Toward Quality EDA Tools and Tool Flows Through High-Performance Computing

Aaron Ng and Igor L. Markov
Department of EECS
The University of Michigan
{aaronnn, imarkov}@eecs.umich.edu

Abstract

As the scale and complexity of VLSI circuits increase, Electronic Design Automation (EDA) tools become much more sophisticated and are held to increasing standards of quality. New-generation EDA tools must work correctly on a wider range of inputs, have more internal states, take more effort to develop, and offer fertile ground for programming mistakes. Ensuring quality of a commercial tool in realistic design flows requires rigorous simulation, non-trivial computational resources, accurate reporting of results and insightful analysis. However, time-to-market pressures encourage EDA engineers and chip designers to look elsewhere. Thus, the recent availability of cheap Linux clusters and grids shifts the bottleneck from hardware to logistical tasks, i.e., the speedy collection, reporting and analysis of empirical results. To be practically feasible, such tasks must be automated; they leverage high-performance computing to improve EDA tools.

In this work we outline a possible infrastructure solution, called bX, explore relevant use models and describe our computational experience. In a specific application, we use bX to automatically build Pareto curves required for accurate performance analysis of randomized algorithms.

1 Introduction

As the software industry matures and commercial tools become more sophisticated, the importance of quality control grows. EDA software is especially difficult to test, thanks to long runtimes and the fact that solutions to hard combinatorial and numerical problems cannot always be verified immediately. The value of quality control is underlined by economical considerations. EDA tools are much more expensive than office applications, and since they sell considerably fewer licenses, it is more critical to satisfy first adopters rather than rely on their bug reports. Commercial EDA tools are also distinguished by frequent updates and relatively short average lifespans — while spreadsheet formats and email protocols remain unchanged for many years, the rapid advance of semiconductor technologies requires new design optimizations. This never-ending demand for new EDA tools makes quality control more challenging because rigorous simulation must be completed and result analyzed under time pressure. A case in point, empirical data reported in recent academic work on place-and-route [19] required several CPU-months and was produced on a custom-designed grid-like computational facility PUNCH (Purdue Computational Hub). This motivates a closer look at EDA benchmarking [1], with an eye on new types of automation and increased productivity.

Unlike office applications, EDA tools are typically used in design flows assembled by users. Such flows include dozens of applications, often from different vendors, and the quality of end result is largely a function of how the tools interoperate. While each EDA tool addresses a certain niche, such tools are rarely valuable in isolation. Therefore the number of configurations in testing and benchmarking can be very large, suggesting the use of massively-distributed computing systems. Cost considerations for high-performance computing suggest two types of distributed systems — clusters and grid-computing networks. A number of high-performance clusters have been built in the industry and academia [16, 31] using commodity components, often at a very low cost. Incidentally, seven of the world's top-10 supercomputers of 2003 [30] are clusters [21]. Many hardware and EDA vendors already own high-performance clusters which run EDA tools — either in production use or for regression-testing. The recent availability and popularity of cheap 32-bit and 64-bit Linux clusters underlines the need for logistical support, i.e., automatically handling heterogeneous EDA tools, design flows, input and output files, performance data and user-friendly reporting.

Among other tasks, simulation and formal verification are particularly time-consuming. However, regression-testing is demanding for practically every tool — nightly builds and regression runs are common both for hardware designs and EDA tools. Such runs can automatically catch designer and programmer oversights, and ensure easy diagnostics because relatively few changes can happen in one day. However, such checks are often performed with simple-minded scripts and do not identify subtle performance degradation. For example, several postings at the ESNUG mailing list maintained by John Cooley observe poorer results with Synopsys DC/PhysOpt versions 2003.03-1 and 2003.06-1 compared to the version 2003.05-SP2. The newer versions create layouts with 5-10% larger path delay.[1] Given the complexity of even basic algorithms and data structures, EDA tools may contain conceptual flaws in addition to coding mistakes. Identifying such flaws and rectifying them requires considerable effort in empirical evaluation and benchmarking. To this end, a study on placement benchmarking [1] points out the risks of using a small base of benchmarks — many existing placement tools have clearly been tuned to certain benchmark sets and perform poorly on other benchmarks. To be aware of such problems one must use a larger, more diverse set of benchmarks, in conjunction with automatic means to intelligently summarize simulation results in succinct reports. Not only

[1]See http://www.deepchip.com/items/0416-08.html and http://www.deepchip.com/items/0417-01.html

EDA vendors, but also large EDA customers regularly evaluate software because committing to an overly specialized, unreliable tool may threaten time-to-market and jeopardize design projects.

The evaluation of EDA tools typically consists of a large number of relatively independent jobs, perhaps with some dependence constraints [1]. Many of these jobs apply the same tool to the same design problems in different configurations to observe the difference in results, and thus can run in parallel. Often it is convenient to view a chain of several tools as a single tool, e.g., to determine robust configurations that produce good results or fit into prescribed constraints on runtime, memory usage, number of licenses and technology requirements. Explicitly supporting abstractions of this kind is one of the challenges addressed in our work. Other challenges are in the logistics of scheduling, distributing and launching thousands of jobs, collecting results and evaluating them with minimal manpower. Benefits to hardware engineers include robust and scalable design flows, as well as automatic monitoring of those flows to detect early signs of trouble. Business success of start-ups working in this field, such as Reshape, suggests the value of relevant logistical frameworks to hardware design.

In this paper we propose a logistical framework that greatly automates the evaluation of EDA tools and tool flows, starting with Web-based upload of source code or executables, enabling automatic pairing of tools and benchmarks, facilitating automatic production of Pareto curves (for randomized optimization algorithms) or various tabular reports, and allowing the user to mix and match tool flows that can be subsequently scheduled for empirical evaluation. Such logistical support allows to leverage high-performance computing technologies for quality control and improvement of EDA tools.

The remainder of the paper is structured as follows. In Section 2 we cover related work and outline several desired functionalities. In Section 3 we describe the features and use models of our implemented system. In Section 4 we discuss implementation details of our system. Section 5 summarizes our computational experience. Our conclusions and ongoing work are described in Section 6.

2 Background

Our work builds upon an earlier GSRC Bookshelf project [6] that contributed an extensive online collection of free VLSI design tools, as well as benchmarks and methodologies for evaluating them. Thus, our major goal is to make the GSRC Bookshelf executable, adding support for composing, running and evaluating tool flows. The proposed system is called Bookshelf.EXE, or bX.

Related work includes Flowtracer [13], WELD [10], OmniFlow [4], Odyssey [5], Nelsis [22], ASTAI(R) [3], JavaCAD [11] and MOSCITO [26]. These contribute a number of useful concepts that we use in this work, such as distributed and collaborative computing, various degrees of automated flow management, high-level operations with flows (composition, repetition, evaluation, etc), and user-friendly UI. There are the load share facilities for grids such as PBS and Sun's Grid Engine, which inspire the leveraging of load sharing across heterogeneous hosts. bX aims to bridge the gap between the basic provisions of a load share facility and the users, adding value to a load sharing facility to make it more useful and convenient for EDA tool evaluation. There are also the more widely known distributed computing efforts such as SETI@home. These are different from bX because their servers and computational clients are hardcoded to deal with very specific problems. bX, on the other hand, proposes to address a broader domain, which is to be able execute untrusted user-submitted code, safely and securely, on a distributed network. Also, one of the applications of bX is to evaluate EDA tools, and as such, execution-time resource usage and solution quality across various hardware must be accounted for in a consistent manner. bX is similar to RTDA's Flowtracer with a few of the exceptions being that bX does not perform runtime tracing [20], and we explore, to a greater depth, various use models made possible by a flow management and distributed computing infrastructure.

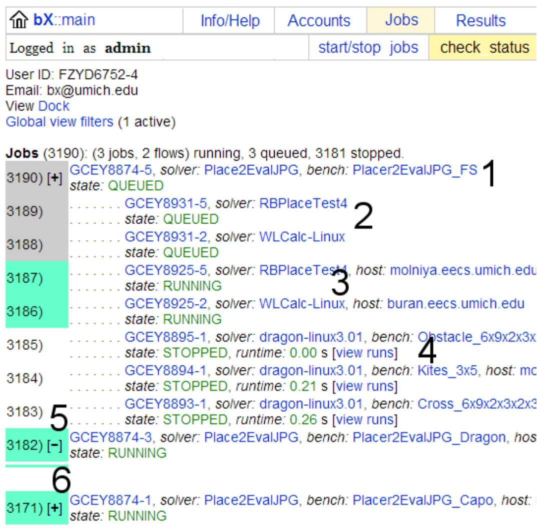
User interface in bX emphasizes simplicity and genericity — tool developers do not need to modify their tools to work within the bX framework (as long as the tools can run in batch mode). It uses typical Web interfaces available through any browser and does not require users to install new software. Installing bX clients (for those who wish to contribute CPU time to bX) is also fairly easy and does not require creating special UNIX accounts or recompiling system kernel. bX is also low-maintenance from the administrator's perspective as it is automated and self-contained, in a sense that users are free to perform actions such as upload tools and benchmarks and execute jobs without any administrative intervention, as long as the actions are within pre-defined policies.

With the advent of distributed computing, the adoption of more rigorous practices in benchmarking [1] demands additional flow automation for verifying correctness, averaging, evaluating combinations of flows, finding most difficult benchmarks, etc. A relatively new challenge is the organization of program competitions and comparisons, both in the simulation and scheduling back-end and at the front-end UI. To further support collaboration, (i) the results of experiments should be automatically available in digest form, emphasizing the main trends, and (ii) computational experiments themselves should be easily reproducible by other researchers. A large number of tools, benchmarks and flows should be available, compatible and easy to manipulate using a common interface, including job scheduling and contributing new items. We also observe that some previously proposed distributed systems do not offer the ability of distributed execution. Distributed execution unfortunately brings up a host of security-related issues.

3 Features and Use Models

bX automates and simplifies large-scale experimentation. The user starts by uploading tools and benchmarks, or by selecting tools and benchmarks made available by other users. The user may then compose tool flows (with various dependencies) and run them on selected benchmarks. The user can monitor the status and progress of all jobs using web-based UI. After the jobs complete, all results are available through the same interface. Important data can be extracted from raw output files and tabulated using presentation tools offered by bX.

Flows and scripts. In general, scripts describe the EDA tools involved in a flow, and inter-tool dependencies that imply the order of tool invocations. The interface to compose flows is web-based and tries to accommodate users with various levels of commitment and expertise: novices may use check-buttons and *script wizards* to compose existing tools, and experts may edit and customize bX-generated scripts to suit their needs. For example, one can set up bX to find a combination of program execution flags that produce best results for a given input. This can be accomplished by a bX script that schedules iterative job runs with varying execution parameters, and bX will automatically schedule and execute the jobs over all designated computational hosts and manage their results. bX scripts are constructed using bX's Script Composer, described in Section 4.



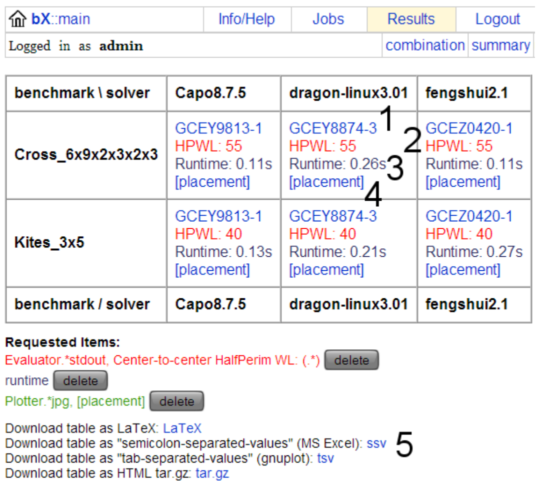
Logged in as **admin**

User ID: FZYD6752-4
Email: bx@umich.edu
View Dock
Global view filters (1 active)

Jobs (3190): (3 jobs, 2 flows) running, 3 queued, 3181 stopped.

```
3190) [+] GCEY8874-5, solver: Place2EvalJPG, bench: Placer2EvalJPG_FS
          state: QUEUED
3189)     . . . . . . . GCEY8931-5, solver: RBPlaceTest4
          . . . . . . . state: QUEUED
3188)     . . . . . . . GCEY8931-2, solver: WLCalc-Linux
          . . . . . . . state: QUEUED
3187)     . . . . . . . GCEY8925-5, solver: RBPlaceTest4, host: molniya.eecs.umich.edu
          . . . . . . . state: RUNNING
3186)     . . . . . . . GCEY8925-2, solver: WLCalc-Linux, host: buran.eecs.umich.edu
          . . . . . . . state: RUNNING
3185)     . . . . . . . GCEY8895-1, solver: dragon-linux3.01, bench: Obstacle_6x9x2x3x
          . . . . . . . state: STOPPED, runtime: 0.00 s [view runs]
3184)     . . . . . . . GCEY8894-1, solver: dragon-linux3.01, bench: Kites_3x5, host: mc
          . . . . . . . state: STOPPED, runtime: 0.21 s [view runs]
3183)     . . . . . . . GCEY8893-1, solver: dragon-linux3.01, bench: Cross_6x9x2x3x2x3
          . . . . . . . state: STOPPED, runtime: 0.26 s [view runs]
3182) [−] GCEY8874-3, solver: Place2EvalJPG, bench: Placer2EvalJPG_Dragon, hos
          state: RUNNING
3171) [+] GCEY8874-1, solver: Place2EvalJPG, bench: Placer2EvalJPG_Capo, host:
          state: RUNNING
```

Figure 1. Real-time monitoring from bX's web UI — *(1) Queued flow. (2) Queued jobs. (3) Running jobs. (4) Completed jobs with links to results. (5) Running flow with expanded view. (6) Running flow with collapsed view.*



Logged in as **admin**

benchmark \ solver	**Capo8.7.5**	**dragon-linux3.01**	**fengshui2.1**
Cross_6x9x2x3x2x3	GCEY9813-1 HPWL: 55 Runtime: 0.11s [placement]	GCEY8874-3 HPWL: 55 Runtime: 0.26s [placement]	GCEZ0420-1 HPWL: 55 Runtime: 0.11s [placement]
Kites_3x5	GCEY9813-1 HPWL: 40 Runtime: 0.13s [placement]	GCEY8874-3 HPWL: 40 Runtime: 0.21s [placement]	GCEZ0420-1 HPWL: 40 Runtime: 0.27s [placement]
benchmark / solver	**Capo8.7.5**	**dragon-linux3.01**	**fengshui2.1**

Requested Items:
Evaluator.*stdout, Center-to-center HalfPerim WL: (.*)
runtime
Plotter.*jpg, [placement]

Download table as LaTeX: LaTeX
Download table as "semicolon-separated-values" (MS Excel): ssv
Download table as "tab-separated-values" (gnuplot): tsv
Download table as HTML tar.gz: tar.gz

Figure 2. Automatic table generation — *(1) Link to job details. (2) Half-perimeter wirelength, extracted from the `stdout` of the flow's Evaluator job. (3) Job runtime, from the job's execution trace. (4) Link to the placement plot image, taken from the flow's Plotter job. (5) The table is downloadable in various formats.*

Reporting and transparency. bX automatically collects job information such as runtime, memory usage, the job process' exit status, all output files generated by the job, as well as the console and error output streams (called `stdout` and `stderr` on POSIX-compliant operating systems). The user may monitor the status, health and progress of flows and jobs in the flow in real-time. Figure 1 is a screenshot of the web UI. bX automatically stores and organizes the large amounts of flow information, making them available later for download or evaluation in bX.

Extracting results from output files for automatic table generation. After jobs complete, users typically end up with multiple output files per job, along with the jobs' execution traces containing data such as runtime and memory. We find that users routinely extract data from these output files, and later organize them in a manner suitable for comparison, either visually or with the help of additional software. However, since users typically have results from hundreds or thousands of jobs, users would be burdened with the additional task of creating and maintaining scripts to perform these tasks. bX simplifies this process for the user, with automatically generated tables. The user may specify the rows and columns of a table and bX will automatically tabulate all matching jobs. The user may then select items to be inserted into the table. Items can be values from job information (such as runtime), values from job output files (such as placement wirelength), or links to output files. The tables are downloadable in a variety of formats. Figure 2 shows a screenshot of a customized table generated in bX, from the results of a completed flow.

Pareto curves and automatic regression testing. When an EDA tool evolves from one version to the next, developers and testers need to compare the quality of result to that of previous versions. However, final results can often be improved just by increasing runtime, e.g., considering more possibilities. Given that increased runtime is undesirable, it is important to reason about trade-offs. Empirically, one can plot quality of results versus runtime — such plots are often called *Pareto curves*. They are particularly appropriate to evaluate randomized algorithms which can produce different outputs if launched many times on the same input data. The performance of a single run is statistically described by average quality of result and average runtime. However, one can also perform two runs and take the better result. Plotting average best-of-two versus double the average runtime will generate another point on the Pareto curve. Similarly, average-best-of-N quality can be plotted against runtime of N starts. In practice, such data can be collected by clever over-sampling which is much faster than producing all datapoints independently by direct averaging. Either way, this process entails many iterations of dispatching tool runs, collecting and processing raw results, generating Pareto points, checking convergence criteria (enough samples for a reliable result) and checking the curves for leveling off, i.e., when additional runtime does not significantly improve the result anymore. This requires a tedious effort involving hundreds of jobs for every tool-benchmark pair evaluated; each job taking at least as long as the tool requires to process the benchmark.[2] Because of the logistical complications just described, Pareto curves are practically unavailable unless they can be built automatically. To this end, bX supports automatic construction of Pareto curves, and we report on this feature below in Section 5. Observe that all the job runs that make up the sampling pool for Pareto points are independent of each other, exhibiting parallelism that is easy to exploit in a distributed system. In addition to Pareto curves, bX users may construct regression test suites in bX by selecting built-in test primitives such as basic less-than/greater-than comparison, average percent difference, and comparing outputs with expected outputs. The user may also upload custom tests to be used in regression test suites. The test suites can then be launched at every tool update and left to run unattended; and the results collected later to identify regressions or failures. Features such as automatic Pareto curve generation minimize user interaction and increase throughput of jobs in regression testing through automation and distributed execution.

[2]For example, a VLSI placer may take 30 minutes to run on a circuit with 200,000 standard cells and several hours on a circuit with 1M standard cells.

4 Implementation

bX, at its core, is a system that executes and manages jobs on a distributed network.

Infrastructure. Technologies used in bX are selected based on their relevance and the existence of foundations for easily building security upon. C is used for the back-end hosts. XML-RPC is the language between distributed bX entities. XML-RPC is a specification and a set of implementations that allow software running on disparate operating systems, running in different environments to make procedure calls over the Internet. XML-RPC was selected over technologies like CORBA and SOAP because it is simple and suits our needs. XML-RPC uses XML for encoding, and HTTP for transport. One of the benefits of this is that securing transmissions is as easy as switching to use HTTPS instead of HTTP. NFS is used to solve the problem of sharing files across nodes on a geographically distributed network. NFS was selected because after setup, NFS allows remote file accesses to work transparently, similar to local file accesses. NFS is also actively developed, with a focus on security. An HTTP server acts as a portal between the user and bX. PERL scripts on the HTTP server allow the user to interact with bX through dynamic web interfaces. Roughly a third of bX's over 30,000 lines of new code is for the web front-end, which drives the UI. The remainder of the code is for the back-end. The bX back-end currently only runs on Linux because of the dependency on the `/proc` file system for capturing execution-time job information. In the future, more UNIX-based operating systems may be supported.

Managing failures. A distributed computing environment is prone to failures. bX's fault tolerance is made up by the fault tolerance of its sub-components. For example, by using NFS and HTTP at the application layer, we do not have to worry about the reliability of the transport layer, such as the correct transmission of data packets. By propagating the benefits of the solutions of reliable sub-components up to the top level, the problem can be simplified to failures such as power loss or network disconnections. Catastrophic failures such as hardware failures can be solved by redundancy, but this is a significant topic on its own. At the application layer, good database accounting is used to ensure that bX always transitions from state to state cleanly, is never in an intermediate state, and is always able to recover to a stable state after a failure. The underlying levels are abstracted away from the user level, which sees computational resources as either *available to run the user's queued jobs* or not, and errors are usually more high-level, such as access permissions on a particular tool.

Security considerations. From a high-level perspective, a robust implementation of user policies and access control is necessary for security. bX implements access control using groups and permissions similar to UNIX operating systems. Since bX is a common element across the distributed network, it can be designed to be the sole interface between users and all aspects of the network. For the most part, this allows for implementations of solutions independent of the underlying infrastructures of heterogeneous hosts. At a low-level, needs for security are generally due to the nature of distributed computing. It is necessary to secure:
(i) data transmissions between hosts on the network, (ii) the execution environment of the computational hosts, and (iii) the data stored on and reported by computational hosts. To protect the data transmissions, we plan to enable the security implementations of NFS and XML-RPC. The other concerns can be addressed by having administrative control over the hosts in the distributed infrastructure. For example, with respect to item (ii), there are serious concerns for running untrusted processes on a distributed network of hosts. As a start, the user processes can be confined using the chroot() system call, severely restricting its access in a host's file system. A process must also be restricted in its access to system calls. With software like Systrace [25], fine-grained policies may be generated for each process, constraining a process' access to the system.

Table 1. Hardware used in bX

Role	Machine configuration
Central server and Web UI	CPU: Dual 2GHz AMD Athlon Memory: 2GB
Computational hosts (3)	CPU: Dual 2GHz Pentium 4 Xeon Memory: 1-3GB
PBS commander	CPU: 2.8GHz Pentium 4 Memory: 1GB
PBS cluster	CPU: 2.4GHz Memory: 1GB

Script Composer. To facilitate the production and reproduction of flows, flows are instances of bX scripts. bX scripts are PERL scripts using bX's Script Composer API. The API is used to explicitly define the components of a flow and dependencies between components of a flow. The API provides an abstraction over the many underlying details of executing jobs such as flow management and distributed execution. By building on top of a language like PERL, we also inherit the power and flexibility of a programming and scripting language.

Automatic Pareto curve generation. The user initiates Pareto curve generation by selecting a tool and a benchmark to run. The user also provides bX with a regular expression for extracting a figure of merit from the outputs of the jobs, which represents solution quality. bX then starts creating an initial sampling pool by running a certain number of jobs. After the jobs complete, a Pareto curve is generated from the sampling pool and evaluated for convergence and leveling-off. If the resultant plot is not sufficiently accurate, the sampling pool is doubled by running more jobs, and the process is repeated. The stopping criteria for Pareto curve generation require that curves should be strictly decreasing, and that the values of the last 4 points in a curve not vary by more than 1%. bX will also stop Pareto curve generation when the size of the sampling pool for a curve exceeds 1000 jobs.

5 Computational Experience

bX is run over the University of Michigan TCP/IP intranet, but can be trivially extended to a geographically distributed TCP/IP network. Table 1 describes hardware used in bX. For executing jobs, bX makes use of a combination of regular machines, and a PBS [24] cluster. The regular machines execute jobs locally, and the PBS cluster is commanded by bX via a special machine that manages PBS jobs.

Usage statistics. A prototype version of bX serviced users from a number of universities: the University of Michigan - Ann Arbor, Purdue University, Binghamton University, University of California - Santa Barbara, and University of Waterloo, hosting 92 tools (159 MB), 571 benchmarks (320 MB) and the results of 11331 jobs (8.3 GB). bX has been used for the evaluation of EDA tools for circuit layout (Capo, Dragon, Kraftwerk, FengShui, mPL) as well as verification and Boolean satisfiability (Chaff, zChaff, zRes, Berkmin, GRASP, Cassatt).

Sample applications. Figure 1 shows a screenshot of multiple instances of a flow in progress, as seen from the web UI. Figure 2 shows how results of flows can be automatically tabulated in bX.

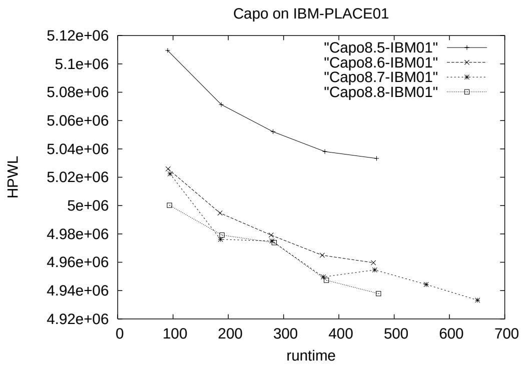


Figure 3. Pareto plots automatically generated from runs of different versions of Capo with the IBM-PLACE01 benchmark. The plots suggest steady improvement through all four versions of Capo on this benchmark.

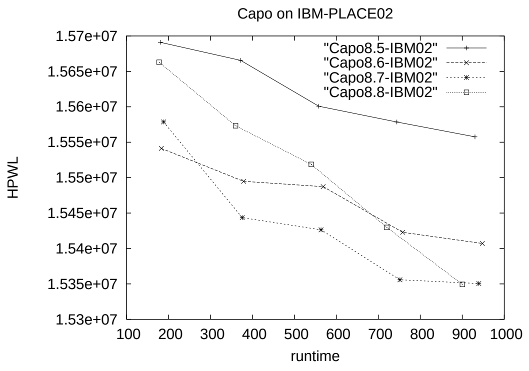


Figure 4. Pareto plots automatically generated from runs of different versions of Capo with the IBM-PLACE02 benchmark. The plots suggest steady improvement in versions 8.5 through 8.7 of Capo, but a regression in version 8.8 on this benchmark.

Figures 3-5 describe empirical evaluation of free software for VLSI placement (Capo [7]) and circuit partitioning (MLPart [8]), which are comparable to industrial tools. Figures 3 and 4 are Pareto curves automatically generated by bX, demonstrating the solution quality versus runtime relationships for different versions of Capo run with IBM-PLACE benchmarks. Figure 3 suggests that on the IBM-PLACE01 benchmark, the performance of Capo steadily improved through the four versions. However, Figure 4 suggests that while versions 8.5 through 8.7 of Capo steadily improved on the IBM-PLACE02 benchmark, version 8.8 of Capo took a step back. For the plots here, a total of 1760 jobs were run, having a combined runtime of 57.77 CPU hours. The jobs were distributed over 3 computational hosts at 2 jobs per host at a time. bX took 16.25 hours to produce the plots. Ideally, it should take 9.63 hours. This discrepancy can be accounted for by a scheduler bug, scheduler inefficiency, network file transfer time, and the fact that the computational hosts were also multi-user workstations where bX processes are set to run at a lower priority than the regular processes on the workstations. On the face of it, all these contributors to the excess time taken allow room for significant improvement in the future.

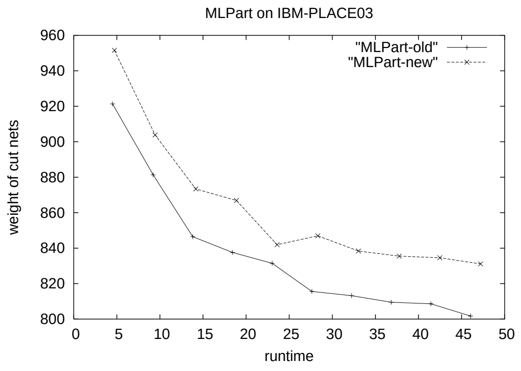


Figure 5. Pareto plots automatically generated from runs of two MLPart versions with the IBM-PLACE03 benchmark. The plots show that the new variant of MLPart produces worse results than the old variant.

As with any system for automation, the goals are to minimize the expense of costly user time, and to assume control of tasks where human intelligence can be substituted with machine intelligence. We measure the success of bX by savings in user interaction time and by an increase in user efficiency. For example, the process of generating the Pareto curves above was reduced to checking boxes to select Capo and the IBM-PLACE benchmarks, and telling bX how to extract a figure of merit from a job's output files. Automatic Pareto curve generation has also allowed some users of bX, who are tool developers, to see the impact of a change to a tool much sooner. For example, Figure 5 compares a development version and a stable version of MLPart. The Pareto plots generated by bX show that the new variant of MLPart produces worse solutions than the old variant, and should not be released as such.

Overhead. bX's job execution process, from user initiation to job completion, can be decomposed into these six steps: (i) user request, (ii) scheduling, (iii) job dispatch to host, (iv) job initialization, (v) job execution and (vi) job result collection. The bulk of the job execution overhead lies in job initialization and job result collection. At the initialization step, bX prepares a customized isolated environment (sandbox) for every job. This procedure involves copying a number of large files over the network, resolving file dependencies for dynamically linked executables, etc. In our experience, typical sandboxes are under 100MB and the initialization step takes under 5 seconds in bX. Maintaining synchronized copies on bX computational hosts may improve the overall response time of bX. Another bottleneck is associated with job completion and the transfer of the output files of jobs to the central server. These costs vary from job to job, dominated by network file transfer times and influenced by NFS performance [32]. They can be magnified by inefficiences of the scheduler and virtual dependencies when new jobs cannot be started before a current job completes.

6 Conclusions and Ongoing Work

We described a distributed-computing system, bX, designed to leverage high-performance computing resources for developing and evaluating high-quality EDA tools. Our implementation was used by half a dozen beta testers. Ongoing work proceeds along the following directions.

Database compatibility. We are working toward storing extracted results in a SQL database, which will facilitate user queries using the SQL language, e.g., as in [28], and more flexible use of statistical primitives. Another area of interest is the experimenting of databases such OpenAccess [23] across a tool flow, in the spirit of interoperability.

Scalability. We consider the expansion in scale from two areas — users and computational hosts. An increase in users will result in the growth of the repository of tools, benchmarks and jobs, increasing the load on the central server. An increase in the number of computational hosts will increase the difficulty in scheduling jobs and managing hosts. However, since computational hosts can either be single machines or clusters of machines commanded through a single machine, this suggests a hierarchical arrangement of resource nodes to prevent a scale explosion. In the future various network configurations and scheduling algorithms may be explored. However, the major bottleneck for scalability in bX is due to security considerations. As mentioned in Section 4, bX currently relies on having administrative control over the computational hosts on the distributed network, to support features like executing user-uploaded software. With more work, we hope to avoid this limitation.

Additional applications. Our work primarily targets EDA tool evaluation and seeks to automate typical operations. At the same time, bX includes a general-purpose distributed computing engine and is certainly applicable in a broader context.

Acknowledgements

The Bookshelf.EXE project has been supported by DARPA and MARCO through the Gigascale System Research Center (GSRC). We thank David Papa from the University of Michigan, other beta-testers of Bookshelf.EXE, and Marius Eriksen who is now at Google.

References

[1] S. N. Adya et al, "Benchmarking for Large-Scale VLSI Placement and Beyond", *IEEE Trans. on CAD*, April 2004, pp. 472-488.

[2] M. Baker, R. Buyya, and D. Laforenza, "Grids and Grid Technologies for Wide-Area Distributed Computing", *Software: Practice and Experience Journal*, Nov 2002.

[3] M. Bauer, P. Penkala, A. Pawlak, D. Stachanczyk, P. Fras, "Collaborative Environment for Testbench Development", *Euromicro Digital System Design Symposium* 2002.

[4] F. Brglez and H. Lavana, "A Universal Client for Distributed Networked Design and Computing", *Proc. IEEE/ACM DAC*, 2001, pp. 401-406.

[5] J. Brockman, T. Cobourn, M. Jacome and S. Director, "The Odyssey CAD Framework", *IEEE DATC Newsletter on Design Automation*, 1992, pp. 7-11.

[6] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Toward CAD-IP Reuse: The MARCO GSRC Bookshelf of Fundamental CAD Algorithms", *Proc. IEEE Design and Test*, May 2002, pp. 72-81 *(http://vlsicad.eecs.umich.edu/BK)*

[7] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Can Recursive Bisection Alone Produce Routable Placements?", *Proc. ACM/IEEE DAC*, June 2000, pp. 477-482 *(http://vlsicad.eecs.umich.edu/BK/PDtools)*

[8] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Improved Algorithms for Hypergraph Bipartitioning", *Proc. IEEE/ACM Asia and South Pacific DAC*, Jan. 2000, pp. 661-666 *(http://vlsicad.eecs.umich.edu/BK/PDtools)*

[9] V. Cardellini, E. Casalicchio, M. Colajanni and P. S. Yu, 'The State of the Art in Locally Distributed Web-Server Systems", *ACM Computing Surveys*, 2002, pp. 263-311.

[10] F. L. Chan, M. D. Spiller and A. R. Newton, "WELD — An Environment for Web-Based Electronic Design", *Proc. IEEE/ACM DAC*, 1998, pp. 146-151.

[11] M. Dalpasso, A. Bogliolo and L. Benini "Virtual Simulation of Distributed IP-based Designs", *Proc. 36th ACM/IEEE DAC*, 1999, pp. 50-55.

[12] S. Fenstermaker, D. George, A. B. Kahng, S. Mantik and B. Thielges, "METRICS: A System Architecture for Design Process Optimization", *Proc. IEEE/ACM DAC*, 2000, pp. 705-710.

[13] *http://www.rtda.com/products/flowtracerEDA*

[14] P. van den Hamer, W. van der Linden, P. Bingley and N. Schellingerhout, "A System Simulation Framework", *Proc. IEEE/ACM DAC*, 2000, pp. 699-704.

[15] K. Hines and G. Borriello, "A Geographically Distributed Framework for Embedded System Design and Validation", *Proc. IEEE/ACM DAC*, 1998, pp. 140-145.

[16] U. Hoelzle, J. Dean and L. A. Barroso, "Web Search for a Planet: The Architecture of the Google Cluster", *IEEE Micro Magazine*, April 2003, pp. 22-28.

[17] E. Koch and M. Grossman, "Flow Management Tool Tracks Files and Jobs in IP Development", *Chip Design Magazine*, October 2003, pp. 24-27.

[18] H. Lavana, A. Khetawat, F. Brglez and K. Kozminski, "Executable Workflows: A Paradigm for Collaborative Design on the Internet", *Proc. IEEE/ACM DAC*, 1997, pp. 553-558.

[19] C. Li et al., "Routability-Driven Placement and White Space Allocation", to appear in *ICCAD 2004*.

[20] *http://www.rtda.com/download/paper2002.pdf*

[21] R. Merritt, "Intel, Clusters on the Rise in Top 500 Supercomputer List", *EE Times*, Nov. 18, 2003 *(http://www.eetonline.com/sys/news/OEG20031118S0011)*

[22] *http://www.ddtc.dimes.tudelft.nl/nelsis*

[23] *http://www.openeda.org*

[24] *http://www.openpbs.org*

[25] N. Provos, "Improving Host Security with System Call Policies", *12th USENIX Security Symp.*, Aug. 2003.

[26] A. Schneider et al, "Internet-Based Collaborative Test Generation with MOSCITO", *Proc. DATE*, 2002, pp. 221 - 226.

[27] B. Schrmann and J. Altmeyer, "Modeling Design Tasks and Tools — The Link between Product and Flow Model", *Proc. IEEE/ACM DAC*, 1997, pp. 564-569.

[28] L. Simon and P. Chatalic, "SatEx: A Web-based Framework for SAT Experimentation", *Proc. SAT 2001 (http://www.lri.fr/~simon/satex/satex.php3)*

[29] P. R. Sutton and S. W. Director, "Framework Encapsulations: A New Approach to CAD Tool Interoperability", *Proc. IEEE/ACM DAC*, 1998, pp. 134-139.

[30] *http://www.top500.org*

[31] *http://computing.vt.edu/research_computing/terascale*

[32] M. Wittle and B.E. Keith "LADDIS: The Next Generation in NFS File Server Benchmarking", *USENIX Summer*, 1993, pp. 111-128.