI Mode / Execute slow move
      - A bunch of functions get logged (instrument if desired)
      - Much faster at locating issues (automatic)

Real World Examples

- Targeting script calls (*to move to native code?)
  - Unreal Script, MKScript, Kismet, etc
  - Instrument “script execute” function
    - Use PET_OTMessagef() or PET_SetFunctionDescf() for extra info
  - Why better than just timing script calls?
    - Automated Instrumentation modes will locate stalls in children functions AND give script context info

Questions ???

Contact Info:

Adisak Pochanayon
Principal Software Engineer
Netherrealm Studios
adisak@wbgames.com


NETHERREALM
STUDIOS

“Automagic” Profiling on MK

- Code Instrumentation
  - Manual
  - Automatic*
- Creating a profiler
- MK @ 60 FPS
  - Detecting spikes