How to Speed Up Software Migration and Modernization: Successful Strategies Developed by Precisiating Expert Knowledge

Francisco Zapata\textsuperscript{1}, Octavio Lerma\textsuperscript{2}, Leobardo Valera\textsuperscript{2}, and Vladik Kreinovich\textsuperscript{1,2}

\textsuperscript{1}Department of Computer Science
\textsuperscript{2}Computational Science Program
University of Texas at El Paso
El Paso, TX 79968, USA
fazg74@gmail.com, lolerma@episd.org
leobardovalera@gmail.com, vladik@utep.edu
1. Computers Are Ubiquitous

- In many aspects of our daily life, we rely on computer systems:
  - computer systems record and maintain the student grades,
  - computer systems handle our salaries,
  - computer systems record and maintain our medical records,
  - computer systems take care of records about the city streets,
  - computer systems regulate where the planes fly, etc.

- Most of these systems have been successfully used for years and decades.

- Every user wants to have a computer system that, once implemented, can effectively run for a long time.
2. Need for Software Migration/Modernization

- Computer systems operate in a certain environment; they are designed:
  - for a certain computer hardware – e.g., with support for words of certain length,
  - for a certain operating system, programming language, interface, etc.
- Eventually, the computer hardware is replaced by a new one.
- While all the efforts are made to make the new hardware compatible with the old code, there are limits.
- As a result, after some time, not all the features of the old system are supported.
- In such situations, it is necessary to adjust the legacy software so that it will work on a new system.
3. Software Migration and Modernization Is Difficult

- At first glance, software migration and modernization sounds like a reasonably simple task:
  - the main intellectual challenge of software design is usually when we have to invent new techniques;
  - in software migration and modernization, these techniques have already been invented.
- Migration would be easy if every single operation from the legacy code was clearly explained and justified.
- The actual software is far from this ideal.
- In search for efficiency, many “tricks” are added by programmers that take into account specific hardware.
- When the hardware changes, these tricks can slow the system down instead of making it run more efficiently.
4. How Migration Is Usually Done

- When a user runs a legacy code on a new system, the compiler produces thousands of error messages.
- Usually, a software developer looks corrects these errors one by one.
- This is a very slow and very expensive process:
  - correcting each error can take hours, and
  - the resulting salary expenses can run to millions of dollars.
- There exist tools that try to automate this process by speeding up the correction of each individual error.
- These tools speed up the required time by a factor of even ten.
- However, still thousands of errors have to be handled individually.
5. Resulting Problem: Need to Speed up Migration and Modernization

- Migration and modernization of legacy software is a ubiquitous problem.

- It is thus desirable to come up with ways to speed up this process.

- In this paper:
  - we propose such an idea, and
  - we show how expert knowledge can help in implementing this idea.
6. Our Main Idea

- Modern compilers do not simply indicate an error,
- They usually provide a reasonably understandable description of the type of an error; for example:
  - it may be that a program is dividing by zero,
  - it may be that an array index is out of bound.
- Some of these types of error appear in numerous places in the software.
- Our experience shows that in many such places, these errors are caused by the same problem in the code.
- So, instead of trying to “rack our brains” over each individual error, a better idea is
  - to look at all the errors of the given type, and
  - come up with a solution that would automatically eliminate the vast majority of these errors.
7. Need for Expert Knowledge

- This idea saves time only if we have enough errors of a given type.
- We thus need to predict how many errors of different type we will encounter.
- There are currently no well-justified software models that can predict these numbers.
- What we do have is many system developers who have an experience in migrating and modernizing software.
- It is therefore desirable to utilize their experience.
- Experts usually describe their experience by using imprecise (“fuzzy”) words from natural language.
- It is reasonable to use the known precisiation techniques – fuzzy logic.
8. Expert Knowledge about Software Migration and Modernization and Its Precisiation

• A reasonable idea is to start with \( n_1 \) errors of the most frequent type.

• Then, we should concentrate on \( n_2 \) errors of the second most frequent type, etc.

• So, we want to know the numbers \( n_1, n_2, \ldots \), for which

\[
 n_1 \geq n_2 \geq \ldots \geq n_{k-1} \geq n_k \geq n_{k+1} \geq \ldots
\]

• We know that for every \( k \), \( n_{k+1} \) is somewhat smaller than \( n_k \).

• Similarly, \( n_{k+2} \) is more noticeably smaller than \( n_k \), etc.

• After formalizing and defuzzifying the \( n_k < n_{k+1} \) rule, we get \( n_{k+1} = f(n_k) \).

• Which function \( f(n) \) should we choose?
9. Which Function $f(n)$ Should We Choose?

- A migrated software package usually consists of two (or more) parts.
- We can estimate $n_{k+1}$ in two different ways:
  - We can use $n_k = n_k^{(1)} + n_k^{(2)}$ to predict 
    $$n_{k+1} \approx f(n_k) = f(n_k^{(1)} + n_k^{(2)}).$$
  - Oe, we can use $n_k^{(1)}$ to predict $n_k^{(1)}$, $n_k^{(2)}$ to predict $n_k^{(2)}$, and add them: $n_{k+1} \approx f(n_k^{(1)}) + f(n_k^{(2)})$.
- It is reasonable to require that these estimates coincide:
  $$f(n_k^{(1)} + n_k^{(2)}) = f(n_k^{(1)}) + f(n_k^{(2)}).$$
- So, $f(a + b) = f(a) + f(b)$ for all $a$ and $b$, thus $f(a) = f(1) + \ldots + f(1)$ (a times), and $f(a) = f(1) \cdot a$.
- Thus, $n_{k+1} = c \cdot n_k$, i.e., $n_{k+1}/n_k = \text{const.}$
10. Empirical Data: Values $n_{k}$ for Migrating a Health-Related C Package from 32 to 64 Bits

Here, $n_{ab}$ is stored in the a-th column (marked ax) and b-th row (marked xb).

<table>
<thead>
<tr>
<th></th>
<th>0x</th>
<th>1x</th>
<th>2x</th>
<th>3x</th>
<th>4x</th>
<th>5x</th>
<th>6x</th>
<th>7x</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>–</td>
<td>308</td>
<td>95</td>
<td>47</td>
<td>13</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x1</td>
<td>7682</td>
<td>301</td>
<td>91</td>
<td>38</td>
<td>13</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x2</td>
<td>4757</td>
<td>266</td>
<td>85</td>
<td>34</td>
<td>12</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x3</td>
<td>3574</td>
<td>261</td>
<td>81</td>
<td>34</td>
<td>12</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x4</td>
<td>2473</td>
<td>241</td>
<td>76</td>
<td>30</td>
<td>11</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x5</td>
<td>2157</td>
<td>240</td>
<td>69</td>
<td>24</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x6</td>
<td>956</td>
<td>236</td>
<td>58</td>
<td>21</td>
<td>8</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x7</td>
<td>769</td>
<td>171</td>
<td>57</td>
<td>19</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x8</td>
<td>565</td>
<td>156</td>
<td>50</td>
<td>17</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x9</td>
<td>436</td>
<td>98</td>
<td>47</td>
<td>17</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>–</td>
</tr>
</tbody>
</table>
11. **Empirical Data: Values** $n_{k}$ **for Migrating a Health-Related C Package from 32 to 64 Bits**

Here, $n_{ab}$ is stored in the a-th column (marked ax) and b-th row (marked xb); e.g., $n_{23} = 81$.

<table>
<thead>
<tr>
<th></th>
<th>0x</th>
<th>1x</th>
<th>2x</th>
<th>3x</th>
<th>4x</th>
<th>5x</th>
<th>6x</th>
<th>7x</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>–</td>
<td>308</td>
<td><strong>95</strong></td>
<td>47</td>
<td>13</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x1</td>
<td>7682</td>
<td>301</td>
<td><strong>91</strong></td>
<td>38</td>
<td>13</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x2</td>
<td>4757</td>
<td>266</td>
<td><strong>85</strong></td>
<td>34</td>
<td>12</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x3</td>
<td>3574</td>
<td>261</td>
<td><strong>81</strong></td>
<td>34</td>
<td>12</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x4</td>
<td>2473</td>
<td>241</td>
<td><strong>76</strong></td>
<td>30</td>
<td>11</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x5</td>
<td>2157</td>
<td>240</td>
<td><strong>69</strong></td>
<td>24</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x6</td>
<td>956</td>
<td>236</td>
<td><strong>58</strong></td>
<td>21</td>
<td>8</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x7</td>
<td>769</td>
<td>171</td>
<td><strong>57</strong></td>
<td>19</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x8</td>
<td>565</td>
<td>156</td>
<td><strong>50</strong></td>
<td>17</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x9</td>
<td>436</td>
<td>98</td>
<td><strong>47</strong></td>
<td>17</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>–</td>
</tr>
</tbody>
</table>
12. How Accurate is This Estimate?

- One can easily see that for $k \leq 9$, we indeed have $n_{k+1} \approx c \cdot n_k$, with $c \approx 0.65-0.75$.

- Thus, the above simple rule described the most frequent errors reasonably accurately.

- However, starting with $k = 10$, the ratio $n_{k+1}/n_k$ becomes much closer to 1.

- Thus, the one-rule estimate is no longer a good estimate.

- A natural idea is this to use two rules:
  - in addition to the rule that $n_{k+1}$ is somewhat smaller than $n_k$,
  - let us also use the rule that $n_{k+2}$ is more noticeably smaller than $n_k$. 
13. Two-Rules Approach

- Once we know \( n_k \) and \( n_{k+1} \), we can use fuzzy methodology and get an estimate \( n_{k+2} = f(n_k, n_{k+1}) \).

- When the software package consists of two parts, we can estimate \( n_{k+2} \) in two different ways:
  
  - We can use the overall numbers \( n_k = n_k^{(1)} + n_k^{(2)} \) and \( n_{k+1} = n_{k+1}^{(1)} + n_{k+1}^{(2)} \) and predict
    \[
    n_{k+2} \approx f(n_k, n_{k+1}) = f(n_k^{(1)} + n_k^{(2)}, n_{k+1}^{(1)} + n_{k+1}^{(2)}).
    \]
  
  - Alternatively, we can predict the values \( n_{k+2}^{(1)} \) and \( n_{k+2}^{(2)} \), and add up these predictions:
    \[
    n_{k+2} \approx f(n_k^{(1)}, n_{k+1}^{(1)}) + f(n_k^{(2)}, n_{k+1}^{(2)}).
    \]

- It is reasonable to require that these two approaches lead to the same estimate, i.e., that we have
  \[
  f(n_k^{(1)} + n_k^{(2)}, n_{k+1}^{(1)} + n_{k+1}^{(2)}) = f(n_k^{(1)}, n_{k+1}^{(1)}) + f(n_k^{(2)}, n_{k+1}^{(2)}).
  \]
14. Two-Rules Approach (cont-d)

• Reminder: for all \( a \geq a' \) and \( b \geq b' \), we have
  \[
  f(a + b, a' + b') = f(a, a') + f(b, b').
  \]

• One can show that this leads to \( n_{k+2} = c \cdot n_k + c' \cdot n_{k+1} \)
  for some \( c \) and \( c' \), and thus, to
  \[
  n_k = A_1 \cdot \exp(-b_1 \cdot k) + A_2 \cdot \exp(b_2 \cdot k).
  \]

• In general, \( b_i \) are complex numbers – leading to oscillating sinusoidal terms.

• We want \( n_k \geq n_{k+1} \), so there are no oscillations, both \( b_i \) are real.

• Without losing generality, we can assume that \( b_1 < b_2 \).

• If \( A_1 > A_2 \), then the first term always dominates.

• But we already know that an exponential function is not a good description of \( n_k \).
15. Two-Rules Model Fits the Data

- Thus, to fit the empirical data, we must use models with \( A_1 < A_2 \). In this case:
  - for small \( k \), the second – faster-decreasing – term dominates: \( n_k \approx A_2 \cdot \exp(-b_2 \cdot k) \);
  - for larger \( k \), the first – slower-decreasing – term dominates: \( n_k \approx A_1 \cdot \exp(-b_1 \cdot k) \).

- This double-exponential model indeed describes the above data reasonably accurately:
  - for \( k \leq 9 \), the data is a good fit with an exponential model for which \( \rho = n_{k+1}/n_k \approx 0.65-0.75 \);
  - for \( k \geq 10 \), the data is a good fit with another exponential model, for which \( \rho^{10} \approx 2-3 \).
16. Practical Consequences

• For small \( k \), the dependence \( n_k \) rapidly decreases with \( k \).

• So, the values \( n_k \) corresponding to small \( k \) constitute the vast majority of all the errors.

• In the above example, 85 percent of errors are of the first 10 types; thus:
  – once we learn to repair errors of these types,
  – the remaining number of un-corrected errors decreases by a factor of seven.

• This observation has indeed led to a significant speed-up of software migration and modernization.
17. Conclusion

• In many practical situations, we need to migrate legacy software to a new hardware and system environment.

• If we run the software package in the new environment, we get thousands of difficult-to-correct errors.

• As a result, software migration is very time-consuming.

• A reasonable way to speed up this process is to take into account that:
  – errors can be naturally classified into categories,
  – often all the errors of the same category can be corrected by a single correction.

• Coming up with such a joint correction is also somewhat time-consuming.

• The corresponding additional time pays off only if we have sufficiently many errors of this category.
18. Conclusion (cont-d)

- Coming up with a joint correction is time-consuming.
- This additional time pays off only if we have sufficiently many errors of this category.
- So, it is desirable to be able to estimate the number of errors $n_k$ of different categories $k$.
- We show that expert knowledge leads to a double-exponential model in good accordance w/observations.
19. Acknowledgment

This work was supported in part by the National Science Foundation grants:

- HRD-0734825 and HRD-1242122 (Cyber-ShARE Center of Excellence) and
- DUE-0926721.