Caching Tool Run Results in Large-Scale RTL Development Projects

Ashfaq Khan
Intel FM6, 1900 Prairie City Rd, Folsom, CA, USA
ashfaq.khan@intel.com

***Abstract* - Caching is a widely used technique to improve efficiency in both software and hardware, including EDA tools and compute/storage infrastructure. But there is significant untapped potential when it comes to sharing tool run results within a large-scale RTL development team. Members of such a large team often repeat various tool runs simply because there's no efficient way to share the results of previous runs among themselves. This causes wasted compute resources and unnecessary wait time for the designers, which ultimately leads to long turn-around time (TAT) and higher cost of the SoC. In this paper, we discuss the challenges in sharing tool run results among the designers of such large teams and explore a few caching methods to address these challenges. We discuss pros and cons of these methods, with particular focus on one of our proposed methods that we found to be the most balanced solution. We present data from a recent SoC at Intel where this balanced method is saving thousands of hours of compute resources as well as significant amount of engineering resources.**

I. Introduction

RTL development for a large-scale SoC generally involves a continuous integration (CI) infrastructure, where designers submit their changes which then get merged with the latest head of the repository, go through acceptance checks and if passed, get published as the new head of the repository. This head of the repository is the start point of a designer activity. This work model, as shown in Fig. 1, is similar to large-scale software development projects, and is necessary to enable efficient distributed development. As such, many of the conventional caching concepts could also apply here [1, 2].

Immediately after cloning the head of the repository, a designer's activity involves the following steps.

1) Configuring the IPs and generating the top-level connectivity or other such content (optional)
2) Iterating between making changes to the collaterals and running the necessary tools until the changes are ready for submission into the CI pipeline

A designer runs various tools to complete his/her intended activity. In addition, most activities also involve pre-work and post-work. Pre-work is where the designer generates or updates necessary collaterals or setups to enable the intended activity. Post-work is where the designer ensures the overall quality of any update he/she made to the repo before it goes to the CI pipeline. Post-work is needed to ensure high acceptance rate in CI since a failed submission in CI negatively impacts the turn-around time (TAT) of the repository for the entire team.

While running these tools, a designer often ends up repeating what may have already been run by another designer or as part of the CI acceptance checks. Example includes having to re-compile the design for a static check tool where the only change done was addition of a new design rule, having to re-compile the entire design for a logic simulation tool where the change done was only for a small portion of the design etc. This can amount to hours of runtime per activity per designer for large designs. Considering the duration of a typical real-life project, this results in thousands of hours of unnecessary tool runs. Being able to identify such scenarios and avoid duplicate tool runs can improve productivity significantly by reducing compute cost and wasted designer bandwidth (even as the designers strive to overlap activities while waiting for a tool run to complete). This ultimately will lead to improved TAT and reduced cost for the entire SoC development.

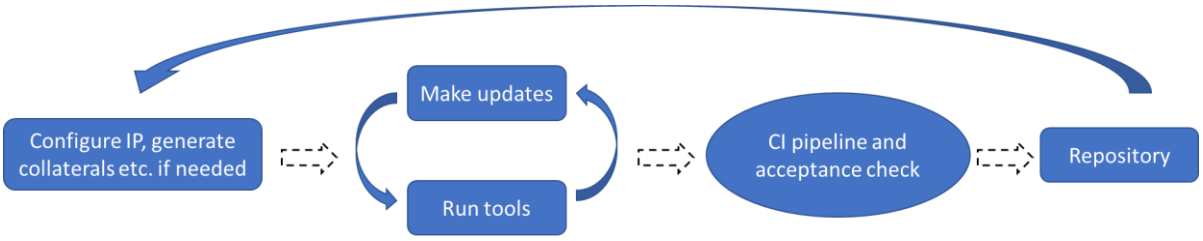

Figure 1. Simplified view of Continuous Integration work model for a large-scale RTL design

In the following sections we will discuss the challenges in achieving efficient caching/re-use for EDA tool runs, describe some of the methods to accomplish such re-use including their pros and cons, and present data on savings from a production project.

II. Challenges and Considerations in Caching/Re-using Tool Run Results

A. Content Re-usability

The very first requirement for successful re-use is that the target content needs to be re-usable! While some of the tool-generated collaterals are re-usable, many are not. Some are not readily re-usable, but can be converted into re-usable collateral with minor work. Generally, this is a result of environment-dependency of the collateral. For example, if a System Verilog file was generated by a tool, it should be re-usable. Meaning, saving it from one user's local run area to another user's local run area (via copying to a central location or by directly copying) should be okay. However, that may not be the case if a tool-generated file uses hardcoded reference to its own run area or has a pointer to the machine it ran on. Copying such content from one area to another could cause issues and may not work at all.

Sometimes though, it is possible to identify such issues and handle them by updating the target collaterals. For example, in the case of the previous example, if a file has hardcoded reference to its own run area (path), at the time of saving the content to a central location, we can replace those path values with the path to the central location. Then at the time of restoring to another user's area, we can replace the path to central location with the user's run area. This is the method we used in our work to make some of the tool-generated files re-usable.

B. Save&Restore vs. Re-generate

Saving some collaterals and restoring them is not entirely free, it requires additional disk space and save/restore run time. If re-generating the collaterals locally is faster than saving and restoring, then it becomes hard to justify any investment in caching. A similar argument is true for disk space requirements. Our experience is that for caching to be a worthwhile mechanism, the design needs to be complex enough such that re-generation of the collaterals will typically be significantly more expensive than saving and restoring them. For any decent sized SoC, that's typically the case.

C. Cache Hit Rate

For caching to be a meaningful option for a project, the cache hit rate needs to be above some threshold. If a user is changing everything in the repository all the time, then there may not be enough room for re-use. However, this is certainly not the case for a large-scale project, where a designer generally works on a small portion of the design at a time. That's one of the main reasons why we found caching to be particularly well-suited for large scale RTL development projects.

D. Cache/Re-use Infrastructure

For the tool-generated content that passes the above-mentioned criteria, the next major hurdle is having an efficient infrastructure to save/restore content and identify if there's a cache hit. Saving and restoring generally involves having a central location (disk) and managing it efficiently. This includes estimating the disk size based on the target content to be saved and having a mechanism to periodically remove old content from the disk such that the disk does not overflow. Some content can also be made part of the repository, although it has its own challenge. We will discuss that as part of the next section.

III. Caching methods

In this section, we describe various caching methods with focus on sharing content within a team. Peer to Peer direct sharing can also benefit from these methods (done as part of our actual implementation), but that's not the focus of this paper.

Method 1 - Storing cache as part of the repository. In this method any content (tool run results or intermediate files) worth sharing among the team is stored as part of the repository. There can be multiple variations depending on who does the storing.

- Method 1A: Where every user is allowed to store the content.
- Method 1B: Where only one or more designated designers are allowed to store the content.
- Method 1C: Where the CI infrastructure handles the storing.

Retrieval is simple in this method since the user of the cache will not have to do anything additional to start using the cache. This method works well for text-based content where the content to store is known precisely beforehand.

It becomes difficult to scale once the cache target increases in size or includes large binary files, and when it's hard to keep track of exactly which files need to be stored.

Method 2 - Storing cache outside of the repository: This method scales well and can accommodate large content and some ambiguity in terms of what needs to be cached (e.g., a directory can be saved without worrying too much about the precise content). Content is stored in a disk, separate from the repository itself. Retrieval is challenging though, since now there needs to be mechanism to identify which stored cache corresponds to the state of the repository that a designer is working on.

We suggest that the storing action is done by the CI infrastructure itself, and while storing the cache a unique tag is generated to identify the state of the repo and its corresponding cache. For example, a Git-based repository would have a SHA id per commit and allows additional tagging as needed [3], which could be used to map the cached content to a particular state of the repository. Retrieval in this method can be done in two ways.

- Method 2A: Where there's a mechanism to detect changes in input to the tool, and cached content is only retrieved when there's no change
- Method 2B: Where user decides whether to retrieve the cache or not (retrieval is a copy command that uses the aforementioned tagging mechanism).

Pros and Cons of the methods: Table I summarizes the pros and cons of each of the methods. Based on our experience, we recommend Method 2B for large design projects with potential multi-generation lifecycle.

Methods 1A, 1B, 1C all share a common limitation that it only works well for content that can be part of a repository, e.g., text files. These methods don't work well at all if the cache target is a large file, potentially binary. We do find EDA tool-generated large binary files worth caching for large-scale projects, which is why we don't recommend these methods for such projects. In projects where that's not the case, these methods could still be effective. Among these methods, 1A is challenging if the CI involves auto-merging of text files, since tool-generated files don't do well in auto-merging. Method 1B works better since the content is only submitted by a set of pre-determined activity owners, but it restricts the work model. Method 1C works the best, but it involves on-the-fly commit of the tool generated content once the checks in the CI pass, which requires careful implementation.

Methods 2A and 2B both support complex cached content and scale well in terms of storing the content. But 2A is very difficult to implement and maintain, since it involves understanding tool inputs. There are multiple mechanisms to identify and track the inputs of a tool (e.g., system commands like *strace*), but automatically detected inputs will often include files that are not directly related to the design content. For example, a tool may be accessing a system file (something from the OS, license server, tool installation directory etc.) which we don't really want to track because it doesn't change the re-usability of the generated content. Such files will have to be detected and excluded from the list of input files that need to be checked for any change. Thus, this method requires identifying all files that a tool accesses and then either include only those that we care about or exclude those we don't care about. Either way is very hard to implement right, and even harder to maintain as the design evolves throughout the project's lifetime.

TABLE I
COMPARISON OF VARIOUS CACHING METHODS FOR A LARGE-SCALE RTL DESIGN TEAM

Method	Key Feature	Pros	Cons
1A: Store in Repo, by all users	Every user submits content to store	1. Cached content always available as part of the repo	1. Doesn't work well for large/complex content. 2. Issue in CI while content merging
1B: Store in Repo, by designated users	Only designated users submit content to store	1. Cached content always available as part of the repo	1. Doesn't work well for large/complex content. 2. Restrictive work model
1C: Store in Repo, by CI infrastructure	CI infrastructure stores as part of the repository (upon successful checks)	1. Cached content always available as part of the repo	1. Doesn't work well for large/complex content. 2. Adds complexity in CI infrastructure
2A: Store separately, Retrieval based on automatic detection of input change	CI infrastructure stores (upon successful checks) in a separate location than the repository	1. Cached content available for copying for as long as project decides to keep the data 2. Scales to any type of collateral and any size 3. No major change in work model	1. Content retrieval involves difficult setup and high maintenance cost due to the need for automated detection of input change
2B: Store separately, Decision to retrieve left to the user	CI infrastructure stores (upon successful checks) in a separate location than the repository	1. Cached content available for copying for as long as project decides to keep the data 2. Scales to any type of collateral and any size 3. No major change in work model 4. Easy setup	1. Doesn't track change in input collaterals.

Our experience is that designers have a reasonably good understanding of what updates they make in the repo and can judge fairly easily if they should use some cached content or not. For example, if a designer only made updates to validation collaterals, they can still re-use all RTL/connectivity content from cache, including any compilation results on those.

Another point to note is that, for some tools, the cached content can be useful even if some of the inputs have changed. Such tools have internal mechanism to detect which parts of the design has changed, and which hasn't. They can use that information to selectively re-use the cached content. That's another reason why we don't find investing in tracking input changes worthwhile. Thus, our overall recommendation is Method 2B, which is what we implemented in our project.

Last but not the least, the final checks on any change by a designer in CI are always done without using any cached content. This ensures that in the unlikely event of a false cache hit causing some errors go undetected in a user's run will be caught in the CI.

IV. Implementation Details

In this section we will detail the implementation of the methods described in the previous section, with focus on Method 2B. We will assume a Git-based repository here [3], but the concepts described here should apply to any other repository management tool as well. Before going into the details of Method 2B, we will briefly touch upon the implementation of the other methods.

A. Methods 1A, 1B, 1C, 2A

Methods 1A and 1B are trivial to implement as all that's needed is for the user to submit the cache content along with his/her regular updates to the CI. There is nothing in the CI infrastructure or repo that needs to be coded to enable these. But as mentioned in the previous section, Method 1A could be a major problem in the CI infrastructure as the tool-generated content tend to be difficult to auto-merge. Therefore, any optimization of the CI infrastructure, such as collapsing multiple submissions into one to reduce the amount of CI check runtime, could be challenging. In Method 1B, this is handled by restricting the use model where only one user or just a few users submit the cached content to the CI. These users coordinate their submissions in a way that doesn't cause the CI infrastructure to handle any auto-merging of the cached content. As one can imagine, this is a very restrictive work model and doesn't scale well for large teams.

Method 1C improves upon the issue of who will update the cached content in the repo by pushing the responsibility to the CI infrastructure. Once all acceptance checks pass, the CI infrastructure does an additional step where it commits the cached content to the repo, in addition to the user-updates, and pushes these changes to the head of the repo along with the user-updates. Additional coding/scripting is needed to create this hook in the CI infrastructure. This script will have the list of what files need to be committed to the repo and will run the Git commit commands accordingly. Another aspect this method needs to handle is to distinguish the changes in the cached content from the changes made by the user. Otherwise, the user will have to manually differentiate his/her changes vs. CI-committed cached content, and make sure he/she only commits the changes he/she made. In Git, this can be done by identifying the cache targets and excluding them from showing up in any status query that the user does (via Git setup files, e.g., .gitignore etc.). For these reasons, this method works fine where the cached content is limited and known ahead of time. It becomes challenging to manage this list as we scale the content we want to cache.

Methods 1A, 1B, and 1C all have the same limitation that they cannot handle large or binary files since everything need to be part of the repository. This is why we have Methods 2A and 2B. The difference between Methods 2A and 2B are that 2A uses an automated method to determine if the cached content is still valid or not. The way we implemented this was by identifying all relevant tool inputs and checking if they changed. The tool inputs can be identified in 3 different ways – manually identifying all relevant inputs, automatically identifying all inputs that the tool used and then *excluding the ones we don't really care about*, automatically identifying all inputs that the tool used and then *keeping only the ones we really care about*. The manual method is tedious and don't scale well. It's workable though if the changes we care about are confined to a manageable quantity, e.g., if all the RTL+UPF files are in a handful of directories. For the semi-automatic way, the initial identification of the inputs can be done through various tracing mechanisms, such as *strace*. The tedious and error-prone part is to identify which input files we actually care about, as a tool not only accesses user-created files, but also accesses many OS (Operating System) files. This could literally mean sorting through thousands of files. Thus, Method 2A becomes very difficult to implement correctly and even more difficult to maintain over time.

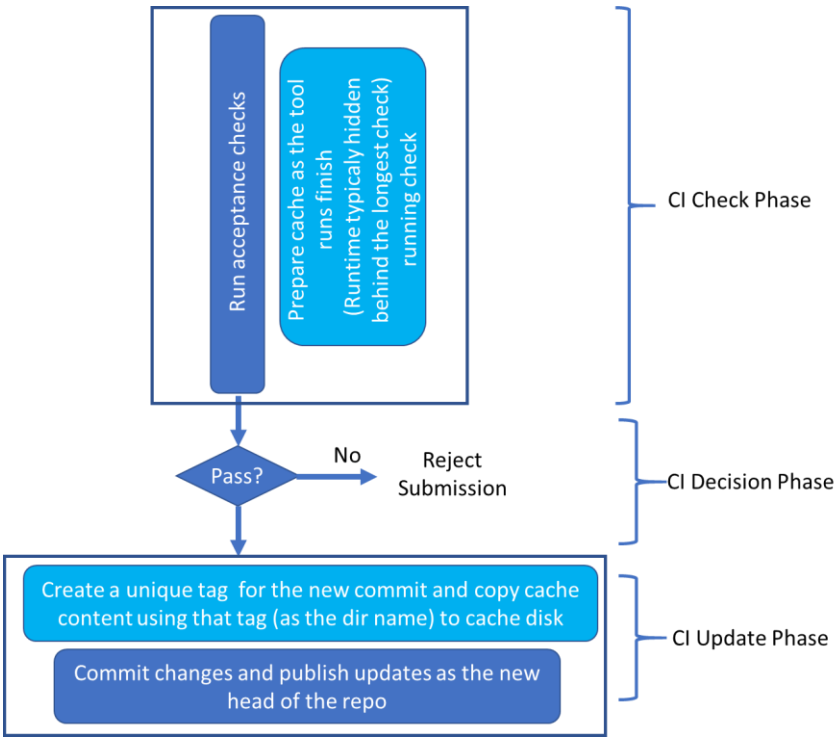

Figure 2. CI work model updates needed to implement caching Method 2B (additional steps highlighted in the light blue sections)

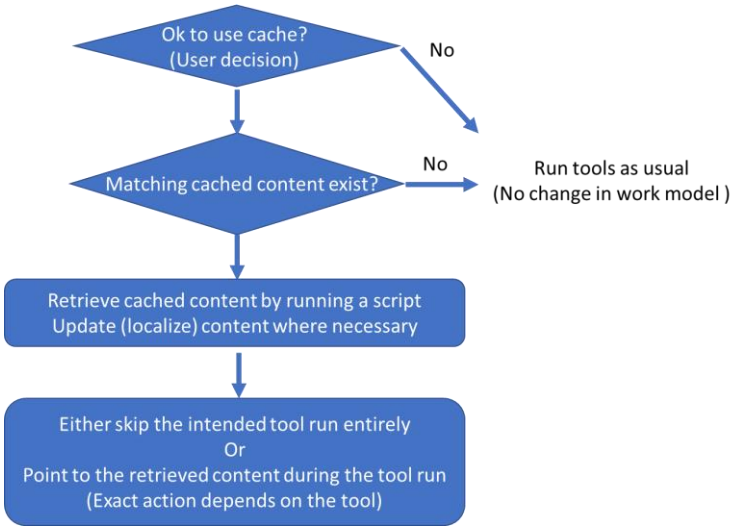

Figure 3. User work model updates needed to implement caching Method 2B (additional steps highlighted in the light blue sections)

B. Method 2B

Methods 2B has two aspects of implementation – work model and coding. Fig. 2 and Fig. 3 show the work model in this method and Fig. 4 shows the pseudo code.

As shown in Fig. 2, while performing the acceptance checks in CI, we now also perform any collateral update needed for caching. A typical example of an update would be compressing files to reduce disk usage and copy time. But updates may also include replacing local path values in the tool-generated content to something generic that can be later converted into the local path of the specific user. These updates can be done in parallel to the checks running in CI to prevent any increase in the overall runtime of CI due to these additional steps.

The other important thing we do in CI is creating a unique tag for the specific commit once the user updates pass the acceptance checks and are committed as part of the new head of the repo. All the cached data are stored in a separate disk under a directory name that corresponds to this unique tag. This disk is periodically cleaned up to avoid disk over usage. The project decides the disk size and cleanup policy based on the amount of data being cached, frequency of successful CI runs, and how long the cached data should be retained. Once the CI acceptance checks all pass, in addition to committing the content, we now also run a caching script (Fig. 4) that copies the target content for caching purposes. Creation of the tag is not done through the caching script, it's left for the CI infrastructure to implement before committing the changes.

Fig. 3 shows the new work model at the users' end. First the user decides whether to use cached content or not. This decision is easier than what it may sound like at the beginning, as most of the time designers have a reasonable understanding of which tools are affected by the changes they made. Our implementation allows them to retrieve the entire cached content for their repo, or only certain sections of it. This gives them more flexibility. In addition, they also have the assurance that the final submission in the CI is always run from scratch without any cached content. Once the user decides to use cache, he/she will then run the caching script (Fig. 4) that checks the tag of the current repo. Then it retrieves the content from the central cache disk if content corresponding to this tag exists. It also does any conversion (decompression, localization of paths etc.) necessary to make the cached content work in the user's environment.

Fig. 4 shows the pseudo code of the caching script. As one can observe, it is very simple – and that is by design. For large scale RTL projects, most caching techniques become very difficult to maintain over time, particularly when the project ownership changes from team to team or task owners move on to new roles. Being easy to maintain was a key motivation in designing this entire caching method, including this script.

This caching script defines the content to copy per target section, including any pre-cache processing (compressing etc.) and post-retrieval processing (decompressing etc.). The same script is used in CI and by all the users. Depending on who is running the script, it either creates a unique tag (for CI run) or identifies it (for user run). Then it copies the desired content from the source to destination and also handles any pre/post processing. For a user run, if no cache exists for the corresponding tag, the script returns with an appropriate message to inform the user that he/she can't use caching.

```
Define Cache_sections
        Define section name
        Define what content to copy (paths of files or directories relative to the top of the repo or run area)
        Define any necessary pre-cache processing or post-retrieval processing

If User is CI then
        Target_sections = all (CI copies all cacheable content to the central disk)
        From_Dir = <User's run area>
        To_Dir = <Central Cache Disk>
        Tag = <Create a unique tag>
Else
        Target_sections = <User's command line input; Default is all>
        From_Dir = <Central Cache Disk>
        To_Dir = <User's run area>
        Tag = <Retrieve the latest unique tag> (Git example: git describe --abbrev=0 --tags --first-parent --match
"<unique tag pattern>")

If User is NOT CI then
        Exit if cached content does not exist for Tag in the From_Dir

For each Section in Target_sections
        If User is CI then
                Perform any pre-cache processing on the content to be cached for this Section

        Copy content for that Section from From_Dir to To_Dir

        If User is NOT CI then
                Perform any post-retrieval processing on the content that was retrieved for this Section
```

Figure 4. Pseudo code for the caching script used to implement Method 2B (the same script is run by the CI and all users

V. RESULTS

We implemented Method 2B in our projects on two categories of tool runs. The first is where IPs are configured and connected to form the SoC, including various validation collaterals (fuse etc.). The second is compilation/elaboration by a logic simulation tool. The first category requires an exact match of the inputs for the cached content to be re-usable, whereas the second category can tolerate certain amount of changes in design and still successfully re-use parts of the cached content. The first category is primarily useful for validation engineers where they are only making changes to the validation collateral and need to re-generate any RTL (design-only) content, but it can also be used by RTL (design-only) engineers in certain cases, e.g., if they are focused on running quality check tools on RTL.

Fig. 5 shows the usage of the caching method 2B in an ongoing project in Intel for the first category of the design activity. Despite leaving the decision to copy to the users, we saw pilot users enthusiastically using this method and in just 30 days we achieved a few hundred hours of reduction in tool runtime. The method continues to be running in the production environment, saving thousands of hours of compute and engineering resources every month. As discussed in section II, we did face and had to overcome the issue that some intermediate files use hardcoded paths to user's run area that had to be automatically converted to the new user's path to make sure the copied content still worked in the new environment.

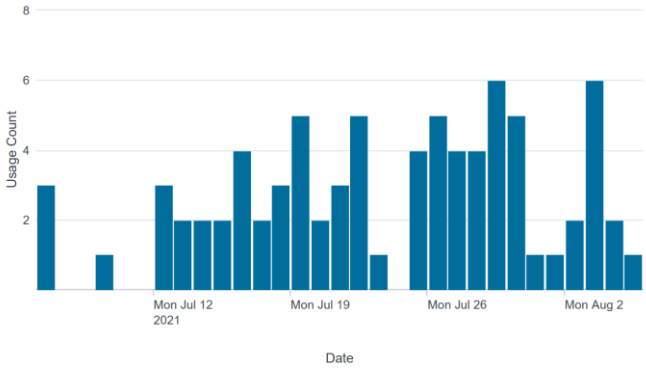

Figure 5. Example of 30-day usage of caching Method 2B in a production project. Each usage represents a reduction of 1-2 hours of computation!

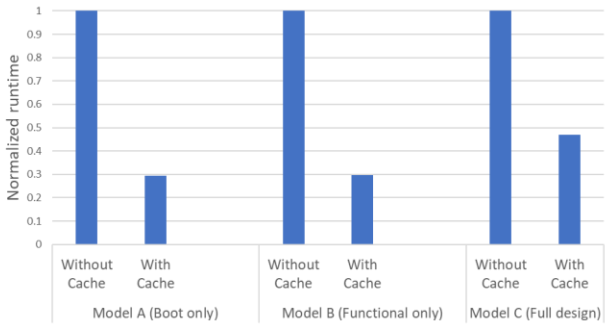

Figure 6. Reduction in building simulation models using cached data

Fig. 6 shows the reduction achieved in building simulation models (generating the binary executable, not including test runtime) in a user's area with little to no modification on top of the content from the head of the repository. As shown there, we have successfully achieved 50-70% reduction in runtime using these caching techniques. This saving is multiplied by the number of users/usecases in a project, contributing to thousands of hours of savings in compute and engineering resources, and corresponding improvement in designer productivity.

VI. CONCLUSION

We described why caching in a large-scale RTL development project can be significantly beneficial and presented methods to achieve successful caching. We demonstrated results from a production project at Intel where we deployed Method 2B to save thousands of hours of computation. As our future work, we plan to work more closely with the vendors of various EDA tools such that they generate more content that is re-usable across all users of a large project, in addition to being re-usable for a local run of the tool. We intend to drive the EDA vendors toward

creating an internal representation of a design as one of the very first steps of a tool run such that the tool can easily identify if any cached content can be re-used. Using various partitioning mechanisms to increase the use of cached content even in the face of some design changes is also a highly desired feature from all vendor tools.

ACKNOWLEDGMENT

Thanks to Narasimhan Iyengar for his support on the deployment of the proposed caching method at Intel.

REFERENCES

[1] John L. Hennessy and David A. Patterson, “Computer Architecture, Fifth Edition: A Quantitative Approach (5th. ed.)”, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011

[2] J. Yan, J. Chen and W. Jiang, "Data Caching Techniques in Web Application," 2014 Enterprise Systems Conference, 2014, pp. 289-293

[3] Chacon S, Straub B., *Pro git*, Apress; 2014.