<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>Fault localization based only on failed runs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Author(s)</strong></td>
<td>Zhang, Z; Chan, WK; Tse, TH</td>
</tr>
<tr>
<td><strong>Citation</strong></td>
<td>Computer, 2012, v. 45 n. 6, p. 64-71</td>
</tr>
<tr>
<td><strong>Issued Date</strong></td>
<td>2012</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td><a href="http://hdl.handle.net/10722/146031">http://hdl.handle.net/10722/146031</a></td>
</tr>
<tr>
<td><strong>Rights</strong></td>
<td>Computer (New York). Copyright © IEEE, Computer Society.; ©2012 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.; This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.</td>
</tr>
</tbody>
</table>
Fault localization commonly relies on both passed and failed runs, but passed runs are generally susceptible to coincidental correctness and modern software automatically produces a huge number of bug reports on failed runs. FOnly is an effective new technique that relies only on failed runs to locate faults statistically.

Program testing and debugging generally consume 50 percent or more of the costs of typical software development projects. Software engineers spend about 35 percent of their time debugging programs, and deploy software knowing that it still contains faults.

When an execution of a faulty program passes through a fault, it may result in an error in the internal program states. The program run generally executes other program statements as well, which might propagate the error to other internal program states. If such program statements produce observable effects, the run will cause a visible failure.

Once they observe failures, software engineers schedule program debugging to locate the faults, fix them, and confirm their removal. However, debugging is still laborious, and fault localization is commonly considered its most difficult component.

Recent research in Asia has significantly advanced automatic fault localization. State-of-the-practice techniques commonly rely on a combination of passed and failed runs. However, a passed run may activate a fault but not reveal a failure. Furthermore, many systems today can detect failures automatically and produce a massive number of useful bug reports on failed runs. We propose FOnly, an effective technique that innovatively relies only on failed runs to locate faults statistically.

STATE OF THE PRACTICE

Researchers have developed many novel fault-localization techniques during the past two decades. To compare these techniques’ applicability, we categorized them according to whether they were originally designed to use passed runs and/or failed runs to locate faults, and, if they were, we identified the respective numbers of such runs. We limited the classification to approaches that locate faults in the source code. By so doing, we excluded techniques such as delta debugging and semi-proving that only reveal the suspicious region of a program’s failure-causing inputs, as well as techniques that suppress or detect failures, such as Eraser.

Table 1 summarizes the categorization. The two leftmost columns indicate the numbers of passed and failed runs. The third column lists an early representative project for each category. The last column shows a recent project for each category presented at a top-notch venue by researchers affiliated with Asian institutes.

**Fault Localization Based Only on Failed Runs**

Zhenyu Zhang, Institute of Software, Chinese Academy of Sciences

W.K. Chan, City University of Hong Kong

T.H. Tse, The University of Hong Kong

Coincidental correctness

The use of passed runs, irrespective of the number of instances, is generally susceptible to coincidental correct-
ness, in which a run activates a fault but does not result in a failure. Approaches such as set differencing, similarity correlation, and sequence similarity do not eliminate variations among sets of program statements that are common to passed and failed runs. Consequently, early algorithms like chislice,\(^6\) nearest neighbor,\(^7\) and Tarantula\(^8\) cannot reliably locate faults if the passed runs suffer from coincidental correctness. Recent Asian research has attempted to address this problem.

CP\(^9\) is a technique that locates faulty blocks in Unix utilities in C by computing the transition frequency among basic blocks in a run and backwardly propagating the fault-failure correlations measured by similarity coefficients along the edges of the program’s control-flow graph so that the code delivering a fault receives a higher rank.

Hong Cheng and colleagues\(^{10}\) localized faults in C utility programs by analyzing the most discriminative graph patterns in a bug report.

Diptikalyan Saha and coauthors\(^{11}\) identified faulty program slices in SAP systems in the ABAP language by taking program loops in a failed run that constructs database queries as starting points of individual slices, and splitting the single failed run into several such slices, some of which are associated with correct database records and deemed as “passed.” However, the passed runs thus generated might still be coincidentally correct.

Additional discussion about coincidental correctness in debugging can be found elsewhere.\(^{12}\)

Table 1. Categories of fault-localization techniques for program debugging.

<table>
<thead>
<tr>
<th>No. of passed runs</th>
<th>No. of failed runs</th>
<th>Early representative example</th>
<th>Recent example project in Asia</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>FindBugs (OOPSLA 2004)</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Approach: pattern matching with static analysis</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Application: locates bug patterns in Java lib/desktop/server programs</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Dynamic program slicing (PLDI 1990)</td>
<td>Saha et al. (India and USA—FSE 2011)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Approach: slicing</td>
<td>Approach: key-based slicing and semantic differencing among traces</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Application: locates faulty slices, without limit in program type</td>
<td>Application: locates faulty slices in SAP systems in the ABAP language</td>
</tr>
<tr>
<td>0</td>
<td>Many</td>
<td></td>
<td>FOnly (China and Hong Kong)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Approach: trend estimation</td>
<td>Approach: locates faulty statements in C utility programs</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Chislice (ISSRE 1995)</td>
<td>Cheng et al. (Hong Kong, Singapore, China, and US—ISSTA 2009)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Approach: set differencing</td>
<td>Approach: control-flow subgraphs as bug signatures</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Application: locates faulty slices in C algorithms</td>
<td>Application: locates faulty methods/blocks in C utility programs</td>
</tr>
<tr>
<td>Many</td>
<td>1</td>
<td>Nearest neighbor (ASE 2003)</td>
<td>CP (Hong Kong—FSE 2009)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Approach: sequence similarity</td>
<td>Approach: propagation of fault-failure correlation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Application: locates faulty statements in C utility programs</td>
<td>Application: locates faulty blocks in Unix utilities in C</td>
</tr>
<tr>
<td>Many</td>
<td>Many</td>
<td>Tarantula (ICSE 2002)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Approach: similarity correlation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Application: locates faulty statements in language interpreter programs in C</td>
<td></td>
</tr>
</tbody>
</table>

**Eliminating passed runs**

Another way to address the issue of coincidental correctness is to completely abandon the use of passed runs. Static analyzers like FindBugs\(^3\) require neither passed nor failed runs. They may, however, produce warnings even if the programs are correct, necessitating additional runs to confirm such warnings. When only one single failed run is available, early techniques like dynamic program slicing\(^{14}\) could produce a set of statements per run. Debuggers similar to Tarantula, such as Ochiai,\(^{15}\) can also be used without any passed runs. The problem is that such brute-force applications are mostly ineffective in fault localization.

To the best of our knowledge, there has been little research on effectively locating faults with one or multiple failed runs and without support from passed runs. In the past, using many failed runs without any passed run was deemed unrealistic because, when testing a program, a large proportion of runs were expected to reveal no failure. However, emerging dynamic (concurrency) bug detectors can monitor runs on the fly and report failures. Based on the bug reports, researchers can use execution synthesis to reproduce failed runs.

**DEBUGGING BASED ON FAILURES ONLY**

Modern software often has the built-in facility to detect failures and report them to the original vendor through the Web. Software debuggers are thus faced with a huge number of automatic bug reports. It would be
A simple but effective means of trend estimation is to find a regression line using the least-squares fitting process. Such a line reveals the tendencies in the samples.

Based on samples of the numbers of times that different failed runs go through the same program entity, FOnly finds a regression line that minimizes the fitting error. It then uses the slope of the regression line and the value of the fitting error to compute a signal-to-noise ratio, which represents an estimate of that program entity’s fault relevance. Program entities with higher signal-to-noise ratios are deemed to be more fault-relevant.

**APPLYING THE FAULT HYPOTHESIS**

Let us use an example to illustrate how FOnly applies the fault hypothesis to perform trend estimation to locate faults.

Figure 1a shows two code fragments from a faulty program known as replace from the Software-artifact Infrastructure Repository (SIR). The upper fragment (lines L115 to L124) shows an if-statement that evaluates a compound Boolean expression consisting of a character-checking function (L115) and a boundary condition (L117). If the Boolean expression is evaluated as true, the function addstr will modify the variable dest (L121); otherwise, dest will remain unchanged.

The lower fragment (L495 to L502) outputs the characters. In the upper fragment, the if-statement (L115) is faulty because an operand of the expected version of the Boolean expression is missing (shown in L116). Note that the faulty statement is more likely to be evaluated as true than the correct version, so an execution has a higher chance of going through L121 than the correct version. Consequently, the variable dest may contain an erroneous value. At the same time, the faulty statement (L115) may produce a result different from its expected version only when the missing function call to isalnum() returns false, which in turn depends partially on its input parameter src. Because it is generally difficult to predict string content statically, the probability of the faulty statement producing an incorrect decision value is hard to know.

We executed the faulty program over each of the 5,542 test cases supplied by SIR. For every statement, we recorded its execution count $c$ with respect to each
program run. For every value \( c_0 \) of an execution count, we tallied the number of program runs \( N(c) \) having a count \( c = c_0 \) and calculated the failure rate, which is the fraction of failed runs among the \( N(c) \) runs.

A comparison of L115 with L121, L497, and L500 indicates that L121 is closest to the faulty statement in terms of the number of lines of code (LOC). L121 is indeed suspicious because it may modify the variable \( dest \) wrongly. At the same time, we inspected the source code to ensure that the fault is not related to the logic in L497 or L500.

We plotted the failure rate against the execution count for each of these four statements and fitted the points using a regression line with the least-squares error among the data points. Figure 1b shows the results. Note that as the execution count increases, the failure rates for L115 and L121 increase faster than those for L497 and L500. Fault-irrelevant statements (such as L497 and L500) thus appear to have gentler slopes. For L500, the statement least related to the fault, the change in failure rate with respect to execution count is least observable.

However, this simple regression line estimation has not considered line-fitting errors. It also requires information on both passed and failed runs. FOnly addresses both of these issues, as detailed in the next section.

**HOW FONLY WORKS**

Consider a program modeled by a list of program entities such as statements. Given a collection of execution results of the program, known as the set of program runs, FOnly conducts fault localization by comparing the suspiciousness of each program entity, measured by its failure trend in the set of program runs. Figure 2 illustrates the process, which consists of four phases: partitioning, calibration, line fitting, and elimination.

**Partitioning phase**

Given any program entity, such as a statement, FOnly divides the set of program runs into disjoint partitions. Every partition contains all the runs such that each run goes through the entity exactly the same number of times (say, \( c \) times). FOnly further computes the proportion of failed runs in each partition, referred to as the failure rate \( F(c) \).

**Calibration phase**

For a given program entity \( s \), \( F(0) \) denotes the failure rate of the partition in which none of the program runs ever go through \( s \). If all the runs in the partition have never gone through \( s \), the latter should not be related to any failure. Hence, if \( F(0) \) for \( s \) happens to not be 0, it should be reset to 0. Likewise, all other partitions for the same \( s \) might have overestimated failure rates. As such, FOnly computes a calibrated failure rate \( G(c) = F(c) - F(0) \), which aims to capture a more accurate estimate of the probability that going through the program entity exactly \( c \) times leads to a failure.

**Line-fitting phase**

Based on the calibrated failure rates, FOnly estimates the failure trend for each program entity in the fitting phase. For any given entity, by pairing up every \( c \) with
the corresponding \(G(c)\) when the latter is defined. FOnly creates a point \((c,G(c))\) in a 2D space. For a fault-relevant program entity, \(G(c)\) should be a discrete monotonically increasing function of \(c\). FOnly estimates the fault relevance of any program entity using line fitting in 2D space.

There are two ways to estimate the probability that going through \(s\) exactly \(c\) times does not result in a failure. The first is to directly use \(G(c)\) and estimate the probability as \(1 – G(c)\). The second way is to use the probability that executing the program entity up to \(c\) times does not lead to any failure. This probability can be estimated to be \((1 – p)^c\), where \(p\) denotes the probability that going through \(s\) only once leads to a failure. Equating the two probabilities results in the formula

\[
G(c) = (1 – (1 – p)^c)
\]

A function \(f(x)\) can be expanded into an infinite Taylor series \(f(x) = f(0) + f'(0)x/1! + …\), where \(f'(0)\) denotes the \(i\)-th derivative of \(f(x)\) at the point \(x = 0\). Hence, the calibrated failure rate can be approximated by \(G(c) = G(0) + G'(0)c/1! = – \log(1 – p) \times c\), which simplifies to \(G(c) = l \times c\). Thus, the calibrated failure rate is modeled by a straight line passing through \((0,G(0))\) with a slope \(l\).

FOnly applies least-squares analysis to minimize the error in line fitting. For a given program entity, the slope \(l\) is given by

\[
l = \sum_{c \in D}[c \times G(c)]/\sum_{c \in D}c^2
\]

and the standard deviation \(\sigma\) is given by

\[
\sigma = \sqrt{\sum_{c \in D}(G(c))^2 – (\sum_{c \in D}[c \times G(c)])^2 / \sum_{c \in D}c^2}
\]

where \(D\) is the set of the possible number of times that any program run might go through the given program entity. However, the number of times that a program run goes through a program entity may vary among entities. Hence, \(l\) for each specific program entity should be normalized to \(\hat{l} = l \times c_{\text{max}}\) before comparison, where \(c_{\text{max}}\) is the largest possible number of times that any program run can go through that particular entity.

To estimate the fault relevance of each program entity in the presence of both passed and failed runs, FOnly computes the ranking score \(R\), which is equivalent to the signal-to-noise ratio and defined as the mean over the standard deviation: \(R = l/\sigma\). The higher the value of \(R\), the more fault-relevant the program entity.

The ranking score’s range is \([-\infty, +\infty]\). If a program entity \(s\) has no sample point, none of the failed runs has gone through \(s\) before resulting in a failure, and hence FOnly assigns a value of \(-\infty\) to \(R\), meaning that \(s\) is least suspicious. If a program entity has only one sample point, the slope \(l\) is undefined, and FOnly assigns a value of \(0\) to \(R\). If the standard deviation is \(0\), FOnly cannot directly compute the ranking score. In this case, if the number of failed runs for \(s\) is \(0\), FOnly assigns a value of \(-\infty\) to \(R\); otherwise, it calculates the limit of \(R\), resulting in a value of \(+\infty\).

**Elimination phase**

Unlike related approaches, FOnly adds a phase to eliminate dependency on the number of passed runs incurred in computing a program entity’s fault relevance. For this phase, it uses a formula that relies only on failed runs.

### Table 2. Faulty programs used in FOnly evaluation.

<table>
<thead>
<tr>
<th>Program (source)</th>
<th>Real-life version no.</th>
<th>No. of faults</th>
<th>Executable LOC</th>
<th>No. of single-fault/multifault versions</th>
<th>No. of test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens (Siemens)</td>
<td>Not available in SIR</td>
<td>7</td>
<td>341-342</td>
<td>5/0</td>
<td>4,130</td>
</tr>
<tr>
<td>print_tokens2 (Siemens)</td>
<td>Not available in SIR</td>
<td>10</td>
<td>350-354</td>
<td>10/0</td>
<td>4,115</td>
</tr>
<tr>
<td>replace (Siemens)</td>
<td>Not available in SIR</td>
<td>32</td>
<td>508-515</td>
<td>30/0</td>
<td>5,542</td>
</tr>
<tr>
<td>schedule (Siemens)</td>
<td>Not available in SIR</td>
<td>9</td>
<td>291-294</td>
<td>6/0</td>
<td>2,650</td>
</tr>
<tr>
<td>schedule2 (Siemens)</td>
<td>Not available in SIR</td>
<td>10</td>
<td>261-263</td>
<td>8/0</td>
<td>2,710</td>
</tr>
<tr>
<td>tcas (Siemens)</td>
<td>Not available in SIR</td>
<td>41</td>
<td>133-137</td>
<td>40/0</td>
<td>1,608</td>
</tr>
<tr>
<td>tot_info (Siemens)</td>
<td>Not available in SIR</td>
<td>23</td>
<td>272-274</td>
<td>23/0</td>
<td>1,052</td>
</tr>
<tr>
<td>flex (Unix)</td>
<td>2.4.7-2.5.4</td>
<td>81</td>
<td>8,571-10,124</td>
<td>18/4</td>
<td>567</td>
</tr>
<tr>
<td>grep (Unix)</td>
<td>2.2-2.4.2</td>
<td>57</td>
<td>8,053-9,089</td>
<td>17/6</td>
<td>809</td>
</tr>
<tr>
<td>gzip (Unix)</td>
<td>1.1.2-1.3</td>
<td>59</td>
<td>4,035-5,159</td>
<td>13/4</td>
<td>217</td>
</tr>
<tr>
<td>sed (Unix)</td>
<td>1.18-3.02</td>
<td>25</td>
<td>4,756-9,289</td>
<td>16/6</td>
<td>370</td>
</tr>
</tbody>
</table>
For any program entity \( s \), let \( N(c) \) denote the number of program runs such that each run goes through \( s \) exactly \( c \) times. FOnly computes the mean number of runs \( \bar{N} \) irrespective of the value of \( c \). It replaces every instance of \( N(c) \) by \( \bar{N} \) in the computation of \( R \) to obtain \( \bar{R} \), which is dependent on failed executions only and free from passed runs:

\[
\bar{R} = \frac{\sum_{c=0}^{\infty} \left[ \frac{c \times (Y(c) - Y(0))}{\sum_{c=0}^{\infty} (Y(c) - Y(0))^2} \right] \times c \times \sum_{c=0}^{\infty} c^2}{\sqrt{\sum_{c=0}^{\infty} (Y(c) - Y(0))^2}}
\]

where \( Y(c) \) is the number of failed runs such that each run goes through the program entity exactly \( c \) times.

Of course, the elimination phase is optional if the set of passed runs can be reliable.

**EXPERIMENTAL EVALUATION**

To validate FOnly’s effectiveness, we conducted an empirical study using faulty programs from SIR. Table 2 summarizes the programs’ statistics. Each of the program versions is seeded with one to three faults to simulate both single- and multifault scenarios, resulting in a total of 186 single-fault versions and 20 multifault versions.

We compared FOnly’s performance with that of four representative statement-level fault-localization techniques: Tarantula, Ochiai, Jaccard, and statistical bug isolation (SBI). Because these techniques were originally designed to work under the assumption that both passed and failed results were available, for every technique we first executed each program version using the whole test pool and then repeated the process using only failed runs. Following previous research, we measured fault-localization effectiveness in terms of the percentage of statements a technique examined (among all the statements ranked) until it found a faulty statement.

Figure 3 shows the overall results for the pass-and-fail and fail-only scenarios. In each plot, the x-axis represents the percentage of code examined, while the y-axis represents the percentage of faults located within the examined code.

As Figure 3a shows, when the examined code ranges from 10 to 100 percent, FOnly’s curve is above, or at least overlaps with, the curves of the peer techniques. For example, when examining up to 20 percent of the code, FOnly can locate faults in 86 percent of the faulty versions, while Tarantula, Ochiai, Jaccard, and SBI can only locate faults in 79, 77, 77, and 77 percent of the faulty versions, respectively. However, when examining the first 5 percent of the code, FOnly is not as effective as the other techniques and has room for improvement.

When no passed runs are available, the ranking formulas for Tarantula, Ochiai, and Jaccard produce the same list of statements. Figure 3b therefore shows one curve instead of three for these techniques. SBI is not included in the fail-only scenario because its formula gives all executed statements the same rank and does not have any fault-localization capability.

As the graph shows, FOnly’s curve is always above, or at least overlaps with, the curves for the other techniques. For example, when examining up to 10 percent of the code, FOnly can locate faults in 57 percent of the faulty versions while the peer techniques can locate faults in only 37 percent of the faulty versions. In the first half of the code-examining range, FOnly always locates more faults than the other techniques; after examining 50 percent of the code, it locates all the faults. Notably, FOnly’s effectiveness is comparable to that in the pass-and-fail scenario, whereas that of the other techniques is significantly lower.

Overall, the results in Figure 3 indicate that FOnly has promising fault-localization capability when statements are used as the diagnostic unit. When passed executions are unavailable or not reliable, its performance does not degrade as dramatically as the peer techniques. Although FOnly is relatively less effective when examining the
first 5 percent of the code in the traditional pass-and-fail scenario, this deficiency is due to the very few failed runs in corner cases, where FOnly has insufficient sample points to make reliable trend estimations.

A straightforward way to enhance efficiency is to apply FOnly at the block level. Because statements in the same block are mostly assigned identical ranking scores, precision will not be lost. Higher efficiency could be gained by a coarser-grained usage of FOnly at, say, the function level, but this will result in less precise fault localization.

To assess the impact of program size and the number of faults on FOnly’s effectiveness, we categorized the subject programs according to their size and whether a faulty version contained single or multiple faults. As Figure 4 shows, in both the pass-and-fail scenario and the fail-only scenarios, FOnly performed better on the medium-scale Unix programs with thousands of LOC than on the small-scale Siemens programs with hundreds of LOC. It located faults in multifault programs almost as effectively as it did in single-fault programs.

The results of single-fault Unix programs with multifault Unix programs, we found that FOnly can locate faults in multifault programs almost as effectively as in single-fault programs.

In recent years, researchers in Asia have contributed significantly to advances in fault localization for program debugging. In addition to tackling existing challenges, they continue to introduce techniques for new classes of real-life problems.

FOnly is a proposed solution to the increasingly common situation developers face when a huge number of bug reports are sent automatically through the Web. It uses trend estimation as a novel method to localize faults and demonstrates the feasibility of using only the results of failed runs, rather than comparing passed and failed runs. Empirical results demonstrate FOnly’s promise.

For corner cases where there are very few failed runs, however, peer techniques that use passed and failed runs are more effective when it is possible to review only a small percentage of the source code. In future work, we plan to apply combinatorial testing to improve FOnly in such situations.

Acknowledgments

This research is supported in part by grants from the Natural Science Foundation of China (project no. 61003027) and the General Research Fund of the Research Grants Council of Hong Kong (project nos. 111410 and 717811).

References

9. Z. Zhang et al., “Capturing Propagation of Infected Program States,” Proc. 7th Joint Meeting European Software


Zhenyu Zhang is an associate research professor at the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China. His research interests include software testing and debugging, with a focus on fault localization. Zhang received a PhD in computer science from The University of Hong Kong. He is a member of ACM, IEEE, and the China Computer Federation. Contact him at zhangzy@ios.ac.cn.

W.K. Chan is an assistant professor in the Department of Computer Science, City University of Hong Kong. His research interests include software testing and analysis of concurrent systems and software. Chan received a PhD in computer science from The University of Hong Kong. Contact him at wkchan@cityu.edu.hk.

T.H. Tse is a professor in computer science and director of The Software Engineering Group at The University of Hong Kong. His research interests include program testing, debugging, and analysis. Tse received a PhD from the London School of Economics. He is a fellow of the British Computer Society, the Institute for the Management of Information Systems, the Institute of Mathematics and its Applications, and the Hong Kong Institution of Engineers. Contact him at thtse@cs.hku.hk.

Selected CS articles and columns are available for free at http://ComputingNow.computer.org.

IEEE CVPR 2012
25th IEEE Computer Society Conference on Computer Vision and Pattern Recognition

16-21 June 2012
Rhode Island Convention Center, Providence, Rhode Island, USA

CVPR is the premiere annual Computer Vision event comprising the main CVPR conference and 36 co-located workshops and short courses. With its high quality and low cost, it provides an exceptional value for students, academics and industry.

Register today!

http://www.cvpr2012.org/