DAISY Dynamic Binary Translation Software

Erik R. Altman
Kemal Ebcioğlu
IBM T.J. Watson Research Center
PO Box 218
Yorktown Heights, New York 10598

Contents

List of Figures

1 Introduction

DAISY (**D**ynamically **A**rchitected **I**nstruction **S**et from **Y**orktown) translates a program at runtime from one instruction set to another in a manner transparent to the user. In other words, the user sees only the input instruction set. This implementation of **DAISY** uses *PowerPC* as the input instruction set and a new VLIW architecture as the target instruction set. This version also works on a page-by-page basis, translating all code it can find on the first page of an application program, then simulating this translated code until control passed to a new page, then translating the second page, and so on. Thus, if some code pages are never executed, they are never translated.
Further (dated) information on **DAISY** can also be found at the DAISY website:
www.research.ibm.com/daisy
and the IBM VLIW website:
www.research.ibm.com/vliw

2 Sample Session with DAISY

We now describe **DAISY** in more detail, beginning with a sample session of a user running **DAISY**. Section 2.2 then gives a rough overview of what **DAISY** has done internally.

2.1 User View

The **DAISY** executable is called `daisy`. A summary of `daisy` usage and its flags can be obtained by typing `daisy` with no arguments. This summary is also reproduced in Figures 1 and 2.

In order to use `daisy`, the `datasize` limit must be at least 256 Mbytes. Under *csh* or *tcsh*, this can be accomplished via the command `limit datasize 512000 kbytes`. It is also useful to have a large amount of paging space, at least 256 Mbytes, and preferably 400 Mbytes. The amount you currently have may be determined by typing `lsps -a`. Contact your system administrator if you need to increase your amount.

The most basic invocation of `daisy` is to give it the name of the executable which is to be translated and simulated. For example, to translate `/bin/ls`, one types

```
daisy /bin/ls
```

The immediate result is the same as that from invoking `/bin/ls` directly, i.e. the contents of the current directory are displayed on the screen, albeit more slowly. In addition, `daisy` appends a variety of statistics about its translation of `/bin/ls` to a file called `daisy.stats` in the current directory. `daisy` also creates 3 additional files in the current directory:

```
ls.vliw_perf_ins_cnts
ls.vliw_perf_ins_info
ls.vliw_spec_ins_info
```

```
Incorrect # of Arguments:
Correct usage is

daisy [<flags>] <Executable> [<Executable Args>]

where <Executable> is a fully linked and executable XCOFF file, and
<Executable Args> are the normal arguments, if any, to <Executable>.
The <flags> are for "daisy".  Currently there are:

-A:  Dump breakpoints for dbx in file "daisy.bk"

-B<info_dir>:  Directory to put info files for "dvstats"
               (Give full path if <Executable> changes curr directory)

-C:  Run without DAISY translation.

-E1: Trace all branches     in xlated   pgm.
-E2: Trace branch-and-links in xlated   pgm.
-E3: Trace cond branches    in xlated   pgm.
-E4: Trace all branches     in xlated   pgm and libs.
-E5: Trace branch-and-links in xlated   pgm and libs.
-E6: Trace cond branches    in xlated   pgm and libs.

-F:  Unroll loops as long as ILP improves.

-G:  Perform cache simulation.
-Z:  Save cache trace (as <Executable>.trace in <Executable> directory).

-H<num1>: If DAISY compiler invoked <num1> times, clr xlation table.
-J<num2>: If DAISY compiler invoked <num1> + <num2>
          times, finish execution in native mode.

-I:  Do NOT dump files needed by "dvstats"
-N:  Dump VLIW code in "daisy.vliw"

-K:  Override xlated program’s segv handler

-L<num>: Do -M action if hit <num> dyn crosspage entry pts
-M<num>: If <num> = 1 stop, if 2 exec native code, else cont

-P:  Support Power architecture instead of PowerPC

-Q<num>:  If <num>==0, use real join pts, o.w. cnt only when pt starts
          a continuation.  W/no -Q, cnt whenever instruc encountered.

-R:  Use LOAD latency with LOAD-VERIFY, o.w. available immediately

-S:  Do NOT split complex record ops: op. into op,cmpi
```

Figure 1. (1) Usage info from invoking `daisy` **with no arguments.**

```
Correct daisy usage flags (cont)

-T:  Save xlation to disk (in ./save.daisy) for possible reuse)
-U:  Retrieve prev xlation from disk (in ./save.daisy) for reuse)

-V<num>: Running AIX version <num> (Default=414)

-W:  Dump to stdout, a copy of stats appended to "daisy.stats"

-X<num>:  Must have done "dvstats" on previous run
  <num> = 0,1:  Use prof-dir-feedback w/cnts.
  <num> = 2,3:  Use prof-dir-feedback w/prob.
  <num> = 0,2:  Generate code for 0-count targets only if executed.
  <num> = 1,3:  Always generate code for 0-count targets.

-Y: STB r3,X ==> STB   r3,x           | STH r3,X ==> STH   r3,x
    LBZ r4,X ==> RLINM r4,r3,0,24,31 | LHZ r4,X ==> RLINM r4,r3,0,16,31
    No LD-VER needed, but copy-prop info for r3 not go to r4

Summary statistics are displayed to "stdout" and are
appended to the file "daisy.stats".  Cache simulation
results are placed in the file "cache.out", while the
cache configuration is read from the file "cache.config"
in the current directory or from the directory specified by
the CACHE_CONFIG environment variable.

Additional configuration parameters are read from the
file "daisy.config", if it exists.  For example "num_alus"
specifies the number of ALUS.  The format of "daisy.config"
is one quantity (e.g. "num_alus") per line followed by
whitespace and the numerical value, e.g. 16.  Blank lines
are ignored and # is a comment prefix for the line.  If
"daisy.config" exists in the current directory, it is used,
otherwise if "daisy.config" exists in the directory specified
by the environment variable DAISY_CONFIG, it is used.  Defaults
are used and a warning issued if neither is present.

The -L and -M can be very useful as debugging aids.
"daisy" can be run several times with the target program
using binary search to assign values to -L<num>.  For example use
-L1000 -M2 to see if the DAISY translated program crashes in the first
1000 page crossings.  If it does, then try -L500 -M2 in binary search
fashion.  Continue this until the precise page causing the crash is
located.  Assume the last value for which the program runs correctly
is -L319.  Then do an additional run with -L319 -M1 to determine the
PowerPC address of the page entry causing the problem.

NOTE:  The -H and -J flags used to be used for similar purposes, but
       use of -L and -M is much preferred.
```

Figure 2. (2) Usage info from invoking `daisy` **with no arguments.**

These binary files are used to create a myriad of statistics using a post-pass tool called `dvstats`. `dvstats` takes as arguments the base name of the translated program, as well as the directory in which the 3 files above may be found. For example,

```
    dvstats ls .
```

The final "." specifies that dvstats should look in the current directory for the 3 files listed above. Running `dvstats` produces several more files, all with the prefix `ls.`. Probably the most interesting is `ls.histo` which has the infinite cache ILP attained by the translated code. Infinite cache ILP is measured simply as the ratio of *PowerPC* instructions used to execute the original program (`/bin/ls`) to the number of VLIW instructions used to execute the original program. The `ls.histo` file also reports how many and what type of operations were packed into each VLIW instruction. Other files report opcode frequencies and critical paths through the code, along with the ILP attained on each of those paths. These matters are discussed in more detail in Section 11.

One other file is of significant interest here. If the `-N` flag is given with `daisy`, a dump of the VLIW code is produced in a file called `daisy.vliw`. This file can be more than a megabyte in size, even for a program like `/bin/ls` because shared library code (e.g. **printf** and **malloc**) is translated too, and because there is a significant explosion in code size – typically a factor of 4, but varying widely. Note too that any previous `daisy.vliw` file in the current directory is destroyed when the `-N` flag is specified. Section 8 gives a description of the syntax of the VLIW instructions in `daisy.vliw`

2.2 Internal View

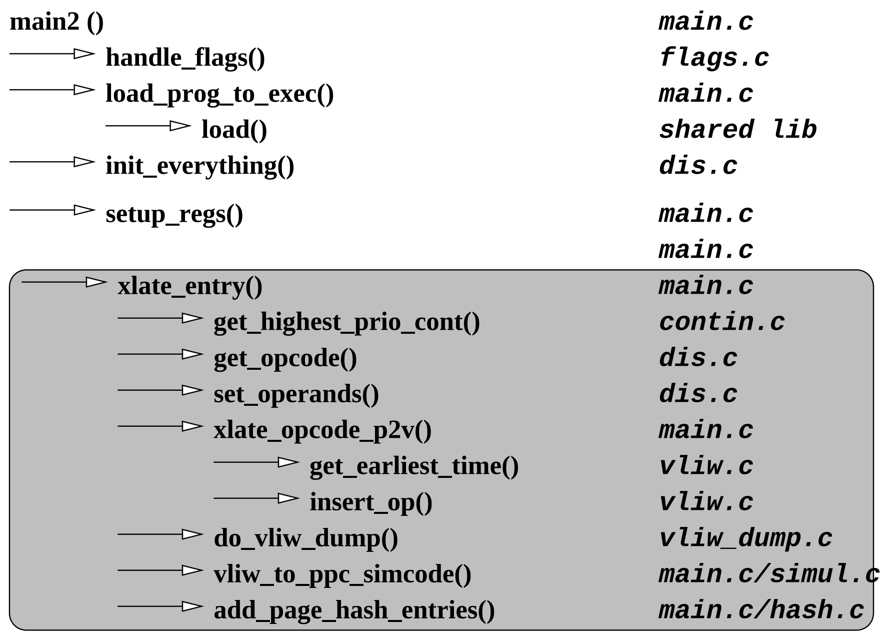
Figures 3 and 4 depict high level call-graphs of what `daisy` does internally as it translates `/bin/ls`. Figure 3 depicts how `daisy` initializes and configures itself, then begins translating. Figure 4 depicts `daisy` in the steady-state when it is mixing translation and execution of `/bin/ls`. Note that the shaded portion of Figures 3 and 4 is common and corresponds to the actual work of translation. Below is a fuller description of the internal processing depicted in Figures 3 and 4.

- The **main()** function is in `main.c` (Actually **main2()** is in `main.c`. **main()** is in `r13_base.s` so as to properly setup the stack for the translated program, e.g. `/bin/ls`. The value of the stack to use for the translated program is the `sp_val` parameter passed to **main2()** by **main()**.)

- **main2()**, first checks for any flags such as `-N` via a call to **handle_flags()** in `flags.c`.

- **main2()** then opens the file to be translated (e.g `/bin/ls`) and reads the header in order to find the number of instructions in the text section. (This is mainly used for informational purposes to know what fraction of the execution time was spent in shared libraries.)

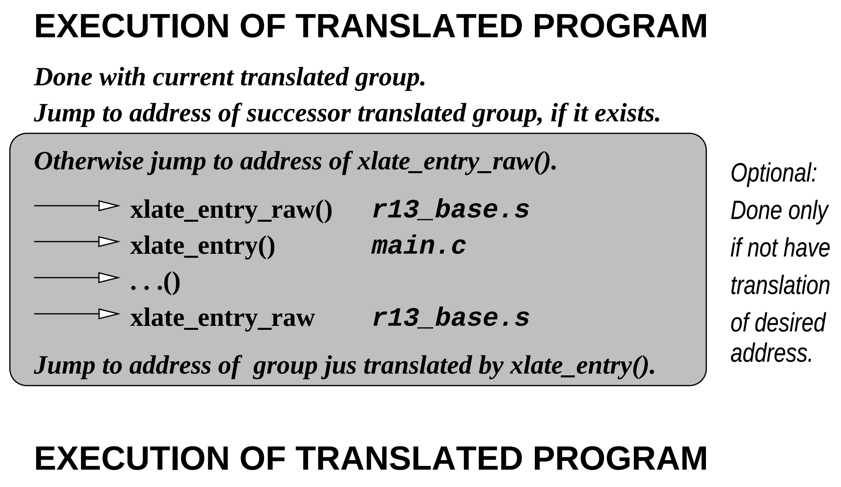
- **main2()** then loads the target program into memory via a call to **load_prog_to_exec()**, which in turn invokes the AIX function **load()**, which reads the target XCOFF file and performs all necessary relocation.



```
main2 ()                                        main.c
  ──▷ handle_flags()                            flags.c
  ──▷ load_prog_to_exec()                       main.c
        ──▷ load()                              shared lib
  ──▷ init_everything()                         dis.c

  ──▷ setup_regs()                              main.c
                                                main.c
  ──▷ xlate_entry()                             main.c
        ──▷ get_highest_prio_cont()             contin.c
        ──▷ get_opcode()                        dis.c
        ──▷ set_operands()                      dis.c
        ──▷ xlate_opcode_p2v()                  main.c
              ──▷ get_earliest_time()           vliw.c
              ──▷ insert_op()                   vliw.c
        ──▷ do_vliw_dump()                      vliw_dump.c
        ──▷ vliw_to_ppc_simcode()               main.c/simul.c
        ──▷ add_page_hash_entries()             main.c/hash.c
```

Figure 3. Initialization, configuration, and start of translation under daisy.



```
EXECUTION OF TRANSLATED PROGRAM
Done with current translated group.
Jump to address of successor translated group, if it exists.
Otherwise jump to address of xlate_entry_raw().                Optional:
  ──▷ xlate_entry_raw()      r13_base.s                        Done only
  ──▷ xlate_entry()          main.c                            if not have
  ──▷ . . .()                                                  translation
  ──▷ xlate_entry_raw        r13_base.s                        of desired
Jump to address of  group jus translated by xlate_entry().     address.

EXECUTION OF TRANSLATED PROGRAM
```

Figure 4. Steady state of daisy mixing execution and translation.

- **main2()** then does a variety of initializations via the call to `init_everything()`. One of these initializations is to call `vliw_init()` in `vliw.c`. `vliw_init()` in turn reads additional configuration information such as the number of registers and function units via a call to **read_config()** in `rd_config.c`, which in turn reads this information from the file `daisy.config`. If the environment variable, *DAISY_CONFIG* is set, **read_config()** assumes it specifies the directory in which to look for `daisy.config`. If *DAISY_CONFIG* is not set, then **read_config()** looks for `daisy.config` in the current directory. If `daisy.config` does not exist, **read_config()** provides a default configuration.

- Note that **DAISY** has its own version of **malloc()**. If **DAISY** invokes **malloc()**, it perturbs how the translated program runs.

- **main2()** then does various initializations dealing with profile-directed feedback and tracing the branches in the original *PowerPC* program (e.g. `/bin/ls`).

- Next **main2()** makes sure that the VLIW registers match the *PowerPC* registers at the start of the program, e.g. VLIW `r3` and *PowerPC* `r3` both contain **argc**. The call to **setup_regs()** which does this, also sets up several other values for use during simulation. Section 8 discusses this in more detail.

- In order to run correctly, the translator and the translated/simulated program must use different stack pointers. Thus **main2()** sets up the simulator (and hence the translated program (e.g `/bin/ls`)) to use the translator's stack pointer (`sp_val`). When the translator is re-invoked to translate subsequent pages it will use its own stack pointer pointing to a statically allocated stack. This is done because we know that the translator can make do with a small fixed stack size, but we do not know if this is true of the translated program.

- **main2()** then sets up a couple of additional values for use by the simulator in **r13_area** and invokes **xlate_entry()**.

- As depicted in Figure 4, **xlate_entry()** is the main entry point of the translator, and takes one argument, the address from which it is to translate. Subsequently when a page needs translation, the simulator branches to **xlate_entry_raw()**, in `r13_base.s`. **xlate_entry_raw()** (1) saves registers live during the execution of simulated VLIW code, (2) sets up registers and the stack frame for use by normal C code such as the **xlate_entry()** function, and (3) calls **xlate_entry()**. When **xlate_entry()** returns, it passes in register `r3` the address of the simulation code for the VLIW code just generated. **xlate_entry_raw()** then restore registers used during exeuction of simulated VLIW code and then branches to the address passed in register `r3`.

- **xlate_entry()** does some initialization of its own, then enters a doubly nested `while` loop. The outer `while` loop checks via a call to **get_highest_prio_cont()** in `contin.c`, if there are any open paths on the page that remain to be translated. Initially there is one such open path — the entry point to the page. Translation on a path stops whenever a branch is encountered. If the branch is conditional, two new path continuations — the target and the fall-through — are added to the open path list (assuming that both are on the current page). Additions to the open path list are made via calls to **add_to_xlate_list()**. Unconditional branches of course place only their target in the path list [1].

- Once a path continuation is found, several checks are made as to whether *PowerPC* operations from this path should be scheduled into earlier VLIW instructions — thus increasing parallelism, or if a new scheduling region should begin at the current operation — thus decreasing code explosion.
- In either case, scheduling of *PowerPC* operations from the current path is done in the inner **while (TRUE)** loop.
- A call to **get_opcode()** partially parses the *PowerPC* operation.
- Then several special types of *PowerPC* operation are checked for — *illegal ops, load/store multiple ops,* and *load/store update ops.*
- Scheduling on this path stops if there is an *illegal op.* If execution comes here, something is probably wrong with the original program. If this path is actually executed, the simulator will print a message that an *illegal op* was encountered and terminate the run.
- *Load/store multiple* and *load/store update* operations are split into simpler primitives and then scheduled like any other operation. There is one caveat to this. `lwsx`, `stwsx`, (and `lscbx`) are not split. Instead **DAISY** acts as if these were VLIW primitive operations. *This should change at some point!*
- For other *PowerPC* operations, parsing is completed by a call to **set_operands()**. Another illegal check must be made here because some operands of `mtspr` and `mfspr` are illegal, while others are not.
- Legal opcodes are sent to the routine **xlate_opcode_p2v()**.
- **xlate_opcode_p2v()** divides instructions into *branches* and *ALU ops.*
- A variety of optimizations can be done with ALU and memory operations. These include *copy propagation, combining, load-motion above stores, load-store alias analysis,* and *dead-code elimination.* All of these are checked by **xlate_opcode_p2v()**.
- Additional splitting of complex ALU ops into simpler ones is done in **xlate_opcode_p2v()**. In particular *record* form ops (e.g. `and.`) that set **CR0** are split (e.g. to `and` followed by `cmpli`). Likewise `rlwimi` is split and the individual pieces scheduled.

[1] `Branch-and-link` instructions can optionally place the fall-through path in the list as well.

- If none of the optimizations or splittings apply — the most common case — **xlate_opcode_p2v()** calculates the earliest time at which the op may be scheduled due to data dependences. This is done via a call to **get_earliest_time()**, which is in `vliw.c`. **xlate_opcode_p2v()** then calls **insert_op()** to schedule the operation at or after this earliest time, as function unit and register availability allow. Exactly how `insert_op` does scheduling is discussed in Section 4.

- Branch operations are split into several different types, the main being *direct onpage branches*, *direct offpage branches*, and *indirect branches* through the `Counter` or `LinkReg`. `RFI` – return from interrupt, `SC` (or `SVC`), and `traps` are also handled, although `SC` and `traps` are infrequent in user code and `RFI` is non-existent.

- Returning up the call chain to **xlate_entry()**, once all operations on all open paths are scheduled, the outer `while` loop exits.

- **xlate_entry()** then checks if the user wants a dump of the VLIW code. The `-N` flag discussed above in Section 2.1 corresponds to the **do_vliw_dump** variable.

- In order to generate statistics at the end of a program run, **DAISY** must know when a program ends. Programs usually end via a kernel service call to location **0x3600** (or **0x37D0** in AIX 4.3). The value in register `r2` determines the service requested. Thus when the **DAISY** simulator sees that it is executing the translation of location 0x3600, it checks if `r2` contains the **_exit()** service value. If so, it writes its statistics to disk, as described above in Section 2.1, and then terminates. The call to **dump_exit_chk()** sets up the simulator to perform this check.

- The **DAISY** simulator is actually a translation of the VLIW code back to *PowerPC* operations which then run natively on the *PowerPC* machine. This is detailed in Section 9. The call to **vliw_to_ppc_simcode()** performs this translation. The simulation code not only models the execution of VLIW instructions, but also keeps count of how many instructions execute, and can also call a cache simulator.

- We want subsequent branches to the entry point of this page to go to the translation we just created, not to the translator. In order to do this **DAISY** maintains a hash table mapping *PowerPC* instruction address to their corresponding VLIW translation. The call to **add_page_hash_entries()** creates this mapping. Note that not only the entry point of the page goes into this hash table, but any other points at which a new scheduling region is begun, as was described above.

- After this, **xlate_entry()** returns to the **main2()** function. **main2()** knows that simulated code for the first group – the entry point of a program is always at the start of **xlate_mem** – the start of translation memory, and sets up an AIX-style function pointer so that an indirect branch can made to this first translation.

- **main2()** then branches to this address to begin executing the translated program (e.g `/bin/ls`).

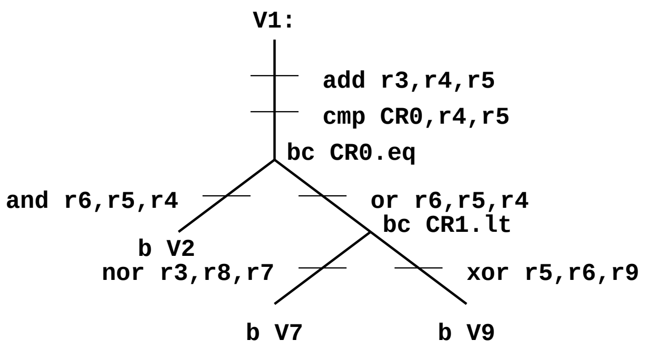


Figure 5. Sample VLIW Tree Instruction.

- When the translated program comes to a new page or performs an indirect jump, it will re-invoke **xlate_entry()** to do the same tasks we just outlined for the first page.
- Subsequent invocations of **xlate_entry()** actually occur indirectly, as can be seen in Figure 4. The simulation code branches to **xlate_entry_raw()** which is in `r13_base.s`.
- **xlate_entry_raw()** saves *PowerPC* registers that may be live (e.g. `r1-r12`, `fp0-fp3`, and the `XER`). It then sets up the TOC (register `r2`, which is used as a *Table Of Contents* in AIX to access global variables) and stack pointer (`r1`) for the translator. It also copies parameters passed by the simulator to standard AIX parameter registers. For example the address to translate is copied from `r27` to `r3`. Upon return to **xlate_entry_raw()**, control is returned to the simulated VLIW code, as described above.
- This process continues until the simulation code detects the translated program is exiting, as described above.

3 VLIW Tree Instructions and Generic Machine Model

DAISY currently targets a VLIW machine whose instructions are trees, a format developed by Ebcioglu [1] and his collaborators over the years. A sample tree is shown in Figure 5. There are several points to note:

- The tree instruction splits at conditional branch instructions. The left path corresponds to the branch condition being false, while the right path corresponds to the branch condition being true. In Figure 5 there are two conditional branches. The edge terminating at `b V2` corresponds to the false path of `bc CR0.eq`. The true path of `bc CR0.eq` splits at the branch `bc CR1.lt` with the path ending at `b V7` corresponding to the false path of `bc CR1.lt` and the path ending at `b V9` corresponding to the true path.

- The branch conditions are evaluated prior to executing the tree. For example, the `bc CR0.eq` uses the value of `CR0.eq` that existed upon entry at the label `V1`, not the value computed by `cmp CR0,r4,r5`.

- With one exception, all input values (like `CR0.eq` in `bc CR0.eq` or `r4` and `r5` in `add r3,r4,r5`) are evaluated prior to the execution of the VLIW instruction. This is termed *parallel semantics*, i.e. all the inputs are read in parallel at the start of the VLIW instruction, and all the outputs are written in parallel at the end of the VLIW instruction.

- *Aside:* Parallel semantics are not essential to VLIW tree instructions. Sequential semantics in which each operation executes and writes its results before the next executes are also possible. However sequential semantics restrict how VLIW instructions may be formed. For example, it would not be legal to write `CR0` via the `cmp` and read it via `bc CR0.eq` in the same VLIW, as occurs in Figure 5. Sequential semantics do have an advantage. They allow large VLIW tree instructions to be easily split into smaller trees in hardware if the machine on which they are running does not have sufficient function units.

- The one exception to parallel semantics in **DAISY** is memory references. Since the compiler cannot always know if a store and load are independent, it must have some way of guaranteeing that the load receives the correct value, and that memory is updated correctly and in the proper order. One way to do this would be (1) to allow at most one store per VLIW instruction and (2) to allow no loads “after” any store in a VLIW instruction. Unfortunately, these restrictions significantly reduce the amount of instruction level parallelism achievable. Thus **DAISY** leaves it to the hardware to insure that stores are properly sequenced to memory and that loads receive the proper value, even if it is from a store in the same VLIW instruction. The hardware may introduce stalls if there is too much such aliasing, so it is still useful for the compiler to do as much disambiguation as possible. (Other optimizations are also possible from such disambiguation work.)

- Since the tree in Figure 5 has two conditional branches, there are three exits, `b V2`, `b V7`, and `b V9`. In general, there is one more exit than there are conditional branches.

- ALU and memory operations are associated with edges of the tree. In Figure 5, there are 5 edges. Whether ALU/memory operations on each edge actually execute depends on the evaluation of the conditional branches. If an operation is on an edge that matches conditional branch conditions, it is executed, otherwise not. For example, the `and r6,r5,r4` on the edge from `bc CR0.eq` to `b V2` is executed only if `CR0.eq` is not set.

- In this way, the conditional branches in a VLIW instruction act not so much as branches, but as masks indicating which set of instructions in the VLIW tree are to execute. Since the branch conditions are evaluated at the start of the VLIW instruction, the set of ALU and memory operations to be executed can be determined at the start of the instruction as well.

- As a practical matter, each VLIW tree instruction executes in one cycle, which does not leave sufficient time — at least at the frequencies we wish to achieve — to evaluate which operations should execute, and then execute them. Thus all operations in any likely implementation are executed and the mask determined by the conditional branches is used to enable write-back of those results which actually should have executed.

- As can be seen in Figure 5, each VLIW instruction explicitly identifies its successor instructions, i.e. VLIW `V2`, `V7`, or `V9`. Thus VLIW instructions branch on every cycle. There are two problems with such branching:

  – Encoding 3 (or 4) completely independent targets in a VLIW instruction is wasteful of encoding space.

  – Fetching new instructions from a new and arbitrary location every cycle is difficult, all the more so if the conditional branch conditions must be computed to determine the next VLIW instruction to fetch.

  To overcome these difficulties, the eventual **DAISY** architecture will almost certainly place restrictions, on the set of successor VLIW instructions to the current VLIW instruction. One likely restriction is that all successor instructions lie in the same Icache line. If this is the case, the fetch unit can begin fetching this cache line immediately without waiting to see how the condition codes evaluate. Then at the end of the cycle, when the condition codes have been evaluated, it can pick the proper successor instruction out of the cache line. *The* **DAISY** *translation software currently does not take into account such restrictions.* This shortcoming needs to be fixed, possibly through a "final-pass" assembler which organizes the VLIW instructions — duplicating them in some cases — in order to meet the cache line restriction.

- Note that register `r3` is written twice if the path from `V1` to `b V7` is followed. In such cases, the operation nearer the VLIW exit takes precedence. Thus, if the `V1` to `b V7` path is followed, the `nor r3,r8,r7` result takes precedence over the `add r3,r4,r5` result.

Note that VLIW tree instructions are quite similar to predicated execution, with the conditional branches in the VLIW tree instruction acting as predicates. For example, Figure 6 shows equivalent predicated code for the VLIW tree instruction in Figure 5. Two points are of particular note:

- Operations may have multiple predicates. For example the `xor` operation from the VLIW tree instruction in Figure 5 has two predicates, `<CR0.eq TRUE, CR1.lt TRUE>`, in Figure 6.

- Branches to the next VLIW tree instruction become conditional branches, again possibly with multiple conditions, for example, `bc <CR0.eq TRUE, CR1.lt TRUE>,V9` in Figure 6.

Another possible task in the **DAISY** *compiler* is to make it support an architecture with explicitly predicated operations, as in Figure 6.

There is of course more to the **DAISY** machine model than simply the instruction model. Many of these other parameters, however are more easily parameterized:

```
add   r3,r5,r5,<TRUE>
cmp   CR0,r4,r5,<TRUE>
and   r6,r5,r4,<CR0.eq FALSE>
or    r6,r5,r4,<CR0.eq TRUE>
nor   r3,r8,r7,<CR0.eq TRUE, CR1.lt FALSE>
xor   r5,r6,r9,<CR0.eq TRUE, CR1.lt TRUE>
bc    <CR0.eq FALSE>,V2
bc    <CR0.eq TRUE, CR1.lt FALSE>,V7
bc    <CR0.eq TRUE, CR1.lt TRUE>,V9
```

Figure 6. Sample Predicated Code Corresponding to VLIW Tree Instruction.

- The **DAISY** compiler currently supports machines having from 64 to 256 general purpose registers, 64 to 256 floating point registers, and 64 to 256 condition register bits (or equivalently for *PowerPC*, 16 to 64 *4-bit* condition register fields). The maximum number is specified in the file `max_resrcs.h` and the number used during translation can be set in the `daisy.config` file as described in Section 2.2. (**Caution:** *There are currently bugs if more than 64 of any one kind of registers are specified.*)

- The **DAISY** compiler currently supports machines having up to 16 ALU and/or memory operations plus 8 conditional branches per VLIW tree instruction. (Larger numbers should be possible, as these values are generally parameterized in the code, however there seems to be some bug . . ..) The exact numbers for a particular run are also set in `daisy.config`.

- Function units can be clustered, so that results computed in a cluster are available as inputs in the next cycle within that cluster, but only after a 1 cycle delay in other clusters. The eventual **DAISY** architecture will likely be clustered, and the software may need to be modified to support the precise model adopted. Much of the code for clustering is in `cluster.c`. The clustering parameters are set in `daisy.config` and may be changed for each translation.

- This version of **DAISY** does not distinguish between ALU types, and in particular does not distinguish between integer and floating point operations.

- Instruction latencies are configured at compile time, not execution time. (That is they are configured when **DAISY** is compiled, not when **DAISY** is run.) Several sets of latencies have been used, and these are controlled by `#ifdef` statements in the file `latency.h`.

- **DAISY** can model up to 3-level cache and TLB hierarchies. Specifying the `-G` flag with `daisy` enables cache simulation. When cache simulation is enabled, **DAISY** reads the cache configuration from the file `cache.config` and the TLB configuration from the file `tlb.config`. Both `cache.config` and `tlb.config` are picked up from the directory specified by `DAISY_CONFIG`

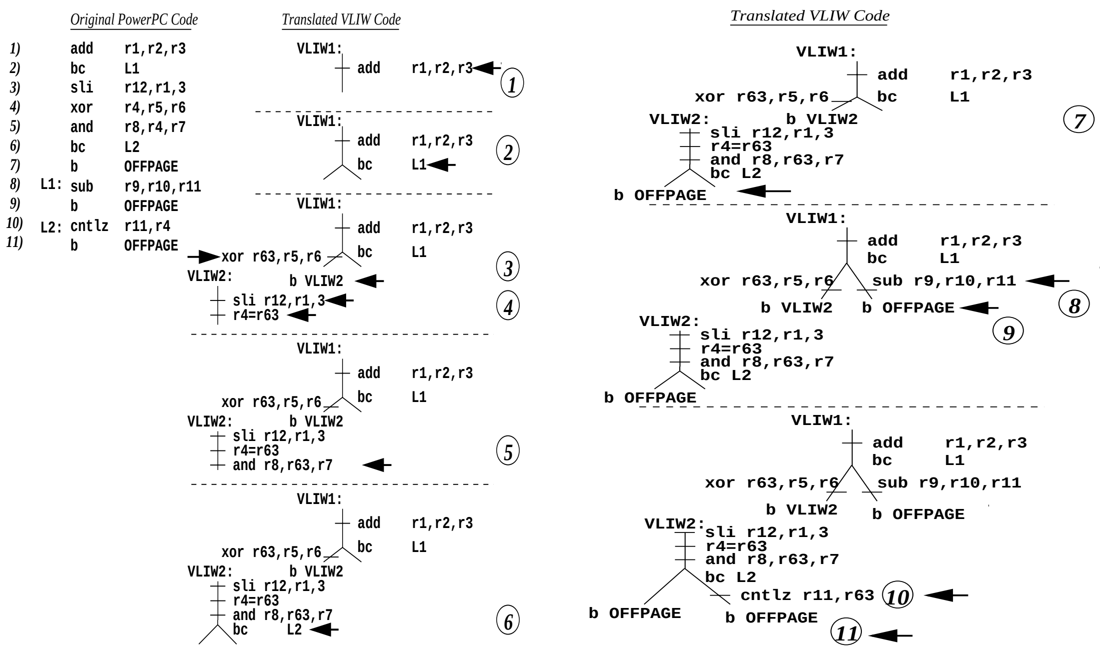


Figure 7. Example of conversion from *PowerPC* code to VLIW tree instructions.

or from the current directory, just as is the case with `daisy.config`. The cache and TLB simulation functions are the same and are mostly in the file `cache_simul.c`. The TLB's just look like an independent cache hierarchy with 4K lines. Cache statistics are written to the file `cache.out`, while TLB statistics are written to the file `tlb.out`.

4 DAISY Scheduling Algorithm

The scheduling algorithm is at the heart of **DAISY** translation from *PowerPC* code to VLIW code, as it is the basic mechanism for extracting instruction level parallelism from *PowerPC* code. The algorithm is illustrated in Figure 7. A detailed description of why each scheduling step is performed is contained in Figure 8. However, there are three major points:

- Operations 1–11 of the original *PowerPC* code are scheduled in sequence into VLIW's. It turns out that two VLIW's suffice for these 11 instructions.
- Operations are always added to the end of the last VLIW on the current path. If input data for an operation are available prior to the end of the last VLIW, then the operation is performed as early as possible with the result placed in a renamed register (that is not architected in the *PowerPC*). The renamed register is then copied to the original *PowerPC* register at the end of the last VLIW. This is illustrated by the `xor` instruction in step 4, whose result is renamed to `r63` in VLIW1, then copied to the original destination `r4` in VLIW2. By having the result available early in `r63`, later instructions can be moved up. For example, the `cntlz` in step 11 can use the result in `r63` before it has been copied to `r4`. (Note that the scheduler assumes the parallel semantics described

The conversion begins by placing an empty path whose continuation is instruction 1 (`add r1,r2,r3` as the sole entry in the `PathList`. I.e. `PathList` is initially {`[(),1,1.0]`}, where each triple in the `PathList` is of the form: {*<List of Instructions in Path>, <Continuation of Path>, <Relative Probability of Reaching the End of This Path Given That Control Has Arrived at the Entry Instruction>*}. The *current path* is defined to be the highest probability path in the `PathList`. Scheduling then proceeds as follows:

1. An empty VLIW (`VLIW1`) is created and added to the current path, and *PowerPC* instruction 1, `add r1,r2,r3`, is inserted in it. The continuation of the current path now becomes instruction 2. So `PathList` is {`[(1),2,1.0]`}.
2. *PowerPC* instruction 2, `bc L1` converts `VLIW1` from a segment to a tree, with the left branch representing the fall-through path of `bc` and right branch representing its target, `L1` (instruction 8). Note that the condition for `bc` has been computed prior to `VLIW1`, and hence the `bc` and `add` can be executed in parallel, assuming resource constraints allow it. Since instruction 2 is a branch, the current path is removed from `PathList` and two new paths are created, one whose continuation is the fall-through instruction 3, and another whose continuation is the target `L1` (instruction 8). Both new paths are placed in the `PathList`, making it {`[(1,2),3,0.7], [(1,2),8,0.3]`}. Assume the branch of instruction 2 is guessed to be taken 30% of the time.
3. Since the fall-through path is calculated to have a higher probability it becomes the current path. *PowerPC* instruction 3, `sli r12,r1,3` depends on the result of instruction 1, `add r1,r2,r3`. Hence it must go to a new VLIW. Hence `VLIW2` is created on the current path, with the fall-through tip of `VLIW1` pointing to it. The continuation of the current path is set to instruction 4. `PathList` becomes {`[(1,2,3),4,0.7], [(1,2),8,0.3]`}.
4. *PowerPC* instruction 4, `xor r4,r5,r6` does not depend on any result yet produced. Hence it can be executed in `VLIW1`. However, in order to maintain precise exceptions, we rename the result to register `r63` (which is not in the *PowerPC* architecture) and copy `r63` to `r4` in `VLIW2`. So `PathList` = {`[(1,2,3,4],5,0.7], [(1,2),8,0.3]`}. If an exception (say an external interrupt) occurred just before executing `VLIW2`, the emulated *PowerPC* machine appears to have completed instructions 1 and 2, and is at the point immediately prior to instruction 3. The results of instruction 4 are still in `r63` and are not yet committed to an architected register, at the point of the interrupt.
5. *PowerPC* instruction 5, `and r8,r4,r7` depends on the result of the `xor`. Because of our aggressive scheduling this result can be used in `VLIW2` by noting that the desired value of `r4` is in `r63`, yielding `and r8,r63,r7`. The continuation of the current path is set to 6. `PathList` becomes {`[(1,2,3,4,5),6,0.7], [(1,2),8,0.3]`}.
6. *PowerPC* instruction 6, `bc L2` has no data dependences and hence can be scheduled in `VLIW2` in a manner analogous to `bc L1` being scheduled in `VLIW1`. But we do not schedule branches earlier than the last VLIW on a path, in order to maintain precise interrupts. The current path is replaced by two paths: one continuing with the fall-through instruction 7, and one continuing with the target `L2` (instruction 10). Assume this second branch is also guessed to be taken with 30% probability. `Pathlist` becomes {`[(1,2,3,4,5,6],7,0.49(0.7`$\times$`0.7)], [(1,2),8,0.3], [(1,2,3,4,5,6),10,0.21(0.7`$\times$`0.3)]`}.
7. Of the 3 paths, the fall-through path of instruction 6, is now most likely, so it becomes the current path. Its continuation is *PowerPC* instruction 7, `b OFFPAGE`. It is placed on the left tip of `VLIW2` since branches are scheduled in order. Since this branch has no onpage continuations, this path is removed from `PathList`, and the next most probable path becomes the new current path. `Pathlist` becomes {`[(1,2),8,0.3], [(1,2,3,4,5,6),10,0.21]`}.
8. The *PowerPC* instruction 8 `sub r9,r10,r11`, the `L1` target, is the continuation of the current, highest probability path. This target continues from the right tip of `VLIW1`, since that is the location of the branch that inserted it in `PathList`. This `sub` instruction has no data dependences with earlier instructions, and hence can be scheduled on the right tip of `VLIW1`. The continuation of the current path becomes 9, and `PathList` becomes {`[(1,2,8),9,0.3], [(1,2,3,4,5,6),10,0.21]`}.
9. *PowerPC* instruction 9, `b OFFPAGE` is next on the current path. It is handled just like instruction 7, and hence a `b OFFPAGE` is placed on the right tip of `VLIW1`. This path is then removed from `PathList`, which now becomes {`[(1,2,3,4,5,6),10,0.21]`}.
10. The only open path remaining in the list is the one that continues with *PowerPC* instruction 10 `cntlz r11,r4`, the `L2` target from `VLIW2`. It is dependent on the result of instruction 4, `xor r4,r5,r6`. As noted, this value of `r4` is available in `VLIW2` itself in `r63`. Hence instruction 10 can be scheduled on the right tip of `VLIW2`. The continuation of the current path becomes 11, and `PathList` becomes {`[(1,2,3,4,5,6,10),11,0.21]`}.
11. *PowerPC* instruction 11, `b OFFPAGE` is next on this path. It is handled just like instructions 7 and 9, and hence a `b OFFPAGE` is placed on the right tip of `VLIW2`. This path is then removed from the list. As there are no more entries in the PathList, and no more entries to process, the algorithm terminates. The translated code is ready for execution beginning at `VLIW1`.

Figure 8. Description of conversion from *PowerPC* to VLIW.

in Section 3, in which all operations in a VLIW instruction read their inputs before any outputs from the current VLIW instruction are written.)

- The renaming scheme just described places results in architected *PowerPC* registers in original program order. Stores and other operations with non-renameable destinations are placed at the end of the last VLIW on the current path. In this way, precise exceptions can be maintained.

The **DAISY** code for performing this scheduling is contained largely in the file `vliw.c`, and in particular, the function **insert_op()**. **insert_op()** is passed a *PowerPC* operation to schedule and an earliest time at which that operation may be scheduled. To get this earliest time, the caller of **insert_op()** typically calls the function **get_earliest_time()**, which is also in `vliw.c`. **get_earliest_time()** returns the earliest time at which the current *PowerPC* operation may be scheduled based on data dependences with already scheduled operations.

The first VLIW in a group has time 0, the subsequent VLIW(s), time 1, etc. There may be multiple immediate successors if the fist VLIW instruction contains a conditional branch. **DAISY** does not assume that the hardware has interlocks and if it encounters a long latency operation such as `divide` and no useful work to be done between the `divide` and its use, it pads the gap with empty VLIW instructions.

insert_op() first tries to place the *PowerPC* operation at the end of the VLIW instruction at the *earliest time*. To do this, **DAISY** requires (1) that a function unit be free on which to compute the result, and (2) that a non-*PowerPC* architected register be available to place the result if the *earliest time* is earlier than the current VLIW instruction. If these function unit or register constraints do not permit placing the operation at the *earliest time*, the next time is tried, until eventually a VLIW instruction is found in which the *PowerPC* operation may be scheduled. This is guaranteed to happen at some point, because if all of the existing VLIW instructions on the path are full, **insert_op()** creates a new empty VLIW instruction and appends it to the current path. Note that this scheduling algorithm never needs to spill registers, since having a register for the result is a requirement of scheduling an operation. Of course, if the *PowerPC* operation is scheduled at the end of the current VLIW instruction or later, it immediately places result in the *PowerPC* destination register.

Although it may not be clear from this example, the scheduling algorithm produces groups of VLIW tree instructions which are themselves trees. That is the groups are trees of trees. The reason for this is in the second bullet above: namely *PowerPC* operations or their *commit* from a rename register are always scheduled in the last VLIW instruction on the current path. Even if the *PowerPC* operation has been seen before on some other path, it is replicated again at the end of the current path. Thus VLIW groups have no *join* points.

Sometimes, of course, it is essential to have a join point. Loops for example need a join at their entry, or the scheduling algorithm would unroll them forever. This is accomplished by terminating the current path and starting a new VLIW group after a particular *PowerPC* operation has been seen sufficiently many times.

The data structures used by the scheduling algorithm, in particular the representation of VLIW tree instructions is described in Section 6.

5 DAISY Optimizations

5.1 Combining

The combining optimization essentially does symbolic arithmetic, and is very useful for noting when address expressions are the same, as well as for generating induction variable values in `for` loops, without serialization. For example, a loop body may have a statement like `addi r3,r3,4`. Thus if `r3` has the value $\texttt{r3}_0$ in iteration 0, then in iteration 3, `r3` will have the value $\texttt{r3}_0 + 12$. Combining determines this and eliminates serialization on the `addi r3,r3,4` instruction. This then allows iteration 3 operations, which have no other data dependences to be scheduled immediately.

Address expressions, particularly those involving the stack also benefit from combining. The stack is often bumped at the start of a procedure, e.g. `addi r1,r1,-96`. This may be followed by a load operation, e.g. `lwz r4,8(r1)`. Combining notes that this load operation can be scheduled immediately (before the `addi`) as `lwz r4,-88(r1)`.

There are some subtle caveats on the use of combining. For example, there are some cases when wraparound from 0xFFFFFFFF to 0x00000000 makes combining impermissible. One such case involves `addic`. For example, assume that `r4` is initially 0xFFFFFFFF, and a loop has the instruction `addic r4,r4,1`. Then the first time that this instruction is executed, the `CA` bit of the XER is set since the new value wraps around to 0x00000000. However, the second execution of this instruction clears the `CA` bit since there is no carry going from 0 to 1. However, combining might rewrite this second iteration as `addic r63,r4,2`, which would incorrectly set the `CA` bit.

DAISY currently implements combining by maintaining a linked list of combined values for each GPR. Each entry in this list contains

- A GPR number,
- A time,
- A displacement,
- A pointer to the next list element.

This list is kept in reverse chronological order, and thus to find the proper base register and displacement at a particular time, the list is searched until the *time* is less than or equal to the time of interest. The desired value of the current GPR is then obtained from the sum of the *GPR number* and *displacement* in the list. This linked list is added to whenever a combinable operation is encountered. For example, `addi` and `oril rx,rx,0` (or any `copy`) are combinable operations. This list is associated with the **TIP** structure and is called `gpr_val`.

5.2 Copy Propagation

Copy propagation is essentially a special case of combining where the displacement must always be 0. However, it is useful to separate the two, because not all operations have a displacement field.

For example, the operation `xor r3,r4,r5` is not helped if we know that at a particular time `r4` is equivalent to `r63+8`. However, if we know that `r4` is equivalent to `r61`, then we can schedule the `xor` early as `xor r60,r61,r5` (assuming that `r60` is an available destination register and that `r5` maps to itself at the time the `xor` is scheduled. Copy propagation information is kept for each GPR, FPR, and CCR in a linked list akin to that used for combining. The only difference is that there is no displacement field. These lists are part of the **TIP** structure and are called `gpr_rename`, `fpr_rename`, and `ccr_rename`.

5.3 Load-Store Must Aliases

There are many cases, particularly in unoptimized code where it can be easily proven that a particular load refers to the same location as a previous store encountered on this path. For example:

```
xor    r3,r9,r10
stw    r3,8(r1)
...
lwz    r5,8(r1)
addi   r6,r5,1
```

In such cases, operations depending on the result of the `load` can be aggressively scheduled using the source of the `store` operation. In the example above, the `addi` instruction could be scheduled immediately after the `xor` as `addi r63,r3,1`, i.e. the `addi` need not serialize on the memory accesses. This is true even if there are other stores between the *must-alias* load and store. If some such store writes to the same memory location, then whatever method is used to ensure memory consistency (e.g. load-verify) will raise an exception, and the `addi` result in `r63` will be discarded. However, in the normal case, where there is no such problem store, the `addi` is computed far earlier than it was in the original code. We sometimes refer to this optimization as *load-store telescoping*.

These load-store *must-aliases* benefit significantly from combining and copy-propagation. Combining and copy-propagation make it easy to determine if address expressions are equivalent even if they do not refer to the same register. Furthermore, *load-store telescoping* can be done through multiple sets of stores and loads.

DAISY implements its *must-alias* analysis via the `stores` field of the **TIP** structure. The `stores` field is a list of all the `stores` which have occurred on this path through the group. Whenever a load is encountered, its address is compared to those in `stores` for a match. In the case of multiple matches, the latest match, of course wins.

5.4 Load-Verify

In order to get reasonable ILP, it is essential to execute load instructions as early as possible, and not just in those cases where *must-alias* can be proven. Since the translator cannot be certain when the memory location is ready, *as early as possible* in this context means when all of the register inputs to the load are ready. (There are exceptions such as `sync` operations which order memory accesses, but we will ignore those here.)

Such speculative scheduling of loads requires some means to check whether the loaded value was correct. The `load-verify` instruction is one way of performing this check. Whenever a load operation is scheduled speculatively (and moves past at least one store), a `load-verify` instruction is inserted in the original order. This `load-verify` instruction reloads the value and checks whether it matches the original value. If so, execution continues normally. If not, a `load-verify exception trap` occurs, and the **DAISY** VMM retranslates starting at the problem load. When this new translation is executed, the load executes first and gets the correct value.

If a particular `load` repeatedly incurs such traps, **DAISY** retranslates the original group without speculating the load instruction. Currently **DAISY** does this if a particular load traps 10 times. Such a retranslate policy drops the number of `load-verify traps` from hundreds of millions to hundreds in the SPEC95 benchmarks.

Memory mapped I/O locations present a particular problem for speculative loads. Such locations normally cannot be read or written speculatively because they can have real physical side effects such as displaying a character on the screen or launching a missile. (Even loads can have such side effects. For example some I/O cards have only 1 address, and in order to read multiple values from that address, sequence through the set of values read from that address.) Thus, any hardware on which speculative loads are supported must be able to recognize such I/O accesses and quash them. On *PowerPC*, this can be done via the WIMG bits, for example.

5.5 Dead Code Elimination

DAISY employs a very simple and limited form of dead code elimination. It notes whether a register has already been written on the current path through the current VLIW instruction. If so, then the previous write is dead and is deleted.

Actually there is one exception even to this simple scheme. If the previous write to the register, also wrote something else, then it generally cannot be deleted. For example, if the previous instruction writing `r3` was `addic r3,r4,1` and the current instruction is `xor r3,r9,r10`, then the `addic` cannot be deleted because the `CA` bit would not then be set in the `XER`. However, leaving two or more writes to a single GPR can be problematic for some implementations which logically `OR` the bits of each operation trying to write to a register. In this example, the `addic r3` value and the `xor r3` value would be incorrectly `OR`'ed together for such implementations.

5.6 Unification

As noted in Section 4, the scheduler forms groups of which are trees of tree instructions (at least in the most general case). Thus the same *PowerPC* operation may be represented multiple times in the same group, if it is past one or more join points in the *PowerPC* code. Such operations may be aggressively scheduled from multiple paths so that they are at the same control point in the VLIW code. For example, consider the following *PowerPC* code:

```
    nor  r7,r22,r23
    cmpi cr0,r31,0
    beq  L1
    xor  r5,r7,r8
    b    L2
L1:
    and  r9,r7,r10
L2:
    addi r3,r4,1
```

The final `addi` instruction is common to both paths through this group, and hence we may get VLIW code such as:

```
V1:
         nor  r7,r22,r23
         cmpi cr0,r31,0
         addi r63,r4,1
         addi r62,r4,1
         b    V2

V2:
         beq  VL1
   VL1:  xor  r5,r7,r8
         or   r3,r63,r63
         b V3
         and  r9,r7,r10
         or   r3,r62,r62
         b V4



V3:      b    OFFGROUP

V4:      b    OFFGROUP
```

It is unnecessary for `V1` to have two computations of `addi rx,r4,r1`. The unification optimization tracks the PowerPC address from which each scheduled instruction comes and notes whether some other path has already computed this value. If another path has computed the value, and the destination register (e.g. `r63`) is unused all along the path where it is now needed then the operations may be *unified*. In other words, the VLIW code would be

```
V1:
```

```
         nor  r7,r22,r23
         cmpi cr0,r31,0
         addi r63,r4,1
         b    V2

V2:
         beq  VL1
   VL1:  xor  r5,r7,r8
         or   r3,r63,r63
         b V3
         and  r9,r7,r10
         or   r3,r63,r63
         b V4


V3:      b    OFFGROUP

V4:      b    OFFGROUP
```

Actually, a variety of other conditions must be met in order to unify operations. For more information, look at the `unify.c` file. These restrictions sufficiently limit the number of times when unification may be employed, so as to yield a relatively small gain from its use.

5.7 Unrolling

Because of the limited time available for translation, **DAISY** does not currently do software pipelining. It instead unrolls several times so as to get good overlap among the original loop iterations. To do this, **DAISY** checks if the *PowerPC* instruction currently being translated has been seen along this path previously. If it has, then this path contains a loop in the original *PowerPC* code. After passing the loop header N times (where N may be specified in the `daisy.config` file and is typically 2), **DAISY** serializes.

That is, **DAISY** starts a new group at this loop header address. This has the effect of peeling the first N iterations from the loop, and is useful for small iteration count loops. This new group formed by **DAISY** gets M copies of the loop body and then loops back to the start of this group. M is also specified in `daisy.config` and is typically 4.

6 DAISY Data Structures

The two most fundamental data structures used by **DAISY** are in the file `vliw.h`. They are the **VLIW** `struct` defining a VLIW tree instruction and **TIP**, the `struct` which defines an edge of a VLIW tree instruction. (For a description of VLIW tree instructions, see Section 3.) There are 9 fields in the **VLIW** `struct`:

- **time** indicates how many VLIW instructions distant from the start of the group, the current VLIW instruction is. The first VLIW instruction of a group has **time** 0.
- **group** is a unique identifier indicating the VLIW group to which this group to which this instruction belongs. As noted in Section 4, VLIW groups are trees of tree instructions.
- **group_entry** is the *PowerPC* address of the entry point of the **group** to which this VLIW instruction belongs.
- **entry** points to the first edge (or **TIP**) in the current VLIW instruction. For example for the VLIW instruction in Figure 5 on page 12, the **entry** edge is the segment from `V1` to `bc CR0.eq`.
- **xlate** is the address of the simulation code for this VLIW instruction. (This value is not filled in until scheduling for a page is complete.)
- **visited** is a bit used by the simulation code generator and by the VLIW code dumper to know whether they have processed this VLIW instruction previously.
- **resrc_cnt** is used to determine the number of each type of function unit which have been used by this VLIW instruction. As noted in Section 3, this version of **DAISY** does not distinguish between floating point and integer operations. This is reflected in **resrc_cnt**, which sees only 1 type of ALU. To change this, the **ALU** `#define` earlier in `vliw.h` must be split into `#define` **ALU_INT** and `#define` **ALU_FLT**.
- **cluster** is similar to **resrc_cnt** but is used when **DAISY** is targetting an architecture with clustered function units.
- **mem_update_cnt** is used with the combining optimization.

The **TIP** `struct` is used extensively by the **DAISY** scheduler described in Section 4. Its name derives from the fact that the last edge or *tip* on the current scheduling path is where the current *PowerPC* operation or the *commit* of the current *PowerPC* operation's value to a *PowerPC* register is placed. Some of the more important fields in the **TIP** `struct` are:

- **vliw** is a back pointer to the overall VLIW. This is essential in determining whether a function unit is available to add an operation to the current tip.
- **op** is a linked list of the operations on the current tip. For example, in Figure 5 on page 12, the initial **TIP** beginning at `V1` would have an **op** field of `cmp, add`.
- **num_ops** is the number of operations in the **op** list.
- **left** and **right** are the false and true paths from a conditional jump in a VLIW tree instruction (as described in Section 3). In Figure 5, **left** would be the edge from `bc CR0.eq` to `b V2`, while **right** would be the edge from `bc CR0.eq` to `bc CR1.lt`. The terminating branch of an

edge, such as `bc CR0.eq` is associated with the edge above it. Note that unconditional onpage branches entirely disappear, with the succeeding ALU operations scheduled as though they were in a straight line with the predecessor of the unconditional branch. Leaf branches of a VLIW tree instruction have **left** set to **0**. Offpage leaf branches place the address of the offpage target in **right**.

- **gpr_vliw**, **fpr_vliw**, and **ccr_vliw** are used in allocating non-*PowerPC* registers for rename results. They are bit vectors, with one bit per register to indicate whether the register is free in the current tip. Note that for a register to be free for a result, it must not only be free in the current tip, but in all tips preceding the current tip until the tip at which the result is calculated. For example, consider Figure 7 on page 16, when scheduling the *PowerPC* `xor` operation. The current tip begins at `VLIW2`. The rename register `r63` had to be free not only in this tip, but in its predecessor, the tip from `bc L1` to `b VLIW2`.

- **gpr_rename**, **fpr_rename**, and **ccr_rename** are used to determine the register in which a value resides at a particular time. If, as with the `xor` result in Figure 7, an operation is renamed, subsequent operations have to know when and where to get the rename value and when to get the value from the *PowerPC* register. For example the `and` which uses the result of the `xor` in Figure 7 must know to obtain its input from `r63` instead of `r4` in `VLIW2`.

- **ca_rename** and **ov_rename** are similar to **gpr_rename** above except they indicate whether the `CA` and `OV` bits from the *PowerPC* `XER` register have been renamed in a GPR extender. The need for this is covered in more detail in Section 7.1.

- **stores** and **last_store** are used for tracking stores on the current path. This is essential for the *must-alias* optimization described in Section 5.3, as well as in knowing when a `load-verify` operation is needed on a uniprocessor. (See Section 5.4.)

- **avail** indicates the time at which each machine resource (not just registers) is available. This is used heavily in the scheduling algorithm described in Section 4, and roughly speaking the value returned by the bf get_earliest_time() function in `vliw.c` is the maximum **avail** time of each of an operation's inputs.

- **cluster** is a supplement to **avail**. In a machine with function unit clusters, a result is ready at its **avail** time only in its own cluster.

- **loophdr** is a bit vector to track which instructions have been seen on the current path from the entry point of this VLIW group. If an instruction has been seen previously, it means there is a loop.

- **gpr_writer** and **mem_update_cnt** are used by the combining optimization.

- **prob** is the probability that this tip executes given that the entry point of this group executed. These probabilities are established through a simplified Ball-Larus set of heuristics. These probabilities inevitably become small after a few conditional branches are encountered on the current path, thus making it difficult to know which operations to schedule — a problem alluded to in Section 1.

- **orig_ops** counts the number of *PowerPC* operations which (1) have their result placed in a *PowerPC* register on this **TIP**, or (2) are a store, or (3) are a branch. This information is used to calculate the infinite cache ILP attained, as described in Section 2.1.

- **br_condbit** specifies which of the (up to 256) condition code bits is being tested by the conditional branch terminating the **TIP**. The sense of the condition (branch if `TRUE` or branch if `FALSE`) is obtained from **br_ins**, the original *PowerPC* conditional branch.

Another set of important data structures are the opcode formats in `dis.h`. The **OPCODE_BASE** `struct` defines basic information used in parsing *PowerPC* operations such as the primary and extended opcodes. Each *PowerPC* operation also has a unique number indicated by the **op_num** field. The **op_num** values are defined in the file `dis_tbl.h`. The **page** field of **OPCODE_BASE** refers to the page number in an old 1990 manual describing the Power (not *PowerPC*) architecture. These numbers are probably of little use at this point. However, one of the carryovers of this is that the mnemonics used in `dis_tbl.h` and `dis_tbl.c` are generally Power, not *PowerPC* mnemonics. This should probably be changed at some point.

The file `dis_tbl.c` contains a list of all the Power and *PowerPC* opcodes formated to initialize the **OPCODE1** `struct` in `dis.h`. **OPCODE1** has two fields, the **OPCODE_BASE** just described and **OPERAND_IN**. The **OPERAND_IN** `struct` lists the source and destination operands of each Power and *PowerPC* operation. Operands can be either explicit in the opcode format, in which case `dis_tbl.c` used a field name such as `RT` for them, or implicit, in which case `dis_tbl.c` explicitly identifies them. For example `CA` is an implicit destination of `addic` (or `ai` for Power). The field names such as `RT` are defined in the file `opcode_fmt.h`. Fields are divided into those such as `RT` which are renameable (generally corresponding to GPR's, FPR's, and condition code fields/bits), and those which are not renameable.

Returning to the file `dis.h`, the **OPCODE2** `struct` is defined as well. **OPCODE2** is similar to **OPCODE1**, with both sharing the same **OPCODE_BASE**. However, where **OPCODE1** symbolically names some inputs and outputs, **OPCODE2** explicitly names all inputs and outputs of a parsed instruction. For example, **OPCODE1** deals with the general instruction, e.g. `and RA,RS,RB`, while **OPCODE2** deals with a specific instance of it, e.g. `and r3,r4,r5`.

These **OPCODE2** *format instructions define not only the input PowerPC instructions to* **DAISY***, but also the VLIW primitive operations.* This ties the VLIW architecture very closely (too closely) to the *PowerPC* architecture. This should be changed at some point so the VLIW architecture can define its own set of primitive ALU and memory operations.

7 Translation Idiosyncrasies

7.1 *PowerPC* Translation Idiosyncrasies

PowerPC has many branch instructions which decrement the `ctr` and branch depending on whether `ctr` is zero, possibly in conjunction with some other condition. Such branches become serializing, since they both read and set `ctr`. In a practical terms, such branches limit parallelism by requiring that no more than one loop iteration execute per cycle. To overcome this problem, it is useful to make `ctr` one of the non-*PowerPC* architected GPR's. For example **DAISY** makes the *PowerPC* `ctr` synonymous with `r32`. In this way the value in `ctr` can be explicitly decremented with the result renamed (e.g. to `r63`, then committed to `ctr/r32`. The renamed value can also be explicitly compared to 0, and `and`'ed with some other condition if need be. In programs with small tight loops, we have observed significant improvement from these actions.

In addition to branching on the `ctr` value, *PowerPC* branches can of course branch on values of condition register bits. It can further branch on the conjunction of a condition register bit values and whether the `ctr` is 0. For good, high frequency performance, such compound branches must be broken into simple primitives, which ultimately branch based solely on the value of a single bit.

For similar reasons, **DAISY** places the *PowerPC* link register in `r33`. The `MQ` register from the Power architecture is in `r34`. This should probably be changed at some point since the *PowerPC* architecture does not define the MQ register.

Although it does not do so in this version, it would be worthwhile for **DAISY** to maintain a non-*PowerPC* architected register such as `r35` with the value 0. (This convention could be enforced by software, i.e. the **DAISY** translator, not by hardware. Since the user has no access to the non-*PowerPC* registers, this can work.) Because the *PowerPC* load and store instructions, as well as `addi` follow the convention that if the `RA` field is 0, it means literal 0, not the contents of `r0`, having a VLIW register dedicated to 0 makes scheduling much easier. To see this more clearly, consider a load instruction with `RA` field *not* equal to 0. Assume this load is scheduled early with its result placed in a rename register. Because of copy propagation, at the time of this early load the value of the `RA` register may not be in the same register as indicated in the *PowerPC* operation, and may in fact be in `r0`. For example, the original *PowerPC* operation may be `lwz r3,8(r4)`. However, at the time the `lwz` is scheduled, the value of `r4` may reside in `r0`. However the instruction `lwz r3,8(r0)` does not mean load from the address `<r0+8>`, it means literal address `8`. To avoid this problem, the **DAISY** VLIW architecture ought not define `r0` to mean literal `0`. To handle those loads and `addi`'s where literal `0` is needed, `r35` could be used.

Yet another problem arises from the *PowerPC* contains `mtcrf` instruction. The `mtcrf` instruction moves any combination of 8, 4-bit condition register fields from a GPR to the condition code register. Since the VLIW architecture has more than 8 condition register fields, extending the `mtcrf` instruction must be handled. If the VLIW registers are 64 bits and the VLIW has 16, 4-bit condition register fields, then a simple extension of `mtcrf` could be done, although the instruction encoding would likely use more than 32 bits. Since, only one field is moved in many cases the **DAISY** software assumes the architecture will have an additional modified format, `mtcrf2`. The `mtcrf2` instruction has 3 operands:

1. One 4-bit condition register as the destination for the instruction.
2. A GPR instruction containing the 4-bit source field for the move.
3. An immediate value specifying which 4-bit field in the GPR is to be moved to the condition register.

The *PowerPC* condition register must be otherwise dealt with carefully, as it is addressable in 3 ways, (1) as individual bits for operations like `crnand` and conditional branches, (2) as 4-bit condition register fields, as set by `cmp` type instructions and moved by `mtcrf2` type instructions, and (3) as a full 32-bit entity, as used with the `mfcr` instruction for example. Dependences set at one level, must of course be observed at other levels. For example a branch cannot be moved above its compare.

Finally, the `CA, OV`, and `SO` bits of the `XER` register require special attention in order to attain maximum parallelism. The `addic` instruction in particular is heavily used to increment loop index variables. Alas, `addic` not only bumps the value in its destination GPR, it also sets the carry value in `CA`. Unless this `CA` value can be renamed, `addic` instructions must serialize because of the output dependence between them — even if the `CA` value computed is never used — which is the case for the vast majority of code. [2] It would be better if compilers always used the `addi` instruction instead of `addi` in such cases. However, a large body of code exists using `addic`. To get around this problem, **DAISY** places the value of `CA` in an extender bit of the target GPR for an operation such as `addic` — if the `addic` was executed speculatively and its integer result renamed to a non-*PowerPC* architected register. When the integer result is committed to its *PowerPC* architected register, the `CA` extender bit is simultaneously committed to the `CA` bit in the *PowerPC* `XER` register. If an `addic` instruction is not executed speculatively, the VLIW can place the carry value directly in the `CA` bit of the `XER`. (The architecture can tell speculative operations by their non-*PowerPC* architected destination register.) The overflow (`OV`) and summary overflow (`SO`) bits are handled similarly, except that the `OV` extender bit for a speculative operation is both placed in the `XER OV` bit as well as `or`'ed with the `SO` bit already in the `XER` register.

A similar scheme is needed for the bits of the `FPSCR` register, and the various combinations in these bits are modified by different floating point instructions. Several forms of the floating point move register instruction (`FMR`) are then needed to handle updating the appropriate combinations of bits in the `FPSCR` from the `FPSCR` extender bits which must be added to each floating point register, just as `CA` and `OV` are added to each general purpose register.

Even the condition register bits need such extenders if the instructions `fcmpo` and `fcmpu` are allowed to be renamed and executed out of order. **DAISY** currently does `fcmpo` and `fcmpu` in order.

To make **DAISY** emulate the full *PowerPC* architecture including supervisor code, it is important that the machine model be complete, or at least easily extendable. For example, dependencies on mode bits in the MSR should be modelled as well as dependences on segment registers and BAT's. WIMG bits must also be modelled for correct address translation. There are machine specific registers such as HID0

[2]The **DAISY** compiler does not have sufficient time to do a liveness analysis to determine with certainty that the value in `CA` is dead.

on the 604, which also must be modelled. In general, it is probably easiest to pick a specific *PowerPC* implementation, and strictly model it.

7.2 Other Idiosyncrasies

There are many data structures in **DAISY** which are live only during the formation of a single group. It would be useful to have a special `malloc` for such structures, with all such memory being automatically free'd after the translation of each group is complete. **DAISY** has an incomplete and awkward implementation of this special `malloc`.

Simulation code must be managed in a parallel fashion to the actual VLIW binary code. That is when VLIW code for a group is formed, the simulation code must also be formed (until we have real hardware). Likewise when a VLIW group is destroyed the simulation code must also be destroyed. Code should be written with this in mind so that VLIW code and simulation code are handled by paired function calls or some like-minded construct.

There are several shortcomings to the current **DAISY** implementation which could and should cleaned up. First, **DAISY** has no simple, clean way to get a scratch register for use by the compiler in generating intermediate values or values needed only by **DAISY** and not the original *PowerPC* code, e.g. the destination of an `LRA` instruction. Next, **DAISY** has no simple clean way to delete operations. This is particularly painful, when a *PowerPC* instruction maps to several VLIW primitives, and it is not known ahead of time if those primitives will be able to be scheduled early. (This happens, for example, on *PowerPC* `loads` going to VLIW `load` – `load-verify` operations.) If all the operations cannot be scheduled early, then they must often all be deleted. The current **DAISY** does not make this easy. Finally, the current **DAISY** assumes that the VLIW primitive operations will look very much like *PowerPC* operations, (but with more registers). This problem results from the front and back ends sharing a common opcode table. Clearly this problem should be avoided in new implementations.

8 Dumping VLIW Instructions

An assembly language dump of the generated VLIW code is generated in the file `daisy.vliw` if the user specifies the `-N` flag when running **DAISY** (`daisy`). The code to perform this dump is largely in the file `vliw_dump.c`. This code traverses each VLIW group — i.e. a tree of VLIW tree instructions — and emits the VLIW instructions. The most natural way to traverse a VLIW group is by starting with the **entry TIP** of the first VLIW instruction and following its **left** and `right` links in depth-first fashion to each exit of the group. Unfortunately this depth-first traversal does not group together the operations in a single VLIW tree instruction. Thus the dumper first traverses the group to find the individual VLIW tree instructions, and then for each such instruction emits the code.

Emitting the code for a VLIW tree instruction subdivides into emitting the code for each **TIP** of the instruction. As noted in Section 6 the ALU and memory ops are contained in the **op** field of the tip (in reverse order). The tips are glued together using the **left**, **right** and **br_condbit** fields of **TIP**. Figure 9 illustrates the output corresponding to the two VLIW instructions in Figure 7 on page 16.

There are several points to note about the dump in Figure 9:

```
V1:
T1_27738b58:                       # 0xd01b7244, Cnts Offset: 0
    cax       r1,r2,r3
    bc        12,2,T1_27738e4c     #             SKIP
T1_27738c54:                       #             Cnts Offset: 0
    xor       r63,r5,r6
    b         T1_27738d50
T1_27738e4c:                       # 0xd01b7260, Cnts Offset: 4
    sf        r9,r10,r11
    b         OFFPAGE

V2:
T1_27738d50:                       # 0xd01b7254, Cnts Offset: 8
    rlinm     r12,r1,3,0,28
    or        r4,r63,r63
    and       r8,r63,r7
    bc        4,5,T1_27739044      #             SKIP
T1_27738f48:                       # 0xd01b7258, Cnts Offset: 8
    b         OFFPAGE
T1_27739044:                       # 0xd01b7268, Cnts Offset: c
    cntlz     r11,r63
    b         OFFPAGE
```

Figure 9. Example of VLIW dump output with `-N` flag.

- Power, not *PowerPC* mnemonics are used.
- VLIW tree instructions begin at labels of the form `Vn:`, where `n` is a count that starts at 1 and increases for the entire execution of **DAISY**.
- **TIP** edges are labelled as `Tn_12345678`, where the `n` indicates which invocation of the **DAISY** translator produced this tip. As can be seen from Figure 9, this count begins at 1. To be more precise, this count is the **xlate_entry_cnt** variable bumped on each invocation of the **xlate_entry()** function discussed in Section 2.2. The `12345678` is the hex address of the **TIP** `struct` for this tip, as described in Section 6. The **TIP** `struct` memory is reclaimed after each page translation, so an address does not uniquely identify a tip. However the < *invocation count*, **TIP** *address* > pair do.
- Onpage branches and intra-VLIW conditional branches use tip labels to indicate their targets. For example, the first `bc` operation in VLIW instruction `V1` branches to the last tip in `V1`. As discussed in Section 3, this `bc` is not really a branch, but a mask to indicate which operations in `V1` should execute, `cax, xor` or `cax, sf`. Such conditional branch/mask instructions are commented with a `SKIP` annotation in the dump as can be seen in Figure 9. The `b T1_27738d50` in the middle **TIP** of `V1` is actually a branch to the next VLIW instruction `V2`.
- The hex numbers after the `#` such as `0xd01b7244` in `V1` indicate the address of the *PowerPC* branch terminating the tip. In other words, the `bc 12,2,T1_27738e4c` was at address 0xd01b7244 in the original *PowerPC* program. Note that the branch terminating the middle tip of `V1` does not have an address. This is because this branch does not correspond to any *PowerPC* branch, but instead to a branch between VLIW instructions.
- Each exit tip of a VLIW instruction has a counter associated with used for tracking the number of times that this tip executes during the execution of the translated program. The `Cnts Offset` values in Figure 9 refer to the location of the associated exit tip's count value as an offset from the start of the counts area. These `Cnts Offset` values can sometimes be useful in debugging as described in Section 10. *Caution:* For large programs like `gcc`, the counts area may not be large enough to hold a count value for all tips. In such cases the `-I` flag can be specified with `daisy` to suppress generation of such counts. (This also speeds simulation at the expense of not collecting as much many statistics on the translation.)

9 VLIW Simulation Code

Most of the simulation code resides in three files, `simul.c`, `ppc_map.c`, and `ccr-v2p.c`. The entry point for generating simulation code is **vliw_to_ppc_group()** in `simul.c`. In much the same manner as a VLIW assembly language listing is dumped, **vliw_to_ppc_group()** traverses the VLIW instructions in a *group* and generates *PowerPC* code to emulate the action of the VLIW instructions. (Section 8 describes how VLIW instructions are dumped.)

```
V1_SIM:
    cax     r31,r2,r3                   # cax   r1,r2,r3
    stw     r31,R1_SHADOW_OFFSET(r13)
    lwz     r31,CR0_OFFSET(r13)         # bc    12,2,T1_27738e4c
    mtcrf   0x80,r31
    bc      4,2,LSIM1                   # Inverted branch sense
    liu     r31,LSIM2_HIGH              # Construct target addr
    oril    r31,r31,LSIM2_LOW           # i.e. LSIM2
    mtlr    r31
    blrl                                # Branch to orig bc targ

LSIM1:
    xor     r31,r5,r6                   # xor   r63,r5,r6
    stw     r31,R63_SHADOW_OFFSET(r13)
                                        # VLIW regs: par semantics
    lwz     r1,R1_SHADOW_OFFSET(r13)    # Set r1 at end of V1
    lwz     r31,R63_SHADOW_OFFSET(r13)  # Set r63 at end of V1
    stw     r31,R63_OFFSET(r13)

    liu     r31,V2_SIM_HIGH             # Construct V2 simul addr
    oril    r31,r31,V2_SIM_LOW
    mtlr    r31
    blrl                                # Branch to V2 simul code

LSIM2:                                  # Targ of bc 12,2,T1_27738e4c
    subfc   r31,r10,r11                 # sf    r9,r10,r11
    stw     r31,R9_SHADOW_OFFSET(r13)
                                        # VLIW regs: par semantics
    lwz     r1,R1_SHADOW_OFFSET(r13)    # Set r1 at end of V1
    lwz     r9,R9_SHADOW_OFFSET(r13)    # Set r9 at end of V1

    liu     r27,0xd01b                  # Offpage target address
    oril    r27,r27,0x8180              # is 0xd01b8180 ==> r27
    lil     r26,1                       # Direct Offpage Branch=1

    liu     r31,xlate_entry_raw_HIGH    # Construct xlate_entry_raw
    oril    r31,r31,xlate_entry_raw_LOW # address
    mtlr    r31
    blrl                                # Branch to xlate_entry_raw
```

Figure 10. Generation of Simulation Code for a VLIW Tree Instruction.

Figure 10 has simulation code for VLIW instruction `V1` in Figure 9. This simulation code is generated as follows:

- The first ALU/memory operation of `V1` is `cax r1,r2,r3`. Thus the simulation code must perform this `cax` (or `add`) operation as well. Most of the VLIW registers are kept at known memory locations. However, for efficiency reasons and for ease of interfacing to other *PowerPC* code, some of the VLIW registers are actually kept in *PowerPC* registers. In particular, VLIW registers `r1-r12` are kept in real *PowerPC* registers `r1-r12`. Likewise floating point registers `fp0-??` are kept in real *PowerPC* registers. Thus the inputs for the `cax` instruction come directly from *PowerPC* registers `r1` and `r2` as shown in Figure 10.

  The destination, `r1`, is of course also kept in a real *PowerPC* `r1`. However, recall from Section 3, that VLIW instructions use parallel semantics, and hence any uses of `r1` in VLIW instruction `V1` should get the old value of `r1`, not the new value computed by `cax`. Thus the `cax` result is placed in `r31`, and then placed in a shadow version of `r1`. At the end of the VLIW instruction, this shadow version of `r1` will be copied to the real `r1`, as explained below.

  Since only VLIW integer registers `r1-r12` are kept in the corresponding *PowerPC* register, the other *PowerPC* integer registers are available for scratch computation. In particular note:

  - `r31` is used to hold the result of the `cax`.
  - `r13` is used to access the location of the shadow register for `r1`. `r13` is used throughout the simulation code to access needed values – not just registers and their shadows, but simulation counts, hash tables, and other values. This is analogous to the use of `r2` in AIX code as the `TOC` pointer.

- The next operation in `V1` after the `cax` is a `bc`. Note that the `bc` in Figure 9 reads condition register bit `2`. Thus the simulation code must first read bit `2` as well. The simulator keeps all 64-256 bits of the condition register in memory — none are kept in real *PowerPC* registers, as with some of the integer and floating point registers. The condition register bits are kept as 4-bit fields, with one 4-bit field stored in the high order 4 bits of a full 32-bit word.

  Thus the simulation code in Figure 10 first loads the 4-bit field (`CR0`) from memory into *PowerPC* register `r31`. In order to branch on bit `2`, the value of `CR0` must be placed in the real *PowerPC* condition register, thus the `mtcrf` instruction which places the VLIW `CR0` value in the *PowerPC* `CR0` field.

  It then remains to actually branch. The simulation code corresponding to the target of the branch can be arbitrarily far away. Why can the target be arbitrarily far away? Note that there may not be room in the currently allocated chunk of memory to place all of the simulation code for the current VLIW instruction. In such cases, the simulator places an unconditional branch to an arbitrary 32-bit location where the simulation code continues. The code for to do this branching is in the **dump()** function in `xlate_mem.c`.

Since the target can be arbitrarily far, the sense of the conditional branch is inverted, so that the original branch target occurs on the fall-through. In the VLIW code of Figure 9, the operation is `bc 12,2` — branch if bit `2` is on, whereas in the simulation code of Figure 10, the operation is `bc 4,2` — branch if bit `2` is off. Immediately following the simulation fall-through are 4 instructions, culminating with `blr`, to construct and branch to the 32-bit address of the simulation code for the target, `LSIM2` of the original branch.

- `LSIM1` in Figure 10 has the code for the fall-through of the VLIW branch and the target of the simulation code branch. At the fall-through of the VLIW code is the operation `xor r63,r5,r6`. Just as with the `cax` operation earlier, the `xor` result is computed into *PowerPC* `r31` and then saved in a shadow location for `r63`.

- The `xor` operation terminates the ALU/memory operations on this path through VLIW instruction `V1`. Thus the values in the shadow registers can be copied into the real registers. The next three operations in Figure 10 do this. Since, as noted above, VLIW `r1` is kept in *PowerPC* `r1`, setting the real register for `r1` consists of a single `lwz` operation from the shadow location to `r1`. VLIW `r63` is not kept in a *PowerPC* register, so its shadow is first loaded into *PowerPC* `r31`, and then stored into the memory location for VLIW `r63`.

- Once this is done the simulation code must branch to the next VLIW instruction on this path, `V2`. The full 32-bit address of the simulation code for `V2` is constructed into *PowerPC* `r31`, the moved to the *PowerPC* `LinkReg`. The simulation code for `v2` is then jumped to.

- It remains to look at the simulation code for the target of the conditional branch in `V1` and the fall-through of the simulation code that xbranch. This code is at `LSIM2` in Figure 10.

- The operation starting this edge in the VLIW code in Figure 9 is `sf r9,r10,r11`. This `sf` (or `subfc`) is handled just as were the `cax` and `xor` operations earlier, with the result being stored in the shadow location for `r9`.

- With the `subfc` is complete, the ALU/memory operations for this path through VLIW `V1` are finished. Just as on the other path, shadow values are copied to their real registers. Since `r1` and `r9` were set on this path, these registers are updated.

- The culmination of this path differs from the other path however, in that the target is not another VLIW instruction, but an `OFFPAGE` location.

- Branching offpage works as follows in Figure 10. *PowerPC* register `r27` is loaded with the *PowerPC* `OFFPAGE` address, `0xD01B8180` in this case. *PowerPC* register `r27` is loaded with the branch type (`1` for `OFFPAGE`) of this branch. Then the address of the function `xlate_entry_raw` is constructed into *PowerPC* register `r31`, which is then copied to the *PowerPC* `LinkReg`. The final `blr` goes to `xlate_entry_raw`, whose function was described in Section 2.2.

The generation of the simulation code just described is largely carried out by functions in the files `simul.c`, `ppc_map.c`, and `ccr-v2p.c`, as noted above. The routines in `simul.c` generally handle the control flow through the VLIW instruction, while the routines in `ppc_map.c` deal with the ALU and memory operations. Operations on the condition register — `cmp, fcmpo, crnand, mcrf`, etc. are dealt with in `ccr-v2p.c`.

The simulation code in Figure 10 is slightly simplified from the simulation code often generated. For example, there is no code to increment the number of times each VLIW exit tip is executed. The code to do such a bump follows the style of code in Figure 10, loading the address of the count based on an offset from `r13`, incrementing the count, and storing it back.

There is also no simulation code to call a cache simulator, trace dumper or other functions written in C. The simulation code for such functions is very similar to the code for dealing with offpage branches. Parameters such as a load address are placed in predefined registers. Then the address of the *raw* cache handler routine is constructed and placed in the *PowerPC* `LinkReg`. A `blrl` is then used to branch to the raw cache handler. Using a `branch-to-LinkReg-and-link` instruction permits the raw cache handler to return to the simulator as with ordinary function calls. The raw cache handler copies the parameters from predefined registers to AIX standard registers, such as `r3, r4, ...`, and then branches to a cache simulator written in C.

Note that the simulator uses the *PowerPC* `LinkReg` to perform indirect branches. Since this is the case, the `LinkReg` for the translated program (e.g. `/bin/ls`) must be kept elsewhere. As noted in Section 7.1, the `LinkReg` for the translated program is defined to be `r33`, while its `CtrReg` is defined to be `r32`, with the Power MQ register occupying (for now) `r34`.

The offset locations from `r13` are accessed both by assembly language and C routines. In C, they are accessed via the **r13_area** pointer. In order to keep the offsets consistent between the C and assembly language routines, the offsets are generated automatically into files `resrc_offset.h` and `r13_hdr.s`. The utility that does this is `gen_resrc` and it is created from the `gen_resrc.c` source file. **Do not manually alter the offsets in** `resrc_offset.h`**!**

10 Debugging DAISY

Debugging **DAISY** is often tedious and difficult — at least debugging problems that occur in the translated code, as opposed to the **DAISY** translator itself. For problems like `assert` failures in **DAISY** itself, your favorite symbolic debugger can be employed. For the translated code however, debugging is generally done at the assembly language level. This owes to the fact that **DAISY** takes as input a binary executable (at least in application mode) and translates it along with any shared library functions invoked (1) into VLIW code and (2) into *PowerPC* simulation code for that VLIW code. Comparison of input and output do not lend themselves to easy identification of bugs.

The most useful debugging strategy uses a binary search approach. On every crosspage and indirect branch, **DAISY** can optionally stop execution and (1) continue **DAISY**-style execution, (2) report the address of the next *PowerPC* instruction which would have been executed, or (3) execute native *PowerPC* code directly without being translated into **DAISY** VLIW code.

Binary search can be used to isolate which crosspage or indirect branch results in the manifestation of a bug. For example, suppose `/bin/ls` does not translate correctly, as may be evidenced by files not being displayed in the same way they are if `/bin/ls` is invoked directly. Then **DAISY** can be run with a specification to shift from execution of the translated VLIW version to the *PowerPC* version after **N** crosspage or indirect branches. With `/bin/ls`, an initial value of **N** $= 1000$ might be reasonable. If a correct listing of files results, then the bug must not occur in the first 1000 crosspage or indirect branches. Hence **N** must be increased, to say 2000. If the output is now incorrect, then the bug must be between crosspage or indirect branch 1000 and 2000. Following the binary search technique, **N** is now set to 1500. This process continues until the precise crosspage or indirect branch leading to an error is found. At this point the translated code for the problem page can be examined in greater detail with knowledge that the bug must lie here. Note that this is true even if there are multiple bugs. The binary search technique catches them in sequence, i.e. the first page with a bug is found (and fixed), then the second, etc.

From a user perspective, binary search is controlled by the `-L` and `-M` flags to `daisy`. The number specified immediately after `-L` (e.g. `-L1500`) specifies the number of crosspage or indirect branches before execution switches to native *PowerPC* code. The number after the `-M` flag specifies what action to take after `-L` crosspage or indirect branches have been seen. `-M2` indicates that control should switch to native *PowerPC* code as described, while `-M1` indicates that **DAISY** should stop and print out the *PowerPC* target address of the current crosspage or indirect branch. `-M1` is normally used after `-M2` has isolated the offending page to find its *PowerPC* address.

The `-L` and `-M` flags set the variables `max_dyn_page_cnt` and `max_dyn_pages_action` which are referenced in **xlate_offpage_indir()** in `simul.c`, which essentially checks if execution should stop or if *PowerPC* code should be executed directly from this point forward.

As an aside, properly switching from execution of translated VLIW code to native *PowerPC* code is a bit tricky, because *PowerPC* registers must be set in a specific order. The most difficult is `LinkReg` since `LinkReg` is used to branch to the native *PowerPC* code. If `LinkReg` is live at this point, there is a problem. A trick to overcome this problem can be found in the file `to_native.c`.

As another aside, the problem page normally lies in the first million crosspage or indirect branches. The worst case thus far encountered involved a bug translating `gcc` in which the bug did not manifest itself until more than 26 million crosspage or indirect branches, with execution of these 26 million taking almost an hour each.

Alas even when the problem page is identified, there are still difficulties. The bug may lie either in the translation from *PowerPC* to VLIW code or in the translation from VLIW code to simulation code. Bugs in the *PowerPC* to VLIW code mapping are generally easier to find. In addition the amount of VLIW code for the problem page may be large due to code explosion, making it difficult to spot the bug by inspection.

One approach in this case is to run a native version of the problem program under a debugger in one window, and the **DAISY** version in another window. Since **DAISY** does not reorder conditional branches, a correspondence between the native version and **DAISY** version may be kept. Often after a few branches, the **DAISY** version will branch one way and the native version the other, thus further isolating the bug.

Even without comparing branch by branch, the native version can be useful in isolating which VLIW instructions to examine for the bug. Often the amount of time the native version spends on the problem page (at the problem point) is short. In such cases, a trace of the native execution can be created. For example, under `dbx`, the `tracei` command can be used to do this. This trace can then be matched to a `daisy.vliw` file (as described in Section 8) to identify which VLIW instructions should execute. By identifying a small subset of VLIW instructions to examine, this technique further facilitates finding the bug by inspection.

Sometimes the translated code crashes with an illegal instruction or similar problem. In these cases, the VLIW simulation code near the crash site can be examined under a debugger, since crashes normally report the state of the registers including the `IAR`. The problem is often to find what portion of the original *PowerPC* program was running at the time of the crash, and also to find which VLIW instruction was executing (as opposed to the simulation code address reported by the debugger).

The tip execution counts discussed in Section 8 can be useful in this regard. As noted in Section 9 these counts are bumped at each exit of a VLIW instruction. These bumps occur explicitly in the simulation code including the reading the old value from the proper count offset. These offsets can be matched to the count offsets reported in the `daisy.vliw` file (see Figure 9 on page 30) to find the VLIW instruction that was executing when the program crashed. Once the VLIW instruction is found, the *PowerPC* address of the original code can be found by looking at other annotations in the `daisy.vliw` file, as discussed in Section 8, and as can be seen in Figure 9.

Another useful trick in debugging is to look at the contents of the *PowerPC* link register. As was seen in Section 9, the simulation code for **DAISY** does many of its branches indirectly through the *PowerPC* `LinkReg` in order to be able to branch to random 32-bit addresses. These branches to link register are normally `blrl`, i.e. they branch to `LinkReg` and then set `LinkReg` to the subsequent *PowerPC* operation. When the translated code crashes at some bad address like 0, the contents of `LinkReg` often indicate how the simulated code got to the bad address.

An additional caveat about debugging: Do not set a breakpoint for the translated code until that code has actually been generated. At least under `dbx` this results in illegal instruction errors. This restriction means that a breakpoint must generally be set in the **DAISY** translator itself, with the breakpoint triggering after the problem code is generated, but before it is executed.

Another point of interest about debugging **DAISY** involves segmentation violations. Segmentation violations can correctly occur in translated code because **DAISY** schedules loads speculatively. Not all speculative loads will actually turn out to be needed, and those that do not may access arbitrary addresses, and consequently cause segmentation violations. To overcome this difficulty, **DAISY** has its own segmentation violation handler (in `shm.c`). This handler ignores segmentation violations from loads, but terminates the program on violations from stores. This approach occasionally masks problem segmentation violations in the translated code or in **DAISY** itself. Note that when debugging under `dbx`, the `ignore segv` command must be used so that `dbx` does not catch and complain of segmentation violations that are actually of the acceptable type described above. (The use of `ignore segv` is seems to cause some problems under **AIX 4.1.5**, but seems fine under **AIX 4.3.2**.)

11 Post-Processing of DAISY Output

As described in Section 2.1, **DAISY** produces a variety of output files. Many of these are produced by the `dvstats` utility, which takes as input three binary files produced by `daisy`:

```
ls.vliw_perf_ins_cnts
ls.vliw_perf_ins_info
ls.vliw_spec_ins_info
```

if the input to `daisy` was `/bin/ls`. As with `daisy`, a usage summary for `dvstats` is produced by invoking it with no arguments. An abridged form of this usage summary is reproduced in Figure 11.

Figure 12 shows the first part of the `ls.histo` file produced by `dvstats`. As reported in Section 2.1, the `ls.histo` file reports the infinite cache ILP attained, along with histograms on how many operations were packed into each VLIW instruction. The ILP information is reported for the application as a whole, and for that part in the original, non-shared library portion of the application.

As also described in Section 2.1, `daisy` appends additional statistics to a file `daisy.stats` in the current directory. A sample of this information is shown in Figure 13. A note of caution: programs which invoke the `fork()` library can be translated by **DAISY**, but the statistics produced in `daisy.stats` and the files produced by **dvstats** are not accurate – only one branch of the fork is seen. Luckily few benchmark and standard utility programs that we have run invoke `fork()`.

12 Caveats

- **DAISY** (executable `daisy`) has been developed on a 604E machine. The simulation code generated may cause problems on some other variants of the *PowerPC*.
- As noted in Section 8, for large programs like `gcc`, the counts area may not be large enough to hold a count value for all tips. In such cases the `-I` flag can be specified with `daisy` to suppress generation of such counts. (This also speeds simulation at the expense of not collecting as much many statistics on the translation.)
- Whenever **DAISY** generates code, as with the **dump()** function in `xlate_mem.c`, the **flush_cache()** function in `cache.c` should be called to make sure the generated code is flushed from the *PowerPC* data cache to the *PowerPC* instruction cache.
- In order to use `daisy`, the `datasize` limit must be at least 256 Mbytes. Under *csh* or *tcsh*, this can be accomplished via the command `limit datasize 512000 kbytes` (which actually sets it to 512 Mbytes).
- In order to use `daisy`, it is important to have a large amount of paging space: at least 256 Mbytes, and preferably 400 Mbytes. The amount you currently have may be determined by typing `lsps -a`. Contact your system administrator if you need to increase your amount.

```
The "dvstats" utility is used to obtain statistics about a PowerPC
program which has been translated using "dtest".  Its usage is

         dvstats [flags] <cnts_file> [<info_dir>]

where <cnts_file> must have a ".vliw_perf_ins_cnts" suffix, and contain
counts of the number of times each VLIW instruction was executed.
(The ".vliw_perf_ins_cnts" suffix need not be specified.)  The
The <cnts_file> should be automatically produced by running "dtest".

The <info_dir> is where information about each instruction
is placed by "dtest" for use here.

Valid flags are "-b#", "-c", ,"-e", "-p", and "-s".  Dynamic opcode
frequencies are always produced.  The "-p" flag causes a ".pdf" file
to be created.  This ".pdf" file contains profile-directed-feedback
information.  The "-c" flag turns off generation of the .cnts file
(which can be very large).  The "-e" flag controls how the ".exits"
file is generated. (This file is generated for "dvstats" only).  By
default the ".exits" file is sorted by frequency of group exit.  (Each
group is a tree of VLIW tree instructions.)  With the "-e" flag, the
".exits" file is sorted by the VLIW at the entry point of a group.  In
this way it can be determined how many of the paths through the group
tree are actually exercised.  The "-b#" flag is used to specify the
degree of branch correlation used in producing the ".condbr" file.
The value must be non-negative and less than 16.

Several output files are produced from an executable file "foo":

foo.histo:      Histograms (both textual and graphical) of the
                number of operations executed per VLIW instruction.
                Useful for determining ILP achieved.

foo.exits:      A list of the paths in each VLIW group, and the
                ILP attained on each.  Useful for finding areas
                where compiler is deficient.

foo.cnts:       Lists the number of times each VLIW instruction

foo.pops1:      A list of opcode frequencies in decreasing order
                of use.  These frequencies are those of operations
                on the taken path through a VLIW instruction.

foo.sops1:      A list of opcode frequencies in decreasing order
                of use.  These frequencies are those of operations
                on all paths through a VLIW instruction.
```

Figure 11. Usage Information for `dvstats`**.**

```
1840306 VLIW      Instructions Executed over 1 run
 497645 VLIW User Instructions Executed over 1 run

3332864 Dynamic PPC Ops in Original Program with shared libs

       PPC Path Length
 1.8 = ----------------
       VLIW Path Length

 785864 Dynamic PPC Ops in Original Program -- User only

       PPC Path Length
 1.6 = ----------------
       VLIW Path Length

Specified ALU Operations Histogram:  4662313 ops ( 2.5 ops / ins)
 0:     657355 ( 35.72%   35.72% cumulative)
 1:     244518 ( 13.29%   49.01% cumulative)
 2:     133254 (  7.24%   56.25% cumulative)
 3:     184201 ( 10.01%   66.26% cumulative)
 4:     137968 (  7.50%   73.75% cumulative)
 5:     117211 (  6.37%   80.12% cumulative)
 6:     174626 (  9.49%   89.61% cumulative)
 7:     116383 (  6.32%   95.94% cumulative)
 8:      74790 (  4.06%  100.00% cumulative)
...
```

Figure 12. Start of `ls.histo` file

```
/bin/ls
--------------------------------------------------
Invocations of translator                        333
Number of pages visited                          0
Number of user pages visited                     0
Total number of entry points                     1366
Average number of entry points per page          0
# of groups which were created                   1366
# of groups which executed   > 10 times          0
# of groups which executed 5 -  9 times          0
# of groups which executed 2 -  4 times          0
# of groups which executed      1 time           0
# of direct offpage    branches executed         189778
# of indirect CTR      branches executed         37457
# of indirect LINKREG  branches executed         70240
# of indirect LINKREG2 branches executed         0
# of TOTAL  crosspage  branches executed         297475
# of hash values with single entry point         1264
Max # of entry points for any hash value         2
Hash value with Max # of entry points            484
Max # of open paths                              170
Total PPC ins (some xlated mult times)           33759
Total VLIW instructions                          1
Total size of VLIW          code                 291264
Size of VLIW code from LOAD_VERIFY fails         2420
Size of VLIW code from LOAD_VERIFY recomps       0
Number of              LOAD_VERIFY recomps       0
# Times LOAD-VERIFY instruc failed               10
# Sites LOAD-VERIFY instruc failed               2
Total size of simulation    code                 2835264
Total size of original user code                 13896
Average size of translated page of code          0
```

Figure 13. Example of `daisy.stats` **file produced by** `daisy`**.**

13 Building the DAISY Executables

There are four steps to compile the two **DAISY** executables, `daisy` and `dvstats`:

1. Go to the `daisy` directory of this distribution.
2. Type `make`.
3. Go to the `dvstats` directory of this distribution.
4. Type `make`.

14 Quick Start Running DAISY

9 steps to run **DAISY**:

1. Login to a *PowerPC*-604 machine, if available.
2. Make sure the `datasize` limit is set so that **DAISY** can run. **DAISY** requires at least 256 Mbytes. Under *csh* or *tcsh*, this can be accomplished via the command `limit datasize 512000 kbytes`, which gives 512 Mbytes.
3. Make sure there is sufficient paging space: at least 256 Mbytes, and preferably 400 Mbytes. The amount you currently have may be determined by typing `lsps -a`. Contact your system administrator if you need to increase your amount.
4. Make sure that you have the directory containing `daisy`, the **DAISY** executable on your path.
5. Set the environment variable *DAISY_CONFIG* to where the file `daisy.config` can be found. (A sample version is in the `daisy` directory of this distribution.)
6. Set the environment variable *CACHE_CONFIG* to where the files `cache.config` and `tlb.config` can be found. (Sample versions are in the `daisy` directory of this distribution.)
7. Type `daisy -V432 /bin/ls`. Within 10-20 seconds, the normal directory listing and a new AIX command prompt should appear. The -V432 tells `daisy` that it is running under AIX version 4.3.2. You should use the proper version of AIX for the machine on which you are running. This number can be determined by using the command:

   ```
   lslpp -i | grep adt | grep bos | grep lib | head -1
   ```

   DAISY has been tested under three versions of AIX: **AIX 3.2.5, AIX 4.1.5**, and **AIX 4.3.2**, although there has not been recent testing under **AIX 3.2.5** and performance there is uncertain..

8. The file `daisy.stats` should be in the current directory, and have contents similar to those in Figure 13 on page 41.

9. Type `dvstats -c ls .`. This should complete within a couple of seconds. Look at the file `ls.histo`, which lists the infinite cache ILP attained by the VLIW machine. The `ls.histo` file should be similar to the one in Figure 12 on page 40. (The `-c` flag suppresses generation of a large `ls.cnts` file.)

References

[1] Kemal Ebcioğlu. *Some Design Ideas for a VLIW Architecture for Sequential-Natured Software*. In M. Cosnard et al., editor, *Parallel Processing*, pages 3–21. North-Holland, 1988. (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing).

[2] Kemal Ebcioğlu and Erik R. Altman, **DAISY:** *Dynamic Compilation for 100% Architectural Compatibility*. Research Report RC 20538, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1996.

[3] Kemal Ebcioğlu and Erik R. Altman, **DAISY:** *Dynamic Compilation for 100% Architectural Compatibility*. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 26–37, Denver, CO, June 1997. ACM.

[4] Kemal Ebcioğlu, Jason Fritts, Stephen Kosonocky, Michael Gschwind, Erik R. Altman, Krishna K. Kailas, and Terry Bright, *An eight-issue tree-VLIW processor for dynamic binary translation*. In *Proc. of the 1998 International Conference on Computer Design (ICCD '98) – VLSI in Computers and Processors*, pages 488–495, Austin, TX, October 1998. IEEE Computer Society.

[5] Kemal Ebcioğlu, Erik R. Altman, Sumedh Sathaye, and Michael Gschwind, *Execution-Based Scheduling for VLIW Architectures*, In *Proceedings of Europar'99*, pages 1269–1280, Toulouse, France, September 1999

[6] Kemal Ebcioğlu, Erik R. Altman, Sumedh Sathaye, and Michael Gschwind, *Optimizations and Oracle Parallelism with Dynamic Translation*, In *Proceedings of Micro-32*, Haifa, Israel, November 16-18, 1999.

[7] Michael Gschwind, Kemal Ebcioğlu, Erik R. Altman, and Sumedh Sathaye *Binary Translation and Architecture Convergence Issues for IBM System/390* In *Proceedings of ICS-2000*, Santa Fe, New Mexico, May 8-10, 2000.

[8] Erik R. Altman and Kemal Ebcioglu, *Simulation and Debugging of Full System Binary Translation*, Proceedings of PDCS-2000, Track 4 - Modeling and Analysis, August 8-10, 2000, Las Vegas, NV

[9] DAISY website: **www.research.ibm.com/daisy**

[10] IBM VLIW website: **www.research.ibm.com/vliw**