CAD Tool Development for Multi-Million Gate Designs

Jarrod A. Roy, David A. Papa, James F. Lu, Aaron N. Ng, Igor L. Markov
University of Michigan, EECS Department, Ann Arbor, MI 48109-2122
{royj,iamyou,jflu,aaronnn,imarkov}@umich.edu

ABSTRACT

Many groups in academia and industry are now extending their tools to handle super-sized designs. In addition to scalable algorithms, this requires software infrastructure and development policies to ensure and verify the robustness and scalability of implementations. Specific issues include programming for and executing programs in 32-bit and 64-bit memory models, modular tool infrastructure, diagnostic visualization of super-sized designs, and the use of simulation clusters for thorough evaluation of CAD tools.

1. INTRODUCTION

Recent academic work has demonstrated the ability to combine partitioning, floorplanning and placement in one tool [4]. This promises to significantly simplify SoC layout by essentially performing flat placement with integrated block packing and other optimizations. While this approach requires more exploration before reaching commercial toolchains, it is already clear that it leads to optimization at a scale previously unseen. Indeed, modern academic tools can place 2.5M objects (10M gates) in 32-bit memory space, and significantly more when compiled in 64-bit mode. Typical runtimes for designs with 2-3M objects, on modern workstations, are less than one day.

Recently IBM has sponsored a placement [8, 10] contest to gauge the effectiveness of academic placers on modern multi-million gate designs and has released eight new large benchmarks to foster development in this area. General statistics for these designs are presented in Table 1. These benchmarks are quite interesting in that they contain many fixed obstacles in the core region and a large amount of whitespace. This is depicted for the design adaptec1 in Figure 1. Combined with the fact that they have many movable objects, they are a challenge to state-of-the-art placers in terms of robustness and scalability which is inherent when working with multi-million gate designs.

In this note we discuss infrastructural issues associated with tool development for multi-million gate designs such as the ISPD 2005 Placement Contest benchmark suite. We illustrate these issues using large-scale circuit placement, but the bulk of the discussion applies to EDA tool development at large.

.

Circuits	# Mov. Objects	# Fixed Objects	# Nets	# Pins	Design Util.
adaptec1	210904	543	221142	944053	57.34%
adaptec2	254457	566	266009	1069482	44.32%
adaptec3	450927	723	466758	1875039	33.66%
adaptec4	494716	1329	515951	1912420	27.23%
bigblue1	277604	560	284479	1144691	44.67%
bigblue2	534782	23084	577235	2122282	37.94%
bigblue3	1095519	1293	1123170	3833218	56.68%
bigblue4	2169183	8170	2229886	8900078	44.35%

Table 1: Multi-million gate benchmarks from the ISPD 2005 Placement Contest [10] . Images of adaptec1 can be found in Figure 1.

The remainder of the note is organized as follows. In Section 2 we outline basic issues arising when dealing with 64-bit execution modes. Section 3 emphasizes the need for rigorous simulation infrastructure, including distributed clusters of workstations. Implications for source code development are discussed in Section 4, and Section 5 describes the requirements and the impact of detailed diagnostics on the development process.

2. SCALING TO 64 BITS?

These days 64-bit hardware is more common and more powerful than ever before with a wide variety of platforms from which to choose — Hewlett-Packard HP-PA, Sun Ultra-Sparc, AMD Opteron and Intel Pentium4-Xeon EM64T for example. Many of these platforms support both 32- and 64-bit execution, but usually give the performance edge to 64-bit code (mainly due to the increased number of registers available). This combined with the ability to use far greater quantities of memory than 32-bit machines make 64-bit platforms very attractive, and in some cases essential, for EDA tools.

Unfortunately this enhanced performance and expanded addressable memory space comes at a cost: larger memory footprint. Pointers double in size when switching from 32 to 64 bits which can dramatically increase memory use. Integers remain at 32 bits, so it becomes more convenient to use indices instead of pointers where possible. Indices require less memory, which contributes to better memory locality and increased cache utilization, and allow for range checking when necessary. Note that circuit hypergraphs are not going to have two billion vertices any time soon so 32-bit indices are safe in the foreseeable future.

One must still be aware when transitioning to 64-bits that standard data structures, e.g. in the Standard Template Library (STL), still must use pointers to some extent, therefore a 30-50% increase in memory usage in 64-bit mode can be expected unless the code

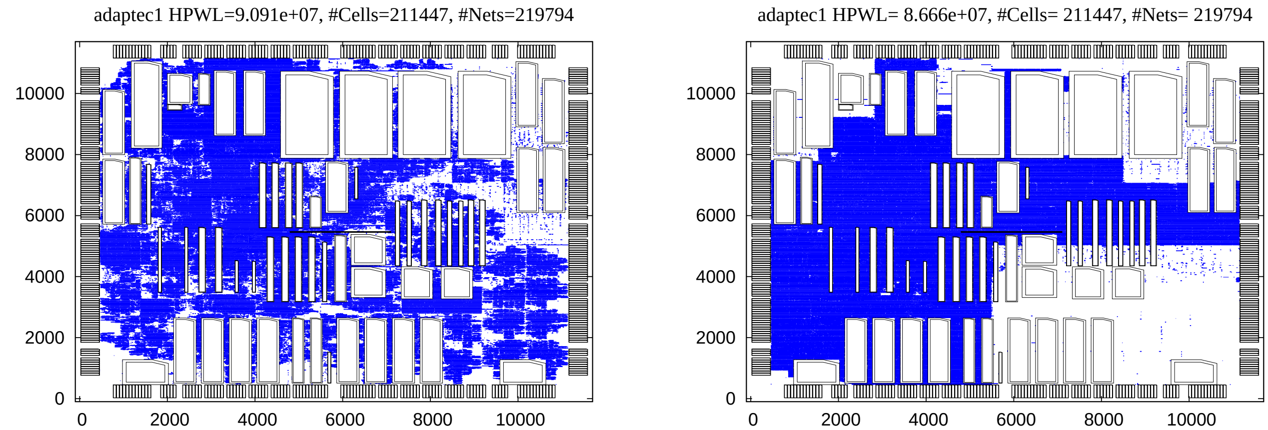

Figure 1: Sample placements produced by Capo [4] of the multi-million gate benchmark Adaptec1 from the IBM sponsored ISPD 2005 Placement Contest [8] . These plots clearly illustrate the effects of using different whitespace allocation techniques.

explicitly manages memory allocation. In summary, if 4GB are not enough for a process, the source code can be recompiled in 64-bit mode, but one should then expect to use closer to 6GB memory, perhaps, with a slight (10%) speed-up.

3. SIMULATION CLUSTERS

Evaluating and debugging tool runs on very large designs is an extremely time-consuming process. Some bugs show up only after many hours of runtime, and require very large datasets. This mandates the use of distributed computing clusters to evaluate every significant change to core algorithms.

3.1 Sample infrastructure

In order to make the most efficient use of programmers' time, we have found it necessary to perform literally thousands of simulations of our algorithms on various available benchmarks to keep abreast of the performance impact of algorithmic changes as well as catching programming errors that are inadvertently introduced. Using benchmarks from several independent sources (for example [3, 10, 13]) is crucial to fairly evaluate our tool [9]. To this end, we use a cluster of over one hundred Pentium workstations dedicated to validating the results of ongoing algorithm development.

3.2 Randomized testing and reproducibility

Several techniques useful to EDA tools include the use (or the possibility for) randomization. Randomization can lead to improved solution quality (the best of 20 runs is usually much better than the average) and can improve the efficiency of testing (every independent run may exercise different paths in the tool) but significantly increases simulation load during testing as regressions must be detected in terms of average quality.

There are ways to "stabilize" the end results of randomized algorithms so that a designer can expect a certain solution even with different random seeds if this is desirable [1]. Similarly, there is a simple way to reproduce the results of randomized runs for debugging purposes, as explained below.

Since random number generation can vary wildly between different platforms and operating systems, we have found it most efficient for purposes of reproducibility to create our own platform-independent random number generation utilities. We override random number generators that may be included with standard data structures (as data structures from different packages may use different random number generators) so that all generated random sequences can be reproduced exactly at a later time with the knowledge of the initial random seed.

4. MODULAR TOOL INFRASTRUCTURE

Due to the sheer size and complexity of code required to build an efficient and robust EDA tool, it is often preferable to divide the tool's workload into well defined sub-problems that can potentially be solved more easily. The tool becomes separated into solvers of various problems and logic that drives the tool by constucting and submitting instances to solvers, interpreting solutions and making decisions about what to do next. These separations of labor make it much easier to understand the flow of the tool and ensure that the tool runs correctly.

4.1 Modular optimizers

The solvers described in the division of labor above should be wholly self-contained so that they can be individually tested for efficiency and correctness. Interfaces should be developed such that solvers can be used as stand-alone applications as well as well as parts of a tool with much larger scope.

Well defined input file formats for each component necessary for individualized testing. Small instances with known solutions aid greatly in the debugging process. Given that the overall task of the main tool will be broken into smaller pieces which can be solved more manageably, it is often very useful to save intermediate problems for later study. For example, a min-cut partitioner can save internally-created partitioning problems for use in debugging and further algorithmic enhancements.

4.2 Internal consistency checkers

Modules must "sanity-check" their inputs as well as the solutions they provide to detect problems as early as possible in development. For example, a wirelength evaluator should not blindly return nonsensical values such as infinite or negative length and array lookups can be subject to bounds checking. These types of internal checks can help identify and localize errors quickly. Runtime and memory penalties are often associated with consistency checking, but can usually be placed in non runtime critical areas of execution. Also once code is deemed mature, many of these checks can be disabled for the sake of efficiency.

5. DIAGNOSTICS

Given the extremely long runtimes on large designs and their large memory footprints, it may be difficult or even infeasible to debug interactively using popular IDEs. Therefore, successful approaches to debugging require articulate, automated and concise diagnostics. Such diagnostics typically takes the form of (i) execution logs, with tunable level of detail, (ii) visualizations that can be quickly adjusted to current needs, and (iii) error reporting.

5.1 Visualization

Visualizing intermediate steps in the running of a tool is extremely important as it can provide much more information about the working of the tool than a single output number such as total power or circuit delay. Displaying something as simple as the placement of cells in a layout is no simple task for multi-million gate designs due to the sheer volume of data. One can use commercially available solutions based on OpenGL, but these generally require specialized graphics hardware for performance and are difficult to customize. For this reason, we emply rasterization techniques to cut down on image size. We assume a reasonable screen resolution for current workstations and coalesce multiple objects into single horizontal and vertical segments when the objects themselves would not be distinguishable due to pixel size. This technique is exhibited in the images in Figure 1 where the benchmark has several hundred thousand small cells which would indistinguishable no matter how accurately drawn given the resolution of the image.

The core area of modern SoC designs is determined largely by pin count when area array I/Os are used. These types of pins are generally large and require significant area resources. For designs to be routable, they must also have sufficient routing resources which inevitably leads to increased area and increased whitespace. The ISPD Placement Contest benchmarks [10] are consistent with these statements. With such a large number of movable objects and relatively low utilization (which equates to high amounts of whitespace), whitespace allocation has a dramatic effect on wirelength, which becomes readily apparent from Figure 1. Two dramatically different whitespace allocation schemes were used produce the pictured placements. The pictures show exactly where whitespace is allocated and can give clues as to where futher algorithmic improvement is possible.

The choice of format for representations is also important for quick and easy visualization. In our development we have found the freely available plotting tool Gnuplot [7] invaluable for producing pictures of our work as it has a very simple interface and allows to easily overlay several different data sets at once. The input format of Gnuplot is also extremely convenient because it can easily be altered by hand or by scripts after being produced, unlike most image formats. For example, using Gnuplot we regularly plot the cut line decisions (see Figure 2 left) and the outlines of areas of local block packing (see Figure 2 right) of our min-cut floorplacer Capo [4] on various benchmarks to see if techniques are working as expected and what decision making processes may need to be altered for better solution quality.

5.2 Memory usage

Memory usage and scalability are paramount concerns when working with multi-million gate designs. Tools should take care to account for their own memory usage and report statistics to aid tool development and use. Figure 3 illustrates some of the useful memory statistics that can be gathered such as page faults (useful to detect thrashing), resident memory, peak memory, and a place in the code responsible for the most memory use. High memory usage can lead to artificially poor performance due to poor cache coherency or heavy swapping in the extreme case when available memory is exhausted. Average memory usage will allow the user to better match hardware with software and determine the feasibility of running larger designs based on extrapolations from smaller ones. Peak memory usage, in addition to the sections of code that consume it, will help tool developers identify areas of the tool for further refinement. Detecting and reporting thrashing due to heavy swapping can also save valuable time since thrashing often nullifies the efficiency of a tool and makes the thrasing system unresponsive.

5.3 Runtime reporting

Like recording and reporting memory usage, keeping internal accounts of runtime for the major components of the tool is necessary for targetting algorithmic refinement. They also provide a sanity check for the tool and a way to predict the runtime of larger tasks. There is a subtlety to measuring both memory and runtime usage: the granularity must not be so coarse as to provide inaccurate information but not so fine as to detrementally impact the overall efficiency of the tool. Figure 3 offers a sample runtime breakdown automatically generated by our tool Capo [4].

```
CapoPlacer took:                 144.506sec
Breakdown by component -
Fiddducia-Matheyses Partitioner:   8.99sec  (6.22%)
Multi-Level FMPartitioner:        64.34sec (44.52%)
Optimal End Case Partitioner:      4.15sec  (2.87%)
Partitioning Problem Setup:        4.78sec  (3.31%)
Optimal End Case Placer:          10.47sec  (7.24%)
End Case Placement Problem Setup: 0.17sec  (0.12%)
Level Statistics:                  0.43sec  (0.30%)
Feedback Processing:               0.49sec  (0.34%)
Block Packing:                    49.28sec (34.10%)
  Block Packing Clustering:  0.07sec  (0.05%)
  Block Packing Annealing:  44.34sec (30.68%)
  Number of Block Packing Instances: 97
    (successful: 84, failed: 13)
  The largest Block Packing instance had 9 macros.
  The largest failed Block Packing
   instance had 3 macros.
Total measured runtime: 143.09sec (99.02%)

Minor page faults: 164649 Major page faults: 0
Current resident memory: 19.5078MB
Peak process memory: 29.168MB
Peak process memory observed in
"Multi-Level FMPartitioner after clustering"
```

Figure 3: Runtime breakdown and memory usage reported by the Capo [4] placer for the benchmark IBM01 pictured in Figure 2.

5.4 Assertions and error reporting

Catching errors early and exiting gracefully is always more beneficial than a hard crash such as a segmentation fault, but it is often not enough. When an imbedded code assertion is tripped, it should provide as much information as possible so that the problem can be reliably reproduced as well as accurate information as to what the tool was doing when it encountered the error. These pieces of information are essential to tracking down and removing errors that may not surface often in a potentially huge codebase. A code base that can also annotate its current activities with useful information upon request can also make a nontrivial difference.

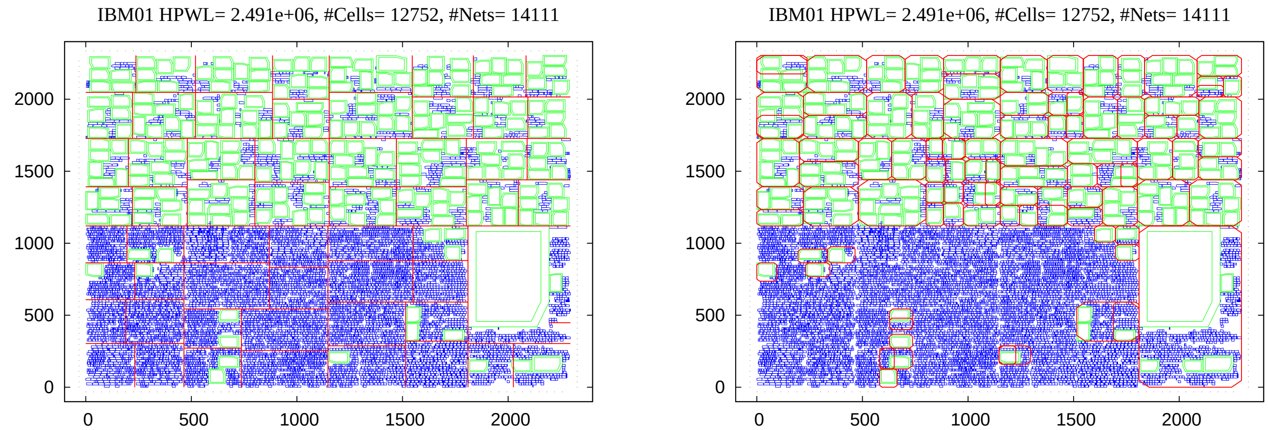

Figure 2: **The use of visualization tools can be extremely effective in identifying possible errors and areas for improvement. Intermediate images offer more insight into the workings of the tool than simple summary numbers produced at the end of a run. The plot on the left shows the cut lines chosen by our min-cut placer Capo [4] for the first 6 layers of bisection on the mixed-size design IBM01 from [3] . The plot on the right shows the same placement instance but highlights the areas of local block packing.**

6. CONCLUSIONS

Developing efficient tools that can handle multi-million gate designs and beyond requires scalable algorithms as a base, but often much more. Any and all mistakes and inefficiencies in a tool are amplified when run on designs of this magnitude. Thus it is necessary to be vigilant when developing a tool to certify that it is up to the task. Doing so requires careful data structure design and implementation while at the same time considering choice of hardware platform, rigorous testing, ease of debugging and informative diagnostics. In this note we have touched upon a few of the areas that we have found to be extremely important while developing our min-cut floorplacer Capo [4] and described some of the techniques we employ to deal with these important issues.

7. REFERENCES

[1] S. N. Adya, I. L. Markov and P. G. Villarrubia, "On Whitespace and Stability in Mixed-Size Placement," in Proc. Intl. Conf. on Computer-Aided Design(ICCAD) (.pdf), San Jose, November 2003, pp. 311-318.

[2] S. N. Adya et al., "Benchmarking for Large-Scale Placement and Beyond," *IEEE Trans. on CAD* 23(4), pp. 472-488, 2004.

[3] S. N. Adya, S. Chaturvedi and I. L. Markov, "ICCAD'04 Mixed-size Placement Benchmarks," in *GSRC Bookshelf*, `http://vlsicad.eecs.umich.edu/BK/ICCAD04bench`

[4] S. N. Adya, S. Chaturvedi, J. A. Roy, D. A. Papa, I. L. Markov,"Unification of Partitioning, Placement and Floorplanning," *ICCAD*, 2004, pp. 550-557.

[5] A. E. Caldwell, A. B. Kahng, I. L. Markov, "Design and Implementation of Move-Based Heuristics for VLSI Hypergraph Partitioning," *ACM J. on Experimental Algorithms*, vol. 5, 2000.

[6] A. E. Caldwell, A. B. Kahng, I. L. Markov, "Hierarchical Whitespace Allocation in Top-down Placement," *IEEE Transactions on CAD* 22(11), Nov, 2003, pp. 716-724.

[7] Gnuplot, `http://www.gnuplot.info`

[8] ISPD 2005 Placement Contest, `http://ispd.cc/contest.htm`

[9] P. H. Madden, "Reporting of Standard Cell Placement Results," *ISPD* 2001, pp. 30-35.

[10] G.-J. Nam, C. J. Alpert, P. Villarrubia, B. Winter and M. Yildiz, "The ISPD2005 Placement Contest and Benchmark Suite," to appear *ISPD* 2005.

[11] D. A. Papa, S. N. Adya, I. L. Markov, "Constructive Benchmarking for Placement," *GLSVLSI* 2004, pp. 113-118. `http://vlsicad.eecs.umich.edu/BK/FEATURE/`

[12] X. Tang, R. Tian, M. D.F. Wong, "Optimal Redistribution of White Space for Wire Length Minimization," *ASPDAC* 2005, p. 412.

[13] X. Yang, B.-K. Choi, M. Sarrafzadeh, "Routability Driven White Space Allocation for Fixed-Die Standard-Cell Placement," *ISPD* 2002, pp. 42-50.