WHITE PAPER

High Performance or Cycle Accuracy?

You can have both

Bill Neifert , Carbon Design Systems
Rob Kaye, ARM

01/11/2012

ATC-100

Abstract

The tradeoff between performance and accuracy is a perennial conundrum when developing models. Software developers generally need models that provide a high throughput but for developing software that is hardware dependent a greater degree of timing accuracy is required. This paper describes a practical approach to building virtual platforms that enables fast context switching between high throughput loosely timed (LT) models and cycle accurate (CA) models. We then explain the benefits of this approach in the SoC design process.

Meeting today's mobile SoC design and verification challenges

Mobile devices such as smart phones and tablets are now the primary connection to the internet for the majority of users. The humble phone has evolved into a device that is a media player, an HD video recorder, a camera, a game player and a myriad of other applications. This evolution is enabled by increasingly complex integrated SoCs and leveraged by a growing diverse community of software developers creating content.

Supporting these development challenges new design and verification solutions are being created. Underpinning these solutions is the need for models of the various blocks in the design. Depending on the design task there are different requirements on the model in terms of accuracy (level of detail) and performance (throughput, measured as the rate of execution of software running on the model). In a perfect world we would like to have a single model that addresses all these requirements. In practical terms this is not possible as increasing detail will add more compute demands and thus reduce the simulation performance.

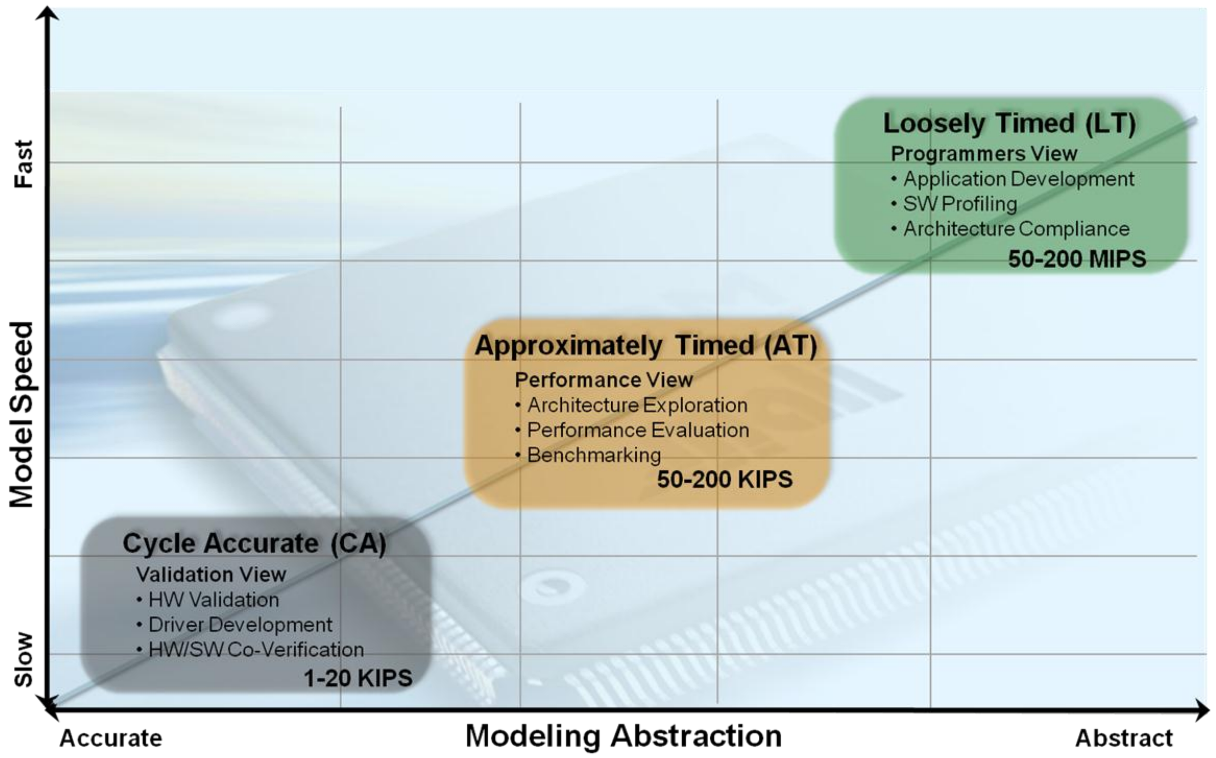

Figure 1: Model Abstractions

Figure 1 breaks the requirements down into three broad areas of application. At the opposite ends of the spectrum we have:

- Validation View: in this application space we need detailed information on the hardware on a cycle by cycle basis. Cycle Accurate (CA) models are required to achieve this level of detail. The expectation is that –f rom a software context – only hardware centric software development will take place as the throughput would not be sufficient for large applications. Booting an operating system would potentially take days of simulation time with we attempted to use CA models

- Programmers View: here the main focus is on development of software more abstracted from the hardware implementation. The focus is on throughput with a goal of running the software as fast as possible while maintaing functional accuracy. Here, loosely timed (LT) models are utilised and these deliver performance of 5-7 orders of magnitude above CA models. At these performance levels a typical OS can be booted in a few seconds to a couple of minutes.

Between these two application areas is the Performance View. In this application area we want to measure and analyse the performance of the system hardware when executing realistic software loads. This requires greater timing accuracy than can be delivered by LT models and greater throughput than can be achieved on CA models. There are broadly two approaches that can be taken to resolving this conflict of requirements. We can either develop a model targetted to this application area - an approximately timed (AT) model – or develop a tools solution that enables the combination of existing CA and LT models. The latter is attractive as it reduces the workload on the model developers (typically the IP providers) and enables re-use of models developed for validation and software development. As the user population working in the performance view space is small compared to the validation and programmer spaces the ROI of developing specialised models is prohibitive and amortisation of development cost across a larger user base is desirable.

In this paper we focus on the integrated LT/CA paradigm to address the needs of the users in the performance view space.

Loosely Timed (Programmer’s View) Models

First, let us look at the Programmer’s View application area and how models are developed and applied to this segment of the overall design requirements. The key requirements are:

- Development of the hardware centric software layers. For this the integrated LT/CA platform that we outline later in the paper is required.
- Development of software prior at the same time as the hardware so that firstly software and hardware are considered together during the design process and secondly the parallelisation reduces the overall development cycle and speeds time to market.

To enable this, models are employed at all stages of the design process. This starts during the development of the processor architecture itself before we start to consider the deployment of the processor into designs. During this stage of the process a model called the ”Architecture Envelope Model (AEM)” is implemented. This can be considered to be an executable representation of the architecture specification. The AEM is utilised to validate that any design of a core is correct against the specification.

The AEM can then be characterised into a specific CPU core by adding implementation specific details to the model and fixing or restricting parameters to the implentation where the architecture specification allows a greater range or optional features.

Both the AEM and the core model have the same key requirements:

- They must be high performance to enable fast software execution
- They must be accurate to the processor and architecture functionality and contain enough implementation detail that software can run unchanged on the model and the target hardware (whilst minimising the impact on performance.

Additional requirements are early availaility to enable software development pre-silicon and the ability to integrate into design environments. This enables both the delivery of complete platforms with models from different sources and at different abstraction levels along with the utilisation of the design, debug and analyis capabilities that these solutions deliver.

In order to achieve the required levels of performance the technology used to implement most LT models today is Code Translation (CT). In CT models the code sequences of the target CPU are translated on the fly to equivalent code sequences for the host workstation. These are then run natively on the host workstation (typically an X86 based machine today). This leads to performance levels orders of magnitude higher than interpreted instruction set simulators. There is a correlation between the size of the code fragment translated and performance: translating longer code sequences will generally produce higher performance at the cost of controllability.

To obtain the highest performance we only want to translate code sequences once. For example translating a subroutine or function each time it is called, or translating loops each time they are executed, will degrade performance. To mitigate this the CT model maintains a pointer to the host memory for each translated sequence. Whenever it can do so the model then executes the previously translated code. This will not always be possible (for example, self-modifying code).

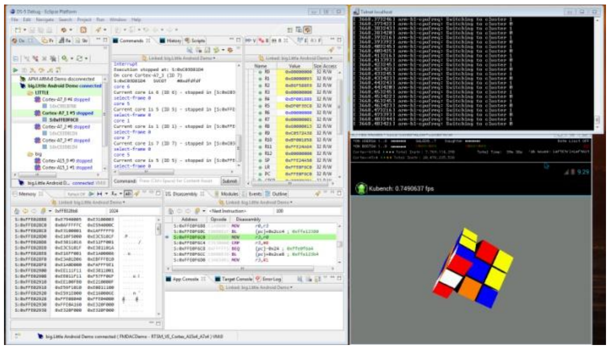

Figure 2: Programmer's View Model example platform

The levels of performance achived can vary greatly. The model usage, the code profile and the capabilities of the host workstation will impact the model performance to differing degrees. For example with a modern (> 2GHz CPU) workstation, running Android™ on a model of ARM® Cortex™-A15 processor x1 you can expect to see throughput in the region of 50-100MIPS.

Performance View Requirements

The previous section looked at the use of LT/PV models from a purely software perspective. The modeling requirements for the performance view are such that we cannot use the LT models unmodified. They do not carry sufficient timing information to make them useful.

Additionally the degree of timing accuracy required depends on the function of the IP being modeled. For example models of interconnect require a higher degree of timing accuracy than a CPU. In the programmer's view this level of detail in the interconnect model is not required and detrimental to the simulation performance. The different degrees of accuracy required in the various models push us further into the hybrid platform environment where models at different levels of abstraction can be merged.

We will now look at the requirements for Cycle Accurate models and the techniques that can be employed to build effective platforms that integrate CA and LT models.

Cycle Accurate (CA) Models

The term “cycle accurate” is another in a long line of seemingly self-explanatory EDA terms, which have somehow morphed into having multiple meanings. Depending upon the source, a cycle accurate model can be a variety of things:

- A model which responds to transactions with a cycle count (which corresponds directly to the SystemC TLM-2.0 definition of approximately timed models) but does not present results every cycle. It cannot be used as a replacement for an RTL representation of the model without transactors and pin adaptors.
- A model which interacts with its environment every cycle but is based upon a functionally approximate version of the implementation of the model for the internals. This means that although the results are presented every cycle they may not be the actual results, which the actual piece of IP would generate, but there are results every cycle and they are functionally correct. It can sometimes be used as a replacement for the RTL representation of an IP block but will typically require pin adaptors. Since complete functionality is not guaranteed however there may be some interaction complexity with other models related to flow-through paths or un-modeled transactions (typically speculative execution, cache fills, etc.)
- A model which corresponds completely to the behavior of the actual IP being modeled. It creates correct results every cycle and creates functionally correct results for each cycle. It models all of the actual cycle behavior of the IP block including speculative execution, cache line fills, etc. It can typically be used as a replacement for an RTL representation although pin adaptors may be needed.

The committee that drafted the TLM-2.0 specification stayed away from defining a standard for cycle accurate models as they had with loosely timed and approximately timed transactions. (Although there were heated discussions on the topic) While this may seem odd, the varying industry definitions presented above certainly indicate some of the underlying issues with creating this description. In addition, since cycle accurate models necessarily describe the cycles of a specific protocol, it would be impossible to specify fully the cycle accurate behaviors of all protocols without IP donations from the various owners.

For the purposes of further discussion in the paper, cycle accurate models will refer to the latter definition of cycle accuracy as presented above. Models of this type are typically generated from the RTL description to maintain the complete internal accuracy necessary to create the best cycles but they can in some cases also be hand-generated or created as part of a high-level synthesis flow.

We talked earlier about the various levels of models in the system: loosely timed for speed, cycle accurate for accuracy and approximately timed for a little bit of both. In the ideal world of course you would need to have various models. They would all be 100% accurate and represent the actual behavior of the underlying hardware. This way you would not need to worry about the accuracy of your system, you could make accurate design decisions and write software knowing that it would run on the real system.

These varying models exist for a reason though. The first and most obvious being that RTL may simply not exist at an early point in the design cycle and the design itself may only be in its initial stages. Obviously, you can’t model something which doesn’t exist yet. Even for IP blocks, which do exist however, there is still a need for higher-level models. The execution speed for accurate models just is not sufficient to do real software development. Many times, software engineers will complain that the actual system itself is not fast enough for their development. Imagine how much they will complain about an accurate model! There is a need to have a model for software engineers, which runs as fast as possible.

Another factor limiting model accuracy is the time required to develop the model. Model functionality is the bar for entry here; all models should be 100% functionally accurate. Injecting cycle accuracy however can be a huge time sink especially when the need for validating the model against the implementation is factored into the equation. It is easy to say that you can stop short of 100% accuracy but knowing when that design tradeoff will actually be important or not is an inexact science. We will talk more about the limitations of stopping short of 100% accuracy.

For ARM IP, there is a family of cycle accurate models which implement 100% of the RTL functionality since they are compiled from the source RTL provided by ARM instead of written by hand. The models are instrumented to take advantage of all the virtual prototype features that are traditionally needed including interactive software debugging and performance instrumentation on the pipelines, caches and memory buffers. Equally importantly though, these models are created to be interchangeable with the Fast Models that ARM provides.

These models are available for pretty much all of the ARM core library and accompanying peripheral models are supported as well. To enable easy adoption, integration and interchange with ARM Fast Models, most processors are accompanied by a virtual reference platform implementation and software for benchmarking and OS booting.

We have talked a lot about models at both ends of the abstraction spectrum so far and have not really talked much about the space in between occupied by approximately timed models. You will see that this corresponds well with the models that are typically available in the market for IP blocks. LT models exist for processors and CA models are available for a vast majority of them. AT models used to be more prevalent, but as processors got more complex, companies stopped providing them. Each one can speak for itself as to why they made that decision.

In customer designs that I have seen they have made a similar move away from AT models for existing IP blocks. Note that this is different from new IP blocks. AT models can be a valuable addition to a high-level synthesis flow since they aid in the design refinement process. In most SoC designs however, the majority of the design is either being purchased from third party suppliers such as ARM or reused from previous projects. In these cases, the value of AT models is much less evident. An LT model can be developed rapidly to model functionality for the software engineer. A CA model can be created directly from the RTL implementation. AT models are the odd-man out. They require substantial design and validation effort to create and maintain. In addition, since they are not 100% accurate, it is difficult to make confident design decisions.

Of course, PV/LT and CA models have limitations as well. PV/LT models are great for developing high-level software but you cannot really use them to do accurate analysis or low level development. They just do not generate bus traffic that is sufficiently representative of the actual core to enable good design decisions. CA models have a similar limitation in that they have all the details needed in the system by they simply are not fast enough to develop high-level software. Booting an OS is an overnight affair and if you want to start up a browser as well then you need to have a few days on your hands.

You can address this problem by partitioning each model into its area of strength and then leveraging that model properly with other models and capabilities to address the problem at hand. This can be done in a few different ways.

Integrated Platforms with CA and PV models

If the primary aim is to develop firmware, combining PV and CA models in a single platform can be a very effective means to get up and running quickly and eliminate the need to create multiple models. The best, and fastest, approach for this is to maintain the processor/memory subsystem in the PV domain and then use CA models where needed. This enables the majority of the system to execute at PV model speeds the majority of the time. Whenever a CA model is active, the whole system will execute at CA model speeds, which reinforces the statement above about maintaining the processor in the PV world, if possible. If there is no LT model available however, then a CA model may be your only solution regardless of where the model is in the design.

Using PV and CA models in a single virtual prototype is well suited for some tasks but you can still be limited by the speed of the CA model. If you have LT and CA models for the components in your system, however there is new technology which lets you take advantage of the best features of both. You can use the LT representation of the system to execute to a point of interest such as a software breakpoint or arbitrary point in time and then swap to a

CA representation of the system to continue execution. This enables the designer to quickly get to a point of interest, be it booting an OS or other important section of code and then change over to a 100% accurate representation to continue execution. Since these models are obviously at different abstraction levels, there are steps, which must be taken in order to ensure a successful swap.

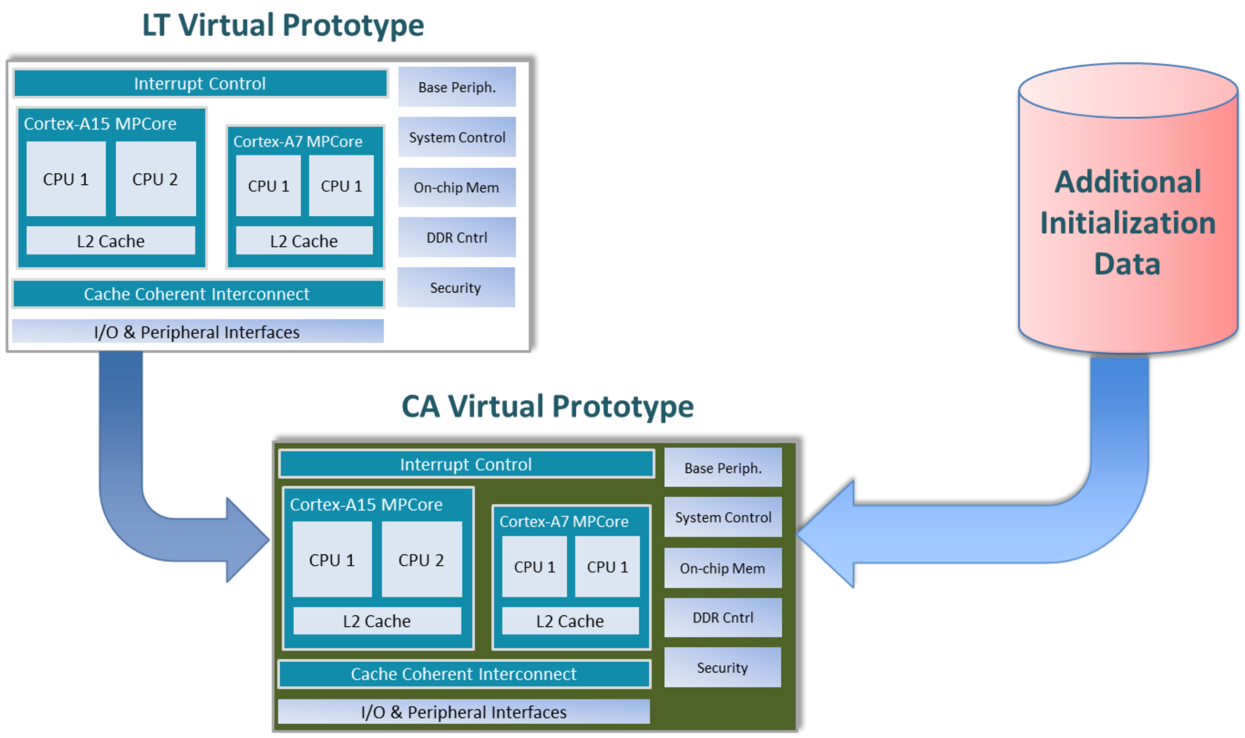

Figure 3: Integrated PV/CA Platform

The swap from PV models to CA models is done by creating a checkpoint of the system. This is not a typical save/restore checkpoint however since the representation of the system is changing underneath. A simple binary dump of system state is not sufficient. Instead, we need to look at the system as a collection of model and manage the swap that way. This brings up the first important rule: all models in the system need to support some sort of check pointing capability either as an architectural checkpoint where the needed state elements are saved or as a binary checkpoint where the representation of the model is not swapped and will instead simply be restored in the next system representation.

To create a checkpoint from an ARM Fast Model system, we extract the needed state data via the cycle accurate debug interface (CADI) which ARM publishes and makes available via is ARM ESL APIs. In all but the most trivial cases, you cannot simply dump this state into the CA representation and continue execution. The Fast Model executes a single instruction at a time with no pre-fetching and no pipeline. The actual processor implementation is typically substantially complex. Since we do not have data to fill all of this state, it should at least start executing in a consistent state so that it can get the correct answer. In addition, not all registers in a piece of IP are simple storage elements. Many of them have side effects from being written (in fact, some of them are JUST side effects and have no physical storage). The side effects must also be modeled to enable correct execution.

Swapping is a one-way process. It is tough enough to put five pounds of data into a ten-pound bag, deciding what data to throw out on the way back is much more difficult. Besides, instead of swapping back, you can simply restart the PV system and re-execute to the point of interest.

The swap itself is managed using the ARM ESL APIs. This means that it is not just for ARM Fast Models, any model, which implements the ESL APIs is a candidate to be swapped. Care must be taken however to ensure that there is sufficient state information in the PV model to go into the accurate model. Any data not in the PV model will need to be mapped in independently in order to continue. Keep in mind that this is purely an architectural swap, which means that only state elements are making the trip across the implementation boundary. If the model relies upon threads or dynamic content, it will likely not be carried across and will cause mismatches. Finally, if you are creating your own swappable model target you need to make sure that the CADIWrite command for any register correctly models the associated side effects.

As might be imagined, testing of the swap functionality is a very important task. Register mapping and side effects can be largely done by inspection but there is no assurance that the IP will respond well to just any random checkpoint. In order to ensure that the swapping is managed correctly, a cohesive test strategy is needed. This is most easily done by creating random checkpoints during program execution and then continuing execution from these test points until the program is completed. At that time, there are various methods to validate the correctness of the end result. Note that intermediate checks can typically NOT be done due to the difference in execution model between the PV and CA models. Writes can occur in different orders, caches can be modeled differently and instructions can execute in different orders as well. The only way to guarantee correctness is to look at the end of simulation. This makes it even more important that multiple, random checkpoints are extracted for testing.

Example application of the CA/PV integrated platform and results

We will now look at an example of this approach, explain how it is implemented and look at the results.

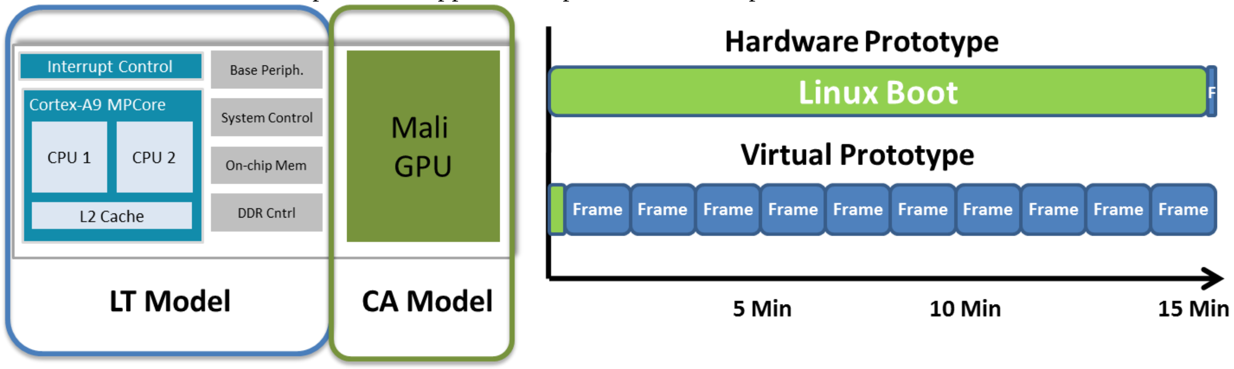

Figure 4: Partitioning the platform for speed and accuracy

Before discussing the results for the LT to CA swap, let us revisit the first case of LT/CA model interaction. For this customer deployment example, an ARM Mali™ GPU was Carbonized and linked together an ARM Fast Model representation of the system. Variations of this exact setup have been deployed at multiple customers, both inside of commercial virtual prototype environments and homegrown system environments. Speed results for all of these were similar but construction time for the homegrown environments was typically substantially longer.

In this example, the processor/memory subsystem is sufficient to boot the Linux OS and get to a prompt. Typical boot time is approximately 15-20 seconds both with and without the GPU present. The number of cycles during which the GPU is accessed during the boot up is sufficiently few enough that the slower CA runtime does not impact simulation. When the system starts processing graphics frames the speed definitely goes down as the CA model now plays a much greater part. Typical frame processing time is around 90 seconds. While you are obviously not going to start watching movies on the CA model, it proved to be a valuable model to debug numerous software setup and configuration issues. By way of comparison, the hardware prototype, which the customer also assembled could process frames much more quickly, in a second or two, but also took nearly 15 minutes to boot Linux. This meant that by the time Linux was booted, the virtual prototype had already processed 9 or 10 graphics frames.

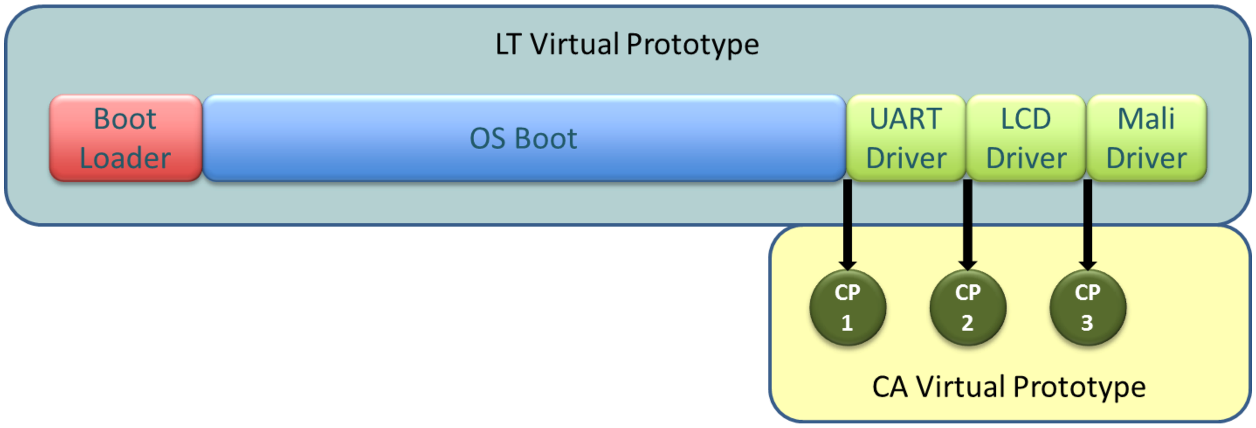

Figure 5: Swap Enabled Platform timeline: applying checkpoints

Running software in a swap-enabled system looks pretty much exactly like running software in any other virtual prototype. When executing in the LT/PV representation of the system, the user executes at exactly the same speed as in a system that is not swap capable and has access to all of the same functionality and debug capability. The primary difference is that the system representation can be swapped to 100% accurate at any point of interest, typically a software breakpoint. If a non-software breakpoint is chosen to halt the system then a small synchronization step is necessary to get the registers in the Fast Model into an examinable state. Breakpoints are typically chosen where they can be utilized for tasks requiring more accuracy. In the diagram above, the points of interest are the starts of various driver code sections inside of the OS kernel. Note that you do not need to do a separate run for each checkpoint. A single Fast Model run can create multiple checkpoints, which can then be simulated independently, in this instance, by driver developers but they can similarly be handed off to engineers debugging detailed interactions of hardware and software, which can only be seen when the system is modeled accurately.

Since each checkpoint is enables 100% accurate execution, there's no need to just take a single checkpoint. You can actually take multiple checkpoints and use them to break up what would be a long simulation if done in a single threaded CA environment into a much faster result. In this case, you use the LT platform to create a series of checkpoints, which can then be executed in parallel in the 100% accurate environment. The results from these multiple runs can then be aggregated together to deliver accurate results for tasks such as performance analysis, power optimization, etc.

We have been discussing a lot of concepts and theoretical ideas in the presentation. At a practical level, the swap capability from ARM Fast Models to Carbonized ARM models exists today and is in active use at numerous customers. Execution time of the both the fast and accurate systems are unchanged from the check pointing and creation and loading of the checkpoint is typically a sub ten second operation.

As you might imagine there are some side effects from making the jump from an untimed simulation environment to a cycle accurate one. Currently caches contents do not survive the transfer. This is an area of investigation for future implementation. Similarly, since the Fast Models do not model speculative execution or branch prediction the core starts with an empty training queue for branch prediction. Thankfully, none of these artifacts affects the functionality of the overall system and the impact on cycle count vanishes quickly.

This functionality exists today for the ARM Cortex-A15, Cortex-A7 and Cortex-A9 processors. Traditional peripheral models for each core are also supported.

Summary

The swap capability we have discussed in the paper presents a methodology, which minimizes the time required to create and validate virtual prototype models for ARM IP. By leveraging the Fast Models from ARM and 100% accurate Carbonized ARM models , it is possible to have a single virtual prototype which is capable of executing at 50-200 MIPS for software debug tasks while maintaining the capability of executing with 100% accuracy to enable firmware development, architectural exploration, performance analysis and system debug. High performance or cycle accuracy: you can have both!