Electronic Design Automation (EDA) tool development: Performance enhancements to circuit extraction

Rithvik Bhogavilli *Poolesville High School*
Poolesville, Maryland

Abstract

The Magic VLSI (Very-Large Scale Integration) Layout Tool is an open-source software that was originally intended for smaller chip designs. However, as chips evolve, the tool must be adapted to meet higher load requirements. Magic currently stores labels for a given chip design in the form of a linked list, meaning that searching for a label is an O(n) operation. To optimize performance, a binned collection and hash table was implemented to replace the linked list storage for labels. Changes to the tools in Magic were tested on the striVe chip and the performance of the given tool was tested using the perf tool in Linux. Using the perf results, a flame graph was generated to then measure the relative number of samples for a given function. The effectiveness of the implemented optimizations was based on the decrease in time spent on functions when using a larger chip design. Replacing the linked list for the binned collection for label storage by location correlated with a general decrease in total time spent on the functions. Although the baseline for label storage by name could not be analyzed, there was still a theoretical increase in efficiency due to the O(1) complexity of the hash table.

I. Introduction

The Magic VLSI (Very-Large Scale Integration) Layout Tool is an open-source software that was a tool made for the planning and laying out of integrated circuits [5]. This tool is easier to use than layout tools of the past due to its increased knowledge of design rules and connectivity. Previous tools such as Caesar and KIC2 had been used for various smaller layouts but provided little support in assiting with routing on chip designs, making the process error prone. Another larger issue with these programs was that it was hard to change a design once it was loaded in the layout tool. This is suboptimal since many design issues are noticed late in the layout process [5]. Magic, although made at about the same time, is more effective as the user gets up-to-date information on whether the design is violating any of the design rules. Magic also makes it easy to modify any existing integrated circuits, allowing for an increased level of debugging an integrated circuit (IC) or test any kind of iteration of an already existing IC [6]. However, like other programs of the past, Magic was originally designed for use on smaller chips. As computer chip designs have become larger and more data intensive, the tool does not have the necessary optimizations for loading and interacting with these designs at a reasonable rate.

In Magic, the layout is split into regions called cells. These cells contain paint that defines the structure of the circuit and the labels that are associated with the paint which describe the paint. Labels are used by the synthesis tools to communicate information about the given area. Labels are accessed inside of the synthesis tool both by region and by name. Thus, there must be two data structures used to access the labels associated with a file.

The current implementation for storing labels involves a linked list, where each node in the linked list contains the information on a label and has a reference to the next stored label. The issue with this implementation is that the efficiency of accessing a label in the list is $\mathcal{O}(n)$, making the current implementation very inefficient with designs that may have tens to hundreds of thousands of labels.

To address the issue of inefficiency when accessing a label by region, the binned plane would be implemented as it is more suited for the task of storing data by region. The binned plane was initially developed as part of a program based off of Magic called ”microMagic” [4]. However, binned planes were not used for label storage and therefore is a novel concept. In this data structure, each cell in the chip design would be a bin and the labels that intersect a given bin are linked together. The labels can then be located through a two-dimensional hash table in which each list will represent a region of the circuit. This data structure is theoretically more efficient as it reduces the search area for a given label to a shorter list of bins in a given area. In a large synthesized digital standard cell layout with labeled nets, the labels all have nearly uniform size and spatial distribution, so the bins are optimal.

In order to optimize the efficiency of searching through labels by content, a hash table could be used. The hash for the table would hash the content of the labels, meaning that chaining could be used for labels that contain the same information but are in different regions. This would reduce the complexity of the search to $\mathcal{O}(1)$ assuming that an optimal hash is used. In the case that there are duplicate labels in separate regions, the bins could be used to filter the duplicates based on the wanted region of a label.

To assess the performance of the various implementations, the Linux `perf` tool will be used. The `perf` tool creates a record that would record the number of CPU cycles per function call and then sort the functions based on the function calls

[1]. The raw output from `perf` would then be put into a flame graph, to get a visual representation of where the program is hanging, allowing the hot spots to be easily targeted [3]. The efficiency improvement is then measured by the decrease in the portion of samples spent on a given function.

This research aimed to implement the aforementioned data structures as a replacement for the linked list used for storing labels. The version of Magic with the newly implemented data structures is then compared against the original version of Magic to measure the increase in efficiency.

II. Materials and Methods

A. Materials

The latest stable release version (8.3) of Magic was used to establish baseline data before the optimizations. The code was written in C and the source code was sourced from the Open Circuit Design website. For testing the performance of Magic, a standardized chip was used. A suitable chip for testing the performance of Magic had to have enough labels for there to be a detectable performance difference between the linked list implementation and the binned collection and hash table implementation. The striVe chip was used due to it being open-source along with there being no issues with using the data for tool analysis [2]. Another factor that was taken into consideration was the concentration of labels. If labels were to be concentrated in one area, then they would all be processed at once, meaning that there would not be any gain in optimizing the search based on area. The striVe chip had a very large layout, meaning that there would be a performance difference in comparison to a smaller chip for reading the labels. To generate the graphs for analyzing the runtime, a flame graph generator was used since it could convert the raw perf output to a graph.

B. Methods

The binned collection was first implemented to store labels based on location. During implementation, the linked list for storing the labels was not removed since many newer features and areas that did not show need for optimization still existed in the code base. The hot spots to be optimized were selected based on previous experience with using Magic to load large chips along with identifying where the linked list implementation was being used for labels. By identifying where the linked list implementations were being called, a performance test could be run on those functions through Magic's commands to quantify its performance.

The first area of concern that was taken into consideration was extraction, which is the generation of a netlist file showing the cells and their connections. The `extract` command in Magic extracts all cells in the window and outputs them into a `.ext` file. The striVe chip was loaded into Magic and then the extract command was run. During this time, Magic was iterating through all labels in the window and adding them to the cumulative layout. In order to get the current performance of the command, `perf` was run with a sampling rate of 99 samples per second to find the functions in the extract command that were taking the most samples in the window to complete execution. The `perf` tool generated a report that was then converted into a flame graph, which organized the results of the records into sections, making it visually more clear to distinguish what functions were taking the most time as shown in Figure 1. To confirm that the functions in question were being called, `gdb` was run on Magic and a break-point was set at the functions in question. The binned collection was added to store labels by editing the `CellDef` data type. The `CellDef` defines the properties for the cell and each cell stores the labels in its designated method of storage. The binned collection was then populated along with the linked list in the creation and deletions of cells. By keeping both data structures, the label storage can gradually be moved from the linked list to the binned collection implementation. The program would still run faster since the traversal through the linked list was causing the program to hang on the function. By changing the iteration to use the binned collection, the program would theoretically run faster. After the implementation was added, another flame graph and performance record was generated and the results of the new flame graph were used to investigate where else to implement the binned collection.

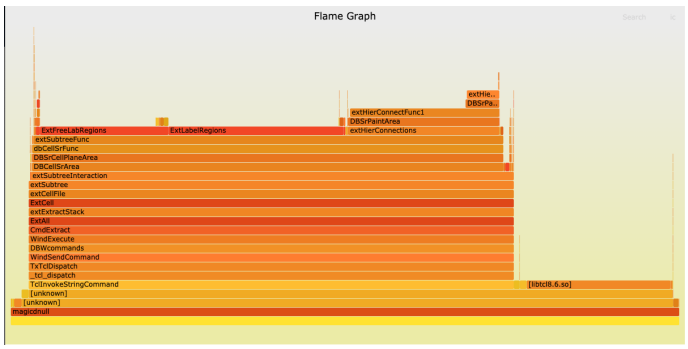

Fig. 1: Sample flame graph generated from performance analysis.

Upon analyzing the new flame graph, other inefficiencies were found in the extraction routine. `extSubtreeFunc` is a function that is called for each of the children cells of a parent cell that is being extracted. In this function, while iterating through the labels to be yanked into the parent cell, labels that were not ”sticky” were also being checked, which should not be done because sticky labels still carry the information of the layer they were on, unlike normal labels. This means that the normal labels would not be of use during extraction. This error was fixed by ignoring any label that was not a sticky label. After this fix was made another flame graph was created in order to ensure that the extraction routine was running at a reasonable level of performance.

The `extHierCopyLabels` function was also a hot spot in the extraction routine so the binned collection implementation was added to replace the linked list traversal. In this case, the search for the labels was not taking into consideration the search area and therefore the function would look through all of the labels for the given cell. The binned collection traversal was used in place to filter labels based on the region that they were in. An example of this implementation in the function is given below:

```
293     for (lab = cumDef->cd_labels;  lab;  lab = lab->lab_next)
294         if (GEO_TOUCH(&r, &lab->lab_rect) && (lab->lab_flags & LABEL_STICKY))
295             if (TTMaskHasType(connected, lab->lab_type))
296             {
```

Fig. 2: Example of the linked list iteration

```
298     BPEnum bpe;
299
300     BPEnumInit(&bpe, cumDef->cd_labelPlane, &r, BPE_TOUCH, "extHierConnections");
301     while(lab = BPEnumNext(&bpe))
302     {
303         if (lab->lab_flags & LABEL_STICKY)
304             if (TTMaskHasType(connected, lab->lab_type))
305             {
```

Fig. 3: The optimized code using the binned collection

Once again, a flame graph was generated and the relative number of samples for the function was recorded.

The next point of interest was net selection. When a net is selected, Magic finds the connected geometry for the net and in the process, calls a function called `DBTreeSrLabels` which recursively searched for all labels with a given mask, hence causing there to be performance issues. This function checked two flags: `TF_LABEL_ATTACH`and `TF_LABEL_DISPLAY`. `TF_LABEL_DISPLAY` looked for labels where either `lab_rect` or `lab_bbox` is in the search area while the other, `TF_LABEL_ATTACH`, looked for labels where `lab_rect` was in the search area. Since the bplane enumeration could only see if `lab_rect` was in the search area, both the bplane and the linked list had to be used depending on what flag was being used. The bplane enumeration was used for when the flag was `TF_LABEL_ATTACH` and the linked list was used when the flag was `TF_LABEL_DISPLAY`. There were also performance issues with the `SelRemoveSel2` function which was thought to be executed when the current selection in Magic was cleared. However, when running `gdb` on the `select clear` command in Magic, this function was not executed. The bplane enumeration was attempted, however the function had to iterate through all labels in the design so adding the bplane enumeration would be detrimental.

After the optimization for finding labels based on location was done, the focus shifted to optimizing the searching of labels by name. Like the bplane implementation, the hash table implementation was added alongside the linked list and the hash table was populated and cleared in the same places that the linked list were whenever a `CellDef` was created or deleted. The hash table was also included with the bplane because if there was only one label with a given name, the hash table would be much faster at retrieving the label in comparison to the bplane. If there were multiple labels with the same name, they are almost always guaranteed to be in different locations meaning that a location search inside the hash table would be fairly efficient.

The hash table implementation would increase performance when labels are being searched by type. Searching labels by type would be useful in the case that labels have the same name in the same position on two different layers. To test the performance of the hash table, the command `select short` was used. This command checked if there was a short between two labels which would mean that it had to find labels based on their names. The change for this function can be seen below:

```
354     for (lab = def->cd_labels; lab; lab = lab->lab_next)
355     {
356         if ((rect != NULL) && !(RECTEQUAL(&lab->lab_rect, rect))) continue;
357         if ((type >= 0) && (type != lab->lab_type)) continue;
358         if ((text != NULL) && (strcmp(text, lab->lab_text) != 0)) continue;
359
360         return lab;
361     }
362     return NULL;
363 }
```

Fig. 4: The original code for checking labels by content

```
    if (text != NULL)
    {
        lab = IHashLookUp(def->cd_labelHash, text);
        if (lab->lab_hashNext == NULL)
        {
            if (rect == NULL) return lab;
            return (RECTEQUAL(&lab->lab_rect, rect)) ? lab : NULL;
        }
    }
    if (rect != NULL)
    {
        BPEnum bpe;
        BPEnumInit(&bpe, def->cd_labelPlane, rect, BPE_EQUAL, "DBCheckLabelsByContent");

```

Fig. 5: The hash table is used in conjunction with the binned collection

A flame graph was generated with this command and two labels. The distance between the labels and if they were in the same net did not matter since the program would still have to search for the labels by name, regardless of their position. The hash table implementation was then added to the `DBCheckLabelsByContent` function. In this function since the hash table would be efficient in the case that there was only one label with a given name, if there were multiple labels with the same name, the bplane enumeration would have to be used in order to take the position of the label in the design into consideration. A flame graph was generated again after the changes were made.

III. Results

The goal of the study was to optimize the performance of the Magic VLSI Layout tool by identifying points in the execution where significant time was taken. Once the hotspots were identified, the code was then replaced to use the new bplane and hash table representations for the label storage.

For all tests of the performance, the striVe chip was used which had 72, 616 labels. The extraction tool was the first area of interest and was recorded in a window of 20 seconds with 99 samples being taken each second.

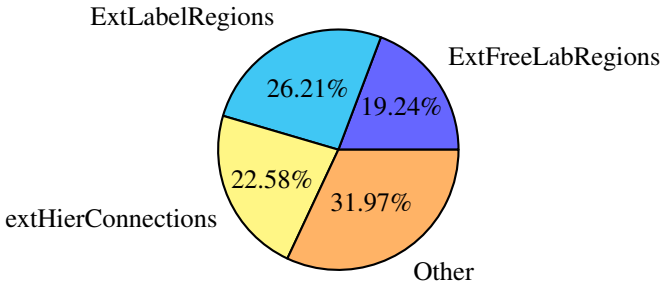

Fig. 6: The three functions that were most prevalent in the extraction were ExtFreeLabRegions, ExtLabelRegions, and extHierConnections. The numbers in the plot represent the percentage of samples spent on the given function.

The figure above shows the main functions that were responsible for most of the time taken for the extraction to complete. Before the error in the code regarding the consideration of non-sticky labels was removed from the extraction code, the extraction exceeded the scope of the sampling window and spent most of its time in the three functions shown in Figure 6. After removing the error in the extHierConnections function, the following was recorded for the extHierConnections function.

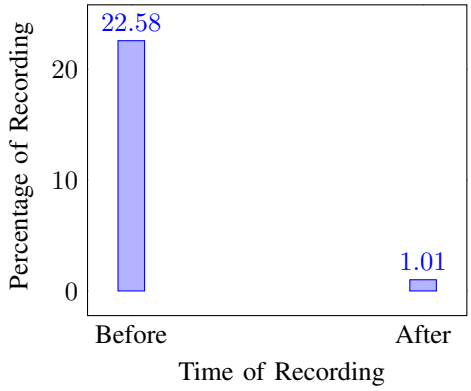

Fig. 7: Before the changes were made, the extHierConnections function was responsible for 22.58% of the calls in comparison to the 1.01% after changing the function's logic.

With the addition of the label detection, the function took much less time to complete execution and therefore was only detected for 1.01% of the samples. After removing the error, the bplane was implemented then tested. With the bplane in place of the linked list, the flame graph showed no noticeable change. This is due to the fact that only one sample was retrieved for the extHierConnections function when recording before and after the bplane.

The next point of interest was optimizing the net selection process. To test this, the striVe chip was loaded and the power net was then selected because it was the largest net in the chip layout. The original net selection results are shown in Figure 8.

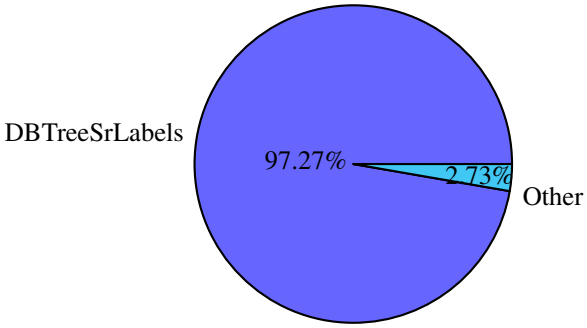

Fig. 8: The figure shows the results of the net selection. The primary hotspot in net selection was the DBTreeSrLabels function which was responsible for 97.27% of the samples during execution.

The DBTreeSrLabels function was primarily limited by what kind of flag was passed to the function. Because of this, the current linked list implementation was kept when the `TF_LABEL_DISPLAY` flag was activated. The bplane implementation was used instead for the `TF_LABEL_ATTACH` flag. With this implementation, the duplicate code handling what data structure was used was moved to a new function and therefore the performance of this new function was analyzed.

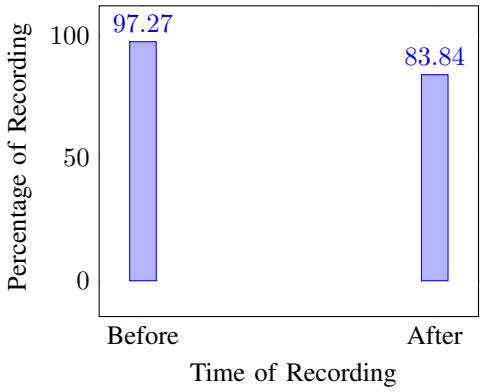

Fig. 9: After implementing the bplane data structure, there was a 15% improvement in the performance.

Despite there being a decrease in the time spent on the function due to the data structure, the overhead of the function due to the interruption handling resulted in more time having to be spent on the function in general. However, there was still a net decrease in the amount of time spent on the function of about 23 samples.

Finally, for the implementation of the hash table, the command that would provide the best test of the effectiveness of the hash table was the `select short` method that would check if there was a short between the two nets of the given labels. When recording the performance of the original program, the `perf` tool did not detect the content searching function. The tool was running at a sampling rate of 30,000 labels per second and since the linked list implementation has a complexity of $\mathcal{O}(n)$, in comparison to the complexity of the hash table which is $\mathcal{O}(1)$, we can assume that the new implementation would also not be detected. The content checking function was confirmed to be run during the testing through `gdb`.

IV. Discussion

The goal of the research was to optimize the label storage in Magic to therefore improve performance time in various tools that come with Magic. Although testing of the optimizations of the program were limited to the striVe and AES chip, initial results have been favorable.

The results indicated a general decrease in the time spent on the functions tested during the sampling period. For the `extHierConnections` function, after correcting the error, there was a decrease of 21.47% in the time spent on the function.

However, when observing its parent function, there was an increase in the time spent on the parent function after implementing the bplane storage. This could be attributed to the general overhead that was involved with the bplane implementation. To traverse over the bplane, another class had to be instantiated which would lead to an increase in overhead which would make the program less efficient for designs with a relatively lower label count. Despite there being a general performance loss for designs of about 70,000 labels and lower, there would still be general performance gains in higher load designs where the setup for traversal would get overshadowed by the actual searching of the bplane. The striVe chip was used in replacement of the lower label count AES chip from before, showing considerable improvement.

For the optimization of net selection, there was less of a noticeable difference in the time spent on the process. This could have been attributed to the fact that not all of the labels can be handled by the bplane implementation. For the striVe chip, there might still have been many more instances where the linked list could have been traversed, and therefore lead to a not as significant decrease in the time spent on the function as before.

For the hash table implementation, the effects of the changes were undetermined due to the performance analyzer being unable to detect a the function being called at a sampling rate of 30,000 samples per second. This would mean that the function is taking less than 30 μs. The function was confirmed to be called using gdb, meaning that the function was already operating at a reasonable speed for the given chip layout. Because the hash table would have a constant time complexity, it would be less likely for it to be detected in the flame graph.

Through the optimizations implemented, Magic will be able to handle larger chip designs and allow developers to interact with them at an increased speed.

For the future, a wider variety of chips could be used in testing the effectiveness of the optimizations discussed in the paper in order to ensure that they do not only apply to the AES or striVe chip. A limiting factor in assessing the performance of the commands run was having to stop the performance recording. Because the recording had to be manually stopped, in some instances, idle time overshadowed the recording of Magic, making the flame graphs less detailed because of the relatively less time spent on the functions of interest.

ACKNOWLEDGMENT

Thank you to Mr. Edwards for providing guidance throughout the internship and giving feedback on where to make optimizations.

REFERENCES

[1] Bangoria, R. (2017, February 1). Analyzing performance with perf annotate. IBM Developer. https://developer.ibm.com/technologies/linux/tutorials/l-analyzing-performance-perf-annotate-trs/.

[2] efabless. striVe [Chip Design]. Retrieved from https://github.com/efabless/striVe/.

[3] Gregg, B. Flame Graph [Computer Software]. Retrieved from https://github.com/brendangregg/FlameGraph.

[4] Ousterhout, J. K. (1984). Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 3(1), 87–100. https://doi.org/10.1109/tcad.1984.1270061.

[5] Ousterhout, J. K., Hamachi, G. T., Mayo, R. N., Scott, W. S., and Taylor, G. S. (1984). “Magic: A VLSI Layout System.” 21st Design Automation Conference Proceedings. https://doi.org/10.1109/dac.1984.1585789.

[6] Taylor, G., and Ousterhout, J. (1984). Magic’s incremental design-rule checker. Design Automation Conference, 160–165. https://dl.acm.org/doi/10.5555/800033.800791.