

# Boosting Debug Productivity - Practical Applications of Verdi Debug Innovations

## Embedded Software Debug Tutorial

Alex Wakefield

March 2017



## **CONFIDENTIAL INFORMATION**

The following material is confidential information of Synopsys and is being disclosed to you pursuant to a non-disclosure agreement between you or your employer and Synopsys. The material being disclosed may only be used as permitted under such non-disclosure agreement.

## **IMPORTANT NOTICE**

In the event information in this presentation reflects Synopsys' future plans, such plans are as of the date of this presentation and are subject to change. Synopsys is not obligated to develop the software with the features and functionality discussed in these materials. In any event, Synopsys' products may be offered and purchased only pursuant to an authorized quote and purchase order or a mutually agreed upon written contract.

# Agenda

## **Synchronized HW SW Debug**

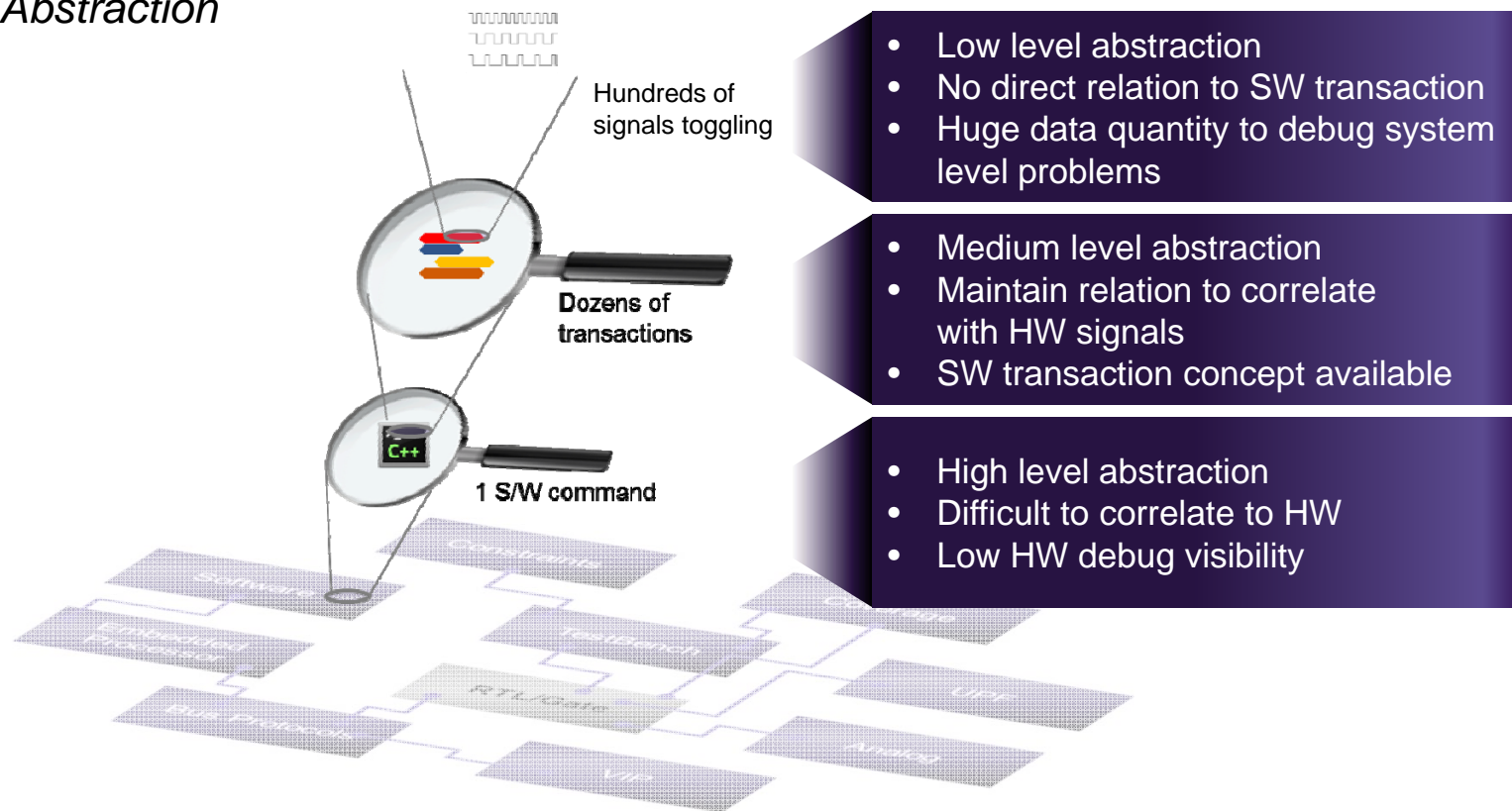
Practical Debug Application: Linux Boot

Performance Profiling of Embedded Software

Coverage Metrics for Embedded Software

# SoC Debug Challenges

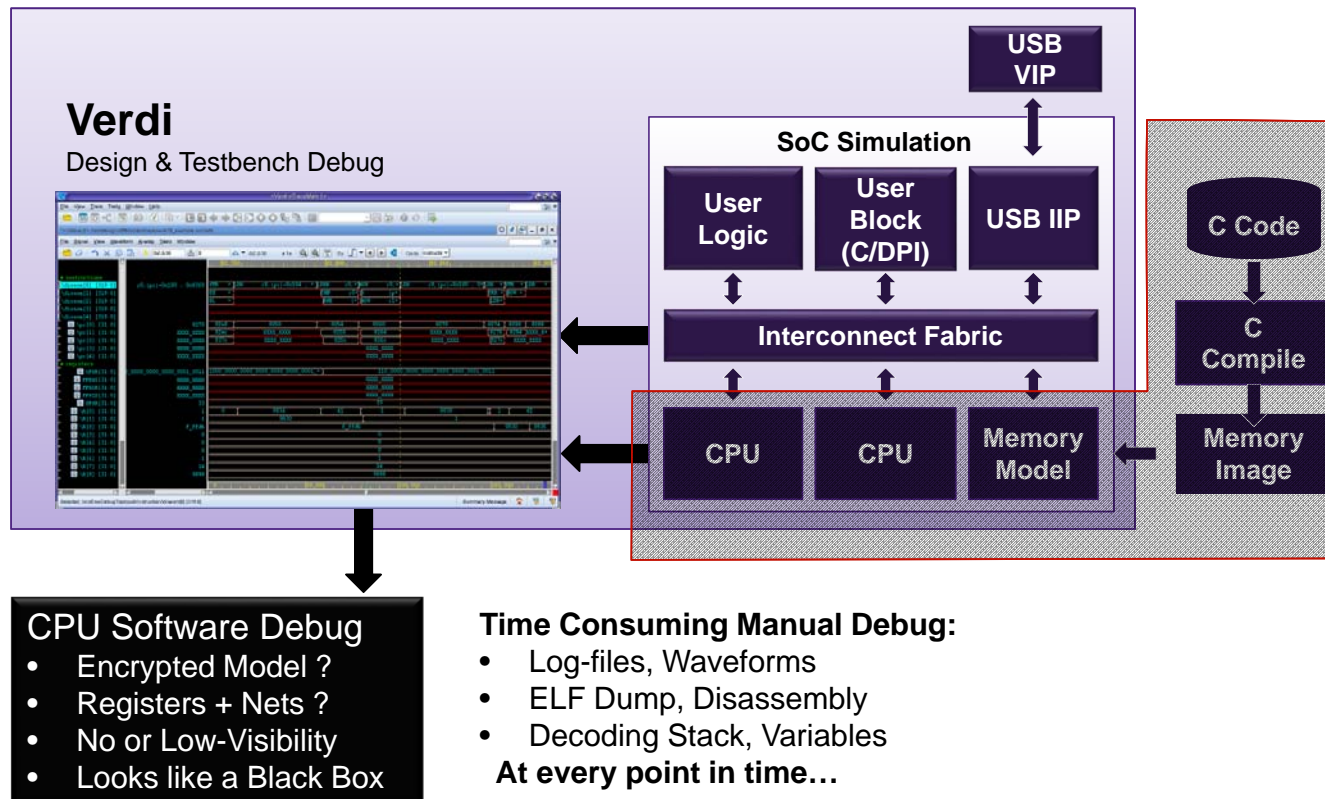
## Levels of Abstraction



**Correlated debug across abstraction levels is required**

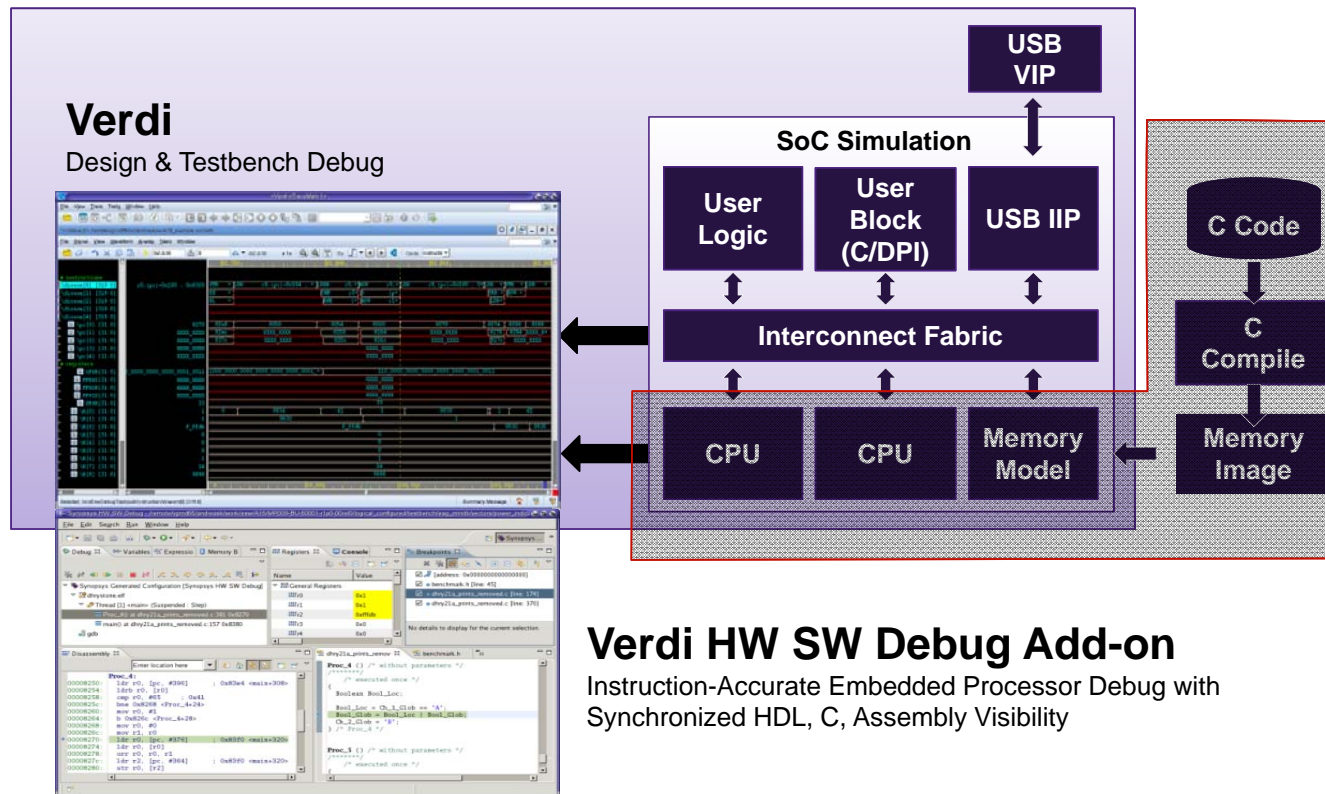
# SoC Debug Challenges

*Lack of Visibility on the Software Side*



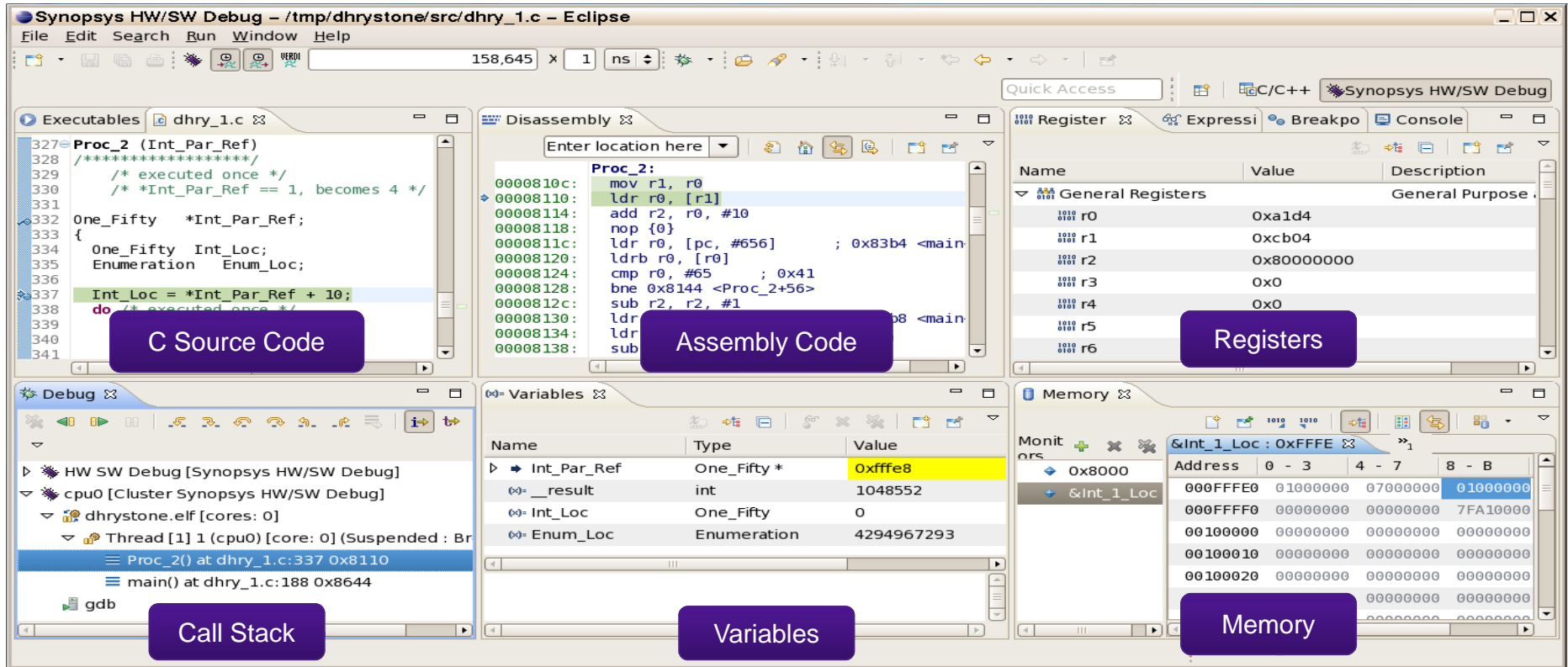
# SoC Debug Challenges

*Lack of Visibility on the Software Side*





# Embedded Software Debug in Eclipse



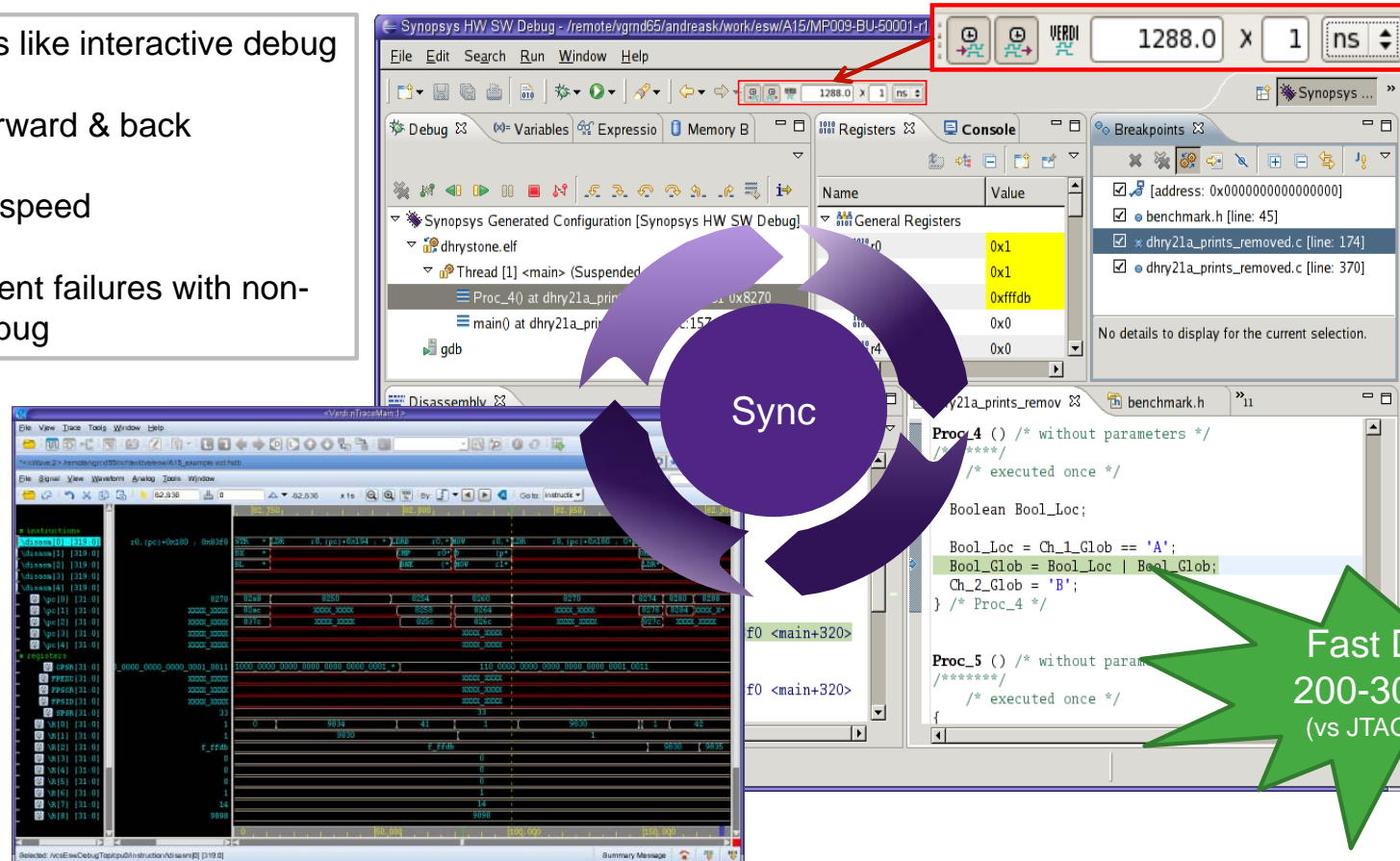
The screenshot displays the Synopsys HW/SW Debug Eclipse IDE interface. The main window is titled "Synopsys HW/SW Debug - /tmp/dhrystone/src/dhry\_1.c - Eclipse". The interface is divided into several panes:

- C Source Code:** Shows the C source code for `Proc_2` in `dhry_1.c`. The current line of execution is highlighted at line 337: `Int_Loc = *Int_Par_Ref + 10;`.
- Assembly Code:** Shows the disassembled assembly code for `Proc_2`. The current instruction is `ldr r0, [r1]` at address `00008110`.
- Registers:** A table showing the values of general registers. The registers are `r0` through `r6`. The values are: `r0`: `0xa1d4`, `r1`: `0xcb04`, `r2`: `0x80000000`, `r3`: `0x0`, `r4`: `0x0`, `r5`: `0x0`, `r6`: `0x0`.
- Call Stack:** Shows the current call stack. The top entry is `Proc_2() at dhry_1.c:337 0x8110`. Below it is `main() at dhry_1.c:188 0x8644`.
- Variables:** A table showing the values of variables. The variables are `Int_Par_Ref` (type `One_Fifty *`, value `0xffffe8`), `__result` (type `int`, value `1048552`), `Int_Loc` (type `One_Fifty`, value `0`), and `Enum_Loc` (type `Enumeration`, value `4294967293`).
- Memory:** A table showing the values of memory locations. The memory location `&Int_1_Loc` is selected, showing a value of `0x00000000` at address `00000000`.

Provides programmer's view of code running on simulated processor

# Post-Process Debug Approach

- Eclipse feels like interactive debug
- Run/step forward & back
- Fast debug speed
- Solve transient failures with non-intrusive debug



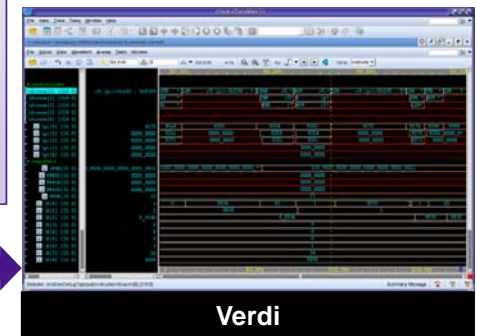
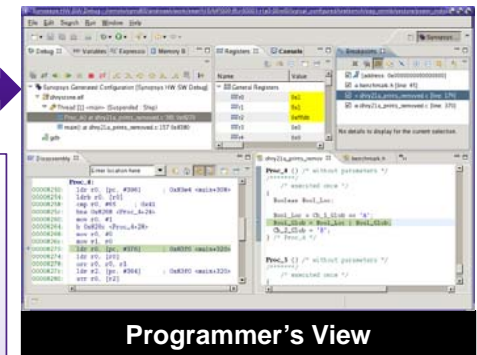
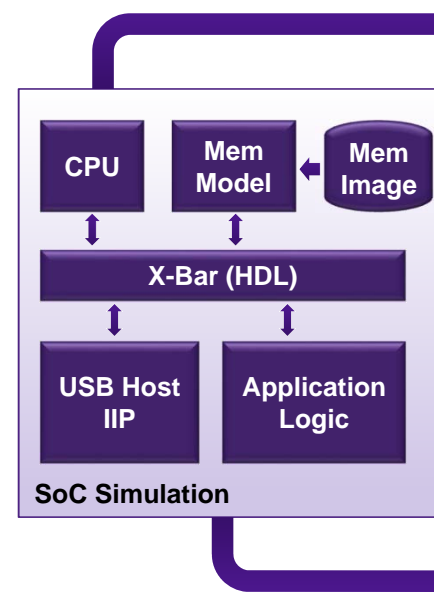
The screenshot displays the Synopsys HW SW Debug interface. At the top, a red box highlights the status bar showing '1288.0 x 1 ns'. Below this, the interface is divided into several panels: Debug, Variables, Expressions, Memory, Registers, Console, and Breakpoints. The Debug panel shows a list of threads, including 'Thread [1] <main> (Suspended)'. The Registers panel shows a table of registers with values. The Console panel shows a list of breakpoints. The Breakpoints panel shows a list of breakpoints with details. A large purple circular arrow labeled 'Sync' is overlaid on the interface, indicating a synchronization process. A green starburst callout in the bottom right corner states 'Fast Debug 200-300 KHz (vs JTAG @ 1Hz)'.



# Verdi HW SW Debug

*Embedded Processor Debug with Synchronized Hardware & Software*

- Enables co-debug between HW & SW
- HW and SW time synchronization
- View C/Assembly source, C variables, stack, memory
- Debug multiple cores
- Simulation supports all ARM® Cortex® cores
- Easy to support additional/custom cores



# Agenda

Synchronized HW SW Debug

**Practical Debug Application: Linux Boot**

Performance Profiling of Embedded Software

Coverage Metrics for Embedded Software

# Debug Challenges of Software Tests



Dealing with different OS –  
Linux , Android, RTOS etc.

How can I debug the OS  
code itself?

How can I debug apps  
running on the OS?

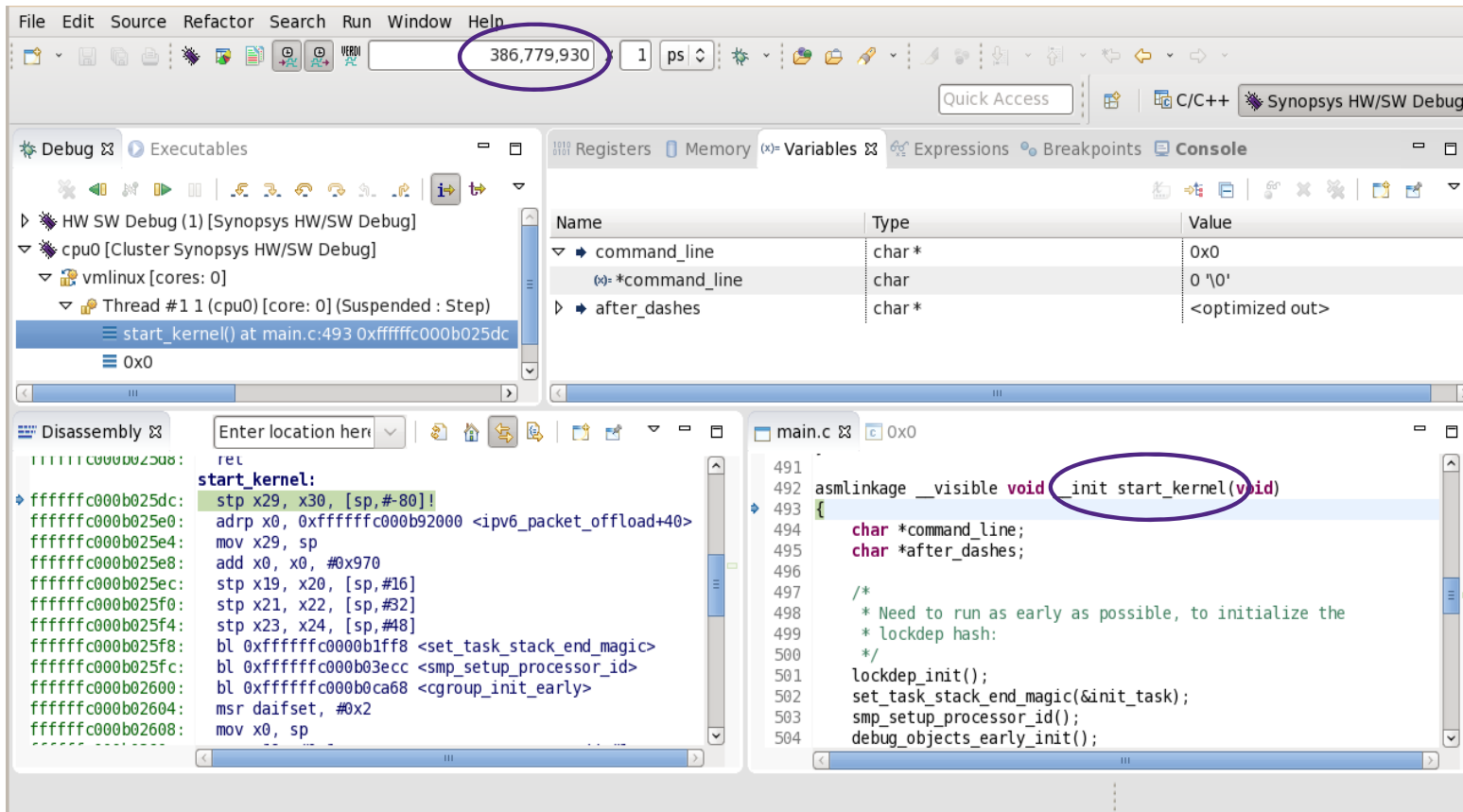
Data sizes are huge! 600M-  
1B cycles for Linux boot!

Debugging effort-intensive  
due to big source code base

Time-consuming test debug -  
multiple features tested

High speed embedded code debug with Verdi HW SW Debug

# Linux Boot Debug



The screenshot displays the Synopsys HW/SW Debug IDE interface. The top toolbar includes a 'Step Into' button (circled in purple). The 'Variables' pane shows the following table:

Name	Type	Value
command_line	char *	0x0
*command_line	char	0 '\0'
after_dashes	char *	<optimized out>

The 'Disassembly' pane shows the following assembly code for the 'start\_kernel' function:

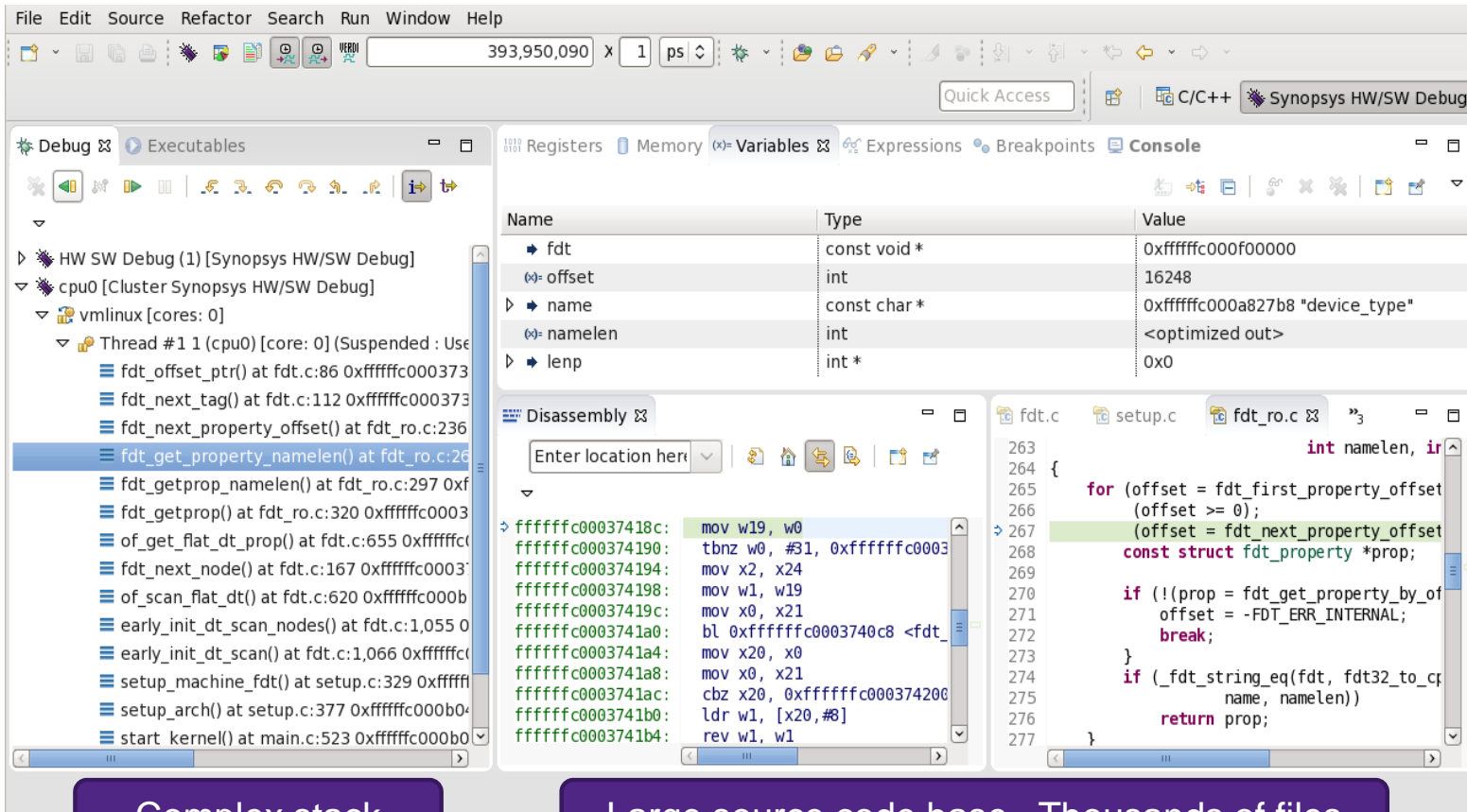
```
ret
start_kernel:
  stp x29, x30, [sp, #-80]!
  adrp x0, 0xffffffffc00b92000 <ip6_packet_offload+40>
  mov x29, sp
  add x0, x0, #0x970
  stp x19, x20, [sp, #16]
  stp x21, x22, [sp, #32]
  stp x23, x24, [sp, #48]
  bl 0xffffffffc000b1ff8 <set_task_stack_end_magic>
  bl 0xffffffffc00b03ecc <smp_setup_processor_id>
  bl 0xffffffffc00b0ca68 <cgroup_init_early>
  msr daifset, #0x2
  mov x0, sp
```

The 'main.c' source file shows the following C code for the 'start\_kernel' function:

```
491
492 asmlinkage __visible void __init start_kernel(void)
493 {
494     char *command_line;
495     char *after_dashes;
496
497     /*
498      * Need to run as early as possible, to initialize the
499      * lockdep hash:
500      */
501     lockdep_init();
502     set_task_stack_end_magic(&init_task);
503     smp_setup_processor_id();
504     debug_objects_early_init();
```

- Long test split into time slices
- Time slice starts at 38M cycles
- Start debug from symbol name (e.g. app\_start, start\_kernel)
- Slice generated from "start\_kernel" symbol
- Run forwards and backwards through this time slice

# Debugging a Complex Software Stack



The screenshot displays the Synopsys HW/SW Debug IDE interface. The top menu bar includes File, Edit, Source, Refactor, Search, Run, Window, and Help. The main workspace is divided into several panes:

- Debug Console:** Shows the execution flow, including Thread #1 1 (cpu0) [core: 0] (Suspended: Use). The stack trace includes functions like `fdt_offset_ptr()`, `fdt_next_tag()`, `fdt_next_property_offset()`, `fdt_get_property_namelen()`, `fdt_getprop_namelen()`, `fdt_getprop()`, `of_get_flat_dt_prop()`, `fdt_next_node()`, `of_scan_flat_dt()`, `early_init_dt_scan_nodes()`, `early_init_dt_scan()`, `setup_machine_fdt()`, `setup_arch()`, and `start_kernel()`.
- Registers, Memory, Variables, Expressions, Breakpoints, Console:** The Variables pane shows a table of variables:

Name	Type	Value
fdt	const void *	0xffffffff000f00000
offset	int	16248
name	const char *	0xffffffff000a827b8 "device_type"
namelen	int	<optimized out>
lenp	int *	0x0

The Disassembly pane shows the assembly code for the current instruction:

```
ffffffffffc0037418c: mov w19, w0
ffffffffffc00374190: tbnz w0, #31, 0xffffffffc00374194
ffffffffffc00374194: mov x2, x24
ffffffffffc00374198: mov w1, w19
ffffffffffc0037419c: mov x0, x21
ffffffffffc003741a0: bl 0xffffffffc003740c8 <fdt_
ffffffffffc003741a4: mov x20, x0
ffffffffffc003741a8: mov x0, x21
ffffffffffc003741ac: cbz x20, 0xffffffffc00374200
ffffffffffc003741b0: ldr w1, [x20, #8]
ffffffffffc003741b4: rev w1, w1
```

The Source Code pane shows the C code for `fdt_ro.c`:

```
263 int namelen, ir
264 {
265     for (offset = fdt_first_property_offset
266         (offset >= 0);
267         (offset = fdt_next_property_offset
268             const struct fdt_property *prop;
269
270     if (!(prop = fdt_get_property_by_of
271         offset = -FDT_ERR_INTERNAL;
272     break;
273 }
274 if (_fdt_string_eq(fdt, fdt32_to_cp
275     name, namelen))
276     return prop;
277 }
```

Two callouts at the bottom of the screenshot highlight key features:

- Complex stack**: Points to the stack trace in the Debug Console.
- Large source code base, Thousands of files**: Points to the Source Code pane.

- Large source code base
- Highly complex stack with many levels
- Variables include integer, string and complex types like linked lists
- Unix console visible in window with timestamps

# Verdi HW SW Debug Demonstration



# Agenda

Synchronized Debug with Verdi HW SW

Practical Debug Application: Linux Boot

**Performance Profiling of Embedded Software**

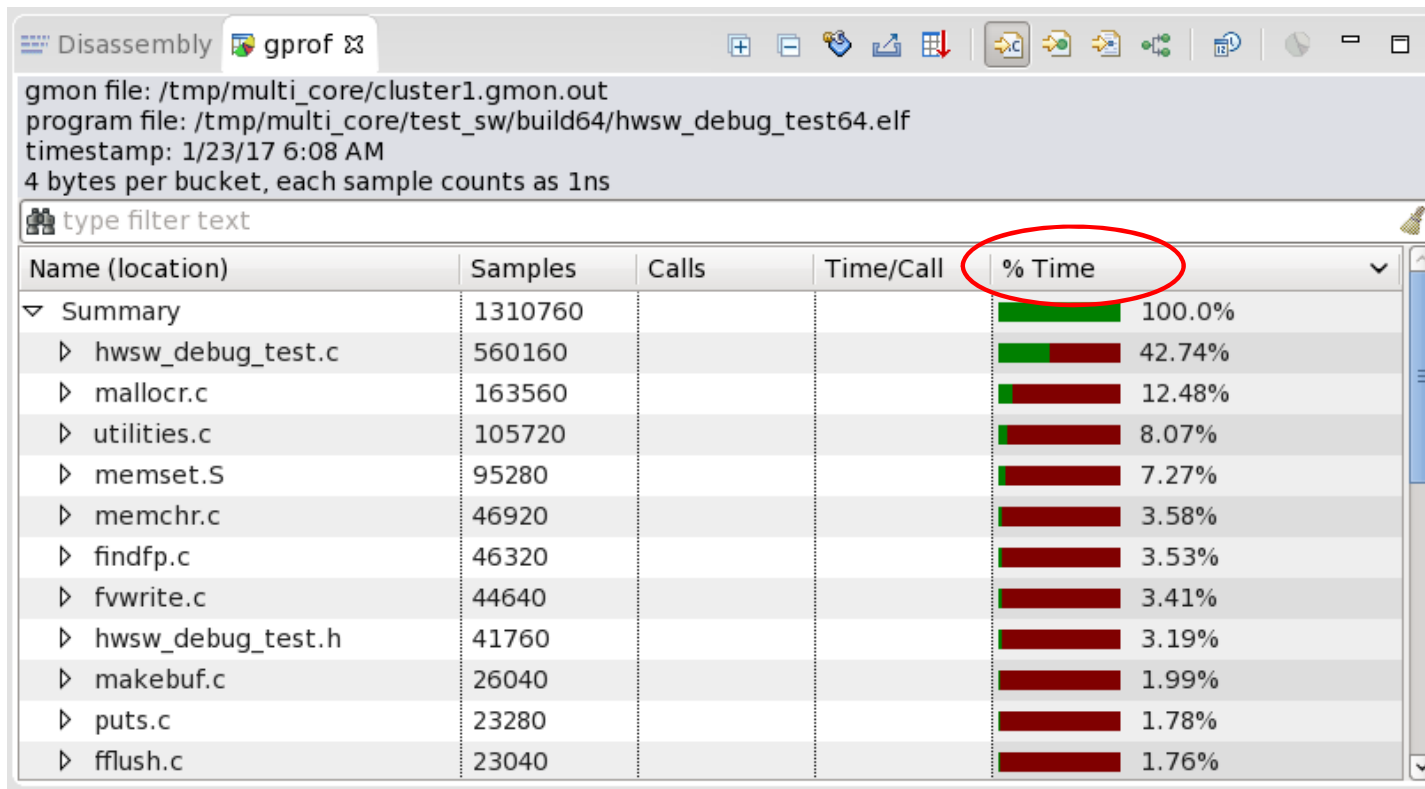
Coverage Metrics for Embedded Software

# Measuring Embedded SW Performance

- **How fast does my embedded software run?**
  - How do I measure this before silicon arrives?
  - How do I measure without modifying the embedded executable?
- **What can I do if it's not fast enough?**
  - How can I get reports of time per function, callers, etc.?
  - Software team can do this with silicon...But how can I do this now?

Really need a standard software profiler for embedded software

# Performance Profiling



- Eclipse Open Platform Profiler Plugin
- Performance Profile of embedded software
- Task call totals
- Assembly level detail
- Non-intrusive: No insertion of code or opcodes
- Measure performance of production firmware

# Agenda

Synchronized Debug with Verdi HW SW

Practical Debug Application: Linux Boot

Performance Profiling of Embedded Software

**Coverage Metrics for Embedded Software**

# Embedded Software Code Coverage

- **How do I meet ISO 26262 Embedded Software requirements?**

- Are any pieces of code not tested/executed?
- Do I have dead code that I do not expect?
- Sounds familiar for verification engineers...Think coverage!

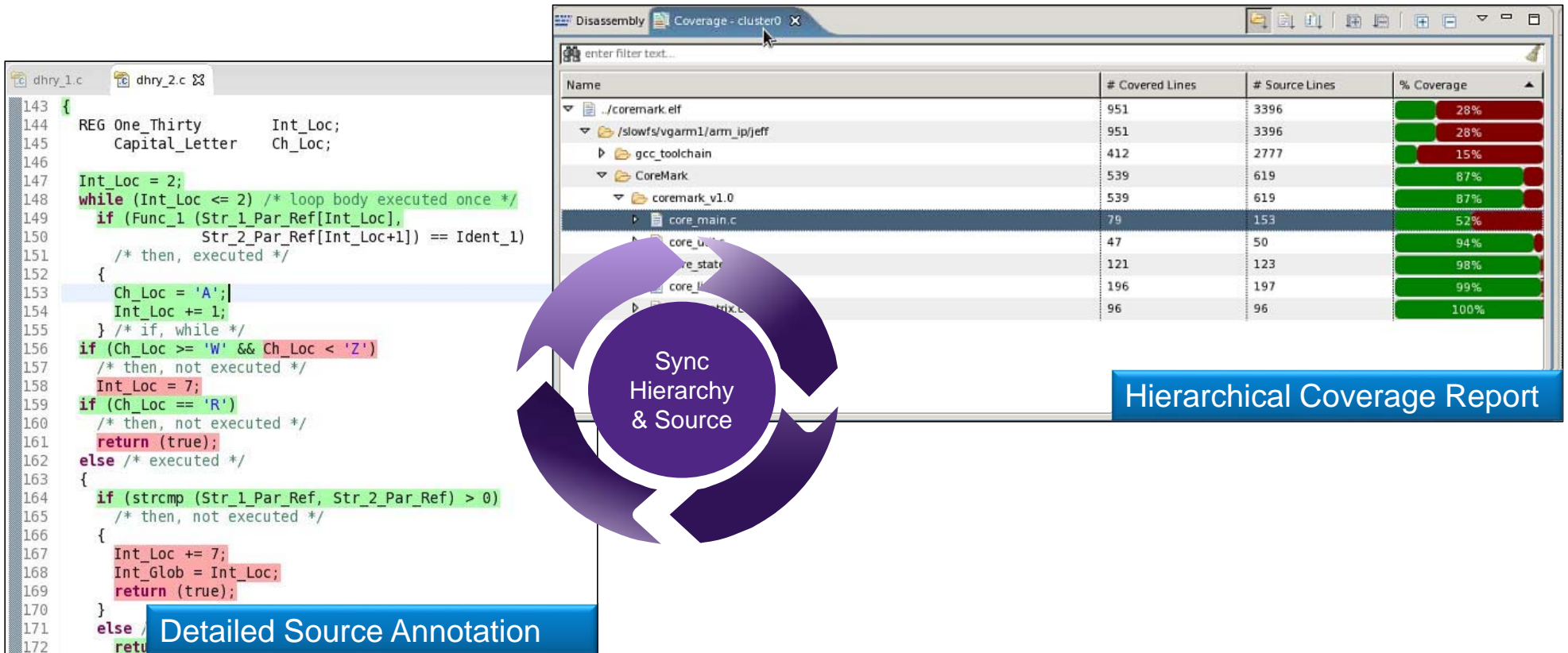


- **Line Coverage – Current method**

- Software teams use coverage (gcov, purecov, etc)
- Problem: gcov heavily modifies the compiled binary (Firmware, OS, App)
- Teams want to run production image without changes

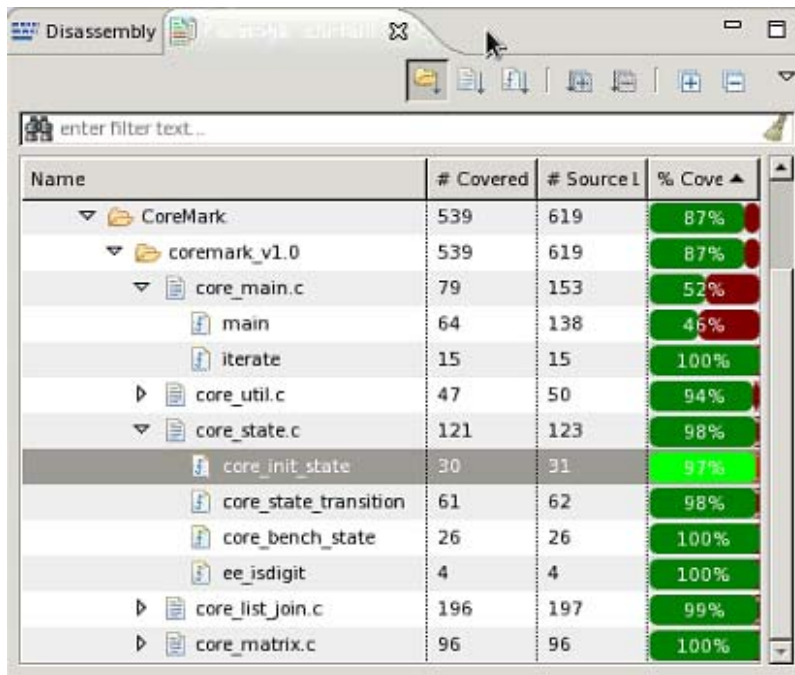
Need a non-intrusive way to measure line coverage

# Embedded Software - Line Coverage



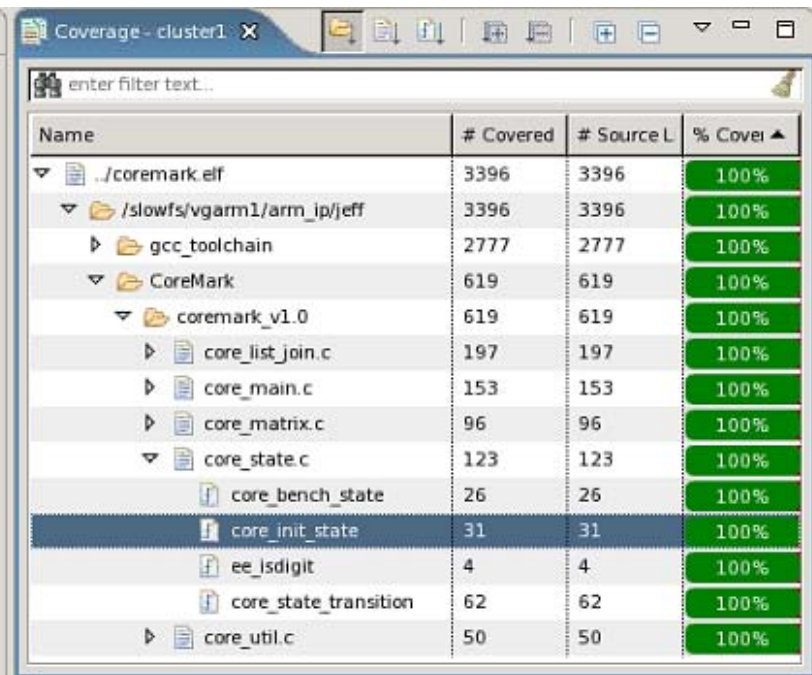


# Embedded Code Coverage on Multiple Cores



Name	# Covered	# Source L	% Cove
CoreMark	539	619	87%
coremark_v1.0	539	619	87%
core_main.c	79	153	52%
main	64	138	46%
iterate	15	15	100%
core_util.c	47	50	94%
core_state.c	121	123	98%
core_init_state	30	31	97%
core_state_transition	61	62	98%
core_bench_state	26	26	100%
ee_isdigit	4	4	100%
core_list_join.c	196	197	99%
core_matrix.c	96	96	100%

Cluster #0



Name	# Covered	# Source L	% Cover
./coremark.elf	3396	3396	100%
/slowfs/vgarn1/arm_ip/jeff	3396	3396	100%
gcc_toolchain	2777	2777	100%
CoreMark	619	619	100%
coremark_v1.0	619	619	100%
core_list_join.c	197	197	100%
core_main.c	153	153	100%
core_matrix.c	96	96	100%
core_state.c	123	123	100%
core_bench_state	26	26	100%
core_init_state	31	31	100%
ee_isdigit	4	4	100%
core_state_transition	62	62	100%
core_util.c	50	50	100%

Cluster #1

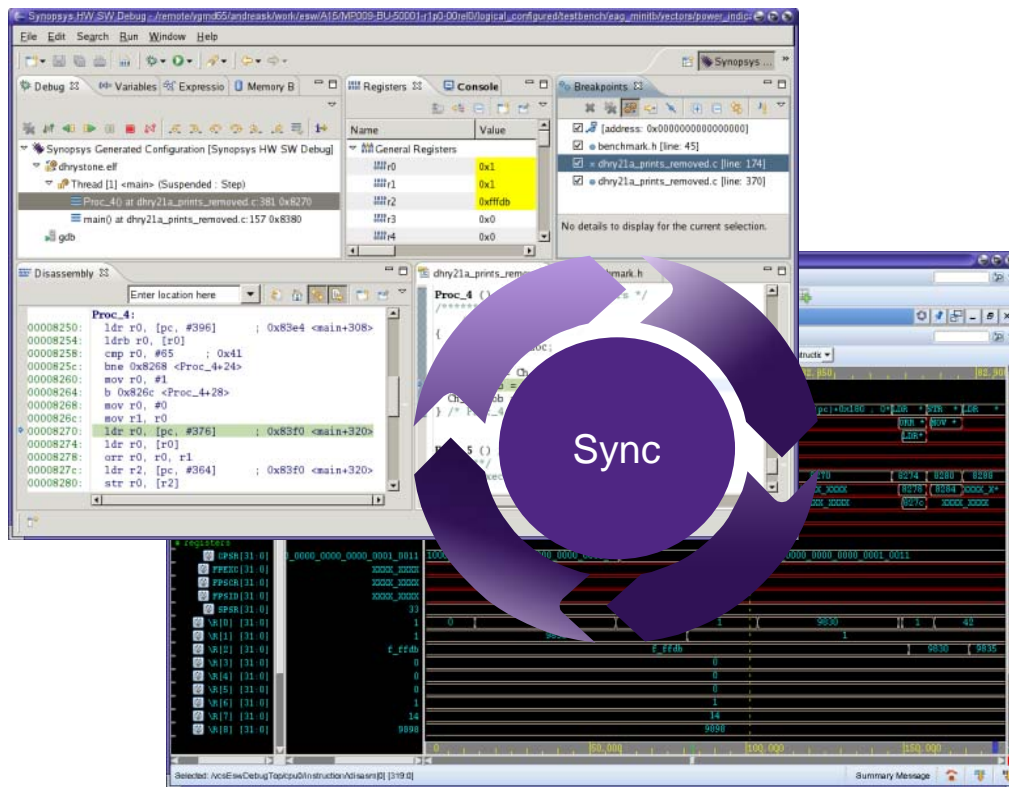
- Support for multiple clusters/cores
- Collect coverage per cluster/core
- Collect coverage per executable/elf

# Verdi HW SW Debug Demonstration

*Performance Profiling, Line Coverage*

# Verdi HW SW Debug

## Boosting Debug Productivity



### • Verification Engineer Productivity

- Debug C-source code on embedded core
- Debug software/hardware issues
- Measure code coverage for ISO 26262

### • Software Engineer Productivity

- Full software development environment
- Pre-silicon debug of boot and applications
- Profile performance before silicon arrives

# Thank You



