Squeeze 0.3.4-Ghent for Tru64Unix User's Manual

Bjorn De Sutter
Ghent University
bjorn.desutter@rug.ac.be

Bruno De Bus
Ghent University
bruno.debus@rug.ac.be

Saumya Debray
University of Arizona
debray@cs.arizona.edu

1 Introduction

This document describes how Squeeze 0.3.4 can be build and used to compact statically linked binaries for the Tru64Unix operating system. Squeeze 0.3.4 has been thoroughly tested on binaries generated for Alpha 21164 and 21264 processors (up till version EV67) and for Tru64Unix versions 4.0D through 5.1.

Section 2 describes how Squeeze is to be build from the source code in the package. Section 3 describes how binaries should be constructed for compaction by Squeeze . Section 4 describes how Squeeze should be executed on the binaries to be compacted. In section 5 some ways of debugging Squeeze are discussed. Finally some platform specific issues are discussed in section 6. We hope to find the time to add some more information on the interal working of Squeeze in the near future.

2 Compiling Squeeze

After the package is untarred, Squeeze can be build in the `src` directory with the following commands:

1. `gmake squeeze_headers`
2. `gmake squeeze_dep`
3. `gmake squeeze`

Gnu's "gmake" is required, as "make" does not provide all the features used to build Squeeze . Also make sure that the environment variable 'OSTYPE' has the value 'osf1'.

Most analyses and optimizations by Squeeze can be turned off or on at run-time by providing the appropriate command-line arguments (see section ??). For constant propagation however, the options can not be selected at run-time. Several options can be set by editing the first few lines of the `eval.new.c` source code file that look like this:

```
#define PROPAGATION_METHOD                        CS

#define BBL_CONSTANT_PROPAGATION                  ON
#define PREDICATE_CONSTANT_PROPAGATION            ON
#define CONDITIONAL_CONSTANT_PROPAGATION          ON
#define INTERPROCEDURAL_CONSTANT_PROPAGATION      ON
#define ELIMINATION                               ON
#define LATE_CALL_JOIN                            ON
#define PROP_DEAD_VALUES                          ON
#define DETECT_DEAD_CALL_EDGES                    ON
```

The first line allows the user to choose between call-site specific interprocedural constant propagation and call-site inspecific propagation. `PROPAGATION_METHOD` has to be defined as CS or CI.

The other options can be set to `ON` or `OFF`. Their meaning is as follows:

- `BBL_CONSTANT_PROPAGATION` — If this is set to `ON`, no constants are propagated over edges between basic blocks during constant propagation (except for the GP-register). In other words, no constant propagation is performed.
- `PREDICATE_CONSTANT_PROPAGATION` — If set to `ON`, additional constants are propagated to one or two of the successor blocks of `beq` and `bne` instructions. The additional constants are found doing a backwards traversal over the code prior to the branch, with the knowledge that the branch condition has to be zero or non-zero for a specific path to be taken.
- `CONDITIONAL_CONSTANT_PROPAGATION` — If set to `ON`, conditional constant propagation is used instead of simple constant propagation. In that case, during the fixpoint calculations, when a condition of a conditional branch evaluates to a constant, the information is propagated to the corresponding successor only.
- `INTERPROCEDURAL_CONSTANT_PROPAGATION` — If set to `OFF`, no constants are propagated interprocedurally, except for the GP-register.
- `ELIMINATION` — Idempotent instructions are detected and possibly deleted if this is set to `ON`. Idempotent instructions are instructions that do not change any register contents (constant or not) and have no side-effects.
- `LATE_CALL_JOIN` — If this is set to `ON`, constants propagated into a procedure from separate call-sites are each propagated separately into the procedure and the join over the values from all call-sites is made at each instruction, instead of joining the values once at the procedure entry-point and then propagate the joined value into the procedure.
- `PROP_DEAD_VALUES` — If this is set to `ON`, dead values are propagated as well, which might be used by later optimizations to simplify instruction sequences.
- `DETECT_DEAD_CALL_EDGES` — If this is turned on, edges modeling unknown calling contexts are considered dead unless proven to be live.

Taking the simpler options speeds-up Squeeze execution at the cost of finding fewer constants.

After editing these options, only the third step of the building process is to be redone.

3 Preparing Binaries For Processing with Squeeze

Binaries that need to be compacted by Squeeze have to contain relocation information and have to be in the correct format (statically linked with relocations performed at link-time, ZMAGIC format). Besides the binary, Squeeze also needs a map of the program layout with respect to code and data sections. Compilation flags for a number of compilers are provided in the table below.

cc	-Wl,-r -Wl,-d -Wl,-z -Wl,-m -non_shared
cxx	-Wl,-r -Wl,-d -Wl,-z -Wl,-m -non_shared
f77	-Wl,-r -Wl,-d -Wl,-z -Wl,-m -non_shared
f90	-Wl,-r -Wl,-d -Wl,-z -Wl,-m -non_shared
gcc	-Wl,-r -Wl,-d -Wl,-z -Wl,-m -static
g++	-Wl,-r -Wl,-d -Wl,-z -Wl,-m -static
g77	-Wl,-r -Wl,-d -Wl,-z -Wl,-m -static

Moreover, the output of the compilation process has to be redirected to a file called `a.out.map`. This map file contains a description of the layout of code and data sections in the program. It might be necessary to edit this file manually for two reasons.

- If during the compilation process, the compiler or linker have produced output on standard output, this information is also gathered in the `a.out.map` file, and should manually be removed.
- For C++ programs, the names of the data and code sections in the map might contain whitespace. The current parser of this map file is not able to handle this. The whitespace in names can in easily be removed by using the `cut` tool, since the names of the sections are of no interest to SQUEEZE and can therefor be shortened to one character.

After compilation the binary has to be stripped with "ostrip -c".

One last restriction on the binaries is that they should not rely on exception handling for correct execution. Exception handling here means exception handling as provided in Compaq's libraries (libc, libf, ...). Some math library procedures such as `sqrt` and `exp` rely on exception handling. There exist versions that do not rely on this (and are a little less accurate) and they can be used by providing the `-D_FASTMATH` flag to Compaq compilers or the `-D_FASTMATH -ffastmath` flags to gnu compilers.

On some gnu compilers, feeding the flag `-Wl,-r` might fail (the problem is that gcc and g++ pass the `-r` flag (from `-W,-r`) too late to the linker). Here is a workaround. Run "gcc -v" : this tells you which spec file gcc is using. Patch the spec file as follows:

- Goto the line following "*link:".
- Make "-r " the first three characters in this line.
- Omit the option "-Wl,-r" in the table above.

4 Compacting Binaries with Squeeze

In order to apply code compaction using SQUEEZE , several steps have to be performed. Please note that SQUEEZE requires a large amount of memory to be executed. The data segment memory limit of a process should be set high enough (with the 'ulimit' command).

4.1 Step 1: Generation of profiling information

To produce profiling information, an instrumented version of the binary can be generated by invoking "squeeze -P Bbl", with or without other options specified below. The produced `b.out` file can then be executed on training input sets. Each execution of `b.out` will produce an `a.out.AltoCounts` file, containing the necessary profile information for SQUEEZE .

When more then one execution of the binary is needed for training purposes, rename the `a.out.AltoCounts` after each execution to some other filename. The execution counts can then be accumulated using the `profadder` program. This program is to be compiled using "gmake profadder". It accepts three command-line options, of which the first two are the filenames of the profiles that need to be accumulated. The result is stored in a file named to the third argument. After accumulating all profiles, the accumulated profile must be named `a.out.AltoCounts`.

4.2 Step 2: Code compaction with Squeeze

To apply SQUEEZE on `a.out`, just type "squeeze". Several files will be produced:

- `b.out`: the code-compacted binary.
- `a.out.litamap`: a specific map indicating dead data in the lita section.
- `a.out.relocatemap`: a specific map indicating which data sections are dead or live.
- `a.out.switchgap`: a file containing the size of padding added to the data segment.

The files that SQUEEZE needs for its execution are: `a.out` and `a.out.map`. `a.out.AltoCounts` is necessary only if the user wants to have SQUEEZE using profile information to optimize the binary.

When code compaction is all that the user needs, his job ends here.

4.3 Step 3: Data compaction with Squeeze

During a third run, SQUEEZE is able to remove dead data from the binary as well. Again it suffices to just type "squeeze". The files produced in the second run of SQUEEZE will now be read and used for dead data removal. After this run, the output files of the first run are overwritten.

Basically, SQUEEZE just looks whether any of the last three files (`a.out.litamap`,`a.out.relocatemap` and `a.out.switchgap` is available. If this is the case, the files are used. Therefor it is necessary to delete these files if you want a second run of SQUEEZE to behave like a run from scratch.

4.4 Squeeze command-line options

SQUEEZE has several run-time options that can be invoked by command-line flags. The most important ones are:

-i [**input-file**] **(default: a.out)** Specifies the name of the file that should be optimized.

-o [**output-file**] **(default: b.out)** Specifies the name of the file being generated.

-r [**number-of-rounds**] **(default: 2)** Specifies how many rounds the optimizer should run the "easy optimization" at most. The C-compiler shipped with the latest versions of Tru64UNIX creates binaries that SQUEEZE cannot handle unless rounds ¿ 0. By "easy optimizations" we mean non-inlining optimization. Currently, SQUEEZE does "easy optimization" first, then inlining, and finally "easy optimizations" again.

-O [**disable-optimize-options**] **(default: none)** Disables certain optimizations. If you want to disbable more than one optimization concatenate the corresponding options with an arbitary delimiter, eg. '+'. The complete list of optimizations is changing rapidly. An uptodate overview can be obtained by running the following command in the master source directory: 'grep DISABLED_OPT *.c'.

-a [**architecture**] **(default: none)** Specializes the binary for a specific architecture. An uptodate overview of supported architectures can be obtained by running the following command in the master source directory: 'grep CONFIG_MACHINE *.c'.

-V [**verbose-optimize-options**] **(default: none)** Specifies the optimizations and analyses of which the user wants more information. An uptodate overview can be obtained by running the following command in the master source directory: 'grep PRINT_OPT_VERBOSE *.c'.

The "-a" flag might seem strange at first and one has to be careful with it. Programs compiled for a generic architecture, and specialized with SQUEEZE to a specific architecture will not work correct on non-compatible prior versions of the architecture. So if compatibility is a concern, don't use this flag. If it is not a concern however, using the flag is useful, even if the program was already compiled for the specific target. The libraries linked with the compiled code contain several tests of the architecture version. At run-time, a specific implementation of some code is then choosen and executed. This test can be eliminated if the flag is used and the library code included for other versions of the processor becomes unreachable and will be eliminated by SQUEEZE .

5 Debugging Squeeze

Bugs and/or questions about SQUEEZE can be submitted to

- debray@cs.arizona.edu
- bjorn.desutter@elis.rug.ac.be

SQUEEZE contains a number of facilities to ease the debugging process.

5.1 Control Flow Graph Inspection

SQUEEZE can output information that is parsed by a tool called `sloop`. `sloop` is able to produce graphical control flow graph representations of procedures.

The command-line option "-S Loop" makes SQUEEZE send this information to the standard output. The additional option "-T [when]" (default: Postopt) specifies when this output is to be produced during the execution of SQUEEZE . A list of possible occasions can be obtained by running the following command in the master source directory: 'grep PRINT_OPT_WHEN *.c'.

Other information than sloop input can also be asked by providing other arguments (possible more than one, concatenated with an arbitrary delimiter, e.g. "+"). For these it might be useful to specify more then one moment during SQUEEZE execution, by concatenating several arguments of the "-T" flag. This is not useful when "-S Loop" is used, since sloop cannot parse files containing more than one occurance of the "-S Loop" output.

5.2 Annotated Assembly Output

SQUEEZE is able to produce annotated binaries by using the "-y" flag. When they are disassembled by the "dis" command, assembly code contains basic block numbers and function names and numbers corresponding to those used internally in SQUEEZE . The following piece of assembly code is an example.

```
       __fflush_unlocked-bbl_3747:
  0x120003424:  22090000       lda     a0, 0(s0)
  0x120003428:  d34013bb       bsr     ra, _xflsbuf-bbl_2048
       __fflush_unlocked-bbl_1989:
  0x12000342c:  a5690018       ldq     s2, 24(s0)
  0x120003430:  496092cb       extwl   s2, 0x4, s2
  0x120003434:  4560900e       and     s2, 0x4, s5
  0x120003438:  4560500b       and     s2, 0x2, s2
  0x12000343c:  417f05ab       cmpeq   s2, zero, s2
  0x120003440:  41df05ae       cmpeq   s5, zero, s5
  0x120003444:  45cb010b       bic     s5, s2, s2
  0x120003448:  f17ffff1       blbs    s2, 0x120003410
       __fflush_unlocked-bbl_1990:
  0x12000344c:  a5e90008       ldq     s6, 8(s0)
  0x120003450:  a5a90010       ldq     s4, 16(s0)
  0x120003454:  41af05ad       cmpeq   s4, s6, s4
  0x120003458:  e5a0000a       beq     s4, 0x120003484
```

This is achieved by defining symbols and relocations in the generated binary corresponding to basic block addresses. We have noticed that this greatly confuses ladebug and dbx. These debuggers should be used on binaries not containing the additional information. The addresses of the instructions in the program do not depend on whether the additional information is included or not, so one could easily execute an unannotated version of the binary with a debugger, while watching the annotated version in some editor.

5.3 Run-time debugging

SQUEEZE has its own built-in debugger. You can activate the debugger by supplying the "-D" flag to SQUEEZE and press "CONTROL-C" during the execution. This brings you to a command-line which allowing to execute commands on the datastructures in SQUEEZE . The available commands are:

- **bbl2fun** *ibbl* or **b2f** *ibbl* Print the name and the internal SQUEEZE number of internal basic block *ibbl*'s function.
- **break** or **b** Break at an option check. Every time SQUEEZE calls `HasOption()`, the option is compared to the value of the breakpoint. "break Factor" would stop SQUEEZE just before `Factor()` is executed. You can add new breakpoint locations by adding "DEBUG_CHECK("Breakpoint_name")" somewhere in the code. Everytime you add or remove a breakpoint, SQUEEZE should be recompiled (third step, see section 2).
- **clearranges** or **cr** Clears all ASK ranges. See later.
- **cont** or **c** Continue SQUEEZE , stop at the next DEBUG statement or breakpoint in the source code.
- **help** or **h** Print a help message.
- **printfun** *ifun* [*file*] or **pf** *ifun* [*file*] Print function *ifun* (the function's name or internal number can be feeded) to *file* (default: stdout).
- **printloops** *ifun* [*file*] or **pf** *ifun* [*file*] Print Sloop-input for the single function *ifun* (name or number) to *ifile* (default: stdout).
- **printvaluespec** *ibbl* or **pv** *ibbl* Print register constants at basic block *ibbl*'s entry point. This can only be applied when constant propagation is being executed.
- **off** Inactivate debugging.
- **quit** or **q** Exit SQUEEZE .

The debugger can also be activated during the execution of SQUEEZE by inserting a "DEBUG;" statement in the source code. The debugger is then activated from the moment the statement is executed.

Another useful technique for debugging SQUEEZE is the use of the "ASK" and the "ASK2" macros. To illustrate this, let us assume we want to implement an optimization on some basic blocks, called MyOptimization. If we are not certain that our optimization performs correct we add an ASK(2) macro before our code and an END_ASK macro after the code.

```
void MyOptimization
{
   INDEX ibbl;
   FOREACH_BBL(ibbl)
   {
      if !(valid_for_this_optimization(ibbl)) continue;
      ASK2("Perform my optimization on ibbl %d",ibbl) // NO ;!
      DEBUG_CHECK("BeforeMy");
```

```
        Perform_Optimization(ibbl);
        DEBUG_CHECK("AfterMy");
        END_ASK
    }
}
```

Every time the code between ASK2 and ASK executes, you get a prompt asking you if you want to execute the code block on basic block ibbl. Valid answers are

- Y: Execute the block this time, ask again the next time the code is entered.
- N: Do not execute the code this time, ask again the next time the code is entered.
- A: Always execute this code block (don't ask again).
- D: Never execute this code block (don't ask again).
- R *INDEX1 INDEX2*: (only for ASK2) Execute only for incoming values (the second parameter of ASK2) in the range[*INDEX1, INDEX2*].

When one of the above answers is given by the user, it only applies to the one ASK code block the user was prompted for. The user can control all ASK blocks in Squeeze by supplying the parameter -A and a numeric value:

- -A 0: Verbose: prompt the use for all ASK code blocks.
- -A 1: Execute all ASK code blocks without prompting the user.
- -A 2: Never execute an ASK code block without prompting the user.

5.4 Limiting Optimization Applicability

It often happens that a bug being introduced by a new optimization in Squeeze only becomes manifest in a limited number of places in the generated binary. These can be found by limiting the applicability of the new optimization to certain regions of the optimized program. One does not want to recompile Squeeze however for each such region.

Two global variables CONFIG_MIN and CONFIG_MAX can therefor be specified on the command-line as arguments of "-n" and "-x". It then suffices to insert conditions into Squeeze once, recompile it and then run it multiple times for different command-line arguments. On can e.g. start with the whole program and then narrow down the interval to find a location in the binary that is buggy.

6 Platform Issues

COFF Binaries (A port to Linux ELF is under way !!) Currently, Squeeze works with COFF binaries only. It assumes that those binaries are statically compiled. It also expects relocation information and symbol table information to be present.

Segments and Sections A COFF binary consists of 3 segments: text, data, bss. Each segment consists of sections.

Code Sections Not all section of the text segment contain code but only sections of the text segment contain code. Squeeze merges the code containing sections into a single sections. It is helpful that those sections are adjacent in the text segment.

Readonly Sections All sections in the text segment are readonly but there are sections in the data segment that are also readonly. It is necessary to know which sections are readonly so that loads from these sections can be "evaluated".

Relocation Information SQUEEZE uses relocation information to to determine basic block boundaries and to find basic blocks that can be the target of indirect jumps. SQUEEZE will also "fix" relocatable data before the binary is written back.

Relocation Types SQUEEZE only considers relocation entries referencing the text segment. Luckily, there aren't all this many. There 8 byte addresses (REFQUAD) and 4 byte gp relative addresses (GPREL32) and one other class of relocation entries.

Refquads indicate the beginning of a basic block that can be jumped to from anywhere.

This is a rather coarse (but safe) assumption. Most of the time the refquad is used to describe a function address which is used in an non-computed jump, it would be nice to know which ones are used for computed jumps.

Gprel32s indicate the beginning of a basic block that is target of a switch statement (computed jump).

It is a big pain to figure out what the possible targets of a switch statement are and it would be nice if the symbol table provided this information.

There is a third type of relocation information used with init and fini sections.

Symbol Table The symbol table is not used all this much. SQUEEZE uses it to associate names with addresses (eg. function name) and to find function boundaries. The latter could also be achieved using procedure descriptors.

Procedure Descriptor Table Procedure descriptors are not used anymore but they contain important information and I might look into them again.

Useful Information That Would Be Nice To Have Some information is approximated in SQUEEZE :

- targets of computed jumps
- Save/Restored register in a function
- Save/Restored register around a function call
- List of functions whose address is taken (in source or implicitly) which might hence be the target of computed jumps.