**RC22957 (97495) December 15, 1999**
**Computer Science**

# IBM Research Report

## Efficient Instruction Scheduling with Precise Exceptions

**Erik R. Altman, Kemal Ebcioglu, Michael Gschwind, Sumedh Sathaye**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

**Research Division**
**Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich**

**LIMITED DISTRIBUTION NOTICE**: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g, payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home

**Abstract**

We describe the **SPACE** algorithm for translating from one architecture such as PowerPC into operations for another architecture such as VLIW, while also supporting scheduling, register allocation, and other optimizations. Our **SPACE** algorithm supports precise exceptions, but in an improvement over our previous work, eliminates the need for most hardware register **commit** operations, which are used to place values in their original program location in the original program sequence. The elimination of **commit** operations frees issue slots for other computation, a feature that is especially important for narrower machines. The **SPACE** algorithm is efficient, running in $O(N^2)$ time in the number $N$ of operations in the worst case, but in practice is closer to a two-pass $O(N)$ algorithm.

The fact that our approach provides precise exceptions with low overhead is useful to programming language designers as well — exception models in which an exception can occur at almost any instruction are not prohibitively expensive.

# 1 Introduction

Binary translation has attracted a great deal of attention of late [1, 2, 11, 13, 14], and signficant progress has been made on fast translation, as well as correct and efficient translated code. In our previous work on the **DAISY** project [3, 4, 5, 6], we have discussed a variety of techniques for quickly translating code while extracting high amounts of instruction level parallelism. Much of this work has been geared towards wide VLIW machines.

Although our techniques work on narrower machines, a significant slowdown is incurred from the fact that our previously reported approaches use valuable instruction bandwidth for **commit** operations, which move speculative values from registers not architected in the original (e.g., PowerPC) architecture to (PowerPC) architected registers. Such movement was designed to place values in PowerPC registers in original program order, thus facilitating precise PowerPC exceptions.

In this paper, we describe the **SPACE** algorithm which eliminates most of these **commit** operations, by keeping a *shadow table* of **commit** operations which are not executed, but which can be referenced when needed, such as when fielding an exception. Our new **SPACE** algorithm still supports precise exceptions, and still requires no annotations in original source programs or their compiled binaries. As well, the approach provides 100% architectural compatibility with existing PowerPC implementations. **SPACE** is also efficient, running in $O(N^2)$ time in the number $N$ of operations in the worst case, while in practice coming closer to a two-pass $O(N)$ algorithm. (This is slightly worse than our previous work which was generally a one-pass $O(N)$ algorithm in practice.) Nonetheless, it is probably still suitable for a dynamic binary translation environment in which the following two criteria must generally be obeyed:

- An interactive user should observe no erratic performance due to time spent in translation, e.g., an application initially taking a long time to respond to keyboard or mouse input.

- As a fraction of the overall runtime, the time spent in translation should be small. Alternatively, code/translation reuse should be high.

Our approach differs from existing instruction scheduling methods that work on single basic blocks or super or hyperblocks [10], in the sense that it handles speculative execution on multiple paths, and produces scheduled code that maintains precise exceptions, even though operations are aggressively re-ordered. By maintaining precise exceptions we mean the ability to indicate a point n in the original code after any exception occurs in the scheduled code, such that according to the current contents of memory and registers at the point of the exception, all instructions before point n have executed, and no instruction after point n has executed. Not re-ordering instructions when there is a possibility of an exception could certainly provide the precise exceptions capability in a simple way; our goal is to aggressively re-order code to obtain better performance, generate efficient scheduled code, and still maintain precise exceptions.

Scheduling with precise exceptions can be important not only for binary translation for 100% architectural compatibility and high performance, where the ability to compile and efficiently run all existing object code software (kernel and user code, including the exception handlers) is required, but also for traditional compilation environments. Maintaining precise exceptions could be dictated the programming language (exception ranges in C++ or Java), or could be useful for debugging scheduled code. In this paper we present a new method to generate efficient code (with reduced overhead operations over prior techniques) that maintains precise exceptions and that can be useful on both narrow and wide

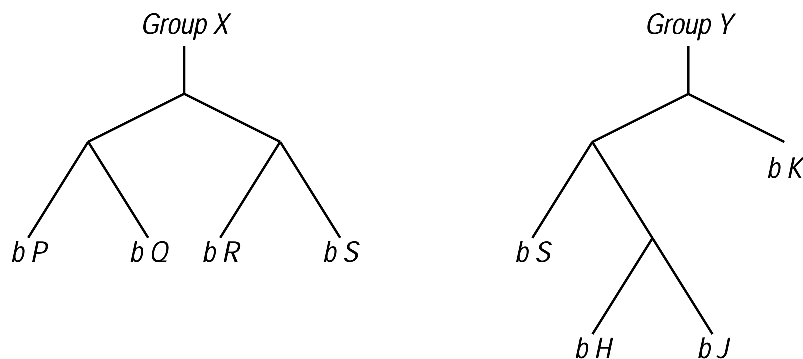



Figure 1: Tree groups **X** and **Y**.

machines . Our technique has low compile time overhead as well. The paper's presentation will be in the context of **DAISY**, our binary translation system.

**DAISY** (**D**ynamically **A**rchitected **I**nstruction **S**et from **Y**orktown) is a system to make an underlying *target* architecture 100% architecturally compatible with existing and quite different *source* architectures [3, 4, 6, 7]. Most of the DAISY work has been in the context of making an underlying VLIW machine compatible with PowerPC. However, most of the ideas are more generally applicable and could be used in support of other architectures such as *IBM System/390*, *x86*, and the *Java Virtual Machine* [3, 15]. The underlying VLIW machine is designed with binary translation in mind and provides features such as speculation support and additional registers over those present in the *source* machine.

The rest of the paper is organized as follows: Section 2 describes our **SPACE** algorithm in detail. Section 3 provides some early results using it. Section 4 discusses related work and Section 5 concludes.

# 2 SPACE Algorithm

Our algorithm for reducing the number of **commit** operations in dynamic binary translation is designed for efficient use in a system in which operations from a *base architecture*, are dynamically recompiled for execution on another *target architecture*. In this paper we focus on PowerPC as the *base architecture*, and a VLIW machine as the *target architecture*. This is in keeping with our earlier **DAISY** work [3, 4, 6].

Our **Shadow Commit Table Algorithm for Precise Exceptions** (**SPACE**) [1] assumes that a separate algorithm has grouped operations from the *base architecture* together in *tree groups* [3, 4]. *Tree groups* have no join points – any code beyond a join point is replicated on two or more paths, yielding a *tree group*, as illustrated by groups **X** and **Y** in Figure 1.

The *target* architecture (in our case, a VLIW architecture) is assumed to have significantly more registers (e.g., 64 or 128 registers) than the *base* architecture (in our case, a PowerPC architecture with 32 integer registers).

During translation from PowerPC to VLIW, several cheap optimizations are performed. Of partic-

[1] In the spirit of out of order execution, the authors have taken the liberty to not only reorder instructions, but apply similar priniciples to the reordering of letters in acronyms. . .

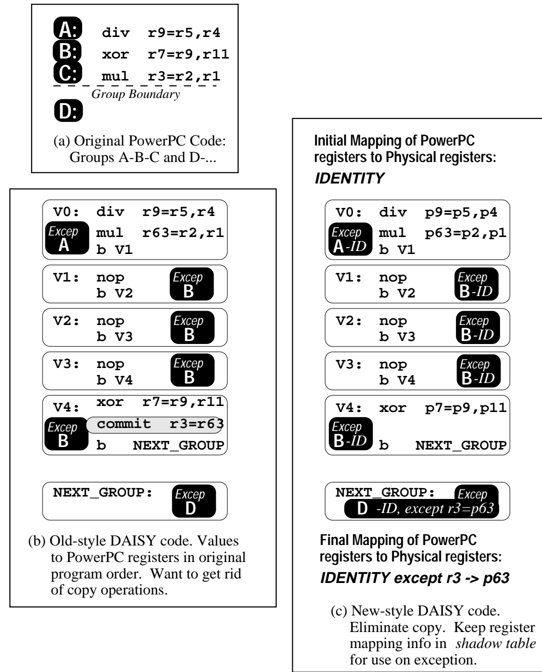



Figure 2: Example of improvement from **SPACE**.

ular interest in developing the **SPACE** algorithm is scheduling. As reported in our earlier work [3, 4], **DAISY** aggressively speculates and renames results, while still supporting precise exceptions. Previous **DAISY** work has accomplished this by reserving a set of hardware registers in the *target* architecture to hold the architected state of the *base* architecture. Updates to the *base* architected state are then performed by using hardware **commit** operations to move speculative results from registers not identified with architected PowerPC state (i.e., target architecture registers `r32` and above) to those onto which PowerPC registers have been mapped (below `r32`) in original program order.

The in-order commit approach is illustrated in Figure 2. Figure 2(a) shows a small fragment of PowerPC code comprising a *group*. Figure 2(b) shows old-style **DAISY** code for this group. The `mul` is reordered with its result placed in `r63`. At the end there is a **commit** of `r63` to `r3` so that all PowerPC registers are updated in their original program order, which in turn facilitates support of precise exceptions, as described below. However, using instruction bandwidth for these **commit** operations can reduce performance, particularly in narrower machines. The **SPACE** algorithm we propose here is a technique for removing such **commit** operations.

In Figure 2(b) — the code generated by the old **DAISY** algorithm — the indications in black boxes such as "**Excep A**" denote at what point in the original PowerPC code one could correctly restart, if an exception occurs at this point in the scheduled code. In the code generated using the **SPACE** algorithm in Figure 2(c), these indications include not only where to restart in the original code, but also what mapping to use to correctly recover the PowerPC `r` registers from the physical `p` registers in the event of an exception. Notice that the **commit** operation in Figure 2(b) has been eliminated in the code generated by the **SPACE** algorithm in Figure 2(c).

**SPACE** does so by maintaining a software *shadow table* indicating which PowerPC register is in which **physical register** at all points in the original program. In Figure 2(c), the mapping upon entry to

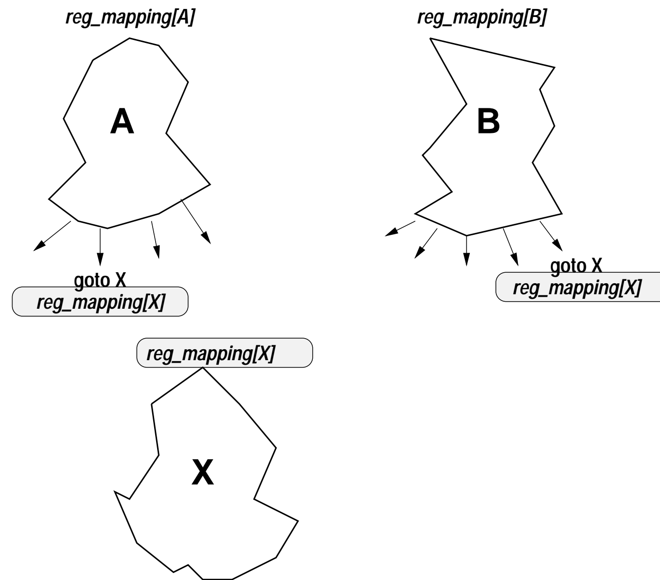



Figure 3: PowerPC to physical register mappings must be consistent at all locations.

the group is the *identity*, i.e., PowerPC `r0` maps to physical register `p0`, `r1` to `p1`, etc. However, at the end of the group, the mapping changes to map `r3` to `p63` instead of to `p3`. When the code is run, the **commit** need not be performed, and the *shadow table* is consulted only in the rare event of an exception. (All PowerPC exceptions first go to **DAISY** system code. This system code consults the *shadow tables* so as to place PowerPC register `r0` in physical register `0` (`pr0`, PowerPC register `r1` in `pr1`, etc., as expected by the VLIW translation of the PowerPC exception handler.) The fact that precise exceptions can be supported with low overhead is useful to programming language designers as well — exception models in which an exception can occur at almost any instruction are not prohibitively expensive.

There are several difficulties to implementing this *shadow table* approach efficiently as is required in a run-time compiler/translator. Most particularly, a group may have multiple predecessor groups, and if care is not taken, each predecessor group may have a different mapping of PowerPC registers to *physical registers*.

The **SPACE** algorithm takes a tree group as input and schedules operations from that group while maintaining consistent *shadow tables*. For example in Figure 3, both groups **A** and **B** have group **X** as one of their successors. Thus the mapping from PowerPC to physical registers must be the same (at least for live registers) when **A** exits to **X**, when **B** exits to **X**, and when **X** starts.

**SPACE** has 4 basic steps, as outlined below:

1. As illustrated in Figure 4, compute the destination registers for operations in group **Q** based on live PowerPC registers in successor groups such as **R**. As illustrated in Figure 5, destination registers are also based on exits from other groups, e.g., **P** that share successor groups with **Q**. This step is $O(N)$ in the number $N$ of PowerPC operations.

2. As illustrated in Figure 6, determine the mapping of PowerPC registers to physical registers at the start of the group, e.g., (**R**). This is based on the mappings in **R**'s predecessor groups such as **Q**. This step requires a hash lookup based on a PowerPC address. In the worst case, this is

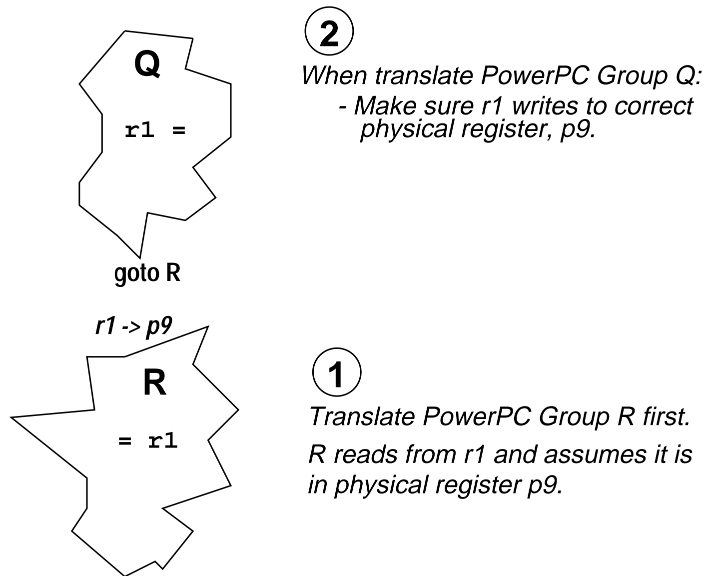



Figure 4: Ensure writes to PowerPC registers go to correct physical registers where successor groups expect them.

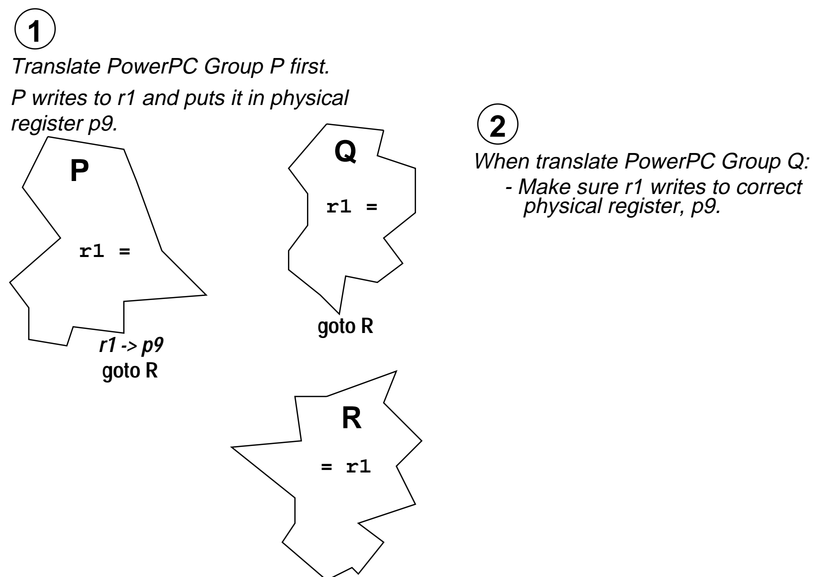



Figure 5: Ensure writes to PowerPC registers go to correct physical registers where other predecessor groups put them.

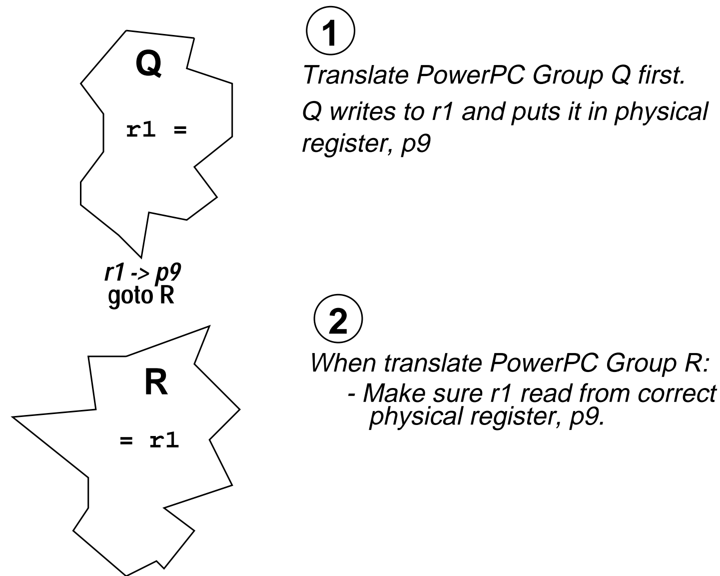



Figure 6: Ensure reads from PowerPC registers come from correct physical registers where predecessor groups put them.

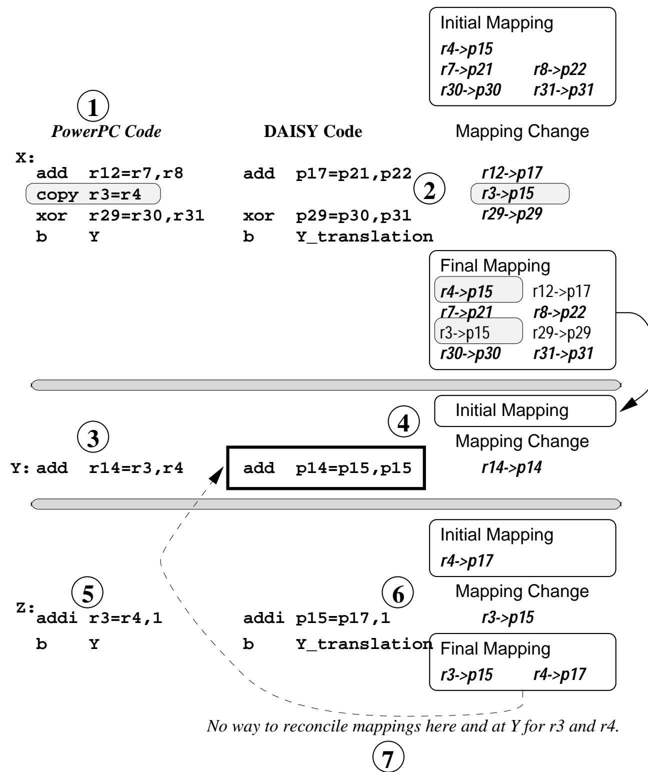



Figure 7: *"Aliasing"* between mappings of PowerPC `r3` and `r4` to physical registers.

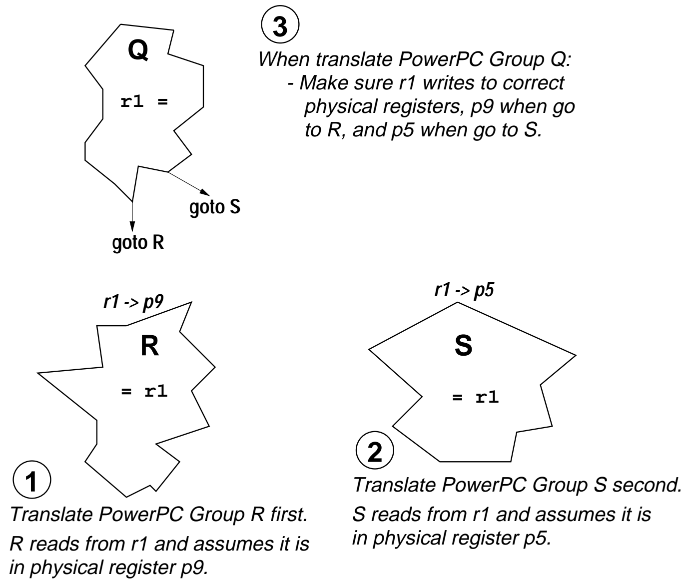



Figure 8: Different successor groups may expect a PowerPC register to be in different physical registers.

$O(M)$ in the size $M$ of the address space, although in practice, it is more likely a constant time operation.

3. Schedule all ops in group **X** using the mapping information from **Step (1)** for choosing destination registers, and from **Step (2)** for knowing where to find input registers. Like many heuristic scheduling algorithms, our approach is $O(N^2)$ in the number $N$ of PowerPC operations, since in the worst case an operation may conflict with all its predecessors. In practice, we do not see many conflicts, and hence average performance is close to $O(N)$.

4. At the end of each group, insert hardware **commit** operations to move any registers where one of two problems exist: (1) Multiple PowerPC registers map to the same physical register, thus causing potential *"aliasing"* problems as illustrated in Figure 7 and described in more detail in Section A.3. (2) The PowerPC to physical register mapping at the end of this group does not match that in a successor group. This is illustrated in Figure 8. Group **Q** has two successor groups: **R** and **S**, and `r1` is live in both, but expected to be in `p9` in **R** and in `p5` in **S**. If the **Q**'s path leading to **R** is judged to be more likely, then the **Q** initially writes the `r1` value to `p9`. A **copy** operation moving `p9` to `p5` is then required on the exit going to **S**.

   Like **Step 2**, this step requires a hash lookup based on a PowerPC address. Thus, in the worst case, it is $O(M)$ in the size $M$ of the address space, although in practice, it is more likely a constant time operation.

Figure 9 contains the entry point (`space (op)`) to a more formal description of the **SPACE** algorithm provided in pseudo notation. **Steps 1 – 4** roughly correspond with the function calls in `space`. More specifically **Step 1** is performed by the calls to `clear_bits (written)` and `calc_pref_regs (op, written)`. **Steps 2** and **3** are performed in the call to `schedule_all_ops (op)`. The actions of **Step 4** are handled by the call to `set_exit_mappings ()`.

```
/*********************************************************************
 *                                                                   *
 *                              space                                *
 *                              -----                                *
 *                                                                   *
 * Entry: Shadow Commit Table Algorithm for Precise Exceptions (SPACE).*
 *                                                                   *
 *********************************************************************/

space (op)
OP *op;         /* First PowerPC op in group                          */
{
   /* 32 PowerPC integer registers  */
   int written[32];      /* Boolean */

   /* Compute whether each operation in this group has a preferred */
   /* destination, and if so, what it is.                          */
   clear_bits (written);
   calc_pref_regs (op, written);

   /* Schedule all operations in the group, keeping in mind the    */
   /* PowerPC to physical register mappings.                       */
   schedule_all_ops (op);

   /* Make sure the register mappings at group exits match those   */
   /* expected at successor groups.  Create an expected mapping    */
   /* for the successor group address if none currently exists.    */
   set_exit_mappings ();
}
```

Figure 9: Entry point for **SPACE**.

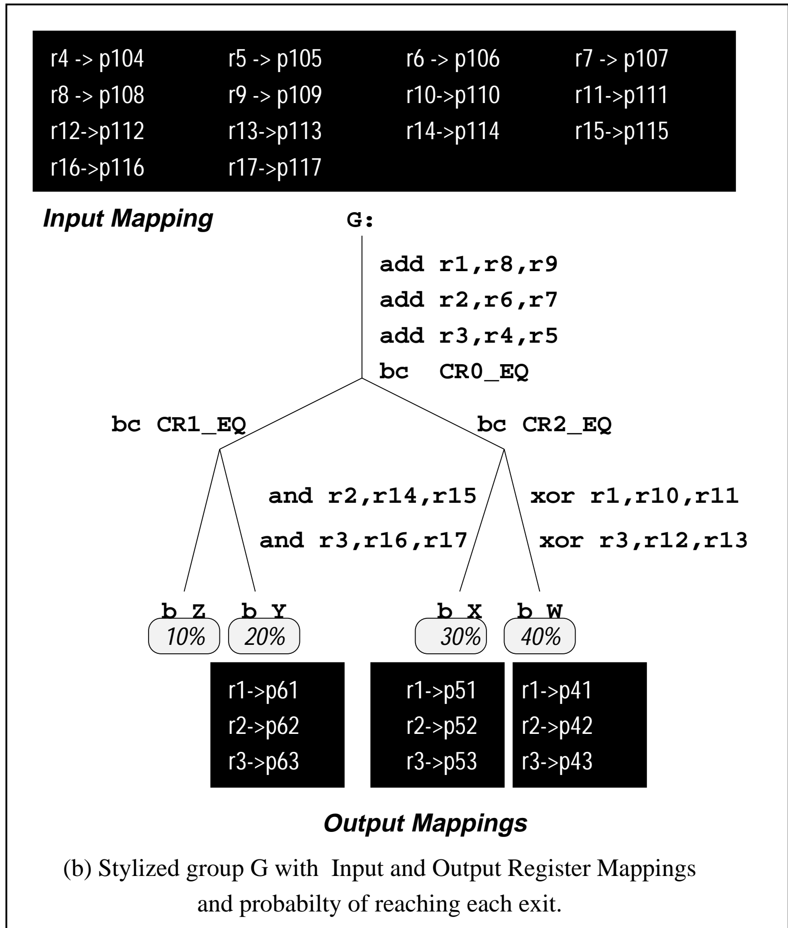
```
G:    add  r1,r8,r9
      add  r2,r6,r7
      add  r3,r4,r5
      bc   12,CR0_EQ,L_1
      bc   12,CR1_EQ,L_Y
      b    Z

L_Y:  b    Y

L_1:  bc   12,CR2_EQ,L_W
      and  r2,r14,r15
      and  r3,r16,r17
      b    X

L_W:  xor  r1,r10,r11
      xor  r3,r12,r13
      b    W
```

(a) PowerPC Code for group G



(b) Stylized group G with Input and Output Register Mappings and probabilty of reaching each exit.

Figure 10: *PowerPC* Group **G**: input to **SPACE**.

In Appendix A we discuss each of these functions in detail and show pseudo-code for each. Here, we illustrate their function via an example.

Figure 10(a) shows the PowerPC code for a tree group, **G**, that is used as input to **SPACE**. This code is shown in more stylized fashion in Figure 10(b). **G** has 4 exits at **W**, **X**, **Y**, and **Z**, with each exit having a different successor group. Consequently each of the exits has a different preferred mapping of PowerPC registers to physical registers, as also shown in Figure 10(b).

**SPACE** discovers each of these exits by recursively descending through the group from its entry at **G**. Upon reaching each exit, **SPACE** checks to see if there is a preferred mapping. In the case of **W**, **X**, and **Y**, such a mapping exits, while for **Z** there is no such mapping. (The lack of such a mapping indicates that no group has yet been translated starting from PowerPC address **Z**.)

After noting the register preferences for each path, **SPACE** begins to go back towards the start of **G** on each path by moving backwards along the same recursive route by which it reached each exit. As operations writing a result to a register are encountered, a preferred destination is noted. Thus, as noted in Figure 11(a), the operation, `xor r3,r12,r13` is marked as having `p43` as its preferred destination, since the successor group at **W** expects `r3` $\rightarrow$ `p43`. Likewise `xor r1,r10,r11` is marked as having `p41` as its preferred destination, as noted in Figure 11(a). If writes to PowerPC

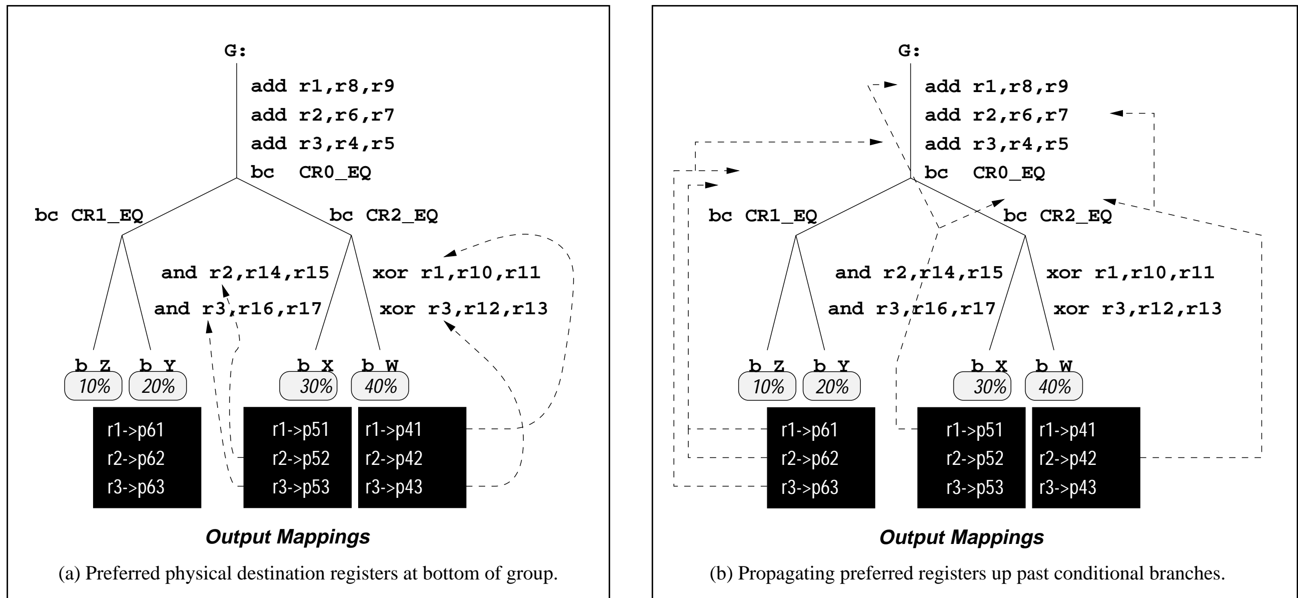



Figure 11: Propagating destination preferences up through group **G**.

registers `r1` and `r3` are encountered prior to this point in **G** (as they will be), there would be no preferred mapping for them coming from the path exiting at **W**, since such writes would be dead along the path to **W**.

Similarly, along the path exiting at **X**, Figure 11(a) shows that `p53` is the preferred destination register for `and r3,r16,r17` and `p52` is the preferred destination register for `and r2,r14,r15`.

Following the return of the recursive trail a little bit higher, the operation `bc 12, CR2_EQ,L_W` is encountered. At this point, the preferences of the two paths leading to **W** and **X** must be merged. **SPACE** does this on a register by register basis:

- PowerPC register `r1` is dead on the path to `W`, but not on the path to `X`, hence the preferred mapping (`r1` $\rightarrow$ `p51`) along the path to `X` is chosen as the preferred mapping above the `bc`, as shown in Figure 11(b).
- PowerPC register `r2` is dead on the path to `X`, but not on the path to `W`, hence the preferred mapping (`r2` $\rightarrow$ `p42`) along the path to `W` is chosen as the preferred mapping above the `bc`, as shown in Figure 11(b).
- PowerPC register `r3` is dead on both the path to `W` and **X**. Hence there is no preferred mapping for `r3`, as shown in Figure 11(b).
- All other PowerPC registers are live on both paths and hence take their preferred mapping from the path to **W**, since it is judged to have a 40% likelihood of being reached from the start of **G** versus only a 30% likelihood for the path to **X**, as shown in Figure 11(b) [2].

[2]**DAISY** interprets code 30 times before translating it and can gather statistics such as the likelihood of reaching particular exits.

As can be seen in Figure 10(b), there are no ALU operations writing to registers on paths immediately adjoining the exits at **Y** and **Z**. As a consequence, when these paths merge at `bc 12,CR1_EQ,L_Y`, the preferred mappings above the `bc` are all chosen from the path to **Y**, which is judged to have a 20% chance of being reached versus only a 10% chance for reaching **Z**. More precisely the following preferences are used `r1` → `p61`, `r2` → `p62`, and `r3` → `p63`.

The preferences from all four paths through the group are merged at the preceding `bc 12,CR0_EQ,L_1`:

- On the **W/X** path, `r1` has a preferred mapping of `r1` → `p51`, and an associated 30% probability. On the **Y/Z** path, `r1` has a preferred mapping of `r1` → `p61`, and an associated 10% probability. Thus the preference `r1` → `p51` is chosen.
- On the **W/X** path, `r2` has a preferred mapping of `r2` → `p42`, and an associated 40% probability. On the **Y/Z** path, `r2` has a preferred mapping of `r1` → `p62`, and an associated 10% probability. Thus the preference `r2` → `p42` is chosen.
- On the **W/X** path, `r3` is dead. Thus the mapping, `r3` → `p63` from the **Y/Z** path is chosen.
- All other PowerPC registers are live on both paths and hence take their preferred mapping from the more likely **W/X** path (which amounts to the **W** path in this case).

The preferred physical destination registers for the three ALU operations at the top of **G** can now be chosen:

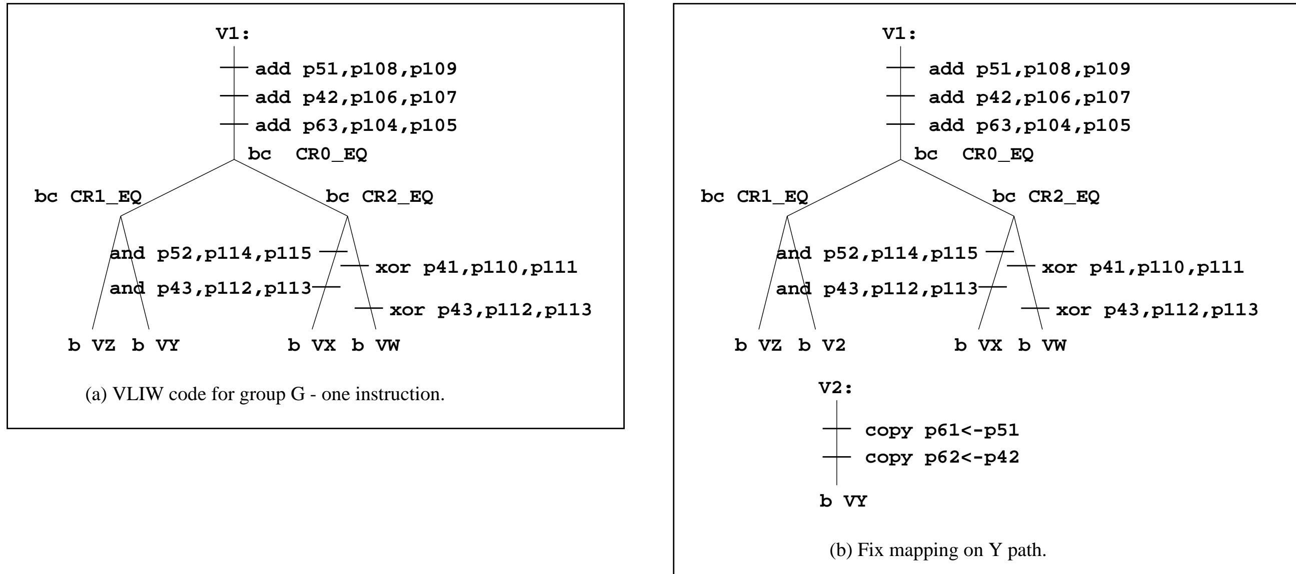
```
add   r1,r8,r9   # r1 -> r51 Preferred
add   r2,r6,r7   # r2 -> r42 Preferred
add   r3,r4,r5   # r3 -> r63 Preferred
```

The preferred mappings for input registers in group **G** can now be computed. Since none of the registers written in **G** are later read in **G**, the input mappings for all registers are those shown in Figure 10(b) at the start of **G**. (This input mapping for **G** was previously set by the output preferences of some group which branched to **G**.) If registers written in **G** were later also read in **G**, the value would of course be read from the physical register to which the value was written, i.e., from the a destination register computed in the manner illustrated above.

Once **SPACE** has computed preferred destination registers, it is ready to schedule operations. The basic **SPACE** scheduling heuristic is the same as in early incarnations of **DAISY**, namely greedily move operations as early as dependence constraints allow subject only to the availability of a function unit on which to compute the value and a destination register in which to put the result. Further details can be found in Appendix A.

Since all of the operations in **G** are independent, they can all be scheduled into a single VLIW instruction, assuming a sufficiently wide machine. This is illustrated in Figure 12(a). (Despite the superficial similarity of Figure 12 with Figures 10(b) and 11, Figure 12 depicts VLIW instructions where all operations execute in parallel, whereas Figures 10(b) and 11 are merely graphical represenations of sequential PowerPC code.)

Simply scheduling operations can leave values in the wrong physical registers, as is the case for the exit to **Y** in Figure 12(a), where the required mapping is `r1` → `p61` and `r2` → `p62`, but the actual mappings are `r1` → `p51` and `r2` → `p42`. As depicted in Figure 12(b), **SPACE** remedies this




(a) VLIW code for group G - one instruction.

(b) Fix mapping on Y path.

Figure 12: Translated VLIW code for group **G**.

problem by inserting two **copy/commit** operations on the exit path. The **copy/commit** operations require an additional VLIW instruction since they are dependent on values computed in the first VLIW instruction. (This reconciling problem is similar to rpelacing SSA $\phi$ nodes by a set of equivalent move operations [12].)

The PowerPC to physical register mappings used in translating group **G** are saved by **SPACE** in a *shadow table* for use both during exceptions (as was illustrated in Figure 2), and for use in determining preferred register mappings when translating later groups such as that starting at **Z**.

# 3 Results

In this section we show the approximate improvement possible with **SPACE**. We obtain these estimates from the current implementation of **DAISY** by using a machine model in which **commit** operations do not consume any resources, such as issues slots or ALUs. Since **commit** operations do not consume any resources, these resources are free to be used for "productive" computation, just as they would be with **SPACE**.

These estimates probably overstate slightly the performance of **SPACE**, since a few **copy/commit** operations are still needed by **SPACE** when the register mapping at the exit of one group does not match the desired mapping at the start of the next group. Since **SPACE** works to minimize the number of such **copy/commit** operations, we do not expect many of them. On the positive side, **SPACE** can eliminate **copy** operations in the original PowerPC code by making them merely an entry in the *shadow table*, a benefit not taken in account by this estimation technique.

Our results are for the SPECint95 benchmark suite and the TPC-C transaction processing benchmark, and use sampled traces as input. Our SPECint95 traces were collected on IBM RS/6000 systems based on the PowerPC architecture. Each trace consists of 50 segments of 2 million consecutive operation samples, uniformly sampled over a run of the benchmark. The TPC-C trace is slightly longer, but

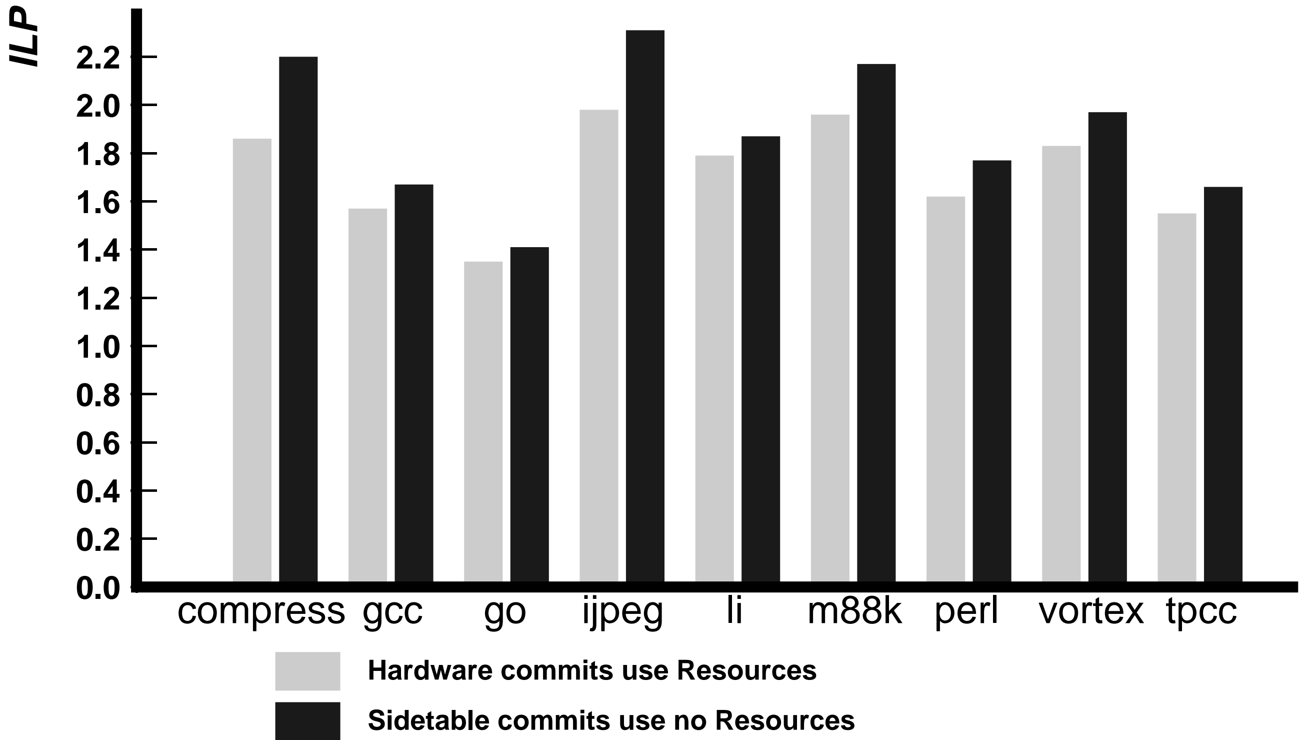



Figure 13: ILP with and without **commit** operations consuming resources.

similarly obtained. The machine that we model can issue up to 4 operations per cycle, has 4 integer ALUs, 2 load/store pipelines, and 1 floating point unit.

Figure 13 depicts the (infinite-cache) ILP attained with and without **commit** operations consuming resources. When **commit** operations do not consume resources, ILP improves by up to 18%, and by an average of 10%, thus showinging that **SPACE** has considerable potential to improve performance over the current **DAISY** without **SPACE**.

# 4 Related Work

Like **DAISY**, a number of other translation systems use renaming of registers to maintain precise exceptions. In [8], Transmeta describes an emulation system which uses a hardware backup/restore mechanism to create a checkpoint of the processor state at the beginning of each translation unit. This is combined with a gated store buffer which allows the undoing of memory store operations. Checkpointing is achieved with a single **commit** operation which copies the entire architected register file and **commits** the conditionally executed store instructions.

In [9], Le describes a binary translator which uses a software based renaming scheme. Similar to **DAISY**, target registers are renamed to support precise exceptions. Register mappings are preserved in a register map within a single translation unit, but are commited to their "natural" locations on group boundaries by compensation code. This can lead to performance degradation if excessive amounts of copy back operations need to be performed between group transitions.

While machine-to-machine binary translation with precise exceptions imposes significant requirements, many of these requirements can be relaxed when translating according to some programming language exception model. Programming languages make fewer guarantees about what state will be accessible and precise in the case of exceptions, and scoping rules (such as `catch/try` blocks in Java and C++) allow analysis of exactly what state will be inspected in the event of an exception.

For example, Moon and some of the authors of this paper [15] describe a register allocation scheme for Java that tries to match register mappings between translation units. However, that work involves register allocation alone, and does not do scheduling as is done in **SPACE**.

# 5 Conclusion

We have described the **SPACE** algorithm for scheduling operations from one architecture such as PowerPC into operations for another architecture such as VLIW. The **SPACE** algorithm supports precise exceptions, but eliminates the need for most hardware register **commit** operations, thus freeing slots for other computation, a feature that is especially important for narrower machines. Preliminary results indicate that up to an 18% improvement may be possible over the previous **DAISY** algorithm. The **SPACE** algorithm is efficient, running in $O(N^2)$ time in the number $N$ of operations in the worst case, but in practice is closer to a two-pass $O(N)$ algorithm.

# References


[1] S. Banerjia, V. Bala, E. Duesterwald, *Efficient Memory Management in a Practical Dynamic Optimizer*, Proceedings of 1999 Binary Translation Workshop, Newport Beach California.

[2] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, J. Yates, *FX!32–A Profile-Directed Binary Translator*, IEEE Micro, vol. 18, no. 2, pp. 56-64, March 1998.

[3] K. Ebcioğlu and E. Altman, **DAISY:** *Dynamic Compilation for 100% Architectural Compatibility*, Report No. RC 20538, IBM T.J. Watson Research Center.

[4] K. Ebcioğlu and E. Altman, **DAISY:** *Dynamic Compilation for 100% Architectural Compatibility*, Proc. ISCA-97, 1997.

[5] K. Ebcioğlu, J. Fritts, S. Kosonocky, M. Gschwind, E. Altman, K. Kailas, and T. Bright, *An Eight-Issue Tree VLIW Processor for Dynamic Binary Translation*, Proc. ICCD-98, Dallas, TX, 1998.

[6] K. Ebcioğlu, E. Altman, S. Sathaye, M. Gschwind, *Execution-based Scheduling for VLIW Architectures*, Proceedings of Europar'99, Toulouse, France, 1999.

[7] K. Ebcioğlu, E. Altman, S. Sathaye, M. Gschwind, *Optimizations and Oracle Parallelism with Dynamic Translation*, To appear in Proceedings of Micro-32, Haifa, Israel, 1999.

[8] E. Kelly, R. Cmelik, M. Wing, *Memory Controller for a Microprocessor for Detecting a Failure of Speculation on the Physical Nature of a Component Being Addressed* U.S. Patent **US5832205**, November 1998.

[9] B. Le, *An Out of Order Execution Technique for Runtime Binary Translators*, Proceedings of ASPLOS-VIII, San Jose, California, 1998.

[10] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann, *Effective Compiler Support for Predicated Execution Using the Hyperblock*, Proceedings of Micro-25, Portland, Oregon, 1992.

[11] C. May, *MIMIC: A Fast System/370 Simulator*, Proceedings of SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, St. Paul, MN, June 24-26, 1987, pp.1-13.

[12] R. Morgan, *Building an Optimizing Compiler*, Digital Press, 1998.

[13] G.M. Silberman and K. Ebcioğlu, *An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures*, IEEE Computer, Vol. 26, No. 6, June 1993, pp. 39-56.

[14] Sun Microsystems, *The Java Hotspot Performance Engine Architecture*, http://java.sun.com/products/hotspot/whitepaper.html, April 27, 1999.

[15] B-S Yang, S-M Moon, S. Park, J. Lee, S. Lee, J. Park, Y.K. Chung, S. Kim, Ebcioğlu, E. Altman, *LaTTe: A Java VM Just-in-TIme Compiler with Fast and Efficient Register Allocation*, Proceedings of PACT'99, Newport Beach California, 1999.

# A Details of SPACE

This appendix discusses the **SPACE** algorithm in detail with pseudo-code for each of the major functions. Reading this appendix is not necessary to understand the paper, but may be helpful in providing extra details of **SPACE**. To help keep this appendix self-contained and to aid the clarity of our exposition, we begin by repeating a small amount of the information from Section 2.

```
/*********************************************************************
 *                                                                   *
 *                             space                                 *
 *                             -----                                 *
 *                                                                   *
 * Entry: Shadow Commit Table Algorithm for Precise Exceptions (SPACE).*
 *                                                                   *
 *********************************************************************/

space (op)
OP *op;         /* First PowerPC op in group                        */
{
   /* 32 PowerPC integer registers  */
   int written[32];      /* Boolean */

   /* Compute whether each operation in this group has a preferred */
   /* destination, and if so, what it is.                          */
   clear_bits (written);
   calc_pref_regs (op, written);

   /* Schedule all operations in the group, keeping in mind the    */
   /* PowerPC to physical register mappings.                       */
   schedule_all_ops (op);

   /* Make sure the register mappings at group exits match those   */
   /* expected at successor groups.  Create an expected mapping    */
   /* for the successor group address if none currently exists.    */
   set_exit_mappings ();
}
```

Figure 14: Entry point for **SPACE**.

Figure 14 contains the entry point (`space (op)`) of the **SPACE** algorithm in pseudo-**C**. (Figure 14 is the same as Figure 9 in Section 2.) As outlined in Section 2, `space` has four basic functions:

1. Compute the destination registers for operations in a group based on live PowerPC registers in successor groups and other groups with the same successor groups as this one.

2. Determine the mapping of PowerPC registers to physical registers at the start of the group. This is based on the mappings at the exit of predecessor groups.

3. Schedule all ops in group **X** using the mapping information from **Step (1)** for choosing destination registers, and from **Step (2)** for knowing where to find input registers.

4. At the end of each group, insert hardware **commit** (or **copy**) operations to move any registers where one of two problems exist: (1) Multiple PowerPC registers map to the same physical register, thus causing potential *"aliasing"*. (2) The PowerPC to physical register mapping at the end of this group does not match that in a successor group.

**Steps 1 – 4** roughly correspond with the function calls in `space`. More specifically **Step 1** is performed by the calls to `clear_bits (written)` and `calc_pref_regs (op, written)`. **Steps 2** and **3** are performed in the call to `schedule_all_ops (op)`. The actions of **Step 4** are handled by the call to `set_exit_mappings ()`.

The following sections provide more detail on each of these functions.

## A.1 Function Calc_Pref_Regs

The primary function of `calc_pref_regs ()` in Figure 15 is to compute a preferred destination register (`op->pref_reg`) for each *op* in the current group. (*Ops* are primitives of the **DAISY** machine, and in most cases are the same as PowerPC operations, although some complex PowerPC operations such as `lwzu` – *load and update* are *cracked* into multiple simpler **DAISY** operations such as `load` followed by `addi`.)

The calculation of preferred registers by `calc_pref_regs ()` is done by recursively following all the paths through this (tree) group. At the end of any path through the group, the `pref_regs` are the `pref_regs` for the group starting immediately after the last instruction in this group, i.e., the group starting at the branch target if the current group ends in a branch, and the group starting at the successor instruction otherwise, as depicted in the function `group_end ()` in Figure 16. If no group starts immediately after this one and no other group shares successors with this group, there are no `pref_regs`.

The `pref_regs` information is then passed back up to the beginning of the group. In a group with no conditional branches, for all those registers not killed in the current group, `pref_regs` is the same at the beginning as at the end of the group.

Conditional branches have two sets of `pref_regs` information, one corresponding to the *fall-through path*, the other to the *taken path*. These two sets of `pref_regs` information are merged into one set by using the preferences of the path judged more likely to execute.

Since `calc_pref_regs ()` uses the function `and_wr_regs ()` when merging the `written` registers from two sides of a conditional branch, registers are treated as killed only if they are killed on all paths. The `merge_prefs` function (Figure 17) ensures that the preferred register assignment comes from a path in which the register is live.

Note that `calc_pref_regs (op, written)` is initially called by `space` with the first (PowerPC) `op` in group. Note also that the array of `written` PowerPC registers is zeroed by `space` prior to its call to `calc_pref_regs`.

## A.2 Function Schedule_All_Ops

Figure 18 depicts pseudo-code for the function `schedule_all_ops`. Five basic steps are performed in `schedule_all_ops (op)`:

1. Determine via the call to `get_reg_mapping` if there is an existing mapping of PowerPC to physical registers for a group beginning at `op->addr`.

```
/*********************************************************************
 *                                                                   *
 *                        calc_pref_regs                             *
 *                        --------------                             *
 *                                                                   *
 *********************************************************************/

calc_pref_regs (op)
{
   /* pref_regs = 32 PPC Regs:  Preferred PPC -> Physical Register Mapping */
   /* has_pref  = 32 PPC Regs:  Is there a preferred mapping               */
   /* pref_prob = 32 PPC Regs:  Probability take path of preferred mapping */

   if (is_condbranch (op)) {

      /* Does group end here for branch taken path? */
      if (!op->right)
           {rpref_regs, has_rpref, rpref_prob} = group_end      (op, RIGHT);
      else {rpref_regs, has_rpref, rpref_prob} = calc_pref_regs (op->right);

      /* Does group end here for branch fall-thru path? */
      if (!op->left)
           {lpref_regs, has_lpref, lpref_prob} = group_end      (op, LEFT);
      else {lpref_regs, has_lpref, lpref_prob} = calc_pref_regs (op->left)

      {pref_regs, has_pref, pref_prob} =
                             merge_prefs (rpref_regs, has_rpref, rpref_prob,
                                          lpref_regs, has_lpref, lpref_prob);
   }
   else {
      if (!op->left)
           {pref_regs, has_pref, pref_prob} = group_end      (op, LEFT);
      else {pref_regs, has_pref, pref_prob} = calc_pref_regs (op->left);

      op->has_pref = has_pref[op->dest];
      op->pref_reg = pref_regs[op->dest];

      has_pref[op->dest] = FALSE;
   }

   return {pref_regs, has_pref, pref_prob};
}
```

Figure 15: Pseudo code for `calc_pref_regs`.

```
/**********************************************************************
 *                                                                    *
 *                              group_end                             *
 *                              ---------                             *
 *                                                                    *
 * Return the preferred register mapping at the exit of the group     *
 * specified by "op".                                                 *
 *                                                                    *
 **********************************************************************/

group_end (op, dir)
{
   /* Branch target is always "right" successor */
   if (is_branch (op)  &&  dir == RIGHT) succ_addr = br_targ_addr (op);
   else                                  succ_addr = op->addr + 4;

   {pref_regs} = get_reg_mapping (succ_addr);
   {pref_prob} = get_reach_prob  (op);

   if (pref_regs) has_pref[0..31] = TRUE;
   else           has_pref[0..31]  = FALSE;

   return {pref_regs, has_pref, pref_prob};
}
```

Figure 16: Pseudo code for `group_end ()`.

2. If not, create an **IDENTITY** such mapping via the call to `create_id_map ()`.
3. Schedule all ops in the (tree) group beginning with `op`, as done by the repeated calls to `schedule_op` in the `for` loop.
4. As each exit/leaf `tip` from the group is encountered, add it to a list of such tips via the call to `add_to_leaf_tips`.
5. When scheduling of all ops is complete, create *shadow tables* for use on exceptions.

**Step 3** requires some elaboration. Operations are scheduled along a path until a conditional branch is encountered. At this point, both continuations — taken and fall-through — are added to a heap of continuations (via a call to `add_continuation`). Scheduling always resumes at the highest priority continuation as determined by calls to `get_continuation` in the `for` loop.

`Get_continuation` returns three values, `cont_tip`, `op`, and `addr`. `Op` is the first operation to schedule in the current continuation, and `addr` is the PowerPC address from which `op` came. `Cont_tip` represents the *tip* or *end* of the VLIW scheduling path. It is thus roughly equivalent to `op`, which represents the tip of the PowerPC scheduling path.

Figure 19 illustrates a series of *tip*'s that could have been generated in creating group **X** in Figure 1. Each dashed line in Figure 19 represents a *tip* at a given point during scheduling. In this case, 7 `tips` are created, corresponding to each call to `get_continuation`. Since continuations occur at conditional branches, roughly speaking, all the ALU and memory operations between conditional branches are associated with a particular tip.

The reality is slightly more complicated. The essence of a tip is as follows:

```
/**********************************************************************
 *                                                                    *
 *                             merge_prefs                            *
 *                             -----------                            *
 *                                                                    *
 * Merge the preferences from the taken and fall-thru paths of a      *
 * conditional branch.  The more likely path´s preferences win,       *
 * assuming they are live, e.g.:                                      *
 *                                                                    *
 *                 R1 Killed     R3 Killed                            *
 *                ____*____________*___________ Path W (Prob = 40%) *
 *           ___B/                                                    *
 *          /    \____*____________*___________ Path X (Prob = 30%) *
 *         /       R2 Killed     R3 Killed                            *
 * ENTRY___A/                                                         *
 *         \                                                          *
 *          \     ______________________________ Path Y (Prob = 20%) *
 *           \___C/                                                   *
 *                \______________________________ Path Z (Prob = 10%) *
 *                                                                    *
 * Thus, between the ENTRY and conditional branch A:                  *
 *     -- a write to R2 takes the preference for R2 from path W       *
 *     -- a write to R1 takes the preference for R1 from path X       *
 *     -- a write to R3 takes the preference for R3 from path Y       *
 *                                                                    *
 * Thus different registers in the same basic block can obtain their  *
 * preferred mapping from different paths.                            *
 *                                                                    *
 **********************************************************************/

merge_prefs (path1_pref_regs[32], path1_has_pref[32], path1_prob[32],
             path2_pref_regs[32], path2_has_pref[32], path2_prob[32])
{
   int    pref_regs[];
   int    has_pref[];               /* Boolean */
   double pref_prob[];

   /* 32 PowerPC integer registers */
   pref_regs = alloc (32);
   has_pref  = alloc (32);
   pref_prob = alloc (32);

   for (reg = 0; reg < 32; reg++) {

      if (path1_has_pref[reg]  &&  path2_has_pref[reg]) {
         /* Both paths have preference for "reg".  Use prefs on most likely */
         /* path to group exit on which "reg" is live.                      */
         has_pref[reg] = TRUE;
         if (path1_prob[reg] > path2_prob[reg]) {
            pref_regs[reg] = path1_pref_regs[reg];
            pref_prob[reg] = path1_pref_prob[reg];
         }
         else {
            pref_regs[reg] = path2_pref_regs[reg];
            pref_prob[reg] = path2_pref_prob[reg];
         }
      }
      else if (path1_has_pref[reg]) {
         /* Only path1 has a preference */
         has_pref[reg]  = TRUE;
         pref_regs[reg] = path1_pref_regs[reg];
         pref_prob[reg] = path1_pref_prob[reg];
      }
      else if (path2_has_pref[reg]) {
         /* Only path2 has a preference */
         has_pref[reg]  = TRUE;
         pref_regs[reg] = path2_pref_regs[reg];
         pref_prob[reg] = path2_pref_prob[reg];
      }
      else has_pref[reg] = FALSE;
   }

   return {pref_regs, has_pref, pref_prob};
}
```

Figure 17: Pseudo code for `merge_prefs`.

```
/********************************************************************
 *                                                                  *
 *                        schedule_all_ops                          *
 *                        ----------------                          *
 *                                                                  *
 ********************************************************************/

schedule_all_ops (op)
OP *op;
{
   initial_tip = make_null_tip ();

   /* Find out how registers are set by any predecessors of this group */
   initial_tip->h = get_reg_mapping (op->addr);

   /* If there are no predecessors, associate the IDENTITY mapping with  */
   /* the start of this group.  If any predecessors are later scheduled, */
   /* their outputs must adhere to this IDENTITY mapping.  The IDENTITY  */
   /* mapping puts PowerPC R0 in Physical Register 0, PowerPC R1 in P1,  */
   /* etc.                                                               */
   if (!reg_map) initial_tip->h = create_id_map (op->addr);

   add_continuation (initial_tip, op, op->addr, 1.0);

   for ( {cont_tip, op, addr}  = get_continuation ();
         (cont_tip, op, addr} != {0, 0, 0};
         {cont_tip, op, addr}  = get_continuation ()) {

      last_tip = schedule_op (cont_tip, op, addr, prob);
      if (last_tip) add_to_leaf_tips (last_tip);
   }

   build_shadow_table (initial_tip);
}
```

Figure 18: Pseudo code for `schedule_all_ops`.

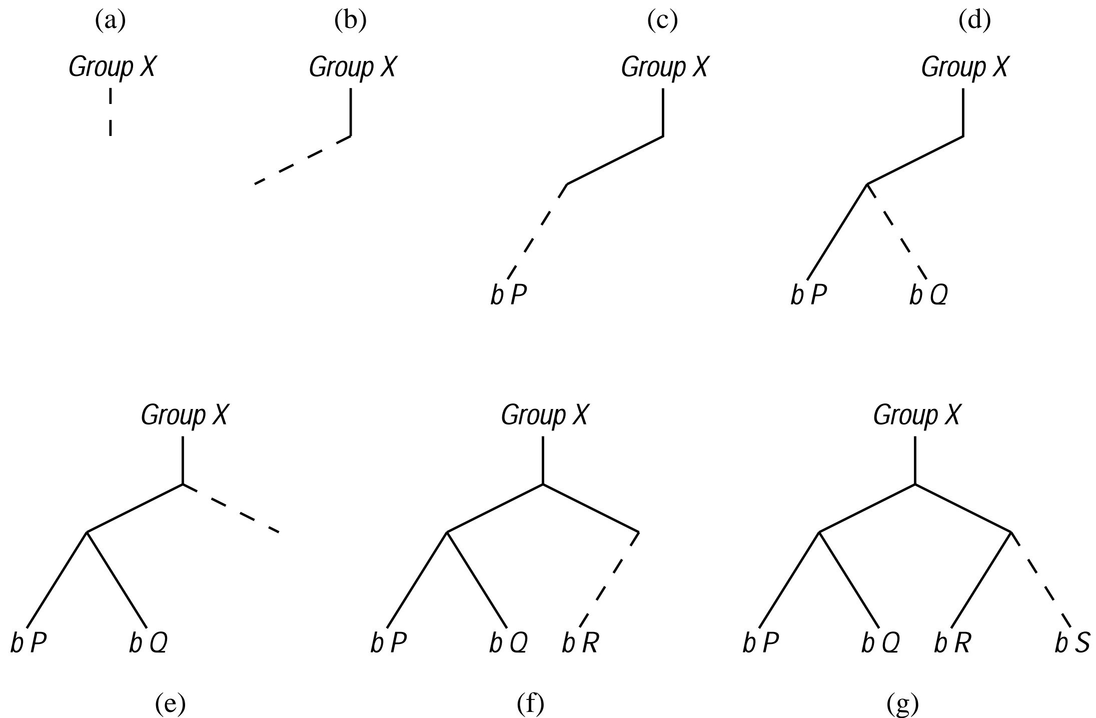



Figure 19: *Tips* in group construction.

```
typedef struct {
   TIP  *prev_tip;      /* Previous  tip in tree of tips         */
   TIP  *left_tip;      /* Fall-thru tip in tree of tips         */
   TIP  *right_tip;     /* Target    tip in tree of tips         */
   OP   *op_list;       /* List of ops associated with this tip  */
   VLIW *vliw;          /* VLIW instruction to which tip belongs */
} TIP;
```

The `prev_tip`, `left_tip`, and `right_tip` fields connect `tips` together to form a tree group. The `op_list` field tracks those operations to be performed on this `tip`, and the `vliw` field is used to track information associated with the VLIW instruction of which `tip` is a part. For example, `vliw` tracks the time (number of VLIW instructions) since the start of the group. `Vliw` also tracks the total number of ALU operations performed in this instruction, so as to ensure that it does not exceed resource limits.

Following this point, it is not actually true that all the ALU and memory operations between conditional branches are associated with a particular tip. Resource constraints and data dependences may not allow all ALU and memory operations between conditional branches to execute simultaneously on the same `tip` / in the same VLIW instruction. In such cases, `tip_m` may be ended and another `tip_n` begun. In this case

```
   tip_m->left  = tip_n;      /* Successor / Fallthru tip is n      */
   tip_n->prev  = tip_m;      /* Predecessor          tip is m      */
   tip_m->right = NULL;       /* Successor / Target   tip not exist */
```

```
/*********************************************************************
 *                                                                   *
 *                          schedule_op                              *
 *                          -----------                              *
 *                                                                   *
 * RETURNS:  Last "tip" on path if the last op scheduled is the last *
 *           on this path through the group, otherwise 0.            *
 *                                                                   *
 *********************************************************************/

schedule_op (tip, op, addr, prob)
OP       *op;
unsigned addr;
double   prob;
{
   if (!op) {   tip->succ_addr = addr;      return tip;   }

   if      (is_store  (op)) earliest = tip->time;
   else if (is_branch (op)) earliest = tip->time;
   else                     earliest = 0;

   forall (op inputs)
      if (tip->avail[input] > earliest) earliest = tip->avail[input];


   for (time = earliest; time <= tip->time; time++)
      if (schedule_op_at_time (tip, op, time)) break;

   /* Did we fail in scheduling op? */
   if (time == tip->time+1) {
      if (earliest > tip->time+1) max_time = earliest;
      else                        max_time = tip->time+1;

      /* Add VLIW instructions until we are at the desired time for op */
      /* When a new VLIW is added, all physical registers currently in */
      /* use, stay in use.  At the "tip" of the path, there should     */
      /* always be 32 registers in use, one for each PowerPC register. */
      for (; time <= max_time; time++)
         tip = make_new_vliw (tip);

      /* o All function units available in new VLIW.                 */
      /* o All PowerPC architected regs consume phys reg in new VLIW, */
      /*   but nothing else ==> If 64 total regs, 32 should be free.  */
      assert (schedule_op_at_time (tip, op, time));
   }

   /* The path bifurcates at a conditional branch.  Duplicate from
      "tip" to "target_tip", all the scheduling information such as
      "avail" times.  Pass on the same scheduling information from
      "tip" to "fallthru_tip".  (Passing the information on is
      efficient, as there is no need to duplicate all the fields and
      then free them for "tip".
   */
   if (!is_condbranch (op))
      return schedule_op (tip, op->left, op->addr + 4, prob);
   else {
      target_tip         = duplicate_tip (tip);
      fallthru_tip       = inherit_tip (tip);
      tip->left          = fallthru_tip;
      tip->right         = target_tip;
      target_tip->prev   = tip;
      fallthru_tip->prev = tip;

      ptake = prob * op->prob_taken;

      add_continuation (target_tip,   op->right, br_targ_addr (op), ptake);
      add_continuation (fallthru_tip, op->left,  op->addr + 4,  1.0-ptake);

      return 0;
   }
}
```

Figure 20: Pseudo code for `schedule_op`.

Likewise, speculatively scheduled operations are assigned to earlier tips, which are easily reached from the end of the current path by following the `prev` field of each tip until a tip is reached, whose VLIW instruction has the time at which we wish to speculatively schedule the operation.

```
/*********************************************************************
 *                                                                   *
 *                        schedule_op_at_time                        *
 *                        -------------------                        *
 *                                                                   *
 * RETURNS:  Non-zero if successfully scheduled, zero otherwise.     *
 *                                                                   *
 *********************************************************************/

schedule_op_at_time (tip, op, time)
{
   if (!is_fu_avail       (tip, time, op))            return 0;
   if (!is_phys_reg_avail (tip, time, op, &phys_reg)) return 0;
   else {
      tip->h[op->dest] = phys_reg;
      mark_phys_reg_used (tip, phys_reg, time);
      mark_fu_used (tip, time, op);
      return 1;
   }
}
```

Figure 21: Pseudo code for `schedule_op_at_time`.

All of this is put to direct use in the `schedule_op` function which is illustrated in Figure 20 and which is called from `schedule_all_ops`.

`Schedule_op` initially determines the *earliest* time at which `op` may execute based on the availability of its inputs. Store and branch instructions are exceptions and are always scheduled in order, i.e., at the last or current `tip` on the path. Function unit and register constraints are checked at each VLIW instruction on the path from the *earliest* until the current VLIW via a call to `schedule_op_at_time`, which is depicted in Figure 21. If `op` cannot be scheduled at any of these times, new VLIW instructions / tips are appended to the current path until the `op` can be scheduled. Multiple (empty) VLIW instructions may need to be appended if `op` must wait for a long latency operation to finish [3].

At the end *tip* of any path, only PowerPC registers are live. Since the VLIW machine has more registers than PowerPC, there is always a physical register for the result of `op`. Likewise, when a new VLIW instruction is added at the end *tip* of the path it is always empty of instructions, and hence there is guaranteed to be a function unit available on which to execute `op`.

Returning to `schedule_op_at_time` in Figure 21, it can be seen that the register and function unit checks just described are performed. Of particular interest is the call to `is_phys_reg_avail`, which is depicted in Figure 22.

`Is_phys_reg_avail` first determines all physical registers which are available for use along the entire path from where `op` is scheduled until the end *tip* of the path. It does so with by **OR**'ing the bit vector of registers in use at each point (*tip*) along the path.

Then via a call to `is_preferred_phys_reg`, `is_phys_reg_avail` checks if there is a preferred physical register for this destination, as was calculated in Section A.1. As can be seen in Figure 23, this is a trivial check after the work of Section A.1.

[3]Depending on the actual implementation of the VLIW machine, empty VLIWs may be eliminated during a *final assembly pass*

```
/*******************************************************************
 *                                                                 *
 *                         is_phys_reg_avail                       *
 *                         -----------------                       *
 *                                                                 *
 * RETURNS: Non-zero if a physical register ("phys_reg") is available, *
 *          zero otherwise.                                        *
 *                                                                 *
 *******************************************************************/

is_phys_reg_avail (tip, time, op, phys_reg)
int time;            /* Earliest time to check -- Typically tip->time */
int *phys_reg;       /* Output */
{
   int        pref_reg;
   BIT_VECTOR regs_used[];

   clr_allbits (regs_used);

   /* After this loop, all 0 bits in "regs_used" represent free phys regs */
   tip = end_tip;
   while (TRUE) {
      or_bits (regs_used, tip->reg_usage);
      if  (tip->time <= time) break;
      else tip = tip->prev;
   }

   if (is_preferred_phys_reg (op, &pref_reg)) {
      if (is_bit_clr (regs_used, pref_reg)) {
         /* There is a preferred register, and it is free.              */
         *phys_reg = pref_reg;
         return 1;
      }
      else if (*phys_reg = get_first_zero_bit (regs_used) >= 0) {
         /* There is a preferred register, it is not free, but another   */
         /* physical register is free.                                   */
         /*                                                              */
         return 1;
      }
         /* There is a preferred register, it is not free, and no other  */
         /* physical register is free.                                   */
      else return 0;
   }
         /* There is no preferred reg, and a physical register is free.  */
   else if (*phys_reg = get_first_zero_bit (regs_used) >= 0) return 1;

         /* There is no preferred reg, but no physical register is free. */
   else                                                    return 0;
}
```

Figure 22: Pseudo code for `is_phys_reg_avail`.

```
/**********************************************************************
 *                                                                    *
 *                        is_preferred_phys_reg                       *
 *                        ---------------------                       *
 *                                                                    *
 * RETURNS: Non-zero if preferred reg ("pref_reg") exists, 0 otherwise *
 *                                                                    *
 **********************************************************************/

is_preferred_phys_reg (op, pref_reg)
OP  *op;              /* PowerPC op */
int *pref_reg;        /* Output     */
{
   if (!op->has_pref) return 0;
   else {
      *pref_reg = op->pref_regs[op->dest_reg];
      return 1;
   }
}
```

Figure 23: Pseudo code for `is_preferred_phys_reg`.

```
/**********************************************************************
 *                                                                    *
 *                        set_exit_mappings                           *
 *                        -----------------                           *
 *                                                                    *
 **********************************************************************/

set_exit_mappings ()

  forall (leaf_tips) {
    hw_regmap = get_reg_mapping (leaf_tip->succ_addr);

    if (!hw_regmap) hw_regmap = create_reg_map (leaf_tip->succ_addr, leaf_tip->h);

    make_pref_reg_assignments (leaf_tip, hw_regmap)
  }
}
```

Figure 24: Pseudo code for `set_exit_mappings`.

If `is_preferred_phys_reg` indicates that there is a preferred register (`pref_reg`), `is_phys_reg_avai` uses it, if it is available over the whole time range. If `pref_reg` is not available, `is_phys_reg_avail` chooses an arbitrary physical register. If none are available, failure is indicated. If `is_preferred_phys_reg` indicated that there was no `pref_reg`, an arbitrary choice is also tried with failure indicated if no physical registers are available.

## A.3 Function Set_Exit_Mappings

Recall from the start of Section 2 that `set_exit_mappings ()` ensures that register mappings at group exits match those expected at successor groups, and that it creates an expected mapping for the successor group address if none currently exists.

Pseudo code for `set_exit_mappings` is given in Figure 24.

`Set_exit_mappings` iterates through the set of `leaf_tips` or (*exit tips of the group*). The set of `leaf_tips` is set in `schedule_all_ops` in Figure 18. At each `leaf_tip`, a check is made via a call to `get_reg_mapping`, as to whether a register map exists for a successor group of this exit.

If there is no successor to this exit and no other group exiting to the same successor, `create_reg_map` associates the current register mapping with the successor address of this tip. Actually, there is an exception to using the current register mapping, as is outlined by the code for `create_reg_map` in Figure 25. If multiple PowerPC registers e.g., **R3** and **R4**, both map to the same physical register, e.g., **P9** only one of **R3** and R4 is assigned to **P9**. Such as situation could arise in a group if there is a PowerPC operation which copies **R3** to **R4**. No **DAISY** code is generated for this PowerPC **copy**, only **DAISY**'s internal mapping tables are updated.

Whichever of **R3** or **R4** is not assigned to **P9** is assigned to some free physical register, e.g., P17. A **DAISY copy** operation from **P9** to **P17** will be generated during the call to `make_pref_reg_assignments`. We will return to `make_pref_reg_assignments` in more detail shortly.

This special handling of multiple PowerPC registers mapping to the same physical register is done in case another path to this successor group is later encountered in which **R3** and **R4** are not mapped to the same physical register. In this circumstance if **R3** and **R4** mapped to the same physical register, there would be no way to make the register assignment for the later path compatible with this path.

More generally, if a successor group is later scheduled starting or ending at this *PowerPC successor address*, their register mapping must adhere to the mapping set here. Since we employ this algorithm at runtime, there is a reasonable chance that this exit will be the primary predecessor to any successor group that is eventually created. Even if it is not the primary predecessor, any other predecessors will be created with this mapping in mind, thus helping ensure that performance is good in all cases.

If a mapping already exists at this PowerPC successor address, then hardware **copy** operations are added via the call to `make_pref_reg_assignments` to make sure the register mapping matches (1) the successor group and (2) any other group exits which branch to this successor group. As noted above, the call to `make_pref_reg_assignments` also generates **copy** operations if the mapping at the exit to this group does not match the required `hw_regmap`, as when two PowerPC registers would otherwise be mapped to the same physical register at group exit.

`Make_pref_reg_assignments` is depicted in Figure 26.

`Make_pref_reg_assignments` determines which physical registers need to be moved to new physical registers in order to make the mapping at the end/*tip* of one group consistent with the mapping for the successor group. Only 32 physical registers are in use at each *tip* — one for each PowerPC register. Those not in use, can of course be ignored. We have endeavored when creating the group ending at *tip* to place values in physical registers consistent with where the successor group(s) want them. Thus a common case is like that for PowerPC register `R4` in Figure 26, i.e., `R4` maps to physical register `P62` at both the *tip* and successor group, and hence no copies need be generated.

More generally however, physical registers can be mapped differently at a *tip* and successor group. A simple version of this case is shown in Figure 26 for PowerPC `R0`, which maps to `P9` at the *tip* and to `P7` in the successor group. In this case a simple **copy** operation is generated to move `P9` to `P7` prior to

```
/******************************************************************
 *                                                                *
 *                        create_reg_map                          *
 *                        --------------                          *
 *                                                                *
 * Associate a register mapping with "ppc_ins_addr" and return it.   *
 * Match "preferred_map", unless "preferred_map" maps more than one  *
 * PowerPC register to the same physical register -- make sure all  *
 * PowerPC registers are mapped to different physical registers.     *
 *                                                                *
 ******************************************************************/

create_reg_map (ppc_ins_addr, preferred_map)
{
   /* 32 PowerPC integer registers */
   rtn_map = map[ppc_ins_addr] = alloc (32);
   unassigned_ppc_reg          = alloc (32);

   phys_reg_used = alloc_and_clear (NUM_PHYS_REGS);

   /* Where possible, assign PPC regs to phys regs according to preference */
   unassigned_cnt = 0;
   for (ppc_reg = 0; ppc_reg < 32; ppc_reg++) {
      phys_reg = preferred_map[ppc_reg];
      if (phys_reg_cnt[phys_reg]++ == 0) {
         rtn_map[ppc_reg] = phys_reg;
         phys_reg_used[phys_reg] = TRUE;
      }
      else unassigned_ppc_reg[unassigned_cnt++] = ppc_reg;
   }

   /* Handle cases where multiple PPC regs mapped to same physical register */
   for (i = 0; i < unassigned_cnt; i++) {
      ppc_reg  = unassigned_ppc_reg[i];
      phys_reg = get_free_phys_reg (phys_reg_used);
      rtn_map[ppc_reg]        = phys_reg;
      phys_reg_used[phys_reg] = TRUE;
   }

   free_mem (phys_reg_used);
   free_mem (phys_reg_used);

   return rtn_map;
}
```

Figure 25: Pseudo code for `create_reg_map`.

```
/*******************************************************************
 *                                                                 *
 *                    make_pref_reg_assignments                    *
 *                    -------------------------                    *
 *                                                                 *
 * Put COPY ops at end of group to reconcile its PowerPC to physical *
 * register mapping with the mapping for the successor group, e.g.: *
 *                                                                 *
 *        Mapping at                                               *
 * Group End     Group Start    Chains/Cycles of Physical Reg Moves *
 * ---------     -----------                                       *
 * R0 -> P9      R0 -> P7       P9  -> P7               => One COPY op *
 * R1 -> P54     R1 -> P43 --+                                     *
 * R2 -> P43     R2 -> P17   |- P54 -> P43 -> P17 -> P54           *
 * R3 -> P17     R3 -> P54 --+                                     *
 * R4 -> P62     R4 -> P62      P62 -> P62              => No  COPY op *
 * ---   ----    ---   ----                                        *
 * PPC   Phys    PPC   Phys                                        *
 *                                                                 *
 * P54 -> P43 -> P17 -> P54 => COPY P54     -> Scratch             *
 *                             COPY P17     -> P54                 *
 *                             COPY P43     -> P17                 *
 *                             COPY Scratch -> P43                 *
 *******************************************************************/

make_pref_reg_assignments (tip, pref_regs)
int pref_regs[32];     /* 32 PPC integer regs:  Prefs at tip successor */
{
   tip_to_succ[0..NUM_PHYS_REGS-1] = NO_MAPPING;
   succ_to_tip[0..NUM_PHYS_REGS-1] = NO_MAPPING;

   for (ppc_reg = 0; ppc_reg < 32; ppc_reg++) {
      curr_phys = tip->h[ppc_reg];
      new_phys  = pref_regs[ppc_reg];

      tip_to_succ[curr_phys] = new_phys;
      succ_to_tip[curr_phys] = curr_phys;
   }

   seen[0..NUM_PHYS_REGS-1] = FALSE;
   for (phys_reg = 0; phys_reg < NUM_PHYS_REGS; phys_reg++) {

      if (seen[phys_reg]) continue;

      pred_reg = succ_to_tip[phys_reg];

      if (pred_reg == phys_reg)   continue;
      if (pred_reg == NO_MAPPING) continue;

      first_reg = find_chain (succ_to_tip, pred_reg, &is_cycle);
      dump_copies (tip, succ_to_tip, tip_to_succ, seen, first_reg, is_cycle);
   }
}
```

Figure 26: Pseudo code for `make_pref_reg_assignments`.

exiting the group at *tip*. More complicated sequences can arise, however as illustrated by the mappings for PowerPC registers R1, R2 and R3. As illustrated in Figure 26, the updates needed to keep their physical registers consistent form a cycle: P54 → P43 → P17 → P54. This cycle can be broken by first copying P54 to a scratch register – many of which are available since only 32 physical registers are in use by the translated code. Then as illustrated in Figure 26, a sequence of **copy** operations can be placed on the *tip* so as to obtain the mapping required at the start of the successor group.

Most of the work of make_pref_reg_assignments goes into finding these *chains* and *cycles* of physical register mappings. The find_chain function in Figure 27 searches for the head of a chain or a cycle, and also determines whether a group of updates is cyclic. When all of this is finally determined, dump_copies in Figure 27 is invoked. Dump_copies finds the end of a chain or cycle, so as not to corrupt the values, and then generates a sequence of **copy** operations on the *tip* to update the mappings, in a manner similar to that illustrated in Figure 26. Make_pref_reg_assignments and its subroutine calls run in time linear with the number of registers.

```
/*******************************************************************
 *                                                                 *
 *                         find_chain                              *
 *                         ----------                              *
 *******************************************************************/

int find_chain (pred_map, base_reg, is_cycle)
boolean *is_cycle;       /* Output: True for sequence like X->Y->Z->X */
{

   /* Search backward until find first register in chain of copies,    */
   /* or until we determine there is a cycle.                           */
   curr_reg = base_reg;
   while (TRUE) {
      prev_reg = pred_map[curr_reg];

      if (prev_reg == NO_MAPPING) {
         *is_cycle = FALSE;     /* X=curr_reg starts chain of copies:  */
         return curr_reg;       /*    X->Y->Z->nothing.                 */
      }

      if (prev_reg == base_reg) {
         *is_cycle = TRUE;
         return curr_reg;       /* Have a cycle of copies:  X->Y->Z->X */
      }

      curr_reg = prev_reg;
   }
}

/*******************************************************************
 *                                                                 *
 *                        dump_copies                              *
 *                        -----------                              *
 *******************************************************************/

dump_copies (tip, pred_map, succ_map, seen, first_reg_in_chain, is_cycle)
{
   if (is_cycle) {
      curr = pred_map[first_reg_in_chain];
      tip = gen_copy_op (tip, curr, scratch_reg);
   }
   else {
      curr = first_reg_in_chain;
      while (TRUE) {                           /* Find end of chain */
         next = succ_map[curr];
         if (next == NO_MAPPING) break;
         else curr = next;
      }
   }

   /* Make COPY´s from end of chain/cycle, e.g. W->X->Y->Z  ==>        */
   /*     COPY Y->Z, COPY X->Y, COPY W->X                               */
   while (TRUE) {
      seen[curr] = TRUE;
      prev = pred_map[curr];
      tip = gen_copy_op (tip, prev, curr);

      if (prev == first_reg_in_chain) break;
      else curr = prev;
   }

   seen[prev] = TRUE;
   if (is_cycle) tip = gen_copy_op (tip, scratch_reg, prev);
}
```

Figure 27: Pseudo code for `find_chain` and `dump_copies`.