3

Practical Considerations

It compiles. Ship it.

–Bart Schaefer, Vice President of Engineering,
Z-Code Software Corporation

We believe that CBI and related research efforts have great potential to make software development more responsive and efficient by giving developers accurate data about how software is actually used in deployment. However, testing this idea requires significant experimentation with real, and preferably large, user communities using real applications. This chapter reports on our experience in preparing for such experiments.

We have selected several large open source applications, listed in Table 3.1, comprising some two million lines of code before instrumentation. We have built instrumented packages using the strategy described in Chap. 2, made these packages available to the public, and are now in the process of collecting feedback reports. We have not yet identified any bugs using these reports: our user base is still too small, and does not provide reports in the quantities needed by our statistical debugging techniques. However, we have demonstrated an end-to-end complete CBI system and feel comfortable in claiming that our approach is technically feasible. While aspects of our system could certainly be improved, at this point all components are good enough to support the deployment of realistic instrumented applications and the collection of feedback reports from a large user community.

Table 3.1. Applications from the public deployment

Application	Lines of Code	Shared Libraries	Plugins	Threads
EVOLUTION	574,224	✓	✓	✓
GAIM	209,639		✓	
THE GIMP	657,156	✓	✓	
GNUMERIC	319,137		✓	
NAUTILUS	129,439	✓	✓	✓
RHYTHMBOX	59,569	✓		✓
SPIM	20,300			

B. Liblit: Cooperative Bug Isolation, LNCS 4440, pp. 39–54, 2007.

The design of a CBI system involves interesting challenges, both technical and social. In the next several sections, we focus on the solutions to technical problems most likely to be useful to the designers of similar systems and experiments: integration with existing native compilers (Sect. 3.1), management of static and dynamic linkage (Sect. 3.2), and correct execution in the presence of threads (Sect. 3.3).

Moving toward the social domain, Sect. 3.4 discusses the privacy and security facets of widespread monitoring of deployed software. Section 3.5 considers CBI from the user's perspective, and presents our approach to ensuring that users remain fully informed about and fully in control of their participation in the CBI system.

Lastly, Sect. 3.6 briefly reviews the current status of our public deployment, and offers general information about the state of this experiment under way.

3.1 Native Compiler Integration

The instrumentor as a whole looks and behaves like a C compiler with a few extra command line flags. It specifically emulates GCC, giving us easy access to a large corpus of open source applications. No manual annotation of source code is required, and all existing configuration scripts and makefiles work transparently. This design lets us instrument millions of lines of open source code and keep up with new releases with very short turnaround. Simply changing an environment variable (`$CC`) builds an application with our instrumenting compiler instead of the standard one.

The meat of instrumentation happens as a source to source transformation after the preprocessor and before the real C compiler. However, we actually need to affect all stages of compilation:

before preprocessing (`cpp0`): Pull in extra headers to declare or define various constructs used by instrumented code. For fixed content it is easier to use fixed headers rather than synthesizing the needed constructs programmatically within the instrumentor.

before compilation (`cc1`): Add sampled instrumentation as a source-to-source transformation. Emit additional static site information into temporary files for use in next step.

after assembly (`asm`): Fuse extra static site information from temporary files into the assembled object file.

before linking (`ld`): Pull in extra libraries containing common runtime support code and data used by instrumented programs.

We use GCC's `-B <path>` flag to specify an alternate directory in which to find the compiler stages. Custom scripts in that directory named `cc1` and `asm` do the extra "before compilation" and "after assembly" work and invoke the corresponding native compiler stages as appropriate.

We also use GCC's `-specs=<file>` flag to augment (but not replace) the standard option specs file with one of our own. An *option specs file*, or simply "specfile," determines how GCC parses its command line arguments. We can add flags of our own, request temporary file names, and so on. A specfile is essentially a tiny