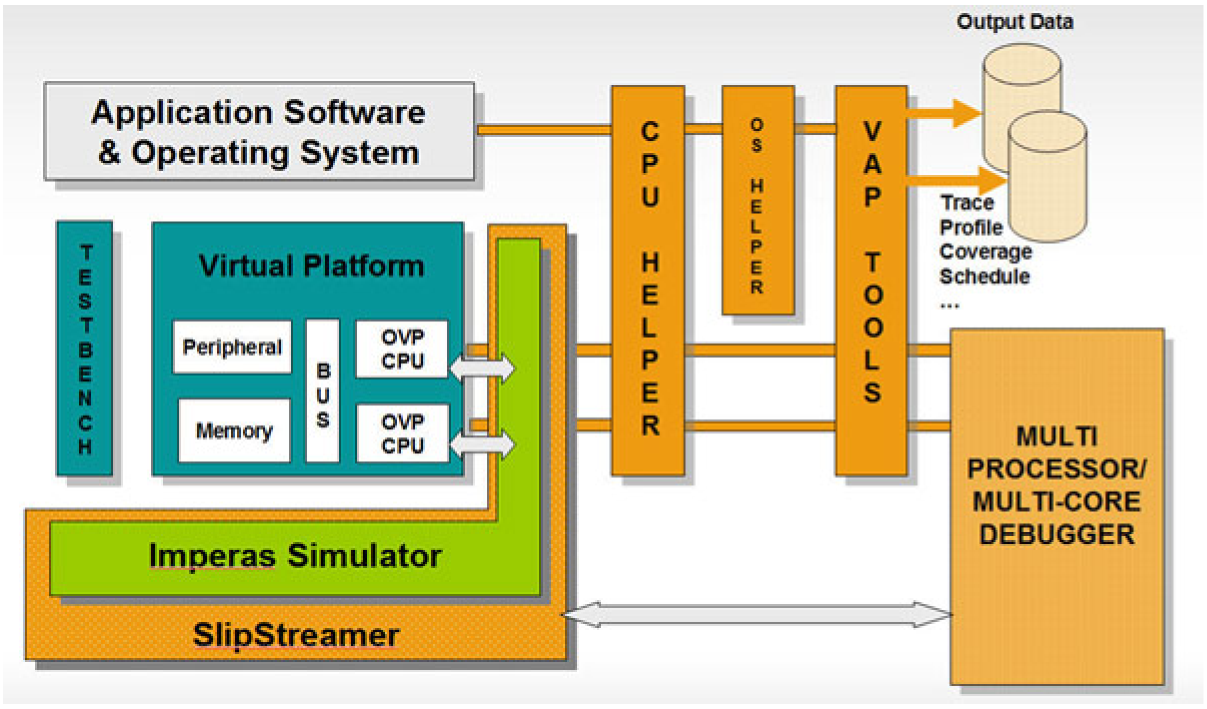
M*SDK - Advanced Multicore Software Development Kit

Overview

The focus of the Imperas products is to save engineering time in the development of embedded software, primarily achieved by making the engineering process significantly more efficient through the use of high-performance simulation and automated, powerful tooling.

The Imperas Multicore Software Development Kit (M*SDK) is a complete embedded software development environment that operates using virtual platforms and is specifically designed to handle complex multicore related issues. M*SDK contains all the capabilities of the M*DEV product, together with a comprehensive verification, analysis, and profiling (VAP) tool set, plus an advanced 3-dimensional (temporal, spatial and abstraction) debug solution, named 3Debug ™, for heterogeneous multicore processor, peripheral, and embedded software debug.



Both the Verification, Analysis and Profiling (VAP) tools, as well as the 3Debug environment contain a range of highly advanced capabilities to handle unusual problems while also enhancing embedded code for multicore designs and complex processor platforms. Custom capability may be easily created to handle specific scenarios. The system uses advanced technology to improve the performance and reliability of the solution, eliminating wasted hours of unnecessary engineering effort.

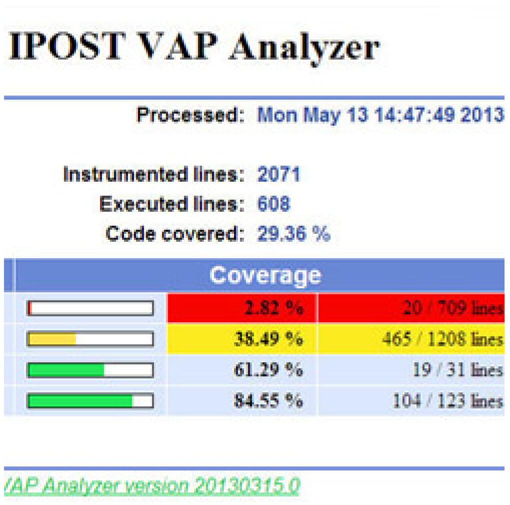
M*DEV - Multicore Developer

M*SDK includes all the functionality and is a superset of the M*DEV product.

M*DEV includes all the platform and peripheral model development technology and high-performance simulation system, using this as a basis for the software development kit components. The M*DEV product is described here.

Verification, Analysis and Profiling (VAP)

The Imperas Multicore Software Development Kit M*SDK™ includes Verification, Analysis and Profiling (VAP) capabilities for systems running single and multicore embedded software. The Imperas M*SDK boasts a number of unique capabilities designed to improve the quality of the final code base in terms of bug elimination and execution performance, as well as reduce the engineering time required to achieve that quality.

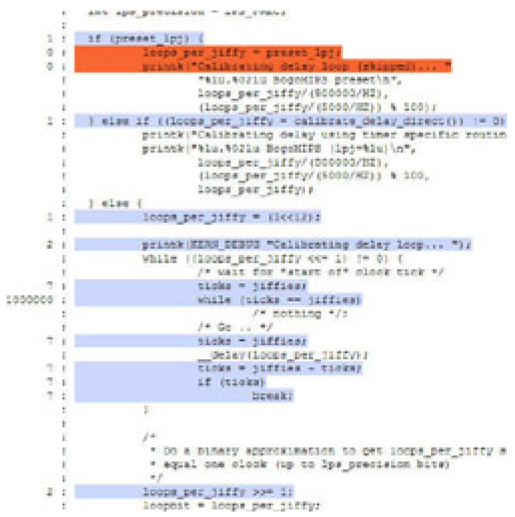


IPOST VAP Analyzer

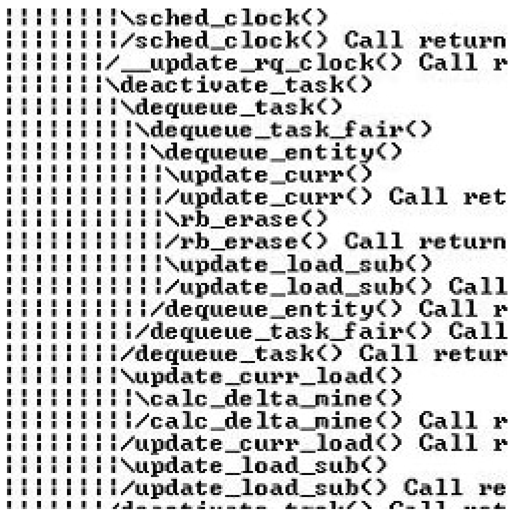
Processed:	Mon May 13 14:47:49 2013
Instrumented lines:	2071
Executed lines:	608
Code covered:	29.36 %

Coverage		
	2.82 %	20 / 709 lines
	38.49 %	465 / 1208 lines
	61.29 %	19 / 31 lines
	84.55 %	104 / 123 lines

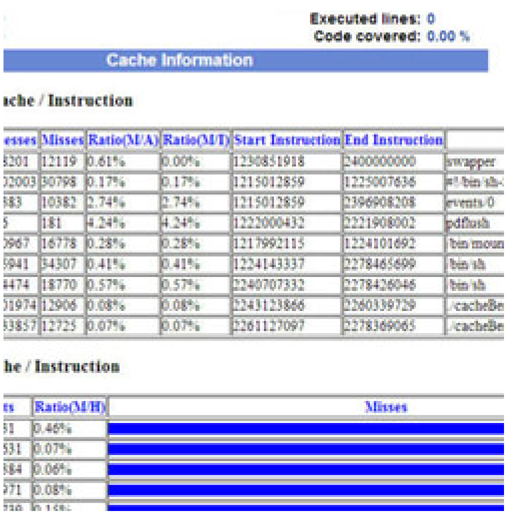
VAP Analyzer version 20130315.0



```
        :   int lpj_precision = lpj_fine;
        :
      1 :   if (preset_lpj) {
      0 :           loops_per_jiffy = preset_lpj;
      0 :           printk("Calibrating delay loop (skipped)... "
        :                   "%lu.%02lu BogoMIPS preset\n",
        :                   loops_per_jiffy/(500000/HZ),
        :                   (loops_per_jiffy/(5000/HZ)) % 100);
      1 :   } else if ((loops_per_jiffy = calibrate_delay_direct()) != 0)
        :           printk("Calibrating delay using timer specific routin
        :           printk("%lu.%02lu BogoMIPS (lpj=%lu)\n",
        :                   loops_per_jiffy/(500000/HZ),
        :                   (loops_per_jiffy/(5000/HZ)) % 100,
        :                   loops_per_jiffy);
        :   } else {
      1 :           loops_per_jiffy = (1<<12);
        :
      2 :           printk(KERN_DEBUG "Calibrating delay loop... ");
        :           while ((loops_per_jiffy <<= 1) != 0) {
        :                   /* wait for "start of" clock tick */
      7 :                   ticks = jiffies;
1000000 :                   while (ticks == jiffies)
        :                           /* nothing */;
        :                   /* Go .. */
      7 :                   ticks = jiffies;
        :                   __delay(loops_per_jiffy);
      7 :                   ticks = jiffies - ticks;
      7 :                   if (ticks)
      7 :                           break;
        :           }
        :
        :           /*
        :            * Do a binary approximation to get loops_per_jiffy s
        :            * equal one clock (up to lps_precision bits)
        :            */
      2 :           loops_per_jiffy >>= 1;
        :           loopbit = loops_per_jiffy;
```



```
¦¦¦¦¦¦¦¦\sched_clock()
¦¦¦¦¦¦¦¦/sched_clock() Call return
¦¦¦¦¦¦¦/__update_rq_clock() Call r
¦¦¦¦¦¦¦\deactivate_task()
¦¦¦¦¦¦¦¦\dequeue_task()
¦¦¦¦¦¦¦¦¦\dequeue_task_fair()
¦¦¦¦¦¦¦¦¦¦\dequeue_entity()
¦¦¦¦¦¦¦¦¦¦¦\update_curr()
¦¦¦¦¦¦¦¦¦¦¦/update_curr() Call ret
¦¦¦¦¦¦¦¦¦¦¦\rb_erase()
¦¦¦¦¦¦¦¦¦¦¦/rb_erase() Call return
¦¦¦¦¦¦¦¦¦¦¦\update_load_sub()
¦¦¦¦¦¦¦¦¦¦¦/update_load_sub() Call
¦¦¦¦¦¦¦¦¦¦/dequeue_entity() Call r
¦¦¦¦¦¦¦¦¦/dequeue_task_fair() Call
¦¦¦¦¦¦¦¦/dequeue_task() Call retur
¦¦¦¦¦¦¦¦\update_curr_load()
¦¦¦¦¦¦¦¦¦\calc_delta_mine()
¦¦¦¦¦¦¦¦¦/calc_delta_mine() Call r
¦¦¦¦¦¦¦¦/update_curr_load() Call r
¦¦¦¦¦¦¦¦\update_load_sub()
¦¦¦¦¦¦¦¦/update_load_sub() Call re
¦¦¦¦¦¦¦/deactivate_task() Call ret
```



Executed lines: 0
Code covered: 0.00 %

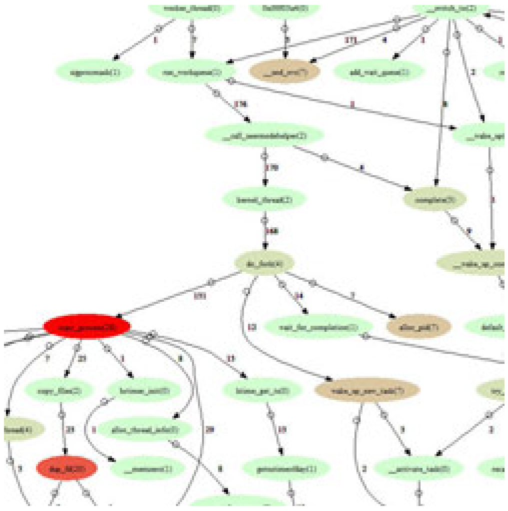
Cache Information

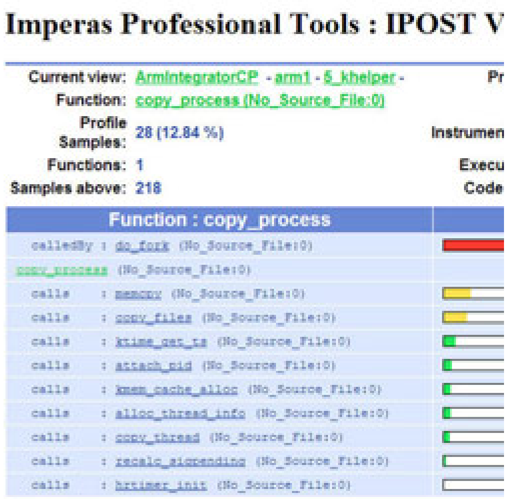
ache / Instruction

esses	Misses	Ratio(M/A)	Ratio(M/T)	Start Instruction	End Instruction	
3201	12119	0.61%	0.00%	1230851918	2400000000	swapper
02003	30798	0.17%	0.17%	1215012859	1225007636	#!/bin/sh-
383	10382	2.74%	2.74%	1215012859	2396908208	events/0
5	181	4.24%	4.24%	1222000432	2221908002	pdflush
0967	16778	0.28%	0.28%	1217992115	1224101692	/bin/moun
5941	34307	0.41%	0.41%	1224143337	2278465699	/bin/sh
4474	18770	0.57%	0.57%	2240707332	2278426046	/bin/sh
01974	12906	0.08%	0.08%	2243123866	2260339729	./cacheBe
33857	12725	0.07%	0.07%	2261127097	2278369065	./cacheBe

he / Instruction

ts	Ratio(M/H)	Misses
31	0.46%	
531	0.07%	
384	0.06%	
971	0.08%	





Imperas Professional Tools : IPOST V

Current view:	ArmIntegratorCP - arm1 - 5_khelper -	Pr
Function:	copy_process (No_Source_File:0)	
Profile Samples:	28 (12.84 %)	Instrumen
Functions:	1	Execu
Samples above:	218	Code

Function : copy_process

```
    calledBy : do_fork (No_Source_File:0)
copy_process (No_Source_File:0)
    calls    : memcpy (No_Source_File:0)
    calls    : copy_files (No_Source_File:0)
    calls    : ktime_get_ts (No_Source_File:0)
    calls    : attach_pid (No_Source_File:0)
    calls    : kmem_cache_alloc (No_Source_File:0)
    calls    : alloc_thread_info (No_Source_File:0)
    calls    : copy_thread (No_Source_File:0)
    calls    : recalc_sigpending (No_Source_File:0)
    calls    : hrtimer_init (No_Source_File:0)
```

As well as providing a complete user accessible range of standard technology to implement the VAP functionality, the system has been extended with unique capabilities that increase the speed and accuracy of code analysis and profiling. These provided capabilities, known as VAP tools aid in the streamlining of software execution and the optimal use of multi-core processor architectures. The system is also CPU- and OS-aware, allowing different operations based on the system state.

Advanced VAP Tool capability in the M*SDK include the following:

- Instruction Tracing
- Function Tracing
- OS Task Tracing
- Code Coverage
- Instruction Coverage
- Memory Analysis
- Cache Analysis
- Exception & Interrupt Analysis
- OS Scheduler Analysis
- Execution Profiling
- Fault Injection
- Protocol Verification
- Shared Resource Introspection
- Heterogeneous Multicore Debug

The system uses all the facets of the virtual platform approach and its power over a hardware prototype, for example, the ability to inspect deeply embedded signals and the rapid inspection of shared memory and key registers. In addition, Imperas technology is able to use ToolMorphing™ to 'merge' the debugger operations with the simulator operation so it underneath the embedded software allowing, for example instrumentation free coverage and profiling, and variable and memory inspection dynamically without embedded software code modification or interaction, thus improving accuracy.

Another unique facet of the system is the ability to enable custom features to target company- or even project-specific profiling, analysis, and verification requirements. For example, specialized communication testing functions between cores in multicore devices, targeting memory location tracing, specific code analysis routines, and many other custom capabilities may be created using a specialized API that provides access to internal simulation and environment features.

The verification aspect of the tools is enhanced with a range of controllability and observability functions. Powerful test scenarios may be applied that leverage exceptions created through various means including assertions operating separately to the embedded code, applying programmed scenarios based on multicore processor interactions. It is this level of capability that is easily leveraged on top of more traditional test sets that can be used to ensure correct operation in all circumstances. This is augmented with a range of coverage options working at varied levels of abstraction to ensure that the entire code base has been fully exercised.

The Imperas ToolMorphing™ technology is used to generate code for the sections of the software design kit required for VAP functions called at specific moments. By tying the simulation code generation process with that of the software design kit, inefficiencies inherent in other tool architectures are eliminated, improving the performance of the entire Imperas solution.

In many tool suites, either an apparent bug can be seen to disappear when a debug environment is applied, or a “fake” bug appear when code is run under a debugger (these are often known as Hiesenbugs). The SlipStreamer™ technology, also executed within the code generation process, separates the embedded software under simulation from underlying system code generation, thus ensuring that code-under-test execution is not affected by the tooling processes, and vice versa. This avoids the time consuming problem of tracking down a bug that is not present, or finding one that hides during debugging.

The Imperas M*SDK represents the state-of-the-art in terms of embedded software development technology, and has been proven to shave many days off intensive engineering effort, thus reducing engineering schedules at critical times during the development process.

Advanced Software Debug using 3Debug

M*SDK includes a powerful three-dimensional debug solution (Temporal, Spatial, Abstract), named 3Debug™, that provides a range of capabilities over and above standard single core debuggers, such as the GNU Debugger (GDB). The M*SDK debug solution has been specifically designed to deal with the issues surrounding multicore processor debug.

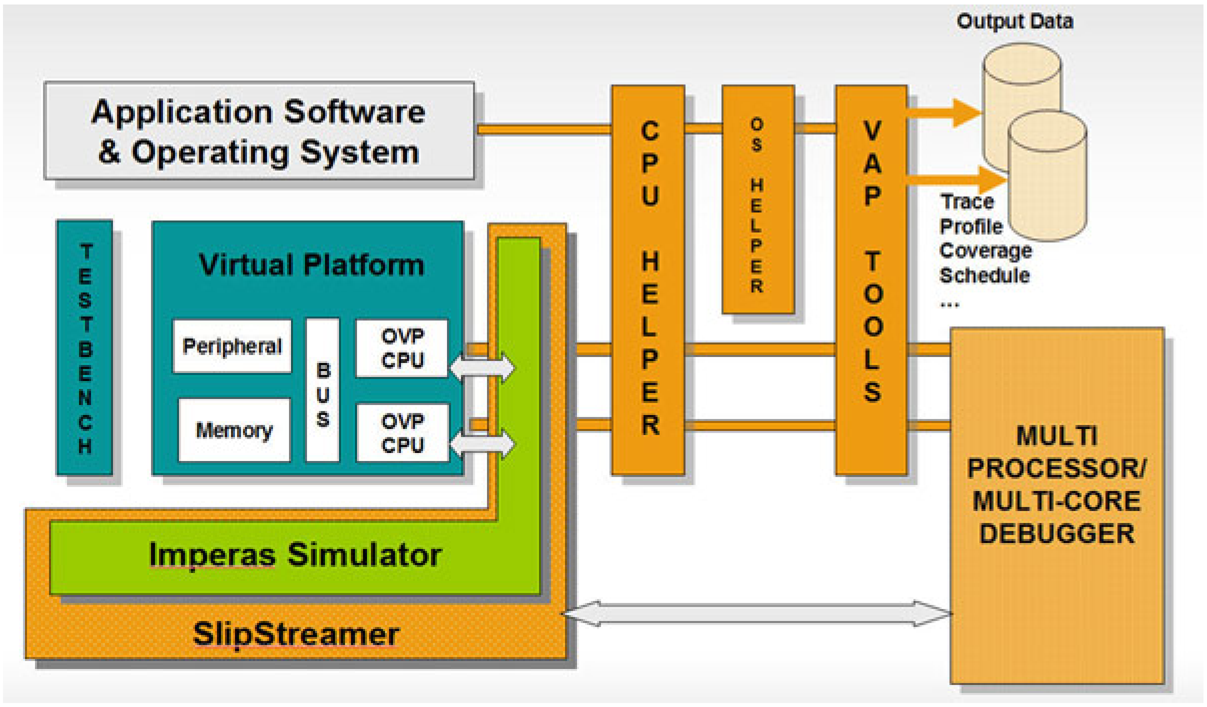
3Debug operates in a familiar, GDB-like, manner and allows the processing of standard GDB commands. It also leverages a TCL input mode, which provides extended scripting capability. TCL scripts may be attached to execution activity in the platform, allowing complex debug operations to be executed based on a sequence of events. The system makes use of user and programmer view modes, allowing ideal use models to be leveraged for platform developers, embedded code developers, or maintenance engineers.

In a multicore scenario 3Debug can be made to focus on specific processors or operate across entire platforms allowing for the debug of all processors and peripheral activity. It can control the specific execution of named processors, freeze all or parts of the platform, and synchronize debugging to particularly execution points across the system. The ability to observe critical signals at important points in time is also improved, for example, watching classic areas of interaction such as shared memory locations which can create many issues during the multicore debug process. Programmable watch scenarios make analyzing memory usage and looking for corrupted locations straightforward.

The 3Debug debugger operates in three dimensions:

- **Spatial.** Provides access across the platform’s multicore components. Trace, debug, analyse, verify etc., software running on any or all processors or cores in the platform at the same time as debugging code modeling OVP peripherals.
- **Temporal.** Monitor and trace events and activity occurring in the system over time creating time ordered complex sequential breakpoints and temporal assertions, while applying unique control to code scheduling on different cores.
- **Abstract.** By creating abstractions of activity occurring in the system, debugging, analysis, verification etc can focus on high level events and operation of the system. For example the built in CPU-awareness and Operating System-awareness provide abstractions allowing such things as simulated OS scheduler tracing, breakpoints on simulated OS task context switch etc.

This level of debug awareness coupled with the programmable nature of the tool makes this debug environment immensely powerful, which leads to extremely efficient debug operations.



3Debug makes use of the CPU- and OS-aware operational nature (capability to understand and control operating system execution) and has been used to discover and repair complex bug conditions. In one example, a large semiconductor company had been operating a modified version of Linux within delivered products. Using 3Debug, engineers found a rare, but critical, fault condition resulting from overwritten memory locations in an unusual scenario. OS Aware control-ability was used to trap the fault condition, which had not been discovered using regular debug systems even after many hours of testing on hardware prototypes and other virtual platform systems.

3Debug sets a new standard for multicore debugging and has been proven on real designs to locate fault conditions inaccessible to other debug solutions. Its ease of use, execution reliability, and operational efficiency make it an excellent choice for multicore embedded software development.

See the model families available from OVPworld
