SYNOPSYS®

WHITE PAPER

Fast Cycle Approximate Simulation Using ARC nSIM NCAM

Key Facts—Methodology—Use Cases

Authors

Igor Böhm
Software Architect and Technical Lead, Synopsys

Alexander Chuykov
Applications Engineer, Synopsys

One of the key factors of successful software (e.g. firmware/application) development is the ability to quickly run and profile software in the absence of target hardware. The earlier in the design process that this is possible, the better, i.e. during the pre-silicon phase.

Typically, the pre-silicon phase is dominated by three activities, each with different challenges:

- Exploring the Hardware/Software configuration space
  - Challenge: to find the near optimal set of configurations from a large design space
- Functional bring-up of the software stack
  - Challenge: to find and eliminate functional software issues
- Pre-silicon optimization of the software stack
  - Challenge: to find and optimize key hotspots in software stack

The DesignWare® ARC® nSIM Instruction Set Simulator aims to address these challenges by providing two simulation modes; a fast-functional mode and a near cycle accurate mode. The fast-functional mode is key to solving the pre-silicon software bring-up challenge. Our new near cycle accurate simulation mode was designed to help solve the challenges of exploring the hardware/software configuration space and to enable pre-silicon optimization.

The end goal is to empower customers to further reduce time to market by providing the right tools to get their software stack production ready before silicon is available and to reduce the risk of costly re-spins due to late design changes.

ARC nSIM NCAM—Hardware Performance Modeling

Although nSIM, itself, is key for early stage software development and bring-up, it is unable to predict micro-architectural performance, as it is a purely functional instruction accurate simulator. However, predicting the target performance of a customer's software stack is important - it affects hardware configuration decisions such as CCM sizes, cache associativity, and branch predictor configuration.

To extract this type of information, customers have been advised to either use a cycle accurate model or resort to FPGA prototyping.

While these approaches are accepted and widely used in industry, they present some risks and challenges:

- Chicken-and-egg problem:
  - Final hardware configuration options depend on the software stack executing on target hardware
  - However, at the time of finalizing hardware configuration options, customers usually do not have RTL, let alone the software stack, in a state to be able to evaluate and explore configuration options. Although risky, it is common to apply domain expertise and extrapolation to settle on a hardware configuration.
- Prohibitively long simulation times:
  - RTL simulation and cycle accurate models derived from RTL are prohibitively slow and are therefore only suitable for small kernel testing. However, the final software stack is much more complex than those kernels. This, in turn, increases the risk of basing decisions on unrealistic conditions, resulting in sub-optimal hardware configuration choices, followed by increased development costs due to delays in schedule.
  - The turnaround time of trying different hardware configuration options using RTL simulation or cycle accurate models is too long. To add to this challenge, it is necessary to rebuild RTL simulation models when configuration options are modified.

The nSIM Near Cycle Accurate simulation Mode (NCAM) has been designed to fill the gap between:

- Fast instruction accurate simulation providing very limited micro-architectural performance detail
- Cycle accurate simulation approaches that are either too slow (simulation speed), inflexible (high re-configuration cost), or unable to run the full software stack (resource constraints)

What is NCAM?

On top of functional (i.e. architectural) simulation, ARC nSIM provides NCAM, an approximate micro-architectural simulation mode capable of estimating target processor performance (e.g. cycles, clocks per instruction (CPI), branch prediction miss rate). This simulation mode enables customers to estimate and optimize the performance of their final software stack running on ARC hardware before signing off on a hardware configuration, long before silicon is available, and without the need for FPGA prototyping and simulation.

- NCAM is not a separate simulator:
  - NCAM is a hardware performance simulation mode of ARC nSIM

    NCAM is not derived from RTL:
- NCAM is derived from micro-architecture specifications therefore it is cycle approximate
- NCAM predicts hardware/software performance
  - NCAM is a hardware timing model exposing KPIs (Key Performance Indicators)
  - # of cycles
  - # of cache misses
  - # of branch mispredicts
  - …
- NCAM is fast
  - CoreMark on NCAM (cycle approximate) takes 3 seconds
  - CoreMark on xCAM (cycle accurate) takes 5 hours

Benchmark	ARC nSIM functional simulation	ARC nSIM NCAM cycle approximate simulation	ARC xCAM cycle accurate simulation
CoreMark	12 MIPS	3.5 MIPS	0.005 MIPS

Figure 1: CoreMark MIPS comparison between NCAM and xCAM

How Accurate is NCAM?

NCAM is derived from micro-architectural specifications describing instruction timing behaviour at an abstract level. Implementing the micro-architecture timing model at a different abstraction level compared to cycle accurate simulators is the key to high speed simulation.

As NCAM is not derived from RTL, it is not cycle accurate. It is cycle approximate, i.e. performance predicted by NCAM might not exactly match performance reported by a cycle accurate simulator derived from RTL. In other words, a program executed on a cycle accurate simulator can produce a different cycle count on NCAM.

So, where is the approximation in NCAM? To give insight how approximations are introduced, consider that an abstract instruction timing model is unlikely to capture all corner case behaviours that the underlying micro-architecture can exhibit. One typically encounters corner case scenarios when operating hardware under sub-optimal conditions (e.g. high branch prediction miss rate). In such situations, the performance predicted by NCAM is likely to be further off compared to a cycle accurate simulator.

While there are no guaranteed accuracy ranges that hold for any application, we have put safeguards and a methodology in place that helps to identify these cases, thereby providing a level of confidence in the obtained results.

Why Use Cycle Approximate ARC nSIM NCAM Technology?

- High Simulation Speed
  - ARC nSIM NCAM is orders of magnitude faster than cycle accurate models (minutes vs. days). It is designed to simulate the full software stack and real workloads, not just small functions or kernels with reduced input
  - High simulation speed directly translates into faster feedback. That, in turn, reduces design and development times
  - Simulation of the full software stack using real workloads greatly reduces the risk of making sub-optimal hardware configuration decisions based on extrapolation from small kernels
- Flexible Integration
  - As NCAM is a simulation mode of ARC nSIM, it integrates into all applications and use cases that are supported by ARC nSIM such as SystemC (OSCI/Acellera and Synopsys Virtualizer), Platform Architect, MetaWare Debugger, and many others
  - The integration in the MetaWare Debugger enables advanced automation capabilities via the MetaWare Debugger scripting language, enabling automated extraction of KPIs for any code region of interest
- Advanced Profiling
  - NCAM provides rich profiling capabilities designed to help you find the needle in the haystack quickly
  - In conjunction with the MetaWare Debugger, it is easy to drill down and attribute important KPIs to functions, statements and individual instructions. As an example, it takes very few clicks to find the function with the most instruction cache misses and drill down to the basic block of instructions where the misses occur.

Dealing with Uncertainty and Approximation

The key ingredient to deal with approximation and uncertainty is a methodology that allows one to judge the confidence in predicted results coupled with the basic knowledge of processor microarchitecture:

The most fundamental guidelines of our recommended NCAM methodology are:

- Do not track and rely on a single key performance indicator alone to detect if there are performance issues
- Instead, track and use a set of key performance indicators to discover performance issues

Cycle Approximate NCAM Simulation Methodology

To demonstrate why following these guidelines is essential to success, let's consider an example where we ignore the recommended NCAM methodology of considering multiple KPIs and just rely on a single KPI. In our example, we try to determine if deadlines for key functions in a hard real-time system can be met when simulating various workloads on a given set of hardware configurations. This information is key in deciding which is the most promising hardware configuration for a given software stack.

Considering only a single KPI might give you the false impression that the deadline can always be met. You may then falsely assume that no further optimisation of your software stack is necessary. What has just happened is that you have put a lot of confidence into a single key performance indicator without any additional evidence supporting that type of strong confidence. There is a real risk that you might be in for a bad surprise once silicon is available, revealing that there are instances where key deadlines cannot be met, and you have to delay your schedule to try to optimise the software stack very late in the process.

The solution to this problem is to follow our recommended NCAM methodology and consider multiple KPIs (e.g. # of cycles, branch mispredicts, cache misses, etc.) to gain confidence in the predicted performance that is backed up by additional evidence.

How can evaluating multiple KPIs improve confidence in the obtained results?

Because it is highly unlikely that all KPIs are within range and program performance is suboptimal. In other words, if one KPI looks suspicious, then the likelihood that the accuracy of other KPIs is lower increases. Therefore, considering additional KPIs provides you with important additional warning signs.

For example, consider at least three KPIs: (1) number of cycles spent, (2) number of cache misses occurred, and (3) the number of mispredicted branches. If cycle counts are within range, cache misses are reasonable for the given application domain, but the branch misprediction rate is very high, you must investigate and figure out why that is and try to optimize the code in that regard.

In this particular case, it may have happened due to a lack of inlining and the presence of many small method calls (e.g. in C++) that are declared in the header but defined (i.e. implemented) in the respective implementation (i.e. .cc) files as mandated by many good software engineering guidelines. With NCAM, it is easy to detect that this is happening. Once the cause of the performance issue is identified the measures to rectify it are readily available and range from using special compiler optimisations (e.g. link time optimisation) to moving critical small method implementations into the program header allowing the compiler to make critical inlining decisions.

Discovering Performance Issues

The key to success is to be able to quickly find the regions in code that exhibit potential performance issues. It is essential to have the right tools that help to quickly find the needle in the haystack. The picture below demonstrates how it all works by using the MetaWare Debugger (MDB) in conjunction with nSIM to debug ARC applications in either GUI (like below) or command-line/ batch mode. In the bottom left corner, the Profiling pane displays NCAM counter values on a per function basis. (It is also capable of building runtime call tree which is particularly useful when you have a deep call stack and need to figure out which of the subfunctions is the bottleneck). The dropdown menu shows some of the counters that can be enabled.

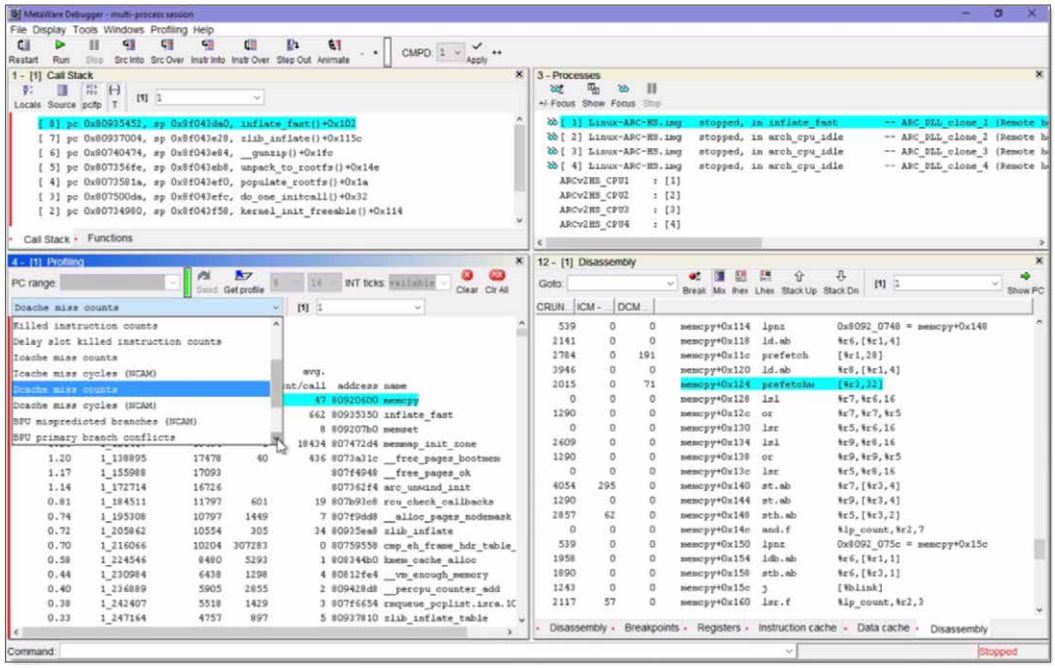

Figure 2: MDB and NCAM in action

Another feature is the Source and Disassembly panes. The Disassembly pane is in the bottom right corner. Apart from the actual disassembly, it can show the counter values attributed to a single instruction. Those are CRUN, ICM and DCM columns on the left-hand side representing cycle counts, instruction and data cache misses respectively.

So, let's check for another KPI which is mispredicted branches available from the dropdown lists in both Profiling and Disassembly panes.

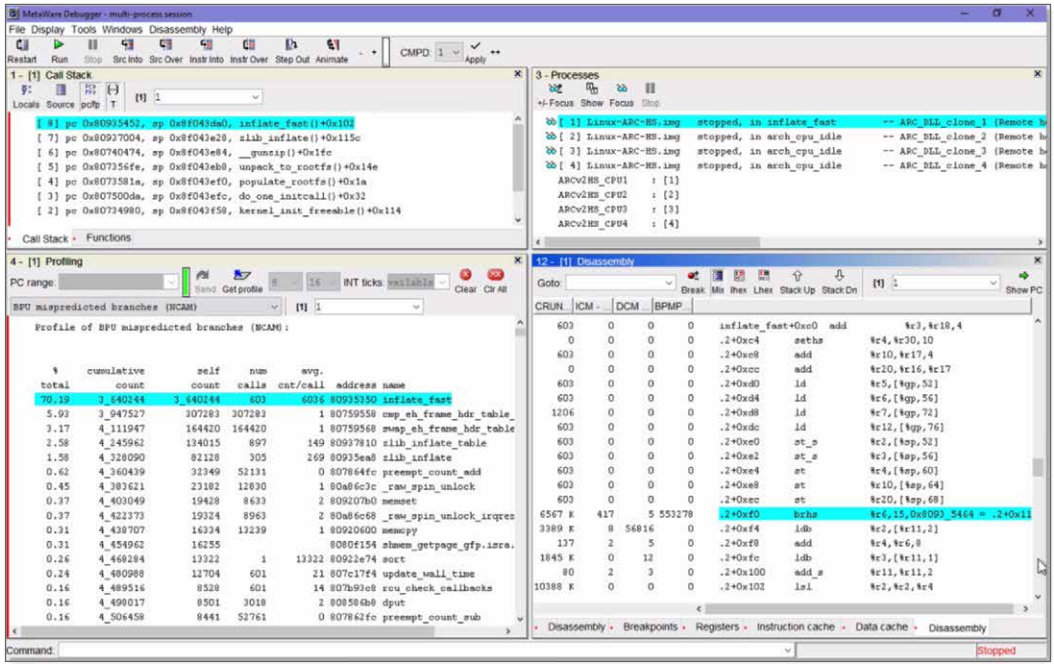

Figure 3: Checking branch mispredictions count with MDB and NCAM

Two steps are required: first check the top contributing function using the Profiling pane. Then narrow down further using the function name symbol and Disassembly pane. As you can now see, there is obviously a bottleneck around a branch instruction which shows extremely high BPMP (i.e. branch mispredict) count. Thus, you get to the problematic code sequence in literally three clicks. The next step involves figuring out why it's happening and tune the C/C++ or assembly code to get rid of the stalls on that specific instruction (probably checking if/else or for/while statements, applying some compiler controls, reorder the code etc.). Repeating this exercise for other frequently executed functions will ensure that performance is good and there will be no surprises when testing the same code on silicon.

ARC nSIM NCAM—Use Cases

To generalize and summarize the NCAM use cases let's go through a couple of real-world examples that we at Synopsys are faced with on a regular basis.

Exploration of the Hardware Configuration Design Space

With ARC nSIM NCAM you can quickly check which of the hardware configuration options affect the target CPI (Cycles Per Instruction) and summarize it in a spreadsheet or other convenient form to estimate the impact of hardware configuration changes on locations of code you are interested in. This is key early input that helps drive the decision to arrive at a given hardware configuration.

- Example: Explore how to dimension/configure the Instruction cache (I$), Data cache (D$) and L2 cache (L2$)
  - Capacity Miss:
    - Increase Cache Size
  - Conflict Miss:
    - Increase Cache Size
    - Increase Associativity (more ways)
    - Decrease Line Size (more sets)

			Benchmark	Benchmark 1		
			Function	Critical Function 1	Critical Function 2	Critical Function 3
#	Item	Configuration Experiment	Input	% change in CPI	% change in CPI	% change in CPI
1	ICCM	Increase ICCM size	Medium	-1%	0%	-7%
2	DCCM	Increase DCCM size	Medium	-5%	0%	-3%
4	I$	Increase cache size	Medium	-20%	-11%	-4%
5	I$	Increase associativity	Medium	-12%	-5%	-7%
6	I$	Decrease line size	Medium	17%	50%	32%
7	D$	Increase cache size	Medium	-4%	-2%	0%
8	D$	Increase associativity	Medium	-1%	0%	0%
9	D$	Decrease line size	Medium	5%	1%	1%
10	BPU	Increase branch cache size	Medium	-5%	-7%	-9%
11	BPU	Increase size of Return Address Stack	Medium	0%	0%	0%
12	BPU	Increase size of Pattern Table	Medium	0%	-1%	0%
13	CSM	Increase CSM size	Medium	1%	0%	0%
14	L2$	Increase cache size	Medium	0%	-1%	0%
15	L2$	Increase associativity	Medium	0%	0%	0%

Figure 4: Tracking the hardware configuration options changes with NCAM

- Example: Explore how to dimension/configure BPU
  - Capacity Misses
    - Increase Branch Cache Size
  - Larger Return Address Stack (RAS) means deeper subroutine nesting levels can be predicted correctly
  - Larger Pattern table size means more global branch history can be tracked for more branch or jump instructions at the same time
  - Example: Explore if critical handlers in a real time system require
  - CCMs
    - Predictable performance—expensive
  - Cached architecture
    - Less predictable performance—cheap

Exploration of the Compiler Optimisation Space

Apart from obvious compiler optimization options such as -O1, -O2, -O3 there are many more specialized options allowing you to specifically tune for

- Code Density
- Performance (inlining, unrolling)

The interaction of these options is non-trivial for large and complex software stacks, especially in the context of a hard-real time system where some functions have real time constraints (i.e. must complete within a time budget) and some can be interrupted any time.

Using ARC nSIM NCAM, you can systematically explore the compiler optimization space using the full application to gain insights into the best combination of optimization options to derive information such as the following table that shows the impact of compile and link time optimizations.

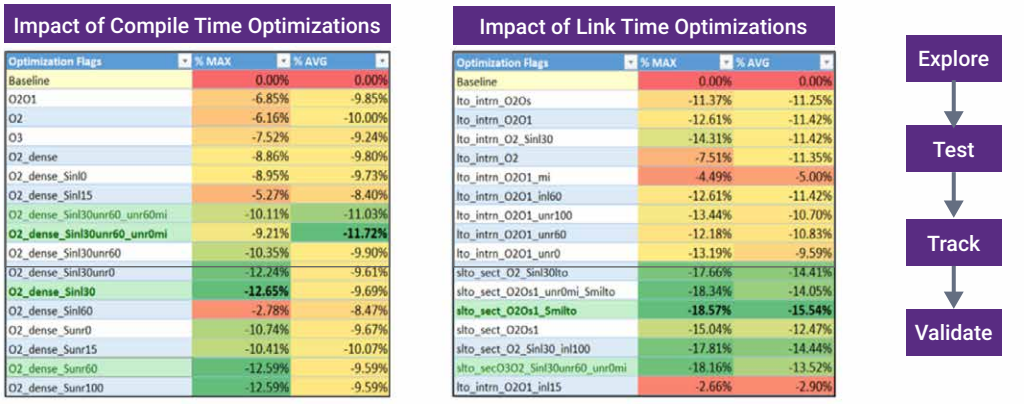

Figure 5: Tracking the compiler options changes with NCAM (table view)

This is particularly useful to track and visualize the performance for such an experiment for a critical function over time to see if the performance envelope can still be met during all application phases (i.e. cold start, warmed up, spikes in workload).

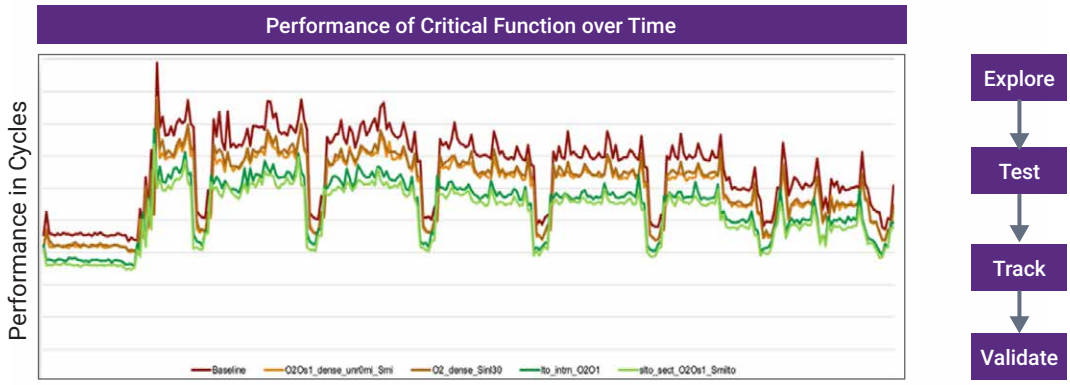

Figure 6: Tracking the compiler options changes with NCAM (graph view)

In the figures above you can see that the -O3 optimization does not provide the best optimization for higher performance. Obviously, the compiler optimizer generates code as fast as possible for -O3. Usually, it is a tradeoff between performance and code size—aggressive loop unrolling, functions inlining, etc. If we run this application on pure nSIM (instruction accurate model), we can give best score for -O3—less subroutines calls overhead. But NCAM is taking into account more things including BPU and caches. Thus, the aggressive code unrolling is not very good for this function. The function has large code size and penalty of instruction cache miss may be higher than subroutine call or keeping loop. In summary, for the complex software optimizations methods, it may be not obvious and require multiple passes to the find best options. Moreover, different functions in the same application may have opposite requirements for optimization.

As you can see, fine tuning the compiler options requires much effort and test case evaluations and NCAM could reveal some non-obvious things when for example increased number of instructions does mean better performance. You could then pick the best options and run them on the slow cycle accurate model to verify and sign-off the best choice.

Generic flow to determine the best set of compiler options

- Select a baseline for optimization options
  - Limits the application performance budget and code size
  - Choose KPIs for application
- Run profiling
  - Get scores for KPIs
  - Find bottleneck functions

- Tune optimization options. Optimization methods depends on KPI and application targets. For example:
  - A lot of Instruction cache misses—optimize cache settings (hardware optimization); decrease unrolling/inlining level or relocate/shuffle functions (software optimizations)
  - A lot of Data cache misses—optimize cache misses (hardware optimization); relocate or shuffle data objects, use prefetch (software optimizations)
- Repeat steps
- If your application has different operation modes, switch to another mode and repeat all steps

This flow allows one to find a near optimal set of hardware configuration (if applicable) or toolchain and software optimization options. Apply this flow along with NCAM for full scale applications or a large set or functions. For small individual functions and kernel (e.g. micro benchmarks), it is better to use cycle accurate ARC xCAM or other cycle accurate simulation strategies (e.g. FPGA).

ARC nSIM NCAM—Conclusion

Cycle approximate NCAM simulation has proven itself to be very powerful, having become an invaluable part of the hardware/software co-design and software development process. Despite this success it is important to understand its limitations as there are situations when using NCAM is simply not recommended in cases where the required level of confidence in results obtained is very high (e.g. close to mandating cycle accuracy). The following simple rules can help to see if NCAM is applicable or not:

When to Use NCAM

- Explore the hardware configuration space
  - Quickly find the set of good hardware configurations for your application
  - NCAM is easily configured for quick experiments
- Application optimization (pre- and post-silicon)
  - Quickly find functions that dominate the runtime
  - Focus effort where it makes the difference
- Compiler optimization
  - Systematically explore large compiler optimization space
  - Find the good set of optimization options

When Not to Use NCAM

- Profiling tiny micro-benchmarks
  - Micro-benchmarks typically explore hardware corner cases and high accuracy is required
  - NCAM is specifically not designed for this use case—use ARC xCAM, RTL simulation or FPGA instead
- Doing performance sign-off
  - Never go to tape-out without performance sign-off executed on a cycle accurate model
  - NCAM is approximate, sign-off requires certainty and accuracy—use ARC xCAM, RTL simulation or FPGA instead

04/07/21.CS611699176 SG-nSIM WP.

Pub: January 2021