Parallel Simulation Accelerates Embedded Software Development, Debug and Test

James Kenney, Simon Davidmann and Larry Lapides
Imperas Software Ltd.
Oxford, United Kingdom
larryl@imperas.com

***Abstract* – For any simulation technology, the key factors for usability are controllability, observability, and performance. For instruction accurate virtual platforms, the controllability and observability have been successfully addressed in various ways, including using APIs for the processor models or tools integrated into the simulation environment. In the area of performance, where near real time simulation performance is required, virtual platforms have been limited to execution in a single thread because of the need for determinism in the simulation. This need is driven by the loss of many of the key benefits of controllability and observability if the simulation is not repeatable. While multiple threads on multiple cores of the host x86 workstation offer the hope of performance improvement, the overhead for synchronizing multiple simulation execution threads to maintain deterministic simulation results has cancelled out any performance gains realized by parallelizing the simulation.**

A new synchronization algorithm has been realized, with much lower overhead, so that significant performance gains have been achieved. Performance gains of over 2.5x have been achieved for symmetric multiprocessor (SMP) systems simulating on a 4-core host machine, while performance gains of over 3x have been achieved for asymmetric multiprocessor (AMP) systems. The same principals have also been applied to accelerating the performance of virtual platforms where the performance bottleneck is one or more of the models used for accelerating specific applications such as image recognition.

Keywords – Virtual platform, parallel simulation, simulation performance, SMP

I. INTRODUCTION

It has long been a goal of simulation technologists, including those working on virtual platforms (software simulation), to use the multiple cores on the host x86 PC to accelerate simulation performance. For Just-In-Time (JIT) code-morphing, or binary translation, virtual platforms [1, 2, 3], a new synchronization algorithm has been developed, enabling significant performance gains. The new parallel simulation accelerator requires no changes to the models in the virtual platforms, the software tools in the simulation environment or the software being executed on the virtual platform.

The requirement for simulation performance in virtual platforms is driven by two factors. The first is the length of individual software tests. This is exemplified by the network server use case where individual tests can include the execution of over 10 trillion instructions. For a simulator running at 100 million instructions per second, the execution time of a single test will be more than 24 hours.

The second factor driving performance is the testing requirements for safety critical systems, such as those in the automotive industry. In this case, the overall test suite may consist of thousands of tests, which need to be executed each day as part of a regression test methodology. In addition, other tools such as code coverage and fault injection need to be added to the simulation environment to meet testing requirements.

When these factors are combined with virtual platforms that have multicore processors, or multiple processors instanced in the platform, or both, the need for parallel simulation acceleration becomes evident. With JIT virtual platforms, the total simulation throughput will stay relatively constant as the number of simulated cores increases (see Fig. 1), however, this means that the simulation throughput per core decreases. Since the number of instructions per core, or the overall number of tests, are going to stay the same as the virtual platform complexity increases (measured by increasing numbers of simulated cores), simulation technology must improve if virtual platforms are to remain a viable technology for embedded software development, debug and test.

The new parallel synchronization algorithm has been implemented and tested in the Open Virtual Platforms simulation environment [3].

II. SIMULATION TECHNOLOGY

A. Just-In-Time Code Morphing Simulation

Just-in-time (JIT) code translating simulators are now widely recognized to be the fastest and most powerful tools for development of instruction accurate virtual platforms. A single-threaded JIT-based simulator is typically capable of delivering simulation performance of billions of simulated instructions per second. Virtual platform simulators operate by dividing time into *quanta*, of fixed or variable size, specified by the platform user. For multiple processor platforms, each processor is

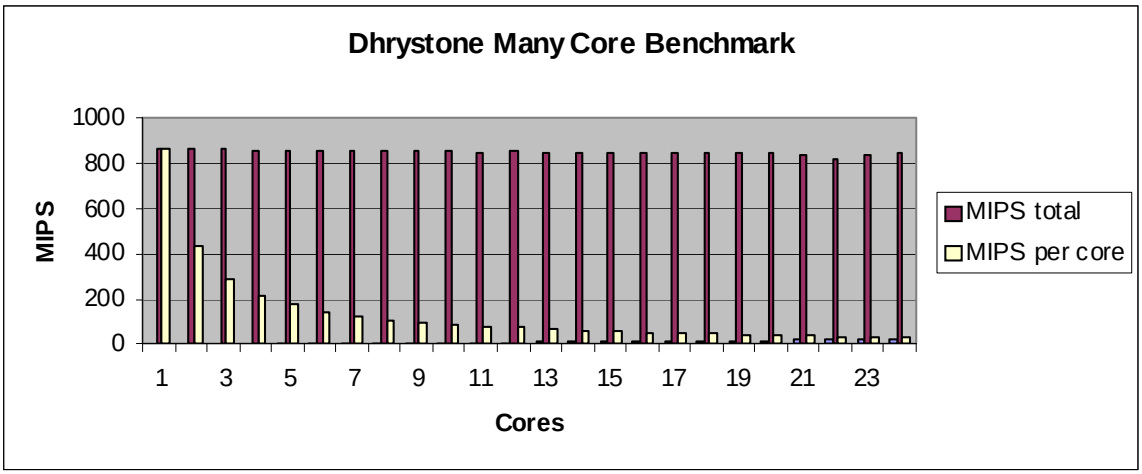

Figure 1. Simulation throughput versus simulated processor cores for single threaded JIT virtual platform shows constant overall simulation throughput, but decreasing throughput per core as the number of cores increases.

simulated in turn for a quantum. When all processors have finished the quantum, time is advanced and simulation resumes for the first processor in the next quantum.

B. Parallel Simulation Algorithm

This paper reports on a new parallel simulation technology which extends JIT code-morphing virtual platform simulation: MultiProcessor target on MultiProcessor host (MPonMP). With the MPonMP technology, parallelism is implemented using POSIX threads (pthreads). Each independent processor core runs in a separate pthread. For example, a simulation of a platform containing an Imagination MIPS P5600 quad core processor model will have a separate pthread for each of the four processor cores. Fig. 2 illustrates the MPonMP technology.

For MPonMP, instructions in a processor model that require synchronized execution are identified by a call to a special function during the JIT code translation phase. For example, in the Open Virtual Platforms (OVP) ARM processor model C code source, both traditional exclusive access instructions (e.g. SWP) and scalable exclusive access idioms (e.g. LDREX/STREX) are identified by calls to this function.

When an instruction that requires synchronized execution is encountered, the simulator first stops all other concurrently executing threads. It then executes the instruction in its entirety before restarting the concurrent threads.

The MPonMP parallel simulations are quantized in the same way as single-threaded simulations. All processors must complete simulation of a quantum before time is advanced to the next quantum. This means that the end of a quantum is also a synchronization point.

III. PARALLEL SIMULATION RESULTS

Three examples have been chosen to illustrate the performance improvements and scalability of the MPonMP technology: a tightly coupled application running directly on the processor cores, applications running on SMP Linux, and running on a 16 core host workstation.

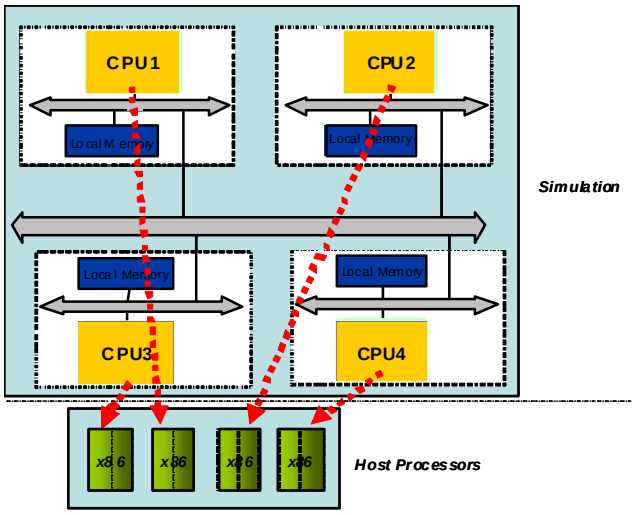

Figure 2. With MPonMP, each processor model in the virtual platform is simulated in a separate thread on the host x86 workstation.

A. Tightly Coupled Bare Metal Application

The parallel primes search application, distributed with the ARM DS-5 product [4], was tested with the MPonMP technology. The program is very tightly coupled; the four cores work together to discover primes using wholly-shared memory, synchronizing at high frequency using a simple mutex library implemented in assembly code. Critical regions of the mutex library are protected using LDREX/STREX instruction pairs. The application was modified to increase the number of primes for which to search from 100 to 1,000,000. The processor model in the virtual platform was an ARM Cortex-A9MPx4, built using the OVP APIs [5]. For this example, a host PC with a 3.5GHz quad core Intel i7-4770K was used.

The simulator quantum size was set to 0.00001 simulated seconds (10 µsec). This means that each core will execute 5,000 instructions per quantum, given the simulated MIPS rate of 500 chosen for each processor. Choice of quantum size is important: if the quantum is very small, very fine grain interaction effects between processors can be modeled, at the cost of slower overall simulation times. Large quantum sizes allow much faster simulation, but may cause odd behavior to be observed in very-tightly-coupled applications. Typically, a quantum size of 500 to 10,000 instructions is a good compromise.

Simulation results for this example are shown in Table 1. All runs generate the same (correct) value for the millionth prime, 15,485,863. The number of simulated instructions differs slightly for single-threaded and MPonMP simulations: both simulation runs are deterministic, but the choices they make about the order in which to handle events are not the same. The runs represent different legal executions of the same application program.

The column *MIPS Rate* is the total number of simulated instructions executed by all cores divided by the elapsed *real time*, in millions of simulated instructions per second. The fastest simulation executes at over 2.7 billion simulated instructions per second on this host machine.

An interesting aspect of the prime search algorithm is that *as primes increase, it generally takes longer to find each prime before it is submitted to the shared database, which requires a synchronizing event.* This algorithm can be used as a tool to establish how effective the MPonMP algorithm is based on frequency of synchronization events between cores. To do this, the prime search application was modified to search for 100,000 primes, starting with a different initial candidate each time. The results are shown in Fig. 3. As expected, the speedup decreases as the synchronization rate increases. Even so, the speedup at the highest synchronization rate is still greater than 1.6 times the single threaded simulation performance.

B. Applications Running on SMP Linux

This example studies the effect of running user-mode benchmarks in a simulation of a virtual platform running a booted Linux distribution. The processor model used here is the OVP model of the ARM Cortex-A57MPx4. The same host machine was used. Once Linux is booted, using a platform capable of booting the same Linaro Linux [6] as the ARMv8 Foundation model and using the FMv1 memory map, a script is run that executes a number of benchmark programs in parallel. The script executes each benchmark a fixed number of times. By default, each benchmark is executed six times. The benchmarks are a Fibonacci number calculation program, the standard Dhrystone benchmark, a program that solves the 8-queens problem, and a program that performs a sieve sort of a large number of random numbers.

One naïve objection to the validity of results presented here might be that the Linux benchmark applications that are run do not communicate with each other: surely MPonMP simulation would not show performance improvement for a set of communicating simulated pthreads, for example?

Although the simulated Linux processes in this example do not explicitly communicate, the cores on which those processors run *do* regularly communicate, to arbitrate access to shared resources of the SMP Linux, such as memory and devices. The MPonMP parallel simulation algorithm makes no distinction between *explicit* communication using user-space constructs such as pthreads and *implicit* communication due to contention for kernel resources: both require correct simulation of exclusive-access constructs.

It is true that communication is relatively infrequent in this example; however, this is very often the case for processes running under SMP operating systems. In addition, the previous example demonstrated that even when communication (synchronization) is frequent, MPonMP can still deliver significant performance benefits.

C. Parallel Simulation Scalability

The previous case studies have used a simulated ARM quad core processor in both bare metal and Linux application scenarios. In each case the host was a four processor x86 PC.

This case study explores how MPonMP scales when the number of simulated cores is increased and when the number of host x86 processors is increased.

Table 1. Simulation results for the parallel primes search application.

Description	Elapsed Time	Simulated Instructions	MIPS Rate	Last Prime Found	Speedup
Single-threaded	99.64s	128,786,424,838	1292	15,485,863	N/A
MPonMP	47.45s	128,786,383,511	2714	15,485,863	2.10x

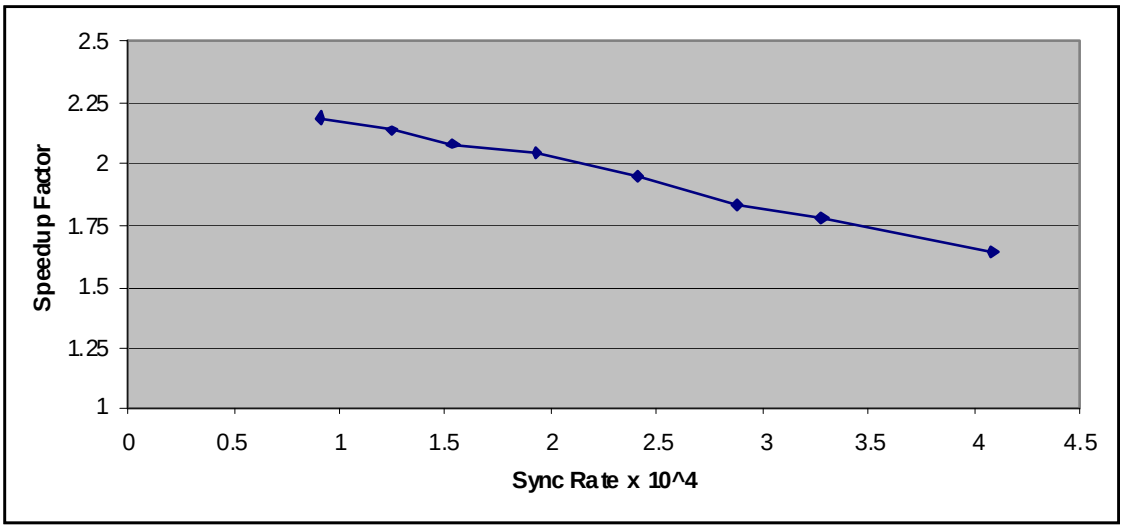

Figure 3. Speedup factor versus synchronization rate for MPonMP simulation.

This example has a platform which includes many CPUs, where each CPU is a single core and has its own program and data memory and has a copy of the program being run, in this case the Dhrystone benchmark. In this example the number of processors instanced in the virtual platform is varied from 1 to 512. The host PC for this example is a 2.7GHz 16 core x86_64.

Fig. 4 shows that as the number of simulated cores increases the simulation throughput increases, until it peaks when there is one simulated core per host x86 processor. After this, simulated throughput gradually declines as more and more cores and simulated memory are added to the simulation. Peaks occur in the graph when the number of simulated cores is an exact multiple of the number of host cores, because this allows the workload to be shared most efficiently between the available cores on the host.

IV. SUMMARY

A new parallel simulation algorithm has been used to enhance JIT-based code-morphing software simulation resulting in increased simulation performance for virtual platforms containing multiple processors. Significant performance increases have been achieved with the parallel simulation technology for both bare metal applications and applications running on the Linux operating system. Additionally, the parallel simulation technology has been shown to scale such that host PCs with larger numbers of processors can be used to simulate virtual platforms with increasing numbers of instanced processor models.

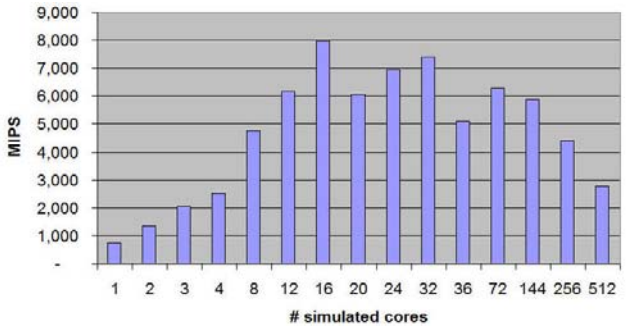

Figure 4. Simulation throughput versus number of simulated cores with 16-core host x86 machine.

REFERENCES

[1] "Just In Time Compilation", Wikipedia article, http://en.wikipedia.org/wiki/Just-in-time_compilation.

[2] Qemu http://wiki.qemu.org/Main_Page

[3] Open Virtual Platforms, OVPsim, http://en.wikipedia.org/wiki/OVPsim

[4] ARM DS-5 Tools (DS500-BN-00004-r5p0-19rel0), http://ds.arm.com/downloads/

[5] Open Virtual Platforms API documentation and user libraries are available at www.OVPworld.org

[6] http://www.linaro.org/downloads/