Chapter VI

Conclusions

6.1 Summary of Results

The results of the entire project can be summarized as follows:

1. A complete AWK-to-C translation system was successfully developed on the Unix System V Release 4 environment. The translator recognizes the new version of the AWK language as defined in Aho, Kernighan, and Weinberger (1988).

2. A test suite containing more than 240 AWK programs was constructed to test the translation system extensively. All the bugs found during the test have been fixed.

3. A performance measurement suite containing ten representative AWK programs selected from the test suite was constructed to measure the execution times of the translator-generated programs. This execution times were then compared with those of the respective AWK programs running by the Unix AWK interpreter *nawk*. The results show that in most cases, the translator-generated program runs faster than its nawk-processed counterpart. The speed improvements vary widely but the average over the ten representative programs is 32%.

4. In order to facilitate the development process, nearly all stages of software development cycle have been largely automated using the Unix file updating program *make*. This includes generating and compiling the source programs, building the translator and the library, testing the system, measuring the performance, packaging the

source code for distribution, and intalling the whole software package into the system.

6.2 Suggestions for Further Development

There are two major areas in which the AWK-to-C translation system could be further developed and improved to make it more useful: the performance of the generated program, and portability.

6.2.1 Performance of the Generated Program

Although the performance measurement suite has shown that most of the translator-generated programs run faster than their interpreted counterparts, the average speed improvement of 32% is hardly satisfactory. Moreover, Table 5.1 also shows that the speed improvements vary widely among the ten representative programs, ranging from -13% to 135%. This could make any AWK programmer reluctant to use the translator as a replacement for the existing AWK interpreter.

Therefore, further development should be done to improve the generated program's performance, or at least to make the speed improvement more consistent and more uniform. Time profiling of the generated program should be performed to analyse the execution behavior of program to see where the performance bottlenecks are and how they could be dealed with to make the program run faster.

The fact that the speed improvements among the ten representative programs vary greatly could serve as a good clue for pinpointing the performance bottlenecks as well. For example, Table 5.1 shows that the test program *walk* has the speed improvement value of -13% while the program *hist* has the value of 135%. Hence, the AWK code of both programs could be examined to find out which AWK language constructs are translated into the C code that performs well and which constructs are not.

Some design adjustments could probably improve the performance as well. One possible target area is in field splitting. The current design of field splitting mechanism used in the generated program is such that every input line is always splitted into fields regardless of whether the program actually accesses any field or not. Since field splitting has to be done on every line read, this could take a good deal of execution time especially if the input lines are very long and composed of a large number of fields. For example, if each input line is splitted into 100 fields but the program make use of only the second field in its action code, then the time spent on splitting the line from the third field upto the hundredth field will be totally wasted, and if the program has to read 10,000 input lines, the wasted time could be tremendous. Therefore, the implementation of field splitting mechanism could be redesigned so that whether the field should be splitted and how far in the line the splitting should go is decided dynamically, depending on demand.

Another area that a different design could improve the performance is the storage allocation/deallocation mechanism. The current design is such that an allocated object will be deallocated immediately after its logical lifetime has ended. Thus, some execution time could be saved if the object to be deallocated is marked as such and then the actual deallocation action is postponed until the run-time organization has exhausted its allocatable space, thereby triggering the so-called garbage collection process.

6.2.2 Portability

Currently, the translator, the run-time library, and the translator-generated programs are portable only among the Unix Systems that have an ANSI C compiler and the ANSI C standard library available. Further development should be done to improve portability so that the software could be ported to the Unix system that has only the so-called traditional K&R C implementation available, and also to other widely used operating systems such as MSDOS or VMS as well.