Comparing Development Costs of C and Ada

March 30, 1995

Stephen F. Zeigler, Ph.D.

Rational Software Corporation

Programming Languages often incite zealotry because of theoretic advantages, whether from market acceptance or from intrinsic features. Practical comparisons of languages are more difficult. Some projects, notably prominently failing ones, cite choice of language and tools as a reason for their failure. Analysis is complex however: big projects aren't done twice in parallel just to see which language/tool choice is better, and even then there would be questions about the relative talent, teamwork, and fortunes of the efforts. This article discusses one case where most variables were controlled enough to make a comparison between development costs of C versus development costs of Ada.

Two companies, Verdix and Rational, merged in 1994 to form Rational Software Corporation. In the process of Due Diligence, records were examined to determine the value of the two companies but also to judge the best technologies and methodologies to continue into the future of the merged company. During that process, the extensive and complete update records of the Verdix Aloha development showed that there might be quantitative basis for comparing the success of two languages, C and Ada, under the fairest known conditions. A cursory pre-merger evaluation resulted in the following assertion:

"We have records of all changes ever made in the development of Verdix products; those records show us that for the lifecycle Ada seems to be about 2x or better for cost effectiveness than C..."

We have now completed a much more extensive evaluation. Our data applies more to development costs than to whole lifecycle costs. Even so, the above cost estimates are substantiated in comparing C and Ada development.

Conditions of the Study

Verdix existed eleven years from March 1983 till its merger in March 1994 as, primarily, a vendor of Ada-related development tools. It entered the Ada market convinced that Ada would form the best language basis for developing reliable software, and that reliability would become the most important concern of large software in the 1990s. The resulting VADS productline of compilers, builders, runtimes, and debug tools became recognized in the industry as one of the finest available. Verdix, and its merge partner Rational, now sell a wide range of development tool products for Ada, C, C++ and Ada95. The Rational software development environment, called Apex, has been merged with the VADS technology described here. The result, Apex2.0, is obsoleting a substantial portion of the C-based code in VADS.

This article is based on internal VADS data, sanitized to protect individual developers. This data was and is used during development to understand code and its history, and for accounting purposes with Rational's certified auditor, Ernst & Young, for the purpose of software capitalization.

The VADS product base was begun in March of 1983. Records of every change were kept from that time, but were not annotated with explanatory notes and categorizations until early in 1986. This study and the relatively low problem rates of the VADS line are in part due to these records. Prior to this 1993-4 study, these records were not examined for comparing C and Ada, nor was there ever an intent to make such a comparison beyond idle speculation.

VADS was begun in the C language. Ada was not used until 1986 since no good Ada compilers were available. As time went on, Ada was used increasingly with the general rule similar to that of the DoD "Ada Mandate": use Ada if more than 30% of a project will be new code. By mid-1991, the amount of Ada and the amount of C in VADS had reached approximate parity.

VADS is built in a common source baseline, meaning that all VADS products are composed from the same source. VADS version 6.2 for the Sun SPARC is about the same as VADS v6.2 for the DEC Alpha, except for architecture-and operating system-specific optimizations.

The development team for VADS has preserved a relatively free-form style that encourages engineers to cooperate and move among the different functional areas of the line. Thus a person might add to an Ada-based tool when they first join, then learn how to build and test an entire tool set, and then move towards an area of specific interest such as “trace tools”; their work in trace tools might have them adding code to the Ada-based runtime system and linker, to the C-based code generators, and to the Ada-based test frameworks. Of the 62 contributors measured in this study, only one did no Ada updates and only one did no C updates; in each case these employees had been with the team for less than six months. Of the 62 contributors, only six are no longer working on the code as of mid-1994. Hiring has been mostly steady with growth of about 5 people per year.

Of the team members, most have Master’s Degrees from good Computer Science schools. Most were considered excellent students. The more experienced contributors tend to work on the C parts of VADS because the C parts were begun first and because of the dictum that developers continue responsibility for any code they write.

The VADS tools, supporting both C and Ada, are used for their own development. The host platform C compiler and linker was used for C builds, but the developers would not normally be aware of this since these foreign tools were hidden within the common build apparatus (vmake), the common source code control system (dsc) and the common debugger (a.db). Contributors therefore see about the same debug/test/edit capability in each C/Ada section, and indeed may often be debugging both C and Ada at the same time. The same design methods were used regardless of language. The test apparatus also applied equally to each of C and Ada.

In summary, the C and Ada areas are worked by about the same people (with a slight advantage to C) and using the same tools under approximately the same conditions.

Data for Overall Development of the VADS Product Line

(Up to Oct. 15, 1994)

	C_FILE	ADA_FILE	SCRIPT_FILE	OTHER_FILE	TOTALS
all_lines:	1925523	1883751	117964	604078	4531316
SLOC:	1508695	1272771	117964	604078	3503508
files:	6057	9385	2815	3653	21910
updates:	47775	34516	12963	12189	107443
new_features:	26483	23031	5594	6145	61253
Fixes:	13890	5841	4603	1058	25392
Fixes/feature:	.52	.25	.82	.17	.41
Fixes/KSLOC:	9.21	4.59	39.02	1.75	7.25
devel_cost:	$15,873,508	$8,446,812	$1,814,610	$2,254,982	$28,389,856
cost/SLOC:	$10.52	$6.62	$15.38	$3.72	$8.10
defects:	1020	122	(A)	(B)	1242
defects/KSLOC:	.676	.096	(A)	(B)	.355

Definitions:

C_FILE:Files of C-based source.
ADA_FILE: Files of Ada-based source.
SCRIPT_FILE: Files of scripts for Make, VMS-DCL, and internal tools such as “vmake” used for virtual make. The Make files are used primarily for C since Ada tools have automated build manager tools, but since release tools and VMS require work for Ada as well, these results are not just lumped in with C.

OTHER_FILE: Files used for documentation, or otherwise indeterminate purpose - all classifications were done automatically.

all_lines: Results from the Unix “wc” command. These include comments and blank lines for both C and Ada.

SLOC: Non-blank, non-comment lines of code. This is sometimes called SLOC, for Source Lines Of Code. Comments and blank lines were not measured for scripts and “other” files.

files: Unix files. C and Ada are distinguishable by their unique suffices. Script files are sometimes indistinguishable and are therefore under-reported with the balance in “OTHER_FILE”.

updates: The source code control system tracks every change updated into each baseline (in this case, the main development “dev” baseline.)This row lists all updates of any kind, into the dev baseline.

new features: The subset of updates that added new features to the “dev” baseline. Features are similar to “function points” as defined in [].

Fixes: The subset of updates that fixed bugs in the “dev” baseline. More than 90% of these Fixes were found during unit test, before beta product release, and are called “internal fixes.” Defects (customer-discovered bugs) are discussed later.

Fixes/feature: This row gives the ratio of the “new features” and “fixes” rows. It means that overall, we could expect to find .52 Fixes in each feature added in C, but only .25 in each feature added in Ada. Features took about 80 lines of additional code, on average.

Fixes/KSLOC: The measure of internal Fixes per 1000 SLOC over all time.

devel_cost: This is the approximate burdened costs for the people spending time on these various projects. This figure is approximated from the base salaries for the people involved. Since the base salary figures do not take into account salary changes during the development period, nor inflation, nor exact assignments, nor time spent on other (e.g. sales and mentoring) activities, nor many other burden costs, we calculate the burdened devel_cost by multiplying base salary information by the company’s maximum burden factor of 2.0. It is meant only as a rough guideline giving an approximation of the bias of different salaried people. Dollars are adjusted for inflation as about 1992 valued.

cost/SLOC: This gives the approximate burdened cost of each SLOC. This value is based on the approximate devel_cost above.

defects: This row is condensed from customer support records. There were 16,440 customer interactions (called “tickets”) which produced about 5072 possible defects (called “CR”s), which eventually produced 2004 deficiency reports (called “DR”s), which eventually resulted in about 1242 actual bugs (called “defects”.)

defects/KSLOC: The classic measure of visible defects (customer-reported bugs) per 1000 SLOC.

NOTES:

1) This summary does not include obsoleted directories, such as for end-of-life’d products (e.g., the VADS for the Mil-STD 1750a 16 bit military computer) or for replaced components (e.g., the original C-based optimizer and runtime.)
2) This summary does not include source from the infrastructure support baselines (e.g., for our internal source code control and documentation) nor from our very large test baselines (e.g., ACVC tests or regression test suites) nor from our user documentation baselines. Also, it includes only the “dev” baseline, rather than released version baselines; these baselines occasionally have separate work done for them, but normally receive a subset of the fixes and a small subset of the features added to the dev baseline.

3) As explained below, fixes for makefiles and others could not be automatically distinguished. (A)+(B) together is 100.

This data is collected automatically by the source code control system in the same way regardless of language. The source code system does not take effect until developers make an update. That is, only updated code is tracked. It is expected that unit tests and a base automated test suite would pass before updates. In practice, some developers update readily and show more fixes, and some update infrequently with (normally) fewer fixes. The decision to update is influenced by how many other people might need or benefit from the results, how many others might be developing and therefore making updates in the same files, and a variety of other factors. Updates cannot be completely tested because the complexity of the product and its many switches, variants, hosts, targets, add-ons and usages make complete testing impossible

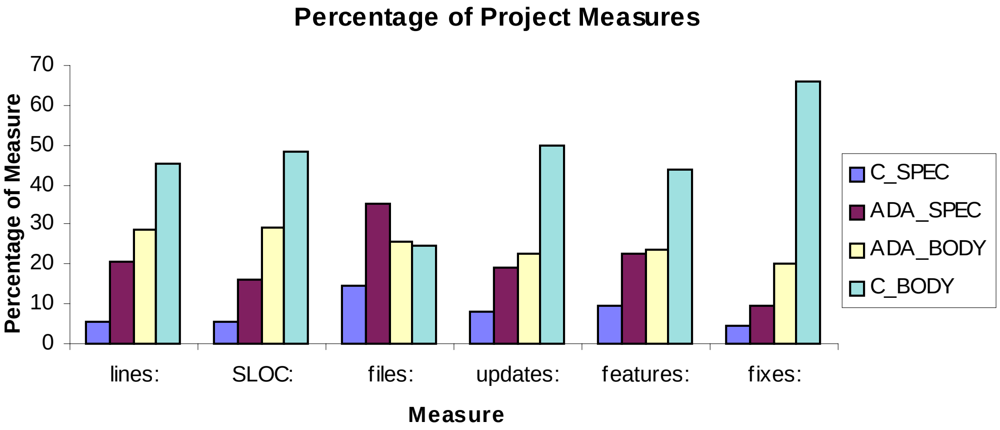
The process of updating requires several inputs from the developer, including a categorization of the general reason for the update Developer update records are code-reviewed by peers. Even so, the "fixed bug" categories may be underreported because some developers do several work items at once and update several fixes in amongst other feature addition-type changes; most of these updates seem to be recorded as "new feature" probably because of the minor stigma of recording bugs; on the other hand, at least one developer marks everything as a bug fix unless it is very clearly new development.

C Line Equals Ada Line?

On the surface, Ada appears more cost-effective than does C. Ada lines cost about half as much as C lines, produce about 70% fewer internal fixes, and produce almost 90% fewer bugs for the final customer. But there are many variables that might explain these effects. We start with the question of whether a C line and an Ada line are comparable.

A first observation is that Ada has rigid requirements for making entities such as subprograms and variables visible globally. This leads to a separation of Ada code into specifications or "specs" and bodies. Could it be that there really wasn't that much Ada as far as functional lines, and that many are repeated specifications in the specs?

	C_SPEC	ADA_SPEC	C_BODY	ADA_BODY
lines:	205087	781921	1720436	1101830
SLOC:	158911	453782	1349784	818989
files:	2252	5443	3805	3942
updates:	6541	15886	41234	18630
new_features:	4844	11253	21639	11778
fixes:	890	1894	13000	3947

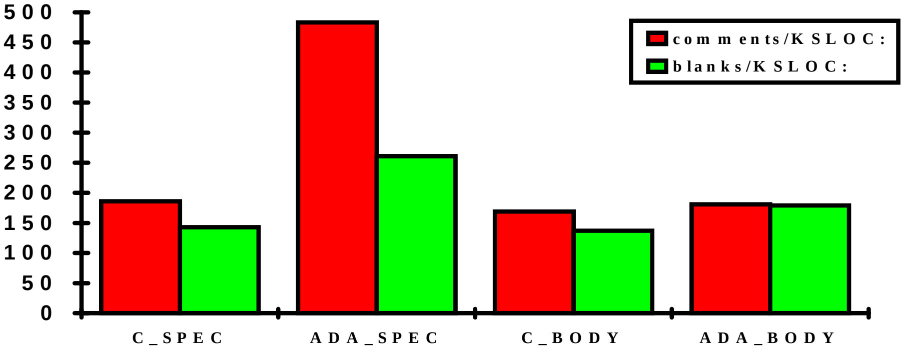


The above data supports the conclusion that Ada has more "specification" file lines than C. Are these "redundant" lines? ADA_SPEC files often provide the body, as in the case of packages with inlined definitions or containing library subprograms, so over half of the above ADA_SPEC lines are actually ADA_BODY lines. In addition, the types, variables, and in fact all non-subprogram definitions are not redundant since their single definition in the ADA_SPEC serves all users.

C bodies also contained significant redundant code. C allows entities such as variables and subprograms to be imported either by definition in a ".h" C_SPEC file, or by an explicit "extern" definition in the C_BODY. When new entities are added to C bodies, some developers choose to avoid changing their C_SPEC .h file because compilers may recompile many files. Thus C_BODY files collect some redundant lines for shared variables and constants. This is not recommended coding practice and is no longer allowed since smarter recompilation and faster machines rebuild quickly even with .h file changes. However, for the purposes of these statistics, we must consider that some C_BODY files have some inflation.

The relationship between comments, blank lines and SLOC reveals a consistent pattern:

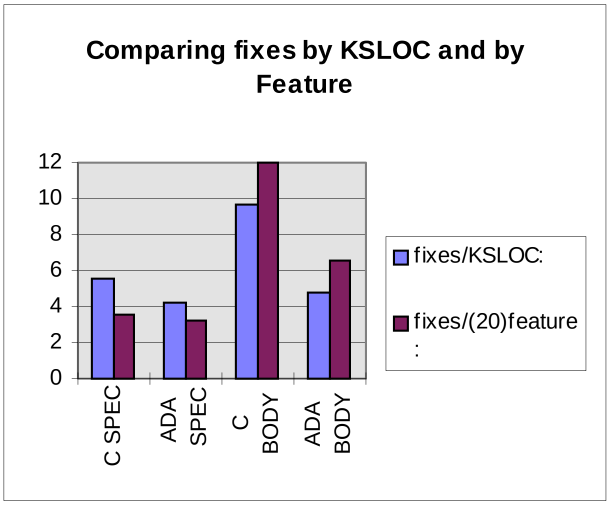
	C_SPEC	ADA_SPEC	C_BODY	ADA_BODY
comments/KSLOC	186	483	169	181
blanks/KSLOC:	143	261	137	179



We see that Ada specification files are consistently more commented and have more white space. This effect is not by requirement. It appears to result from the use of Ada specification files for "understanding" the programs; developers seem to add comments to specification files because since the subprogram prototypes have to stand alone without code, the readers can't fall back on reading code as they would with C. In contrast, C header files are not normally used to navigate C programs; developers tend to go right to the actual subprograms and read code.

Comparing aggregate bug fixes, we see:

	C_SPEC	ADA_SPEC	C_BODY	ADA_BODY
fixes/feature:	.18	.16	.60	.33
fixes/KSLOC:	5.60	4.17	9.63	4.81

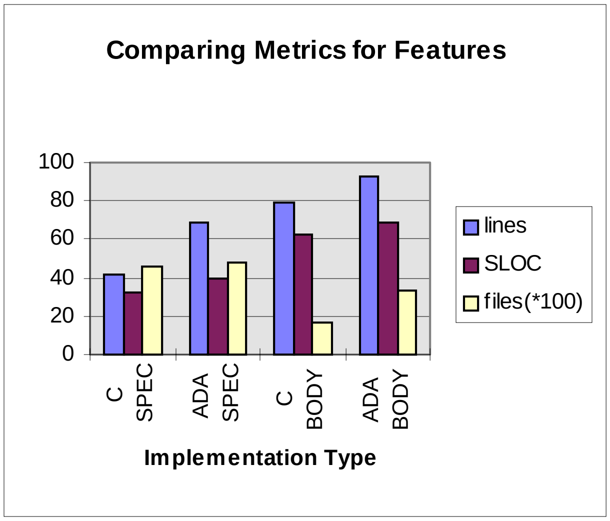


This table indicates that even comparing C_BODY and ADA_BODY, we see that fix rates remain twice as low for Ada compared to C. As expected, the avoidance of C specification changes reduced the number of C header file changes, while the presence of real code in ADA_SPECs increase the fix rate of Ada while decreasing the apparent fix rate of C.

C Feature Equals Ada Feature?

We can understand the effective SLOC in C and Ada a bit more by studying the cost, in lines, SLOC and files, of implementing features. Once again the reluctant additions to C_SPEC files reduces the SLOC in C header files, while body files come out relatively comparable. It is surprising, however, that Ada generally takes more lines to implement features than does C.

	C_SPEC	ADA_SPEC	C_BODY	ADA_BODY
lines/feature:	42	69	79	93
SLOC/feature:	32	40	62	69
files/feature:	.46	.48	.17	.33



This raw data indicates that Ada is slightly more verbose either in SLOC or including all lines. Although a strength of Ada is its high-level, powerful features such as tasking and exceptions, VADS was designed before such features as tasks were available or effective. VADS makes less use of these powerful features, and therefore derives less of the full benefits possible with Ada.

	C	ADA
cost/feature:	$299	$183

In this estimate we amend our cost estimates by measuring feature-to-feature. On a feature basis, for every dollar we spent on Ada features we spent $1.63 on C.

The feature-by-feature costs seem more reliable, but may be complicated by automated code generation and code reuse

Effects of Code Automation and Reuse

In both C and Ada we have tried to make use of reusable and automatically generated code. Some of this code is kept in the same baseline sections as active code. By auto-generated lines we mean lines counted in our line counts that are produced automatically by other programs. By reused code we mean sources that were obtained from partners or as public property but that are then taken over and used for our purposes, including modifications and repairs. Note that about 70% of all code is used in more than one product, as for example the core compiler pieces are all reused for every compiler variant; we don't need to compensate for direct reuse because the source code control system already accounts for this kind of reuse of identical internal code.

	C_FILE	ADA_FILE
reused lines:	134844	175856
auto-generated lines:	276442	7802

These reused and autogenerated lines change the statistics for SLOC in different ways. Reused lines are entered into the normal source/change tracking systems, showing up as a single "feature" with a lot of code associated with it; it may then have repairs and enhancements as would any other active code. Auto-generated code does **not** show up as a feature addition nor does it ever have any fixes or enhancements.

The presence of reused and auto-generated lines underscores the importance of considering features rather than SLOC. It is difficult to adjust for these lines, since they represent work yet do not participate equally with other lines. Since C has by far the greater number of autogenerated lines, its figures will certainly show improved fixes/line ratios as well as higher apparent productivity.

Build-Script Costs for C

The cost per line of scripts was the highest of any category even measured with automated cost distribution. A more detailed analysis would likely reveal that script costs were higher, since they have few tools to support their organization or debugging, and since their effects are often widespread yet their accounting here reflects only the cost of a repair, not the cost to developers who are impeded by the bugs. Scripts have the highest bug rates of any category.

Our cost/feature figure above counts only the cost of developing the code itself, and does not account for the cost of managing C's makefiles and build apparatus (among other things). If we assume that at least half of the script costs are unique to C, then we can calculate cost per equivalent feature as:

	C_FILE	ADA_FILE
cost/feature:	$316	$183

We do not again include costs for C's makefiles in figures that follow. For more accurate cost measures of our historic development, we could take the costs beyond the simple code itself. Complex C programs like ours have become dependent on hand-crafted makefiles; Ada, with compilation order, elaboration order, exceptions, generics and real-time features, was considered too complex to link by hand, and so Ada tools have auto-build capabilities. With ANSI C and C++, "make" complexity is much higher for these languages as well, so makefiles are of reduced importance in the future of C/C++.

Was Ada Used For Easier Jobs?

VADS was not a perfect laboratory for comparing languages: each feature was (normally) done in only one language. What if C was used for the hardest problems and Ada was used for easier problems?

Unfortunately difficulty of project is not easily measured independent of language. Most experts describe real-time code as the most expensive to build and integrate; Ada was used for most of the real time needs of VADS. Experts also agree that sheer size of modules will produce bugs with non-linear growth; C was used for the biggest, oldest sections of VADS. Our experience indicates that dependence on external, changing entities such as OSs, object formats, and optimization requests are one of the most troublesome burdens for development; all of VADS has these sorts of dependencies, though some more than others.

The C language was used for older parts of VADS, and those features directly associated with the older parts of VADS, including the core VADS front end(>200k lines), the core debugger(>130k lines), and the core code generator(>25k lines). C was used for machine-dependent debugger areas (>15k lines each) and code generator areas(>20k lines each in addition to the cores. The debugger includes some difficult areas with real-time programming needs. The C code also shows some big directories that have a lot of duplication and therefore are easier to produce, in the code generator and initialization areas. We also see some directories strongly associated with external sources, for example X, Motif, editors, and Curses.

Ada was generally used for most areas developed after the basic compiler units, and without a strong C legacy from external code. The top 25 largest Ada directories show emphasis on interfaces both to external libraries such as X and from internal data bases such as DIANA. The most difficult code in general is the real time support code, most of which is in smaller units than these big directories but that shows up in the basic runtimes, the Fault Tolerance extensions, the networking support, the target debug support, and the file support systems. Some tool support is present in the form of the optimizer and the cross linker.

The VADS product line involves more than two thousand directories. By measuring the time spent by developers in these directories, we can calculate a rough measure of their “expense”, and examine trends in these more expensive directories to see if they are consistent with the overall trends. The top ten most expensive C directories had these bug fix rates:

Fixes/Feature	Project Directory	Source Lines	Developers
.98	vms_tools	7743	25
.93	amd29k_cg	14248	14
.79	fe	209346	52
.61	d	132492	53
.44	cg	23975	42
.63	optim3	13713	14
.58	m68k_cg	18527	23
.56	unix_d	20307	45
.54	mw_d	40469	9
.17	xwin_d	23292	8

Of the top 25 most expensive directories, 80% were C directories. We have to look farther to find ten expensive Ada directories:

Fixes/Feature	Project Directory	Source Lines	Developers
.71	lib_tools	13689	43
.66	posix_ada	17759	6
.64	sup	18083	43
.47	xlink	16052	15
.38	optim4	42199	17
.38	test_sup	9313	30
.21	new_ts	20438	27
.18	inet	18087	4
.14	i860_cg	12342	10
.12	new_krn	25113	18

Where there are many contributors in every project, we can assume again that our numbers will be statistically unbiased by individual styles. There is no correlation between the number of contributors to a project and its bug fix rate, indicating that the development environment supported parallel developments.

Average fixes/feature in C for “hard” projects:	.62
Average fixes/feature in Ada for “hard” projects:	.39

Regarding C and Ada, we see that for these hardest directories, the ratio of the fixes/feature in the two languages is .62/.39 or 1.58. This ratio is substantially less than the overall 2.26 ratio we observed over the entire operation. We can conclude that in more difficult programming, the use of Ada will not help as much.

Was Any Project Duplicated in Both C and Ada?

It happens that no project was begun in both C and Ada independently. However, several projects were recoded in Ada after beginning in C. There were two motivations for going to Ada at those early times: some were motivated simply because we wanted to be using our own tools; others were moved to Ada because we need to get code onto a cross target where no C compilers existed. One example is the optimizer:

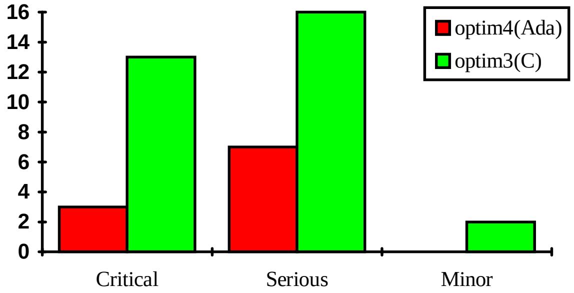
Fixes/Feature	Project Directory	Source Lines	Developers
.63	optim3	13713	14
.38	optim4(Ada)	42199	17

The Ada-based optimizer project was basically a “start-from-scratch”, using an entirely different optimizer data structure and algorithms, and with completely different personnel. The “optim4” optimizer now does more optimizations with greater reliability than did the C-based optimizer.

The optimizer project, being a self-contained user-visible tool of critical importance to at least some users, provides another metric: field-reported bugs:

Project	Critical	Serious	Minor	Years in Service
optim4(Ada)	3	7	0	4 yr.
optim3	13	16	2	4 yr.

Optimizer Project Customer Complaints



The optimizer shows consistency with a general theme, reported later: we saw more bugs escape our testing and getting through to users from C-based code.

Another example coded from C to Ada was the library tools project for VMS. The tools project provide Ada library tools (e.g., create, remove, clean) for VMS.

Fixes/Feature	Project Directory	Source Lines	Developers
.98	vms_tools	7743	25
.22	vms_ada_tool(Ada)	8900	12

In this project there was substantial value carried from the old design, so that the Ada version was able to get a better start.

The runtime was also rewritten in Ada from C but no definitive data survives for the original C effort since development on it occurred mainly prior to 1987 when the source code control records became more accurate.

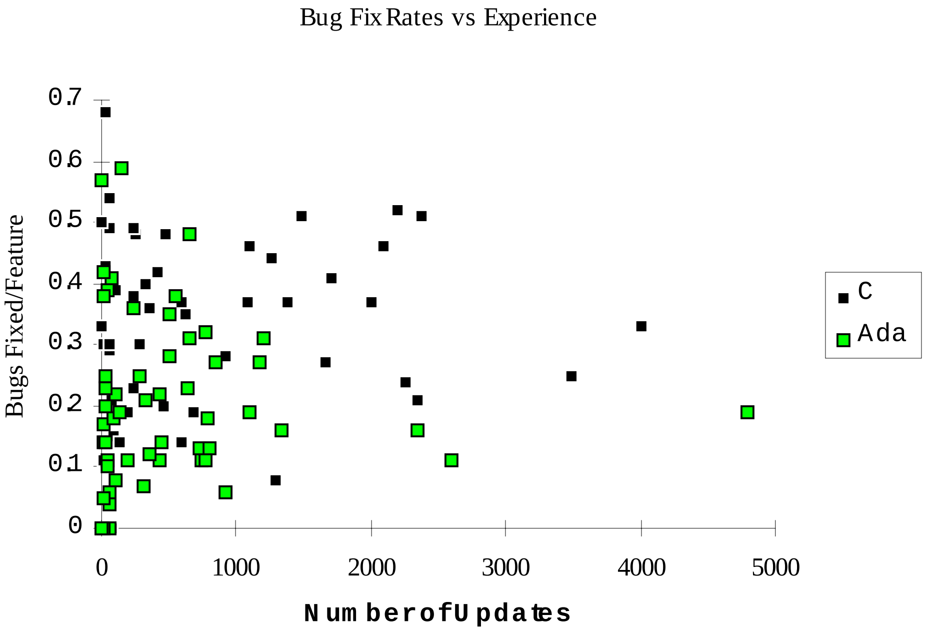
From this small number of projects we cannot conclude any strong statement. It does provide consistent evidence that more fixes occur in C code and that customers see many more bugs.

Programmer Training

Of Verdix hires, 75% had done substantial C programming before joining the VADS project. Less than 25% had done substantial programming in Ada before their hiring. No attempt was made to target Ada experts as hires, though because the business of selling Ada brought the company contact with and recognition from the Ada industry, some experienced hires were made from Ada backgrounds.

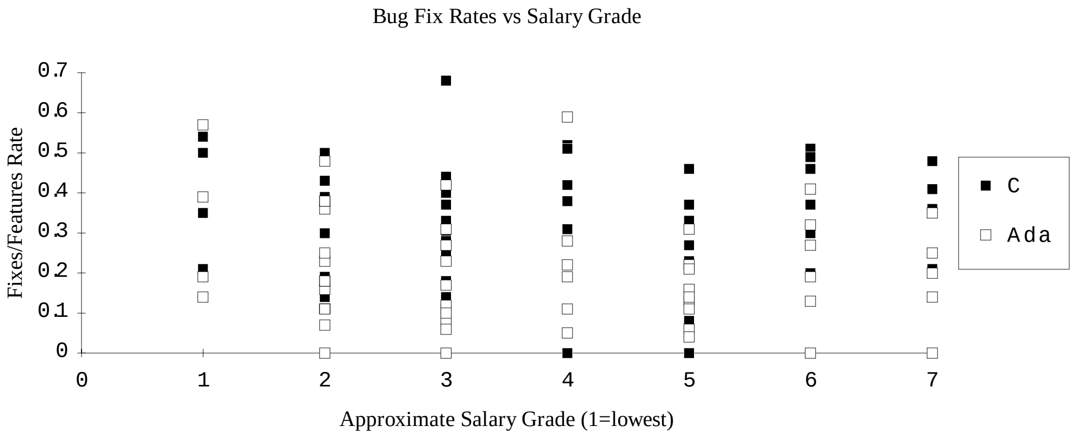
Training was very much "on the job" and "self imposed". Most developers learned by following the code styles that they saw already established in the code they worked from. While few formal code reviews ever occurred, VADS developers expected a constant peer review as other developers wandered in and out of their code. Those with bugs could expect more "help" than those executing flawlessly. Thus, the largest C project was never more than four developers yet records show that nearly every person ever involved in VADS has changed that front end project.

In the following graphs, we compare fix rates with respect to experience, salary and rating. If Ada personnel are substantially different than C personnel we might see patterns in these comparisons.



The above graph plots two spots for each developer: each solid spot represents the average number of fixes per feature for one of the 60 contributing developers when coding in C, while each open spot represents the same average for Ada contributions. There are one C and one Ada spot for each developer; The several spots at (0,0) represent those few people who either never did Ada or who never did C.

We see that C and Ada fix rates are not dissimilar among those with little experience as measured by number of updates, but that as experience increases that Ada fix rates decline significantly. This training effect will be explored in the next section. We also see that there are more experienced people are generally working with C, as is expected since the product line started with C.

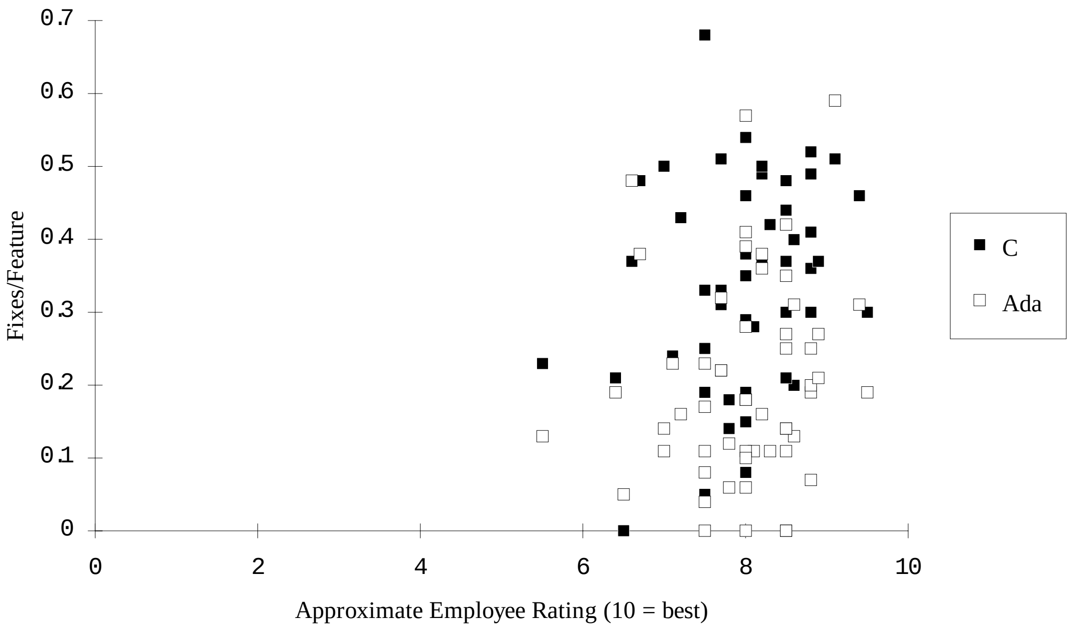


Our data shows little correlation with (approximate) salary. Salary is based on experience and contribution, but not directly on bug fixes. The historic information from the source code control system is used only by the developers themselves. That way there should be no motivation to slant update entries to achieve anything other than accurate records.

In general, developers each do better in Ada than in C, regardless of their level of experience and salary. Only three developers did significant updates for both C and Ada and had lower fix rates for C, and that by only narrow margins. Again we see that more junior people tend to be more in Ada. We also see a small correlation with increasing salary for Ada programming, in that bug rates decline slightly with salary. This effect is seen more clearly in a later graph, where experience as measured in Ada feature updates is compared with fix rates.

Salary does not necessarily reflect skill or general programming effectiveness. We can compare employee rating with bug fix rates and get a measure of where the most skilled people are working:

Bug Fix Rate vs Employee Rating

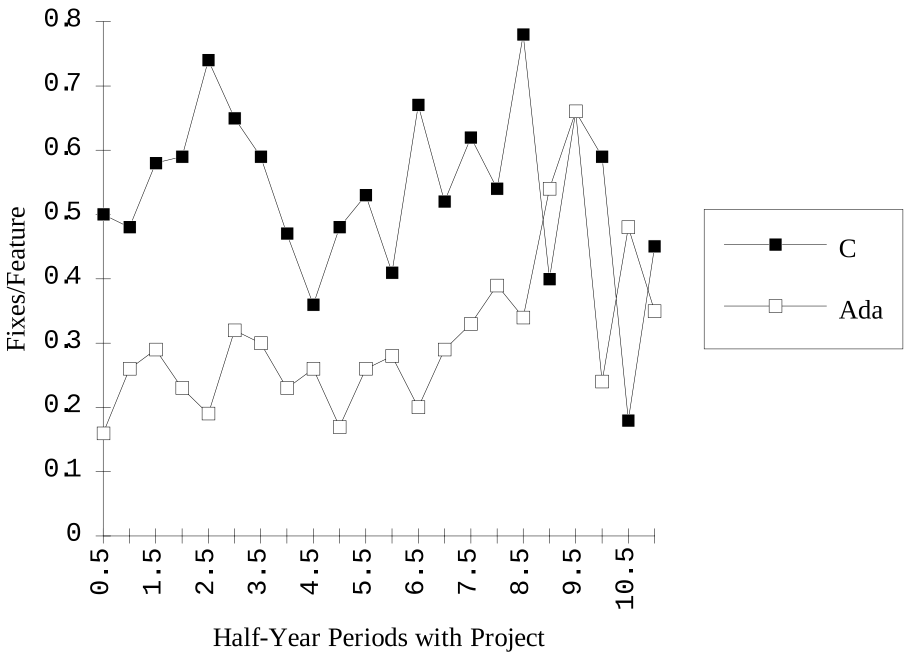


Our data shows no correlation between fix rates and performance Rating.
Ratings are given from 1 to 10 with 10 being superb. The ratings do show that few people on the team are considered mediocre. An approximate rating has been used to protect individuals. This observation underscores that bug rates for internal fixes are not used for ratings. It is expected that more gifted developers would work on more difficult projects and so fix rates would depend more on individual update habits than on true merit.

What is the learning curve for C and Ada?

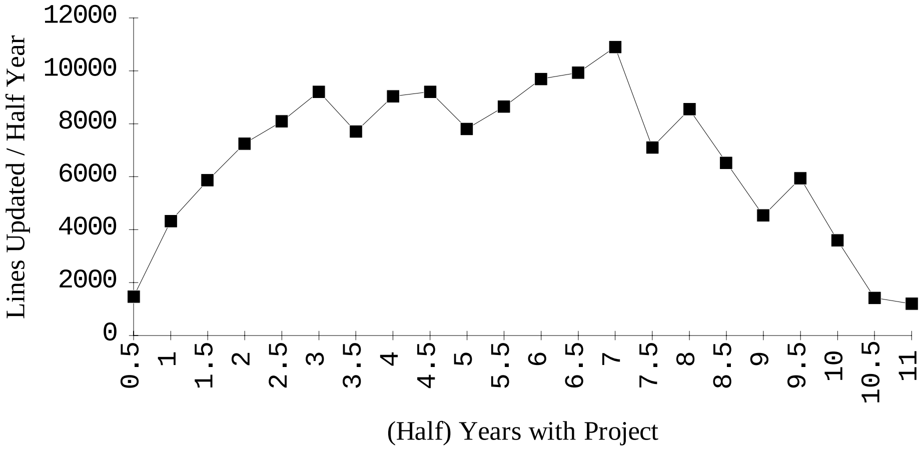
The following graph shows the average contributions of developers in each successive six months after joining the VADS effort. In this graph, the X-axis values represent half-year periods with the project. For example, we see that on the average for all contributors who stayed with the project at least four years, these contributors averaged about .48 fixes per C feature and about .23 features per Ada feature in the period between their 3.5 year and 4th year with the project.

Average Fixes/Feature Over Career in Project



This graph indicates that Ada fixes/feature stay relatively steady until about the 7th year with the project when the number of fixes rises dramatically and erratically. C fix rates start higher but decline through the mid-years, then rise and finally become erratic. The rise in fix rates for those in their seventh or more years with the project is explainable by the gradual shift from developer to manager for those who had been longest with the project; managers typically do much fewer features and spend time helping track down bugs and fixes. The next graph confirms this observation

Lines Updated Over Career

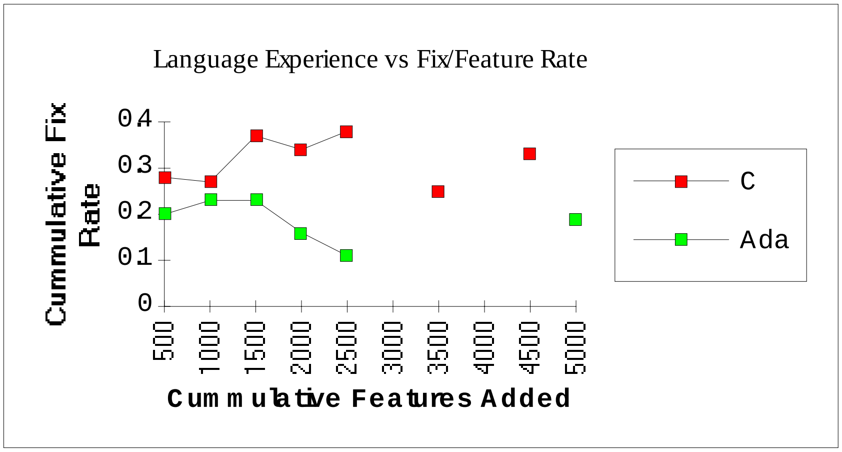


Developer contribution falls off dramatically in the later years after someone has been with the program for a long time. Does this represent burnout or failure to keep up? No, it is the effect of three things: first, these seasoned professionals have been given more management/marketing roles and so have not been able to contribute as much as developers; second, these people are spending more time with mentoring others and helping in difficult subprojects; third, experienced people are given the most difficult assignments. Thus their relative numbers of updates and resultant lines added drop, and their fix rates rise. To repeat, the bug statistics represent bugs **fixed**, not bugs **created**.

Fix rates for Ada start low and stay low until we get out amongst the longtime employees. Of particular interest is that there does not seem to be a "startup cost" with using Ada compared to C. The longtime employees tend to spend more time in C.

The above data does not necessarily tell us how well languages do for expert users, however. The above charts can tell us whet our average fix rates us but it does not really capture language-specific knowledge. It could be that people were make feature additions or changes in areas where they knew little. We can look at language-specific experience instead of gross project experience.

The following graph presents data on learning curves as fix rates for the two languages based on cumulative number of features added for each language. For example, the people who have, in their total contribution on the project, added between 0 and 500 features in C had an average fix rate of .28 fixes/feature.



The fixes per feature rates show a decline in Ada but actually rises in C. Note that values after 2500 features are statistically unreliable as only a few (supercoder) individuals are represented. Even so, we see a distinct rise in bugs for C and a decline in Ada. In each case, more experienced people are likely moving into more difficult contribution areas.

Overall, we can conclude that within the granularity of six months, Ada is not more difficult to learn than is C and that as one continues to learn Ada their code will improve in quality. In contrast, C users do not substantially improve in fix rates after the first six months Finally, we can observe that those people who code primarily in Ada can expect fewer bugs and general improvement, while C users can expect harder going.

Perhaps VADS Programming in C was Substandard?

The VADS bug rates, especially for released products, have been impressively low, whether for C or Ada. Coding standards for C have been aimed at avoiding the known problems with the C language. A partial list is:

- dangling “else”s
- use of “=” for “==”, especially in conditional statements
- use of “/=” for “!=”
- overuse of macros
- use of “cute programming” that sacrifices comprehension for brevity
- use of integer for pointer

Since C compilers have their share of hard-to-find bugs, and incompatibilities of interpretation from on vendor’s C compiler to another, we adopted the general rule that we would avoid buggy areas of compilers as well as of the language itself;

- avoid using the C compiler’s higher optimization levels
- avoid using complex data structuring of nested unions and structures

The most important internal standards have to do with intentional checks and debugability. VADS was built with internal assertions to check both its C and Ada code. We called this “executable documentation”. Asserts in C are designed to make sure that data structures are consistent with each algorithm’s expectations both on entry to and exit from major functional areas. Assertions might be simple (“pointer != null”) or as complex as running two different algorithms and comparing the results.

Some assertions are expensive enough that they cannot be left on at all times, but most are left on even for the product releases in the field; on most compiles the VADS front end expends more than 10% of its time on internal self checks. As a result, most VADS problems are reported as “assertion errors”. While assertion errors occasionally represent false positives, they greatly enhance and early fault detection and debugging.

To further enhance assertion check effectiveness, VADS has internal stress automation: some conditions are very rare and difficult to produce in user code in the presence of shifting optimizations or usage, so stress automation is constructed to force the conditions to cover each case. An example is for code generation when registers are all busy, a difficult condition to produce on RISC machines with hundreds of registers. Stress automation is more effective when working with our TestMate code coverage tools but these were not available during the course of this data.

The VADS debugging tools, including the debugger, its new windowed version, the trace tool, the profiling tool, and the code coverage and test frameworks tools are all designed and used equally well with C or Ada. However, with very large programs no tool can be efficient without customization. VADS developers were expected to generate debugging routines to work in conjunction with the debugger to help isolate problems in complex data structures.

There are more assertions in C code than in Ada code. Ada has stricter typing and runtime checks, so fewer assertions were needed.

Overall, then, we exercised more careful control of C code than we did Ada code.

What About Costs Other Than Adding Features?

The cost of adding features and debugging them is a major part of the engineering costs for products. The engineering costs also include building, testing and releasing products. Their costs would correlate with bug rates since building and testing products has high automation that is broken by bugs. Costs of build/test are generally calculated to be about 35% of the overall development expense of VADS, compared with about 50% spent on new features and enhancements.

These costs are not measured directly in the above figures, as they were not captured in the source control records.

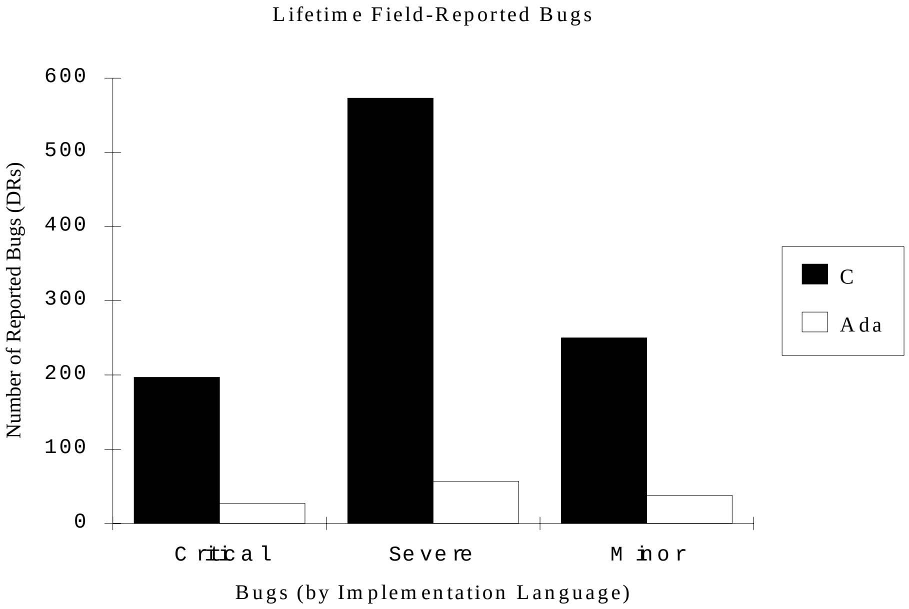
What About the Cost of Customer Support?

Customer Support for VADS represents another cost both inside development as bugs are reported and fixed, and outside development as salespeople, field support and customer service representatives try to assist the customers. Many problems are not the "fault" of the tools themselves, so only a part of the Customer Service cost can be attributed to the tools. However, tool faults are a significant cost both in support and in sales.

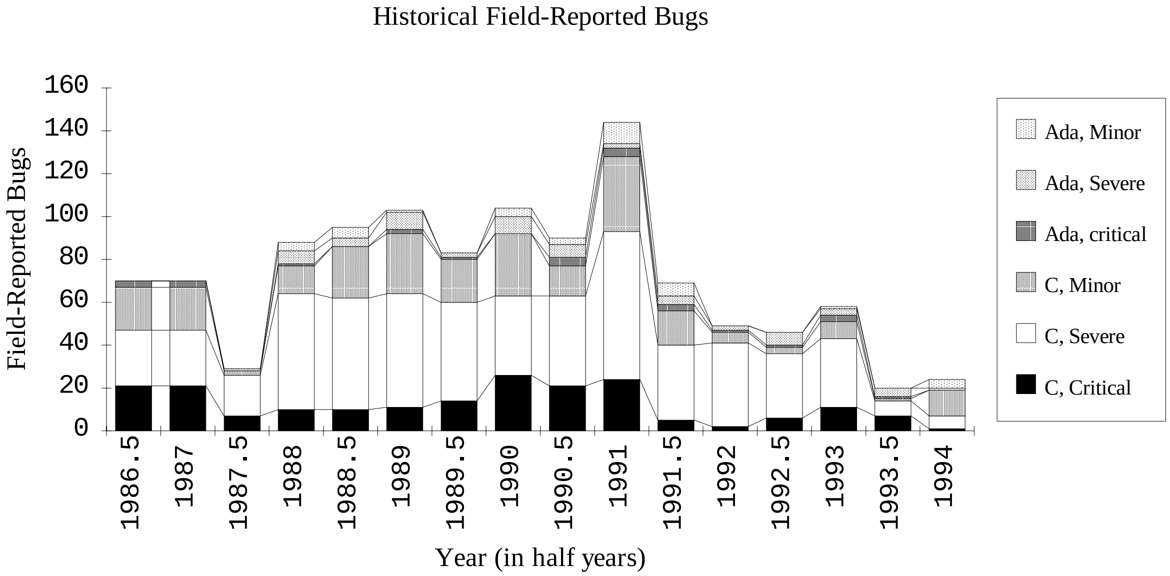
Our Customer Support team's records on customer questions and complaints show the following:

16440 customer interactions (called "tickets")
5072 possible defects (called "CR"s)
2004 deficiency reports (called "DR"s)
1242 actual bugs;(the other DRs were duplicates, not bugs, or requests)

These complaints arose from all aspects of the product line, but many could be attributed to specific parts of our implementation. The following graph summarizes the source of bugs by the implementation language used:



This graph shows that customers were far more unhappy with features implemented in C than they were in Ada. The following graph shows the history of Customer Support complaints attributable to C-based and Ada-Based potions of released product.



This table shows a very encouraging overall trend to fewer customer complaints. We hope that trend will continue. That trend shows more clearly in C-based code than Ada-based. Unfortunately there may be some complicating circumstances. For one, in the last year the merger of the two companies has significantly dislocated both our customer support organization and likely our customers; Thus the 1993.5-1994 numbers for DRs are not likely accurate. For another, since VADS is supported on many hosts and targets, and since some features are available only on a limited basis, the usage of new features and thus their reported bugs tends to lag product introduction. Finally, field reports are dependent on circumstances such as poor corporate performance in 1987, and major new releases in 1991.

In comparing C-based DRs with Ada-based DRs, one could ask whether there was not simply many more lines of C code in use by customers. Prior to 1991 there were about 200k fewer Ada SLOC in customer hands. By 1991, the two code bases had reached parity at roughly 1,500,000 SLOC each.

This table also shows an significant majority of reported user problems as arising from C-based code. Bugs are, unfortunately, not reported uniformly for all aspects of the product except by a few very thorough customers or partners. Therefore one could ask whether C was being used for areas that were more critical to customers and so bugs would more likely cause complaint. We can judge criticality by comparing the ratio of critical complaints against other complaints, on the grounds that users will elevate the importance of a problem based on its impact to them.

On this basis, criticality for various components was judged:

Runtime System	Ada	0.75
Optimizer	C	0.72
Optimizer	Ada	0.42
Code Generator	C	0.41
Cross Linker	Ada	0.28
Front End	C	0.27
Library Services	Ada	0.18
General problems	?	0.08
Prelinker	C	0.06
Autobuilder	C	0.03
Debugger	C	0.00
Support Tools	Ada	0.00
User Documentation	?	0.00

On this metric, there is little reason to believe that bugs would be more frequently reported on C-based code. We can conclude that C is generating many more problems for our users.

Why Does Ada Work Better Than C?

We have related a preliminary finding that Ada performed twice as well as C. Subsequent analysis showed that on a feature by feature basis that Ada was not quite that effective for pure code primarily because redundant code in specifications tended to raise lines of code, but that when effects of makefiles and of external costs were factored back in that Ada costs would be on the order of twice as effective, or half the cost, of our carefully-crafted C. Why?

We have done some analysis of our general development methodology and of our errors. The simple answer is that Ada encourages its users to spend more time in writing to describe their code. That extra effort communicates information to the tools and to all future readers of that code. Future readers then derive benefits, including:

Better Locality of Error. The Ada user tends to be able to assume a lot more about where a problem is arising. Often the bugs are indicated directly by the compiler. When code compiles, it has a higher probability of successfully executing. If it fails, the developer will have to look in fewer places to find the cause. We call this effect “error locality”: an error is local in time if it is discovered very soon after it is created; an error is local in space if it is identified very close (at) the site where the error actually resides.

Better Tool Support. The extra information provided to Ada tools by its users are both a burden and a blessing. For the first several years, the extra information was a burden to Ada tool vendors because they had to do considerably more work to absorb and check the extra information. Now that same information is a blessing that allows the tools to do more for its users. The effect is most pronounced for tasking, where the Ada tools can create parallel, distributed multiprocessing versions of ordinary-looking programs without the users having to do much more than they did for a single processor. Another big win is in machine code, where users can get free access to the underlying hardware but not have to give up the semantic checks and supports of the high level language.

Reduced Effective Complexity. Although Ada is a more complex language than C, the ultimate complexity is a function of the job being done; accomplishing useful complexity within the language may reduce the overall complexity of the developer's job.

Better Organization: A subtler effect of Ada is to encourage better program design. There is a tendency in C of "quick fix" programming. For example, C allows users to create a global variable without registering it in a ".h" file. C allows users to avoid strong typing easily. The effects are subtle but account for some of the more major "progressive design deterioration" that forms the substrate of many extra hours spent not just in debugging but in trying to comprehend old or foreign code.

Is This Experience Applicable Outside Of This Project?

The costs of C should be more pronounced in most other organizations. The VADS team had several advantages that should have made language choice less important:

1) Many teams cannot maintain low turnover. The VADS project consistently kept 95% of its team each year. There is no substitute for knowledge of source.
2) Many teams cannot slowly integrate small numbers of above-average new hires. It is said that good people attract better people, so by keeping standards high and by hiring slowly and carefully, the team has maintained excellent capability. The cost of slow growth is slow time to market, however, which many teams could not tolerate.
3) Many teams do not control as much of their tool chain as does Rational.
   It is a burden to always be a beta-site for our own tools, but it is overall a benefit to be able to customize our environment. Our schedules slip because of dependence on external suppliers more often than because of our own issues.
4) Many teams do not take our aggressive approaches in trying to avoid known C problems and detect problems earlier with thorough testing. We also had advantages of existing test suites and of many users.

The VADS project was complex and very long-term. Many projects do not last as long. It is not clear that Ada's benefits would be as clear for smaller projects. The smallest granularity we've discussed is six months and thousands of lines. For projects less than a year in length, our results are probably that much less applicable.

Will C++ Change The Picture?

Some may look at this study and conclude that C++ will tame C's problems.

Our early experience does not support that conclusion.

Bug rates in C++ are running higher even than C, although we have no where near the ideal comparison platform that we have had with VADS for C and Ada. We do not yet have a large mix of people programming in both C++ and Ada for similar difficulty programs and with history as released products. Our theoretical views of our C++ problems indicate that C++ may allow "run-away inheritance", where many very similar classes are created from a substrate without care to design a smaller number of re-usable classes without many variants; also, existing C++ programs have not yet made good use of templates and so have become cluttered with "container classes" and attendant conversions; finally, so much of C++

goes unseen, hidden behind the notational convenience of the language, that the code can become difficult to understand and navigate. We have had long experience with Object Oriented Software, and believe that OO approaches can yield great benefit if tools can fully support the OO process, and if inheritance complexity can be minimized.

Will Ada94 Change The Picture?

Ada94 is the new version of Ada. It includes OO features similar to C++ except for its multiple inheritance model and its continuance of the strict strong typing of Ada83. It includes support for safety, security, real-time, systems, and other specialty areas.

As with Ada83, Ada94 has received a great deal of design critique and analysis, and is now the first standard OO language. As with Ada83, this fact of national and international standardization with supporting test suites is a great strength for the language and its users.

It is too early to tell how Ada94 for improve code reliability and costs. In theory, once Ada94 tools are mature the language should maintain the strengths of Ada83 while adding support for Object Oriented programming that can be made compatible with C++.

Is the Programming Language All That Important?

The choice of programming language is only one of many factors in project successes and failures. In our opinion, it is NOT the most important.

Among factors with greater influence on project outcomes, we would suggest looking at:

1) Architecture and Design
2) Configuration Management
3) Testing (Effectiveness, Coverage, Rapidity)
4) Support for Iterative/Spiral/Object-Oriented Development
5) Programmer Skill
6) Management Skill

The language choice can do little to make up for weakness in these key areas. A mistake in any of these can kill a project. The language choice may only change expenses by a factor of two. Of course, a million dollars here and a million dollars there and pretty soon you’re talking about real money.

CONCLUSION: Development Costs of C Exceed Those of Ada

Our data indicates that Ada has saved us millions of development dollars. For every development dollar, we could make a case for another three dollars for customer support, sales, marketing and administration costs, spent not only for the extra development but also to calm customers who are affected by our product slips or malfunctions.

Since many programming teams will not be able to switch to Ada or Ada94, Rational Software Corporation is using this data to better our own emerging C and C++ product lines. Our C code is being converted to ANSI C, which can provide some of Ada’s benefits. We are trying to carry as much of Ada’s value into ANSI C and C++ as we can. We are now using our own tools almost exclusively, whether for C, C++, Ada or Ada’s new version called Ada94. We look forward to updating our language comparisons with C++ and Ada94 results.