RC 20538 (08/05/96)
Computer Science

IBM Research Report

DAISY: Dynamic Compilation for 100% Architectural Compatibility

Kemal Ebcioglu, Erik R. Altman

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Abstract

Although VLIW architectures offer the advantages of simplicity of design and high issue rates, a major impediment to the use of VLIW architectures is that they are not compatible with the existing software base. We describe new simple hardware features for a VLIW machine we call **DAISY** (**D**_ynamically_ **A**_rchitected_ **I**_nstruction_ **S**_et from_ **Y**_orktown_). **DAISY** is specifically intended to emulate existing architectures, so that all existing software for an old architecture (including operating system kernel code) runs without changes on the VLIW. Each time a new fragment of code is executed for the first time, the code is translated to VLIW primitives, parallelized and saved in a portion of main memory not visible to the old architecture, by a _Virtual Machine Monitor_ (software) residing in read only memory. Subsequent executions of the same fragment do not require a translation (unless cast out). We describe new very fast compiler algorithms for accomplishing the dynamic translation and parallelization across multiple paths and loop iteration boundaries. We discuss the architectural requirements for such a VLIW, to deal with issues including self-modifying code, precise exceptions, and aggressive reordering of memory references in the presence of strong MP consistency and memory mapped I/O. We also show a method for approaching oracle parallelism levels in the same framework (trading off increased compilation overhead). We have implemented the dynamic parallelization algorithms for the _PowerPC_ architecture. The initial results show high degrees of instruction level parallelism with reasonable translation overhead and memory usage.

Contents

List of Figures

Chapter 1

Background and Motivation

Very Long Instruction Word (VLIW) architectures offer the advantages of design simplicity, a potentially short clock period, and high issue rates. Unfortunately, high performance is not sufficient for success. One of the major impediments to using a VLIW (or *any* new ILP machine architecture) has been its inability to run existing binaries of established architectures. It was argued (and not facetiously) in a recent MICRO conference keynote speech [Hwu94], that architectures which do not run *Intel x86* code may well be doomed for failure, regardless of their speed!

To solve the compatibility problem efficiently, there have been several proposals beyond plain or caching interpreters [Halfhill94]. One has been the object code translation approach (e.g. [SilbermanEbcioglu93, SilbermanEbcioglu92, Sites93, Thompson96]), where a software program takes as input an executable module generated for the old machine, and profile directed feedback information from past emulations, if available. It then generates a new executable module that can run on the new architecture (resorting to interpretation in some difficult cases), and that gives the same results that plain interpretation would. Although many of the nasty challenges to static object code translation (programs printing their own checksum, shared variables, self modifying code, generating a random number and branching to it, and so on) have been addressed, the static object code translation solution still has some problems.

If object code translation is used to emulate applications written for one existing machine on another ([Sites93, Thompson96]), then many primitives may need to be generated to emulate one old architecture instruction, or unsafe simplifying assumptions may need to be made (e.g. about ordering of shared variable accesses, or the number of bits in the floating point representation) to get more performance, in which case full compatibility is sacrificed. This is typically because hardware features to help compatibility with an "important" old architecture were not designed into the new fast machine; compatibility was just not emphasized, or came as an afterthought. For example, the set of condition codes maintained is often quite different in different architectures. This object code translation approach does allow the convenience of running many important applications of the old architecture on the new machine, but does not provide a replacement for the old machine in terms of speed and range of applications.

If the new architecture is fully compatible with the old one by hardware design, but does not run with the best performance on old binaries, ([SilbermanEbcioglu93, SilbermanEbcioglu92]), and the new features of the new architecture that improve performance can be utilized only by object code translation, or recompilation, the solution is still not perfect. Rapid adoption of new architectural features for higher performance may be possible under certain circumstances; scientific and technical computing is an example. But computer designers often underestimate the strong inertia of the user community and software vendors at large, and their resistance to change.

Another approach is to translate the old architecture instructions to a new internal representation (e.g. VLIW) at Icache miss time, by hardware [FranklinSmotherman94, MelvinEtAl88, RotenbergEtAl96]. This approach is robust in the sense that it implements the old architecture completely. But the optimizations that can be performed by the hardware are limited, compared to software opportunities. Also the conversion from the old architecture representation in memory to the internal Icache representation is complex (especially if one attempts to do re-ordering) and can require substantial hardware design investment, and VLSI real estate.

As an alternative we present **DAISY** (**D***ynamically* **A***rchitected* **I***nstruction* **S***et from* **Y***orktown*). **DAISY** employs software translation, which is attractive because it dispenses with the need for complex hardware whose sole purpose is to achieve compatibility with (possibly ugly) old architecture(s). Given the appropriate superset of features in the new architecture (e.g. condition codes in *x86*, *PowerPC*, and *S/390* format), **DAISY** can be dynamically architected by software to efficiently emulate any of the old architectures. Assuming that we can begin with a clean slate for both hardware and emulation software, and adopt a simple design philosophy, what architectural features and compilation techniques are required to make software translation efficient and 100% compatible with existing software? Finally, given the large gap between the parallelism ILP machines are currently achieving and oracle parallelism, what does it take to increase ILP beyond its current levels using the software emulation approach? These are some of the problems we attack in this work.

In the present paper, we will propose a simple VLIW architecture designed specifically for emulation of existing architectures, that is fully compatible with existing software including operating system kernel code, while achieving high levels of ILP. While **DAISY** and this paper focus mainly on a VLIW as the new architecture, the same ideas can be applied any new superscalar design, and potentially to other new ILP architectures that break binary compatibility as well.

Current VLIW compiler techniques are unacceptably slow for dynamic parallelization, which requires real-time performance from a compiler, in order to make the overhead imperceptible to the user. We will describe a new, significantly faster parallelization technique that does object code translation from the old binary code to intermediate code, VLIW global scheduling on multiple paths and across loop iterations, and final assembly into VLIW binary code, all at once. We have implemented this technique for the *PowerPC* and we report the initial encouraging ILP results. Another feature of the new compilation technique is the ability to maintain precise exceptions, so the original instruction responsible for an exception can be identified, whenever an exception occurs. While out-of-order superscalars use elaborate hardware mechanisms to maintain precise exceptions, in our case this is done by software alone.

The paper is organized as follows: We first discuss our new fast dynamic compilation algorithm and various architectural features to support high performance translation in **DAISY**. We then describe the dynamic translation mechanism whereby the VLIW runs the old software with minimal hardware support. Next we discuss the mapping mechanisms from the old code to the VLIW code and back. We then provide some experimental results. This is followed by a Chapter on approaching oracle parallelism.

Chapter 2

The Compilation Algorithm

In this paper, we call the original, old architecture that we are trying to emulate, the *base architecture*. The VLIW which emulates the old architecture we called the *migrant architecture*, following the terminology of [SilbermanEbcioglu93]. The base architecture could be any architecture, but we will be giving examples mostly from the *IBM PowerPC*.

Traditional caching emulators may spend under 100 instructions to translate a typical base architecture instruction (depending on the architectural mismatch and complexity of the emulated machine). So caching emulators are very fast, but do not do much optimization nor ILP extraction. Traditional VLIW compiler techniques, on the other hand, extract considerable ILP at the cost of much more overhead. Traditional VLIW compiler techniques first obtain the intermediate code for a program, then perform control flow analysis and various global optimizations with many passes over the code, and then, for each appropriate region (e.g. loops, superblocks) in the program, create VLIWs cycle by cycle, by examining which operations are ready and choosing the ones to be moved into the current VLIW. Finally register allocation and other optimizations may be performed, and an object file or VLIW assembly file is generated. The total compilation overhead with a traditional VLIW compiler may become very high.

Our goal in **DAISY** is to obtain significant levels of ILP while keeping compilation overhead to a minimum, to meet the severe time constraints of a virtual machine implementation. For this reason we have developed a new, simple and fast compilation technique that still has the potential to extract significant levels of ILP. Unlike traditional VLIW scheduling, we examine each operation in the order it occurs in the original binary code, and find which VLIW it can be placed in right away. Each

```
void TranslateOneEntry (EntryAddr) {
   CreatePathlist ();
   AddToPathlist (EntryAddr);

   while (!IsEmptyPathlist ()) {          /* Create VLIWs for a group of base   */
       x = RemoveFromPathlist  ();        /* instructions starting here, and    */
       CreateVLIWGroupForEntry (x);       /* put their exits into the Pathlist. */
   }

   /****  Convert the tree form of VLIWs into actual binary  ****/
   /****  code, and create a valid entry point in this page  ****/
   AssembleVLIWsIntoBinaryCode ();
}
```

Figure 2.1: Algorithm to translate one entry point in a page.

operation is immediately scheduled in a VLIW (maintaining precise exceptions), as soon as it is disassembled from the binary original code, and converted into RISC primitives (if a CISCy operation). The algorithm then generates binary code from the VLIWs, and the job is done.

The basic compilation algorithm used in **DAISY** is depicted in C-like pseudocode in Figure 2.1. The algorithm first puts the entry point (`EntryAddr`) of a page in a `Pathlist`. If only one *base architecture* executable program is being translated, the first `EntryAddr` is just the entry point of the program. The algorithm then creates a group of VLIWs for the set of *base architecture* instructions reachable from `EntryAddr`. When a branch is encountered in the *base architecture* instructions, translation stops, and the target address of the branch is placed in `Pathlist` (with certain restrictions discussed below in Section A.1.) If the branch is conditional, the address of the fall-through instruction is also placed in `Pathlist`. In addition, `Pathlist` stores the VLIW path from which target or fall-through instruction came. The algorithm then removes from `Pathlist` an `EntryAddr` and the VLIW path to which operations starting at `EntryAddr` should be appended. Operation proceeds as before, halting when `Pathlist` is empty. The newly created VLIWs are then translated into binary (if this is not done directly) and the translator jumps to the start of this binary code to begin execution of the translated program.

Figure 2.2 shows an example of *PowerPC* code and its conversion to VLIW code

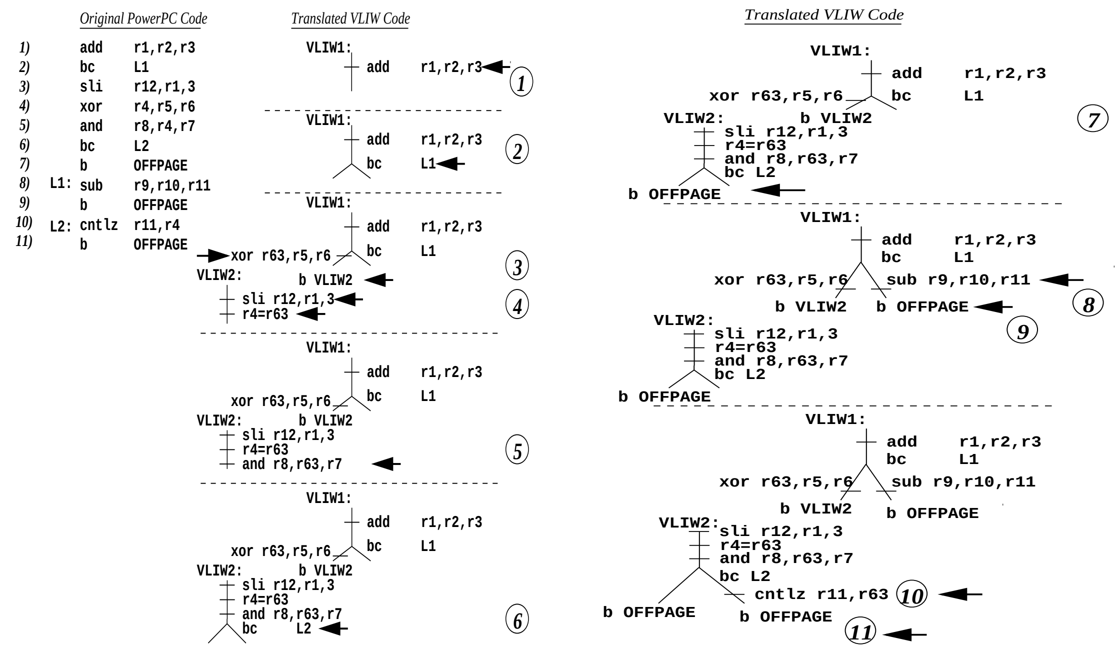

Figure 2.2: Example of conversion from *PowerPC* code to VLIW tree instructions.

using the algorithm in Figure 2.1. Appendix C explains in detail each step of the conversion. Section A.1 discusses the general principles of the conversion in depth. There are however, four major points to note here:

- 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 original architecture). The renamed register is then copied to the original (architected) 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 we use parallel semantics here in which all operations in a VLIW read their inputs before any outputs from the current

VLIW are written.)

- The renaming scheme just described places results in the architected registers of the *base architecture* 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.

- VLIW instructions are trees of operations with multiple conditional branches allowed in each VLIW [Ebcioglu88]. All the branch conditions are evaluated prior to execution of the VLIW, and ALU/Memory operations from the resulting path in the VLIW are executed in parallel.

As this example suggests, the instruction set of the *migrant VLIW architecture* should be a superset of the *base architecture* for efficient execution. [1] This example also raises several questions. How is an OFFPAGE branch handled? How and why is it different than an ONPAGE branch? How are indirect branches handled? These questions are addressed in Chapter 3.

In Appendix A, we present a more complete version of the proposed compilation algorithm, starting with the `CreateVLIWGroupForEntry` function invoked by `TranslateOneEntry` in Figure 2.1. Appendix A describes only the essentials of the rescheduling algorithm. In order to achieve good performance additional (inexpensive) optimizations are necessary. Included in these are *combining* [NakataniEbcioglu89] and some form of hardware and software support to determine whether a speculative load is aliased with a bypassed store [Moudgill96]. Constant propagation can also be useful for converting indirect branches to direct branches (crucial for *S/390* where all branches are indirect). Finally, speculative execution of operations by renaming the result register should include floating point registers and condition code registers as well as integer registers. Through the use of *combining* and renaming condition code registers, *forall* loops that initially appear to serialize on the induction variable can achieve arbitrarily high degrees of parallelism.

[1] CISC instructions such as LOAD-MULTIPLE-REGISTERS which can be directly decomposed into simpler primitives are an exception.

2.1 Essential Architectural Features for Aggressive Reordering

The VLIW must have the usual support for speculative execution and for moving loads above stores optimistically, even when there is a chance of overlap, as discussed in (e.g. [MahlkeEtAl92, Kathail94, SilbermanEbcioglu93, Ebcioglu88, EbciogluGroves90]). In order to keep the paper self contained, we briefly mention these here:

Each register of the VLIW has an additional exception tag bit, indicating that the register contains the result of an operation that caused an error. Each opcode has a speculative version. A speculative operation that causes an error does not cause an exception, it just sets the exception tag bit of its result register. The exception tag may propagate through other speculative operations. When a register with the exception tag is used by a non-speculative commit operation, or any non-speculative operation, an exception occurs. This is illustrated below:

```
    ORIGINAL CODE                  VLIW CODE
                                load  r3'<-[Addr]
bc    L1                        bc    L1
load  r3<-[Addr]                copy  r3<-r3'
```

Register `r3'` is not architected in the *base architecture*. Hence when it is `load`ed, no exception occurs, even if this `load` would normally cause a page fault or segmentation violation. Instead the exception tag bit of `r3'` is set. If the `bc` falls through, the attempt to `copy` `r3'` to `r3` will result in an exception since `r3` is architected in the *base architecture*. However, if `bc` is taken, then execution continues apace and no exception is ever taken.

As discussed above, loads may be moved above stores that cannot be proven not to store into the same location. If there does turn out to be aliasing between a speculative load and a store it passed, or some other processor changed the memory location, the code must be retranslated starting at the load. This allows both the optimistic execution of loads on a single program, and also strong multiprocessor consistency (assuming the memory interface supports strongly consistent shared memory).

It is not always possible to distinguish at compile time which loads refer to I/O space (I/O references should not be executed out of order). A speculative memory mapped I/O space load, will be treated as a no-op, but the exception tag of the result register of the load operation will be set. When the load is committed, an exception will occur and the load will be re-executed — non-speculatively this time.

Note that neither exception tags nor the nonarchitected registers are part of the *base architecture* state; they are invisible to the *base architecture* operating system, which does not need to be modified in any way. With the precise exception mechanism, there is no need to save or restore non-architected registers at context switch time.

2.2 Architectural Features to Support Commonality

If it is single *base architecture* one wishes to emulate in **DAISY**, the primitives instruction operations required for implemention are straightforward. To have the primitives to support multiple architectures simultaneously and keep them fast and simple RISC-like operations, is a harder problem. We have not solved all the problems in this area, but nevertheless we can give a partial list of the issues and potential solutions below.

These issues and solutions are very architecture specific. Since they do not directly relate to the main idea of how to implement a virtual machine in **DAISY**, they may be passed over by the casual reader. We have not finalized the bit representation for instructions in **DAISY**, but again the internal representation of the operations is not essential to the main virtual machine idea presented here.

- Three input `add` operations are needed for maximum performance on *S/390* and *x86* address calculations.

- An `Effective Address Mask Register` is needed for implementing the 31 and 24 bit modes of *S/390*.

- To support *x86* and other architectures simultaneously, **DAISY** requires the ability to take the conditional flags out of bit 8, 16, 32, or 64 registers.

- **DAISY** must have the ability — depending on its current mode — to set the `carry`, `overflow`, `sign`, `zero`, and `parity` bits of the *x86*, as well as *S/390* condition codes, and the unusual `carry` flag set by the arithmetic shift of *PowerPC*.

- To implement *S/390* access registers, **DAISY** must have the ability to specify the `address prefix register` in `load/store` instructions.

- **DAISY** must have the ability to write into a substring of a register (*x86*).

- **DAISY** needs a common intermediate format for floating point registers to deal with *S/390* hex and IEEE representations. Different floating point `load` and `store` operations must be architected to load and store in IEEE and *S/390* formats.

- For good efficiency, **DAISY** must support *Excess 6* arithmetic on 64 bit registers for implementing *S/390* decimal (BCD) operations.

- Since *PowerPC* and *S/390* use a big-endian representation of multi-byte quantities in memory, and *x86* uses little-endian, **DAISY** must support both formats. For peek efficiency, **DAISY** must support these formats in hardware, even for unaligned accesses.

- Finally, **DAISY** must map I/O operations of various architectures into a simple next generation PCI bus interface.

Chapter 3

Page and Address Mapping Mechanisms

In this chapter, we describe the address space layout of the VLIW or *migrant architecture* and how it compares to that of the *base architecture*. We then describe why this layout allows a translation mechanism whereby the *migrant architecture* runs the old *base architecture* software with minimal hardware support. We also discuss why with this layout, a page is a useful unit of translation for dynamic translation. Finally we describe why our approach for **DAISY** is robust in the presence of self-modifying or self-referential code and why all possible entry points to a page need not be known when translating from a particular entry point to that page.

The VLIW (*migrant architecture*) has a virtual memory that is divided into 3 sections, as illustrated in Figure 3.1. The low portion, starting from address 0, is mapped with the identity mapping, where *VLIW virtual address* = *VLIW real address*, and is identical to the *base architecture's* physical address space. (i.e., "real memory" for *PowerPC*, "absolute memory" for *S/390*, "physical memory" for *x86*). In Figure 3.1, for example the *base architecture* virtual page at virtual address `0x30000` is mapped to the *base architecture* physical page at physical address `0x2000` (which is the same as VLIW virtual address `0x2000` in the low portion of the VLIW virtual memory), through the normal virtual memory mechanism of the *base architecture*.

The next, middle portion of the VLIW virtual memory address space, comprises of (1) a read only store (ROM), which contains the Virtual Machine Monitor (**VMM**) software (that accomplishes the dynamic translation between *base architecture* code and VLIW code), (2) a read/write area to store various data structures needed by the **VMM**, and (3) a nonexistent memory area (a hole in VLIW virtual address space).

The middle section (where present) is also mapped with the identity mapping, *VLIW virtual = VLIW real.*

The third and top section is the translated code area, and starts at a large power of 2 address called **VLIW_BASE** (e.g.0x80000000). There are at least two ways in which this section can be mapped:

- For each page in the physical memory of the *base machine*, (= the low portion of VLIW virtual memory) there is an N times larger page in the translated code area of the VLIW virtual address space. To achieve an efficient mapping between the *base architecture* code and VLIW code, N should be a power of 2, so $N = 4$ seems a reasonable value for *PowerPC*, *S/390* or *x86*. (The actual code expansion can be larger or smaller, as described in later sections.) The translation of a page at physical address n in the *base architecture* physical memory, is at VLIW virtual address $n \times N +$ VLIW_BASE. The translated code area is not mapped *VLIW virtual = VLIW real* (since that would require a VLIW real memory area N times larger than the *base architecture* memory). Instead, the **VMM** translates pages when the first execution attempt occurs, and maps it to a real VLIW page frame from a pool of page frames in the upper part of VLIW real storage (discarding the least recently used ones in the pool if no more page frames are available).

- An alternative is to maintain the top section of memory as a hash table of translated entries. The hash table is indexed by the *base architecture* physical address and contains the real address of translated VLIW code. This hash table is maintained by the **VMM**, which adds entries to the hash table as page entry points are translated, and removes them as translations of new pages need the space. This approach has the advantages (1) that code for a translated page can be contiguous, (2) that code need never be moved when a new entry point is discovered, and (3) that there is less wastage — no portion of a VLIW real memory page need be wasted if the actual translation requires less than an $N\times$ expansion. However, this second mapping is more complicated than the first approach, and hence slower.

For simplicity, we shall discuss only the mechanisms of the first mapping in the Sections below. However, the second approach can be extended in a straightforward manner to accomplish the actions described.

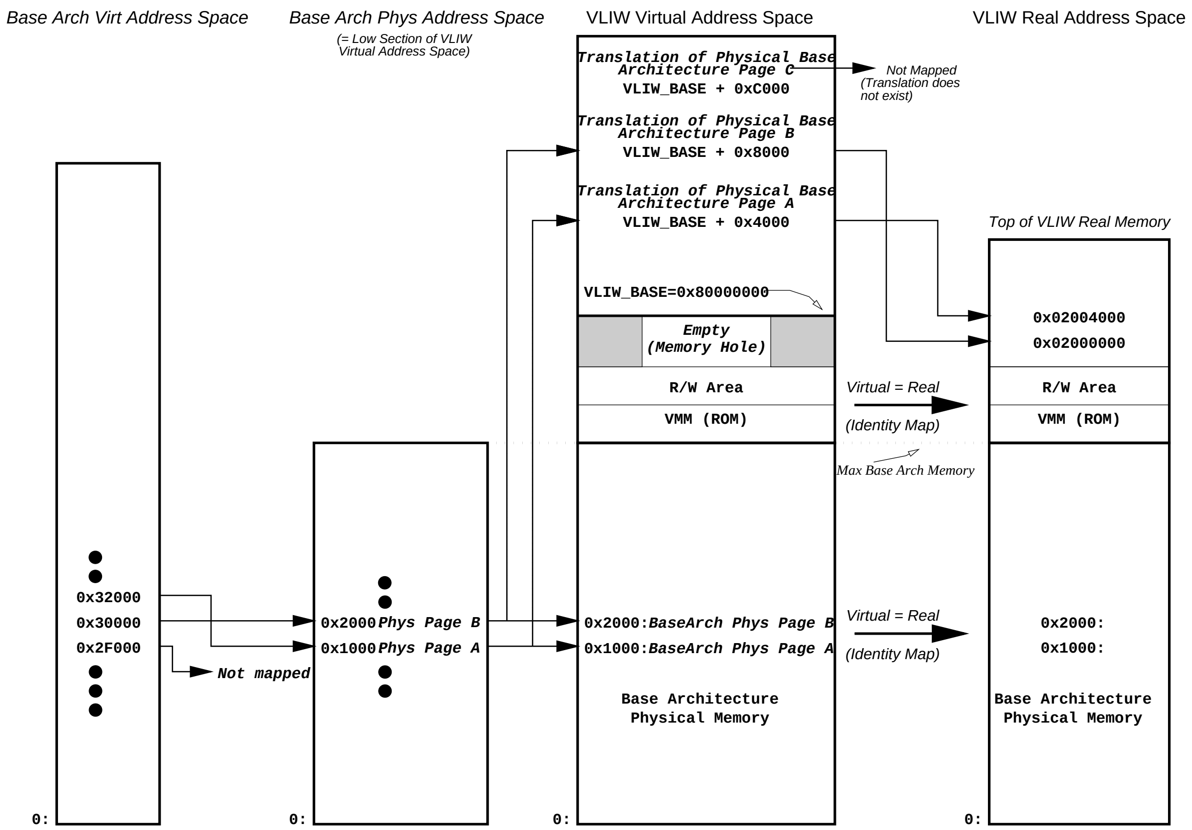

Figure 3.1: VLIW Address Space Layout

3.1 Creation of a Page Translation

Suppose a program running on the base architecture branches offpage to a *base architecture* instruction, whose physical address is n. In the translated version of the same program running on the VLIW, this branch will be executed by branching into VLIW virtual address $n \times N + \text{VLIW\_BASE}$ in upper area of the VLIW virtual address space. Assume the beginning physical address of this 4K byte *base architecture* physical page was $n_0 = (n\&\texttt{0xfffff000})$ (in C notation). If this *base architecture* page has never been executed before, then the corresponding $N\times$ 4K byte page at VLIW virtual address $(n_0 \times N + \text{VLIW\_BASE})$ will not be mapped, and therefore a "VLIW translation missing" exception will occur, which will be handled by the **VMM**. The **VMM** will create a translation for the *base architecture* physical page at physical address n_0, and make the corresponding translated code area page (which begins at VLIW virtual address $(n_0 \times N + \text{VLIW\_BASE})$ and is $N\times$ 4K bytes long) mapped to some $N\times$ 4K byte page frame in the upper area of VLIW real memory. Then the interrupted translated program will be resumed to redo the branch to address $(n \times N + \text{VLIW\_BASE})$, which will now succeed. When that first page of the *base architecture* program branches to a physical address n' in a second, different *base architecture* physical page that has not yet been executed, that page will in turn be translated and mapped in the same manner.

As a concrete example, as shown in Figure 3.1, suppose the *base architecture* program begins when the operating system branches to *base architecture* virtual address `0x30100` (part of the 4K page at `0x30000 - 0x30fff`). The *base architecture* virtual address `0x30100` has been mapped (via *base architecture* page tables) to *base architecture* physical address `0x2100` (part of the 4K page frame `0x2000 - 0x2fff`), whose VLIW translation is at VLIW virtual address $4\times$`0x2100`$+$VLIW_BASE $=$ `0x80008400` (part of the 16K page `0x80008000 - 0x8000bfff`). In the translated code, the branch to *base architecture* virtual address `0x30100` is really executed as a branch to VLIW virtual address `0x80008400`, which belongs to a 16K VLIW virtual page that is not yet mapped. So this branch initially causes a "translation missing" interrupt to the **VMM**. The **VMM** creates the translation of the *base architecture* 4K physical page frame `0x2000 - 0x2fff`, writes it into the VLIW 16K page frame at (say) VLIW real address `0x02000000 - 0x02003fff`, and maps the VLIW 16K virtual page `0x80008000 - 0x8000bfff` to this page frame at `0x02000000 - 0x02003fff`. The interrupted translated program is then restarted, and now the branch to VLIW virtual address `0x80008400` succeeds without an interrupt, and starts executing the translated VLIW code for the first page. Suppose the code in the first page of the program now branches to a second page, at *base architecture* virtual address 0x32200 (part of the 4K page 0x32000 - 0x32fff) which is mapped (via the *base architecture* page tables) to *base architecture* physical address 0x1200 (part of the 4K page frame 0x1000 - 0x1fff), whose translation is at VLIW virtual address ($4\times$0x1200+VLIW_BASE) = 0x80004800 (part of the 16K page 0x80004000 - 0x80007fff). In the translated code, the branch to *base architecture* virtual address 0x32200 is really executed as a branch to VLIW virtual address 0x80004800, which is eventually mapped to VLIW real address 0x02004800 (part of the 16K page frame at 0x02004000 - 0x02007fff), via another interrupt to the **VMM**, that creates the translation for the second page and then restarts the interrupted translated program.

This still leaves the question of how to handle an offpage branch in the *base architecture* to an address q on the same 4K page as n, but where q was not identified as a possible entry point during the translation starting from n. This problem is addressed in Section 3.4. Another concern is self-referential code such as code that takes the checksum of itself or code with floating point constants intermixed with real code or even pc-relative branches. These are all transparently handled by the fact that all registers architected in the *base architecture* — including the *program counter* or *instruction address register* — contain the values they would contain were the program running on the *base architecture*. The only means for code to refer to itself is through these registers, hence self-referential code is trivially handled. The final major concern — self modifying code — is discussed below in Section 3.2.

The above paragraphs describe the logical behavior of the address mappings. In the actual implementation, these multiple levels of address mapping are collapsed to one level, so cross-page branches can execute very efficiently, as will be seen in section 3.4.

3.2 How a Translation of a Page Gets Destroyed

Each "unit" of *base architecture* physical memory (low section of VLIW virtual memory) has a new `read-only` bit, not known to the *base architecture*. (The unit size is 4K for *PowerPC*, $\geq$ 2 bytes for *S/390*, $\geq$ 1 byte for *x86* — perhaps 8 for both.) Whenever the **VMM** translates any code in a memory unit, it sets its `read-only` bit to a 1. Whenever a store occurs to a memory unit that is marked as `read-only` (by this or another processor, or I/O) an interrupt occurs to the **VMM**, which invalidates the translation of the page containing the unit. The exception is precise, so the *base architecture* machine state at the time of the interrupt corresponds to the point just after completing the *base architecture* instruction that modified the code (in case the code modification was done by the program). After invalidating the appropriate translation, the program is restarted by branching to the translation of the *base architecture* instruction following the one that modified the code. If the page currently executing was modified, then retranslation of the page will occur before the program can be restarted. Note that this mechanism naturally handles the case when a new program begins execution in the *base architecture* memory used by an earlier program (e.g. via overlay programming techniques). When code modification events occur frequently, there are optimizations that can be applied instead of retranslating the whole page, but we will not discuss these during this initial conceptual explanation of our ideas for **DAISY**.

3.3 Communicating Interrupts to Base Architecture OS

All exceptions are fielded by the **VMM**. When an exception occurs, the VLIW branches to a fixed offset (based on the type of exception) in the **VMM** area. So far we have described the "VLIW translation not present" and "code modification" interrupts, that are handled directly by the **VMM**. Another type of exception occurs when the translated code is executing, such as a page fault or external interrupt. In such cases, the **VMM** first determines the *base architecture* instruction that was executing when the exception occurred. (The translation is done maintaining precise interrupts as was described in Chapter 2, so this is possible.) The **VMM** then performs interrupt actions required by the *base architecture*, such as putting values in specific registers. Finally the **VMM** branches to the translation of the base operating system code that would handle the exception. When the base operating system is done processing the interrupt, it executes a `return-from-interrupt` instruction which resumes execution of the interrupted code at the translation of the interrupted instruction.

As an example, consider a page fault on the *PowerPC*. The translated code has been heavily re-ordered. But the **VMM** still successfully identifies the address of the *PowerPC* load or store instruction that caused the interrupt, and the state of the architected *PowerPC* registers just before executing that load or store (see Chapter 2). The **VMM** then (1) puts the load/store operand address in the `DAR` register (a register indicating the offending virtual address that lead to a page fault), (2) puts the address of the *PowerPC* load/store instruction in the `SRR0` register (a register indicating the address of the interrupting instruction) (3) puts the (current emulated) *PowerPC* `MSR` register (machine state register) into the `SRR1` register (another save-restore register used by interrupts), (4) fills appropriate bits in the `DSISR` register (a register indicating the cause for a storage exception), and (5) branches to the translation of *PowerPC* real location `0x300`, which contains the *PowerPC* kernel first level interrupt handler for storage exceptions. If a translation does not exist for the interrupt handler at real *PowerPC* address `0x300`, it will be created (but subsequently will not be cast out, to help achieve fast interrupt response later on).

For an *x86* "segment not present" fault (interrupt number 11), e.g. arising from a far call to a procedure in a segment not currently in memory, the **VMM** finds the 11th entry in the *IDT* (Interrupt Descriptor Table, where pointers to handlers for each interrupt type are kept, as well as other information). Suppose that entry is a "task gate" that points to an interrupt handler task (the *x86* architecture can do task switching by hardware). The **VMM** performs the protection checks and state saving and restoring functions associated with the task switch, and branches to the translation of the first instruction of the interrupt handler task. If the handler has not been translated, it will be, when a branch is made to its translation, and it is found to be unmapped.

Notice that the mechanism described here does not require any changes to the *base architecture* operating system. The net result is that all existing software for the *base architecture*, including both the operating system and applications, runs unchanged, by dint of the **VMM** software.

3.4 Mapping a Base Architecture Instruction Address to a VLIW Address

We mentioned that one could find the translation of a *base architecture* instruction at physical address n, by branching to VLIW virtual address $n \times N +$ VLIW_BASE. So, if an instruction is at offset n in the *base architecture* page, its translation is at offset $n \times N$ in the VLIW translated code page. In reality, not all entry points are valid all the time in the VLIW page. In fact, initially, when a branch is first made to a translated code page, the page is created so that all entries are invalid [1]. There is a marker (e.g. a special no-op, or a bit) that indicates a valid entry point. VLIWs have a special cross-page (or indirect) branch primitive, that must branch to a valid entry point (or else there is an exception), as well as intra-page branch primitives that do not have this restriction. Initially, when the translation page is created, all entries are set to invalid. When a branch is made to an invalid entry point at offset $n \times N$, an "invalid entry point" exception occurs; the **VMM** translates the base instructions starting at offset n, and lays out the VLIWs in memory so that the VLIW code for the group of base instructions starting at offset n, begins at offset $n \times N$ in the translated code page, and marks the VLIW at $n \times N$ as a valid entry. (Previously created VLIWs may need to be moved around to create the new entry point). Secondary valid entry points on this page (e.g., at the exits of the first group of VLIWs) may also be created during the translation process. The locations which are not valid entry points are used as plain memory. For example, a VLIW code fragment will start at offset $n \times N$ with a valid entry only at that offset, and may be allocated sequentially in the translated code page thereafter. If for any reason there is no space left on the page, one could branch to an overflow area to continue. (We must keep records in order to free the overflow area when the translation of this page is destroyed by code modification, or cast out).

Return-from-interrupt (`rfi`) instructions are effectively branches to some entry point in a page. If we created translations for every target on a page that an `rfi` branches to, and a sufficient number of external interrupts occured, we could end up with too many entry points on a page. So after an emulated `rfi` branches into a translated code page, a good method is to interpret base instructions until the next subroutine call, cross page branch or backward branch is executed. This technique limits the entry points to loop headers, normal page entry points, and indirect branch targets, and guarantees that we will quickly leave the interpretive mode.

On the other hand when a plain indirect branch (probably a computed branch, C++ virtual call, or procedure return) goes to a non-existent entry point, a valid entry point should be created at the branch target, since computed branches may be executed frequently.

A variety of VLIW primitives can be used to perform a cross-page branch. We first discuss a high-performance implementation of the primitive, `GO_ACROSS_PAGE` which requires some hardware support. Below we discuss some alternatives which may have

[1] The remainder of this paragraph does not apply when a hash table, as described at the start of Chapter 3 is used instead of $n \times N + \mathrm{VLIW\_BASE}$.

lower performance, but require less hardware. The syntax of `GO_ACROSS_PAGE` is:

`GO_ACROSS_PAGE offset(reg)`

The `offset` is added to the register `reg` to obtain an effective address of the *base architecture*. That effective address is first translated to a physical address of the *base architecture*; then it is multiplied by N and VLIW_BASE is added to it; then it is translated to a VLIW real address, which is finally the address of the branch target VLIW. If the *base architecture* physical address is not available, a *base architecture instruction page fault* exception occurs (to a handler in the **VMM** — all exceptions are fielded by the **VMM**). If the translated VLIW code for this page is not available, a *translation missing* exception occurs. If the target VLIW is not marked as a valid entry, an *invalid entry* exception occurs. Otherwise execution proceeds with the target VLIW instruction.

The above description may give the impression of a daunting CISC instruction, but Figure 3.2 illustrates how it can be implemented. Assume the VLIW Instruction Translation Lookaside Buffer (ITLB) maps the *base architecture* 4K virtual page numbers directly into VLIW $N\times$ 4K real page frame numbers that contain the translated code. The software could guarantee that the low order 12 bits of `reg` is 0, or the `offset` is 0, so the low order 12 bits of the effective address `reg+offset` is immediately available. The low order 12 bits of the effective address are shifted left by $log2(N)$ bits, and applied to the Icache (14 bits allows a 64K cache, if 4 way associative). At the same time the upper bits of the effective address are sent to the ITLB. If a VLIW real address that comes out of the Icache directory matches the VLIW real address that comes out of the ITLB, no miss occurs. The target VLIW is then checked for an valid entry marker on the next cycle, while optimistically executing the target VLIW as if it were a valid entry (and recovering before any side effects occur, and causing an exception, in case the target VLIW is an invalid entry). Note that the ITLB and ICache function as a hardware version of the hash table described at the start of Chapter 3, mapping *base architecture* instruction addresses to *migrant* VLIW instruction addresses.

If only an Icache miss occurs, hardware handles it. One could handle an ITLB miss by hardware sequencers, but using a yet lower level of software to implement a "micro-interrupt" ITLB miss handler is simpler, and more in line with the philosophy of the present design. (Note that all software in a VLIW is like horizontal microcode, so no part of the VLIW software is necessarily slower than horizontal microcode.) Here is how the ITLB miss can be handled with a "micro-interrupt": If the *base architecture* is not in real address mode, the **VMM** "ITLB miss micro-interrupt handler" searches the *base architecture* page tables to find the *base architecture* physical address. If this

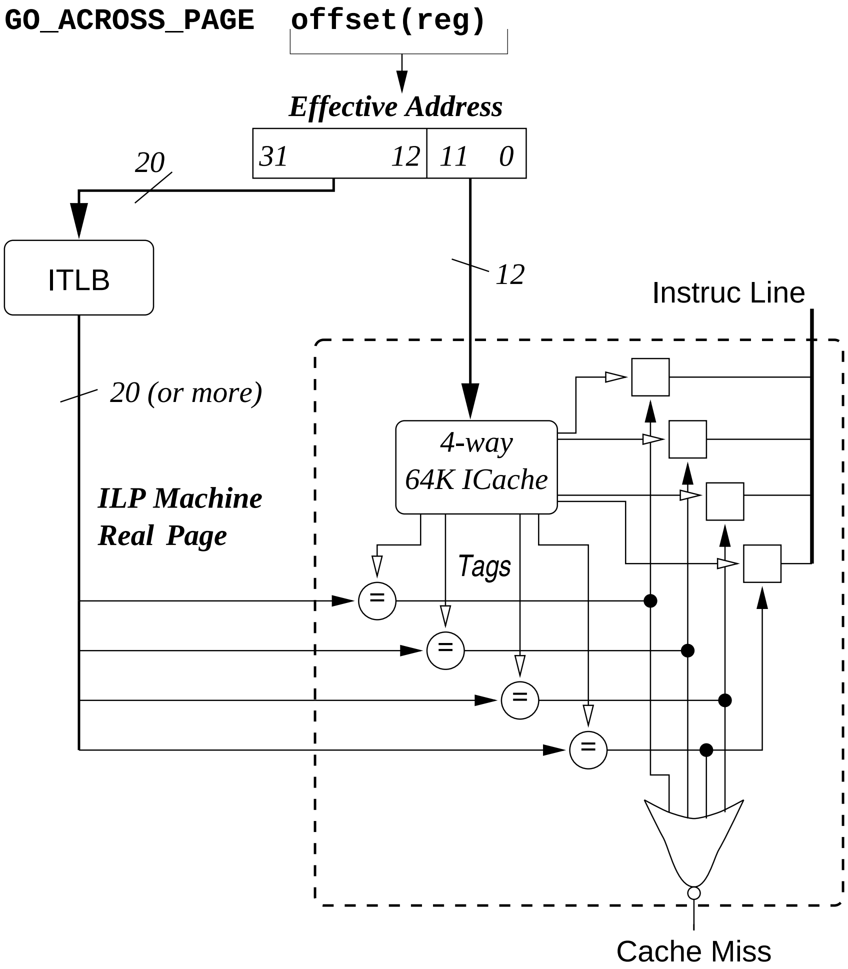

Figure 3.2: Implementation of `GO_ACROSS_PAGE` Instruction.

search fails, a *base architecture instruction page fault* is fielded by the **VMM**, which in turn communicates the page fault to the *base architecture* OS first level interrupt handler. Next, the ITLB miss handler multiplies the *base architecture* physical address by N and adds VLIW_BASE to it, and then searches the resulting VLIW virtual address in the **VMM** virtual-to-real page mapping tables, finally obtaining a real VLIW page frame address. If the latter search fails, a *translation missing* exception is fielded by **VMM**. Otherwise, an entry mapping the effective address page number to the VLIW real address page frame number is placed into the ITLB. An extra bit is appended to the effective address sent to the ITLB, that indicates if the *base architecture* is currently in physical address mode. (Thus, for example, mappings for base page no. 10 physical and base page no. 10 virtual may coexist in the ITLB.) Whenever the assumptions that caused an ITLB entry to be created change, that ITLB entry must be invalidated. Examples of this include `TLB-invalidates` by *base architecture*, code modification events, and cast-outs of **VMM** translations.

Other types of branches are:

- `GOTO offset` just branches to the VLIW at `offset` in the current page (no check for a valid entry). Ordinary intra-page branches between VLIW's use this branch.

- `GOTO lr, GOTO long_offset` branch to the VLIW at the real address given by a link register, `lr` or the `long_offset`. There is no check for a valid entry, and the ITLB is bypassed. Branches to an overflow area may use these primitives.

The `GO_ACROSS_PAGE` primitive, ITLB implementation, and valid entries mechanism described above are intended for reducing the latency of a cross page branch. If we give up the simultaneous ITLB lookup, we could first do the address translation in a prior VLIW, and then send a VLIW real address to the Icache, which has some advantages in Icache design.

```
LRA r1,offset(reg)    ; Translate (reg + offset) to physical address n.
                      ; The VLIW real address for VLIW virtual address
                      ; n * N + VLIW_BASE is then placed in r1.

GO_ACROSS_PAGE2 r1    ; Branch to VLIW real address in r1.
                      ; Check for a valid entry.
```

We can also give up the valid entry point approach. Let the translated code page for a *base architecture* page consist of a vector of pointers. For a base instruction at offset n in the *base architecture* page, vector element n will contain the real address of the VLIW code, or in case the entry at offset n has not yet been created for this page, the real address of a translator routine, that will create the corresponding VLIW code. This costs another level of indirection, but is simpler to manage.

```
LRA        t1,offset(reg)    ; Put translated reg + offset in t1.
                             ; ==> t1 contains real address of pointer
                             ;     to the VLIW code for this entry.

LOAD_REAL  r1,0(t1)          ; Load pointer to VLIW code into r1.

GOTO       r1                ; Go to real address of VLIW code.
                             ; Make no check for valid entry.
```

For additional simplicity, we could even give up the ITLB and simulate a big direct mapped ITLB in VLIW real memory by software in a manner similar to the software hash table described at the start of Chapter 3. In many cases the operations for doing the hash lookup of VLIW real address can be scheduled into VLIW instructions like other operations. Less than 10 VLIW instructions normally suffice for the lookup. These 10 VLIW's may be shared with normal program code, thus hiding their latency.

3.5 Mapping from VLIW Back to Base Instruction Addresses: How to Find the Original Base Instruction on an Exception

As we mentioned, when an exception occurs in VLIW code, the **VMM** should be able to find the *base architecture* instruction responsible for the interrupt, and the register and memory state just before executing that instruction.

The simplest way to identify the original instruction that caused an exception is to place the offset of the base instruction corresponding to the beginning of a VLIW at as a `no-op` inside that VLIW, or as part of a table that relates VLIW instructions and base instructions, associated with the translation of a page.

If the VLIW has an exception semantics where the entire VLIW appears not to have executed, whenever an error condition is detected in any of its parcels, then the offset identifies where to continue from in the base code. Interpreting a few base instructions may be needed before identifying the interrupting base instruction and the register and memory state just before it.

If the VLIW has a sequential semantics (like an in-order superscalar, where independently executable operations have been grouped together in "VLIWs") so that all parcels that logically preceded the exception causing one have executed when an

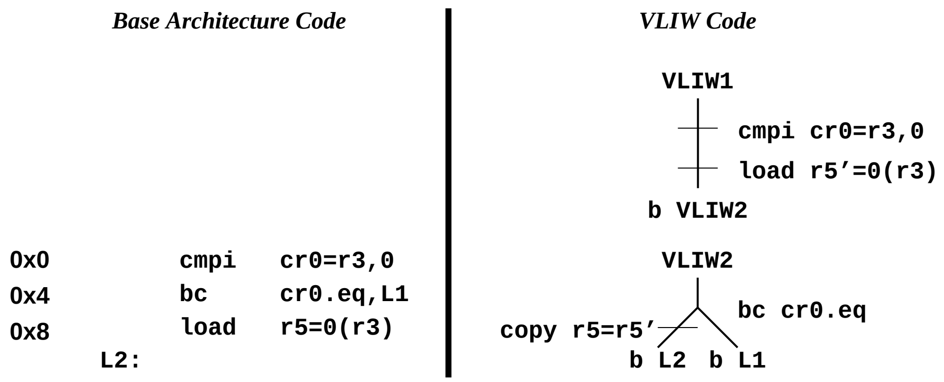

Figure 3.3: Finding the *base architecture* instruction responsible for an exception

exception is detected, the identification of the original base instruction does not require interpretation. Assuming the *base architecture* code page offset corresponding to the beginning of the VLIW is available, the original base instruction responsible for the exception can be found by matching the assignments to architected resources from the beginning of the VLIW instruction, to those assignments in the base code, starting at the given base code offset.

One way to avoid tables and pointers to the original *base architecture* instructions altogether is as follows: Let us assume the VLIW has sequential semantics, and exceptions occur at a parcel of VLIW, (as opposed to a VLIW boundary). In this scheme there are no offsets in the VLIW code that relate it to the *base architecture*, nor any tables. When an exception occurs, find a backward path from the exception causing parcel to the entry point of the group of VLIWs, which is known to have an exact correspondence with a *base architecture* instruction (If the beginning of the group is at offset $N \times n$ in the translation page, the original base instruction must be at offset n in the *base architecture* page).

We describe the scheme with the help of the example in Figure 3.3. Assume that the `load` at address 0x8 causes a page fault. To determine the *base architecture* address of the exception-causing instruction, the **VMM** finds the backward path from the exception causing parcel to the entry point of the group of VLIW's. The exception is registered in `VLIW2` in the `copy r5=r5'` instruction, when the exception bits associated with `r5'` are acted upon. Thus the **VMM** traces from this parcel to the start of `VLIW1`, the entry point of this group of VLIW's. If VLIWs are laid out in a topological order from the entry point, a backward scan in the binary code from the interrupting parcel to the nearest entry point should be able to rapidly identify the path from the entry point to the interrupting parcel.

As the backward path is scanned, {`copy, bc, VLIW2, b VLIW2, load, cmpi, VLIW1`}, the **VMM** remembers the branch directions taken by conditional branches, in this case the fact that `bc cr0.eq` is not taken. Upon reaching the top of the backwards path, the *base architecture* address corresponding to `VLIW1` is calculated: `VLIW1_addr/4` − VLIW_BASE if the code has 4× expansion. In this case the calculation yields address 0 in the *base architecture*. Now the same path is followed in forward order, {`VLIW1, cmpi, load, b VLIW2, VLIW2, bc, copy`}. There has to be a one to one correspondence between assignments to architected registers, conditional branches and stores in the VLIW code path, and assignments to architected registers, conditional branches and stores in the *base* code path. Thus the `cmpi` assignment to `cr0` is matched first. The `load` to `r5'` is passed over since `r5'` is not architected in the `base architecture`. The next correspondence is the `bc` at address 0x4 in the *base architecture*. The **VMM** recorded that this branch was not taken, so the **VMM** moves to instruction at 0x8 in the *base architecture*. The `load` to `r5` in the *base architecture* is matched to the `copy` to `r5` in the VLIW. Since the **VMM** recorded that this `copy` caused the exception, it determines that the `load` at 0x8 is the offending instruction. The **VMM** then puts 0x8 in the register used by the *base architecture* to identify the exception, and branches to the VLIW translation of the exception handler.

3.6 Dealing with Restartable CISC Instructions

Sometimes architectures (*x86*, *S/390*) require that if there is an error condition (e.g. a page fault) during the execution of an instruction, that instruction appears not to have executed, and is restarted after the operating system brings in the page and returns to the interrupted program. In this case the translation of such an instruction requires pre-testing some of the memory operands to see if they will lead to a page fault, before commencing the emulation of the instruction. For example an *S/390* `MVC` (move characters) or `AP` (add packed decimal) instruction has to touch the upper end of the memory operands first, before starting the move (or decimal addition) from the lower end of the operands. This way, either there is a page fault before the instruction has had a chance to cause side effects, or the instruction continues until completion. For really complex operations such as `TR` (Translate) in *S/390* where the translation table is within 256 bytes of a page boundary (danger of a page fault), we can do a trial execution of the instruction without causing any side effects to architected resources; and if that does not cause a page fault, we can proceed with the actual emulation of the instruction. If any interrupt other than a page fault occurs during the instruction, a "micro-level" interrupt handler may need to single step operations in the VLIW code, until a valid base instruction boundary is reached. That boundary can be found, because of the correspondence mechanisms between the VLIW code and the base code, described above.

For the *PowerPC*, this is not an issue since there are no complex memory to memory instructions. For example, the architecture specifies that a *PowerPC* store multiple instruction may have modified some of the memory before causing a page fault, but can still be restarted by the operating system.

3.7 Dealing with Real-Time Requirements

Programs are not supposed to be timing dependent (architectures such as *S/390* state that a program should not be timing dependent to run on all models). But unfortunately they can be. Virtual machines have in general unpredictable timing, so real time performance is a challenge for the present proposal.

Let us first consider the case of programs that must take less than a given time limit. One can do the following to alleviate the real time constraints in this case:

1. Use heuristics to pin translations of certain memory areas in memory, so they will not be cast out (e.g. the interrupt handlers and other code fragments known to need real time performance).

2. Offer an exact method to communicate to the **VMM** indicating that the translation of a routine should be pinned. This requires a software change.

Similarly, for code fragments that must take exactly a given amount of time, not less or more (e.g. loops for waiting for exactly m milliseconds), one could offer a version of dynamic translation that tries the emulate the timing of an old machine by padding the code with delay operations. However, such exact timing dependent programs would suffer with any faster implementation of the base architecture.

Real time interrupt response (to I/O events) is also important. So external interrupts should be enabled during translation (to the external interrupt handler the program will appear to be at the point just before executing the entry instruction of the page being translated). When an external interrupt occurs during translation, one should not throw away the ongoing translation (if the interrupts are frequent enough, one can imagine never completing the translation of this page). One approach is to continue making progress with the translation for a while, and then save the compilation state and take the interrupt. When a return from interrupt instruction tries to execute the entry instruction of the partially translated page again, the **VMM** continues from where it left off. This way, we can get both forward progress in the translation, and good external interrupt response time.

Chapter 4

Data Memory Access Requirements for a Virtual Machine

The data memory accesses by the translated code must go through the translation mechanism of the *base architecture*. For this purpose, a different data translation lookaside buffer (DTLB) can be used, mapping a *base architecture* Virtual Page Number to a *base architecture* Physical Page Number. If the data space relocation is currently turned off for the *base architecture* (it is in real address mode), it is still useful to use the DTLB to restrict access to real memory locations the *base architecture* is not supposed to access (like the real memory area where the VLIW translations are kept), and to implement some specialized protection functions (such as key protection in the *S/390* — each 4K block in *S/390* real memory has a 4-bit protection key) that have not been implemented in the actual VLIW hardware. An *address prefix register* that contains bits indicating whether data space relocation is enabled (and PSW key for *S/390* and so on), can be prepended to the effective address generated by a load or store when accessing the DTLB (i.e. the address sent to the TLB has more bits than the virtual page number). This way the same effective address will correspond to different entries in the DTLB depending, e.g., on whether data space translation is enabled or not. If the base processor tries to access an out of bounds page during real mode, a DTLB miss will occur, and the **VMM** will communicate an appropriate storage exception to the *base architecture* operating system.

The **VMM** will also need some storage space of its own. For example infrequently used registers of the *base architecture*, or storage keys for *S/390*, may be emulated in memory locations. To access these without interfering with the data of the executing program, the processor will need `LOAD_REAL` and `STORE_REAL` instructions that always bypass the DTLB, and access the VLIW real memory directly. The loads and stores that access real VLIW memory will need to be intermixed with normal loads and stores that use the DTLB, without any mode changes in between; that is why we need them.

Chapter 5

Experimental Results

We have implemented the incremental compilation technique for the *RS/6000*, which is essentially the same as *PowerPC* for our purposes. The present version of the incremental compiler is incomplete in a number of ways. For example, the "combining" optimization includes only a small subset of all combining possibilities, and software pipelining is not implemented. Nevertheless, we provide here some preliminary results on a few AIX utilities, an Erastothenes' Sieve program for finding prime numbers (a Stanford integer benchmark), SPECint95 `compress` and SPECint95 `gcc`, a large application.

In our implementation of **DAISY** we have assumed a VLIW machine with primitives similar to the *PowerPC*, but with 64 integer and floating point registers, rather than 32. To measure the amount of parallelism extracted by **DAISY**, we began with a very large VLIW machine with a total of 24 fixed point operations (out of which 8 can be stores), and a total of 7 conditional branches (8 way branching) can be executed in each VLIW, which follows the tree instruction model. We then looked smaller implementations, and in particular at a machine which can issue 8 **ALU/Mem** operations — of which at most 4 can be memory operations, and which can have 3 conditional branches in addition. Efficient hardware implementations of the tree VLIW have been described elsewhere (e.g. [Ebcioglu88]). The implemented incremental compilation algorithm is similar to the one discussed in this paper, although instead of generating binary VLIW code, an assembly level listing is produced.

Since our implementation runs on RS/6000 machines, a set of RS/6000 simulation instructions (in direct binary form) is also generated for each VLIW. These RS/6000 instructions emulate the actions of each VLIW. In effect we use a *compiled simulation* method similar to *Shade* [CmelikKeppel93] for simulating our VLIW machine on the

Program	*PowerPC* Ins *per VLIW*	Average Size of Translated Page
`compress`	6.5	14K
`lex`	4.7	27K
`fgrep`	4.8	17K
`wc`	3.0	13K
`cmp`	3.6	10K
`sort`	3.7	23K
`c_sieve`	4.6	2K
`gcc`	3.0	36K
MEAN	4.2	18K

Table 5.1: Pathlength reductions and Code Explosion moving from *PowerPC* to VLIW.

RS/6000. During transitions between VLIWs, a counter is incremented for each VLIW flowgraph edge. From the edge counts and from information about the static properties of each edge, ALU usage histograms and other statistical data can be obtained at the end of the run. A call to a kernel routine is translated to a real call, so kernel routines are not simulated in this implementation. But since there are many AIX applications that spend most of their time in user and library code, we can learn significantly about available ILP, and tradeoffs in compiler techniques, from the current incremental compiler tool.

Table 5.1 contains the pathlength reductions achieved on various AIX utilities, the Erastothenes sieve program, and SPECint95 `compress` and `gcc`. The pathlength reduction is equal to the number of operations in the RS/6000 execution trace divided by the number of VLIW instructions in the VLIW execution trace. The pathlength reduction can be viewed as an abstract measure of the infinite cache instruction level parallelism for the program. Figure 5.1 indicates how the pathlength reduction changes with the number of resources available in the *migrant* VLIW machine. These benchmarks all achieved ILP around 2 for the most primitive machine, which issues 4 instructions per cycle, 2 of which may be ALU operations, another 2 of which may be memory ops, with only 1 branch allowed per cycle. Performance diverges for the high end machine with ILP of up to 6.5 achieved for `compress`.

By way of comparison, Table 5.2 compares the performance of **DAISY** with a tra-

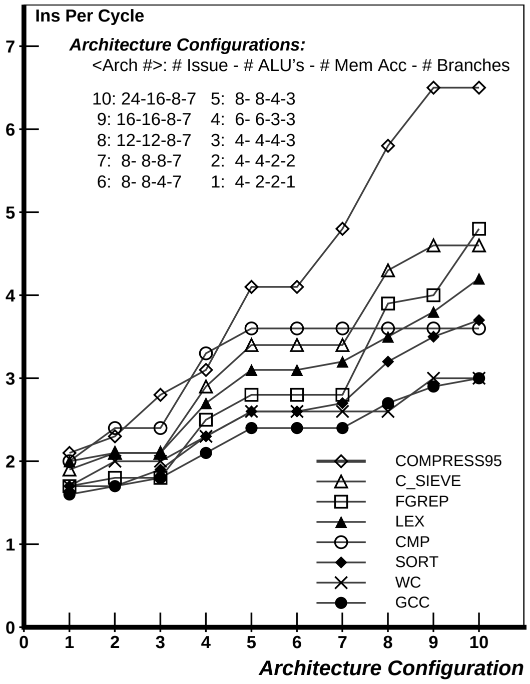

Figure 5.1: Pathlength reductions for Different Machine Configurations

Program	**DAISY** ILP	Trad ILP
`compress`	6.8	7.6
`lex`	3.9	5.4
`fgrep`	4.2	6.8
`sort`	2.5	5.1
`c_sieve`	4.6	3.9
MEAN	4.4	5.8

Table 5.2: Comparison of ILP from **DAISY** and traditional VLIW compiler.

ditional VLIW compiler performing a great number of sophisticated optimizations. [1] As can be seen, the ILP achieved by **DAISY** is less than 25% worse than that achieved by the traditional compiler on these benchmarks, albeit with much individual variation among the benchmarks. For `c_sieve`, **DAISY** actually outperformed the tradional compiler.

This performance was achieved at quite low cost. In the current experiments, **DAISY** required an average of 4315 *RS/6000* instructions to compile each *PowerPC* instruction. However, note that our implementation is currently a research prototype intended for flexible experimentation. We can expect to reduce this number significantly with straightforward tuning, and further with an eventual rewrite of the incremental compiler, when the design matures. As a rough guess, under 1000 base instructions per base instruction seems achievable for implementing our aggressive compiler techniques. Our traditional VLIW compiler breaks our profiling tools, but in order to compare our compilation speed to a standard optimizing compiler, note that the *gcc* compiler executes an average of 65K RS/6000 instructions to generate each machine instruction in its output.

Although we have not implemented a detailed timer so as to obtain precise pipeline and cache effects, our implementation of **DAISY** does include a simple cache simulator. We have measured the benchmarks the following cache configuration:

- 64 kbyte first level data cache with 4-way associativity and 256-byte lines, 0 cycle latency.
- 64 kbyte first level direct mapped instruction cache with 256-byte lines, 0 cycle latency.
- 4 Mbyte second level combined cache with 4-way associativity and 256-byte lines, 12 cycle latency.
- Main Memory, 88 cycle latency

Table 5.3 measures the reduction in ILP from using finite caches. Overall, performance drops by a little over 20% from infinite cache levels, although individual benchmarks, such as gcc fare much worse. (In gcc's case, the large increase is due to a 19% miss rate in the first level instruction cache.) The ILP attained is still

[1] Because our traditional compiler deals only with compilable user code, the comparison in Table 5.2 represents performance for the user portion of the benchmarks. Thus, numbers for **DAISY** differ from those in Table 5.1.

Program	∞ *Cache*	*Finite Cache*	*PowerPC 604E*
`compress`	6.5	2.6	0.2
`lex`	4.7	3.8	1.1
`fgrep`	4.8	3.8	0.7
`wc`	3.0	2.9	0.9
`cmp`	3.6	3.5	0.9
`sort`	3.7	2.2	0.3
`c_sieve`	4.6	4.6	1.2
`gcc`	3.0	0.8	0.5
MEAN	4.2	3.3	0.7

Table 5.3: Reduction of ILP from finite caches and comparison to *PowerPC 604E*

significantly higher than that achieved by a *PowerPC 604E* with 128 Mbytes of memory, with a mean of 3.3 versus only 0.7 for these benchmarks — almost a five fold improvement. Even the difficult `gcc` benchmark achieves 60% higher ILP in **DAISY** than on the *604E*.

Table 5.4 indicates that most VLIW instructions do not contain loads causing cache misses. Hence stalls should be relatively rare. Figure 5.2 indicates the miss rates for the different benchmarks. Most of the miss rates are quite low, with the exception of the second level cache for `c_sieve`, `cmp` and `wc` and the Icache for `gcc`. The high rates on the second level cache reflect cold start misses — there are very few accesses to the second level cache since these three benchmarks are small and have their needs satisfied by the first level cache. The high `gcc` Icache miss rate is a more serious concern and reflects the large working set of `gcc` and the fact that a cross-page jump occurs on average every 10 VLIW instructions.

For comparison, we also measured the performance on the 8-issue machine described at the start of the Chapter (8 **ALU/Mem** ops per cycle of which at most 4 can be memory operations, plus 3 conditional branches). In doing so, we also reduced the size of the first level instruction and data caches from 64 kbytes each to 4 kbytes each, while moving the 64 kbyte caches to the second level, and the 4 megabyte cache to the third level as summarized below.

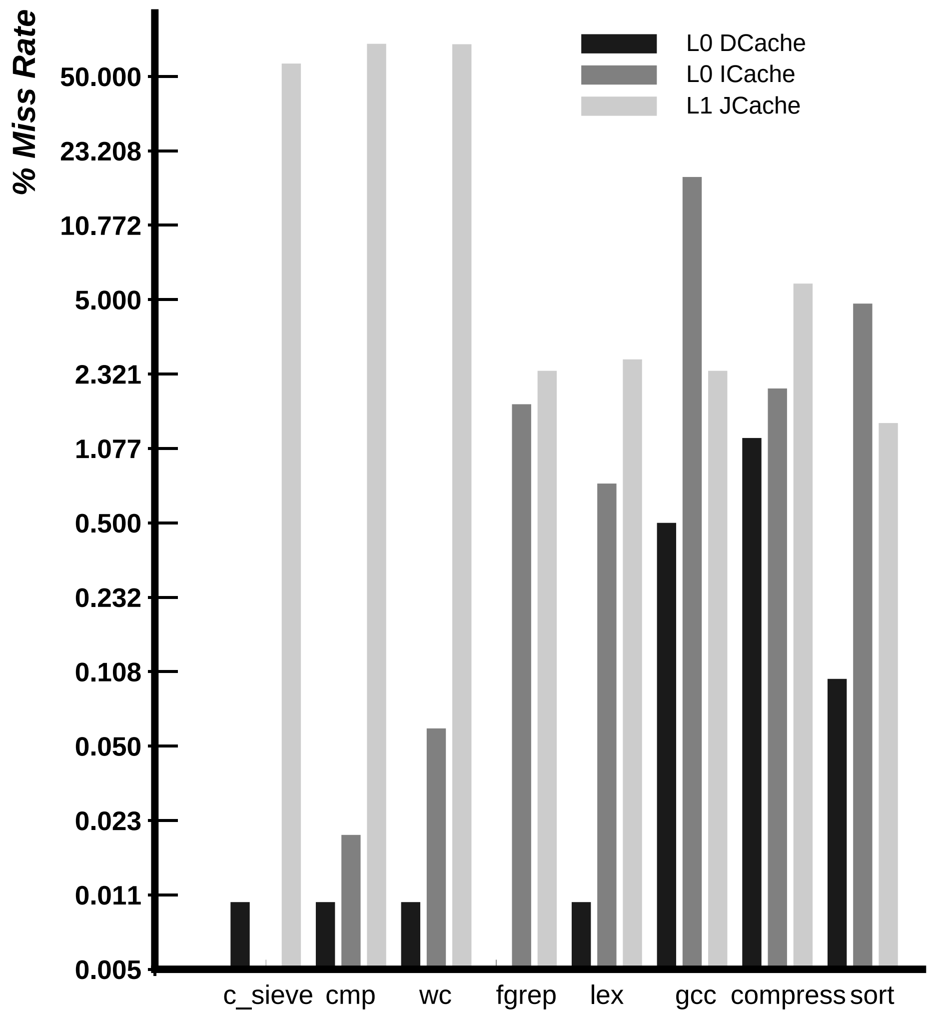

Figure 5.2: Cache Miss Rates for Benchmarks

Program	Loads per VLIW	Stores per VLIW	Mean VLIW's Between		
			Load Misses	Store Misses	Memory Misses
compress	1.39	3.19	165	19.8	17.7
lex	1.77	0.46	9,092	6,193	3,684
fgrep	1.27	0.33	18,075	49,295	13,226
wc	0.65	0.002	13,695	41,086	10,271
cmp	1.29	0.003	16,528	85,000	13,837
sort	2.41	0.84	401	1,683	323
c_sieve	0.89	0.63	92,500	9,024	8,222
gcc	1.71	0.51	96.1	899	86.8

Table 5.4: Load, Store, First-Level Cache Characteristics of Benchmarks

Cache	Size	Assoc	Line Size	Latency
Lev 1 ICache	4K	1	64	0
Lev 1 DCache	4K	4	64	0
Lev 2 ICache	64K	2	128	4
Lev 2 DCache	64K	4	128	4
Lev 3 JCache	4M	4	256	16
Main Memory	–	–	–	92

Table 5.5 indicates the results. Infinite cache parallelism is reduced from 4.2 for the 24-issue machine to 3.0 for the 8-issue machine — clearly the 8-issue machine is making more efficient use of its resources. Finite cache parallelism drops by a similar amount from 3.3 to 2.2. (The large drop for gcc is the result of large Icache miss rates.)

Code explosion statistics for the benchmarks are also in Table 5.1. The average code expansion per actually translated page is 18K/4K = 4.5X (this is just the VLIW code size; empty wasted space on pages due to the 4X fixed expansion may lead to additional overhead, unless used for something else). We have placed little emphasis in our implementation on controlling code explosion and expect to reduce the explosion in future implementations. Notice that only the actually executed pages get translated, so code explosion may be less than that of a static VLIW compiler that

Program	∞ *Cache*	*Finite Cache*
`compress`	4.1	2.5
`lex`	3.1	2.2
`fgrep`	2.8	2.1
`wc`	2.6	2.5
`cmp`	3.3	3.0
`sort`	2.6	1.7
`c_sieve`	3.4	3.3
`gcc`	2.4	0.6
MEAN	3.0	2.2

Table 5.5: Peformance of 8 Issue Machine.

translates all pages of the executable.

Another measure of interest is the number of crosspage branches executed. As discussed in Section 3.4 crosspage branches can be expensive, particularly in low-end implementations of the VLIW. Table 5.6 breaks down the number of crosspage branches in the seven benchmarks. *PowerPC* has 3 distinct types of crosspage branches: (1) direct branches, (2) branches via the `Link Register`, and (3) branches via the `Counter Register`. Notice that there is wide variety among the different benchmarks as to the fraction of instructions which are crosspage branches. From viewing these and other benchmarks, it seems to be the case that larger benchmarks have significantly more cross-page branches with up to 1 in 9 VLIW instructions ending with such a branch.

Our implementation of **DAISY** moves `loads` above `stores`, unless a simple alias analysis reveals that a `load` must alias with a `store` (in which case the `load` is replaced with a `copy` of the source register of the `store`). This speculative movement of `loads` exacts a price when a `load` and `store` turn out to be aliased during execution of the program. In this case, the value must be reloaded and execution re-commenced from the point of the load (with all speculative work discarded). Clearly for high performance it is important that runtime aliasing be a relatively infrequent event. Table 5.7 indicates that for most benchmarks undiscovered aliasing is rare, with the possible exception of `compress` (one failure every 65 VLIW's) and `sort` (one failure every 107 VLIW's). For benchmarks with high amounts of runtime aliasing, an entry point could be retranslated with movement of loads above stores inhibited. However

Program	*Branch Type* Direct	via `Linkreg`	via `Counter`	**Total**	Total VLIWS Exec / Total CR-Branches
`compress`	796	791	253,074	254,661	116
`lex`	255,573	166,981	48,050	470,604	180
`fgrep`	25	8	46	79	667
`wc`	167	269	1,048	1,484	1,111
`cmp`	498	490	943	1,931	625
`sort`	534,394	42,777	520,416	1,097,587	9.2
`c_sieve`	0	1	0	1	372,066
`gcc`	21,809,787	21,476,762	2,406,501	45,693,050	10.5

Table 5.6: Number of crosspage branches in different benchmarks.

Program	Runtime Aliases	VLIWS Exec	# VLIWS / Aliases
`compress`	9023	588K	65
`lex`	2595	24M	9333
`fgrep`	4216	2169K	515
`wc`	4	1438K	359,616
`cmp`	6	1190K	198,394
`sort`	94,359	10.1M	107
`c_sieve`	0	370K	∞
`gcc`	734,708	406M	552

Table 5.7: Number of VLIW's per runtime `load-store` alias

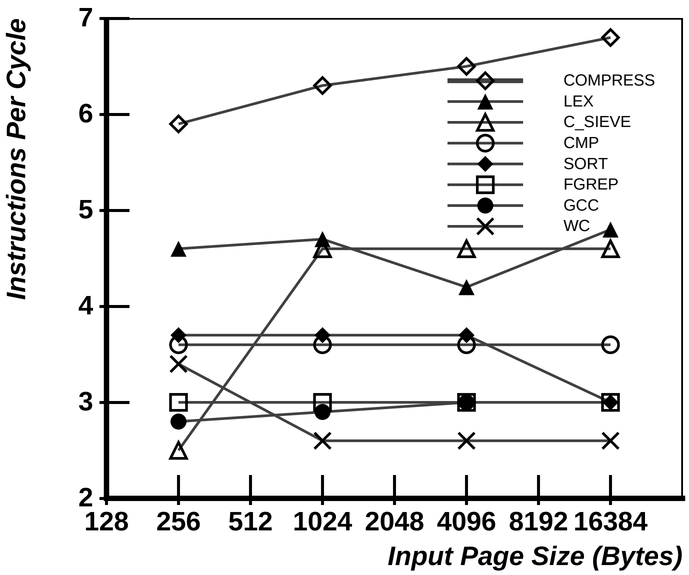

Figure 5.3: ILP versus size of input page

our implementation of **DAISY** does not yet have this feature.

It is also instructive to consider performance changes with page size. For the results presented thus far, we have assumed a 4096 byte page size, in keeping with the value used for *PowerPC*. However, the **VMM** could be made to use page sizes either larger or smaller than 4096 bytes. In particular, we are interested in

1. *Whether significant additional ILP can be extracted by using larger pages.* With the exception of `c_sieve`, Figure 5.3 indicates that the answer is no. The dramatic change in `c_sieve` moving from 256-byte to 1024-byte pages is because a critical loop is no longer split between two pages. We are still investigating the anamoly for `wc` in which 256-byte pages produce better ILP than larger pages. We suspect that with larger pages, **DAISY** is filling VLIW's with operations from a less frequent paths and crowding out operations from more frequent paths.

2. *The amount of reduction in total code size by moving to smaller pages.* Fig-

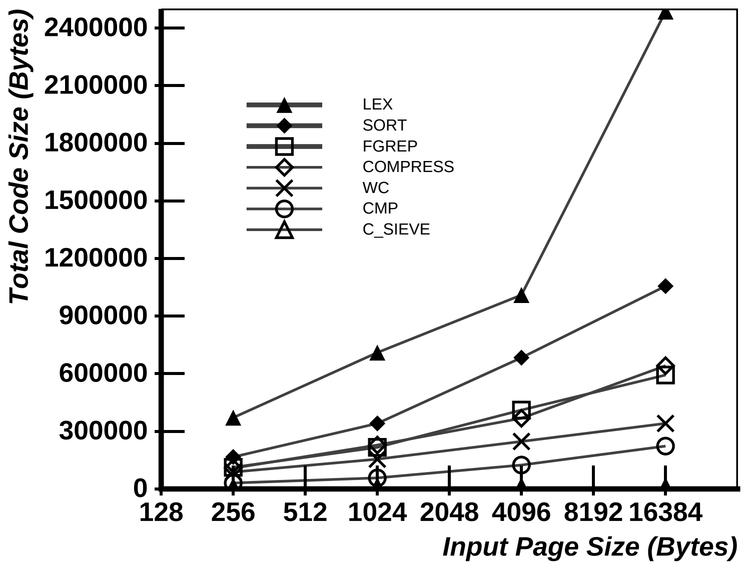

Figure 5.4: Total VLIW code size versus size of input page

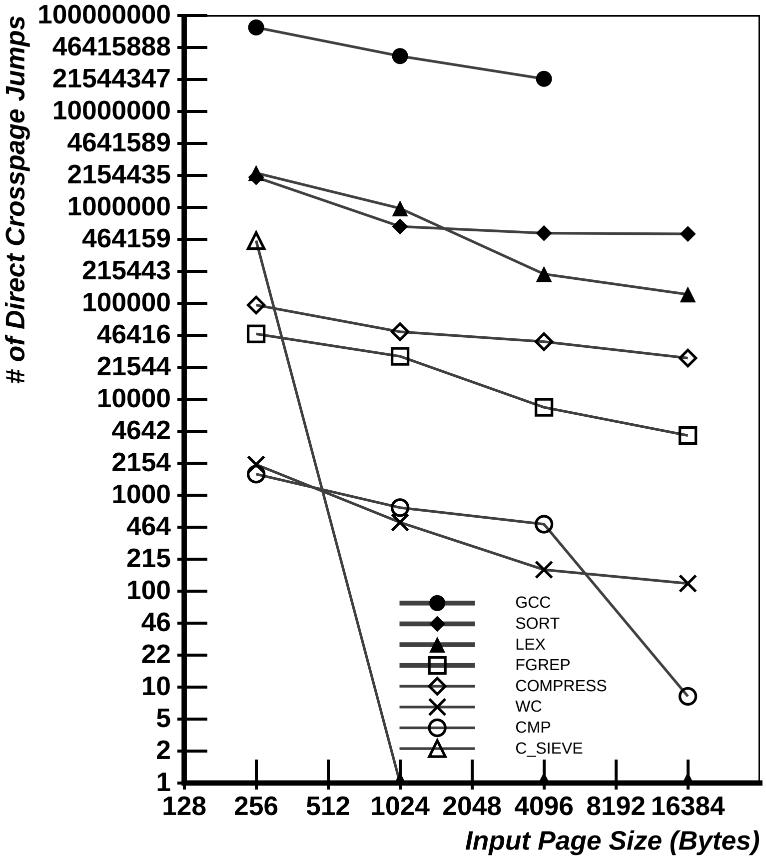

Figure 5.5: Number of direct cross page jumps versus size of input page

# Ins to Compile an Instruction	Unique Code pages	Reuse Factor	Time Change
4000	200	39000	-47%
4000	1000	7800	14%
4000	10000	780	707%
1000	200	39000	-59%
1000	1000	7800	-43%
1000	10000	780	130%

Table 5.8: Overhead of Dynamic Compilation

ure 5.4 indicates that code size generally increases slowly with page size.

3. *The change in the number of direct crosspage jumps from changing the page size.* This measure is of particular importance in low-end implementations where such jumps can be expensive. This relationship is depicted in Figure 5.5, and it is difficult to ascertain a general rule.

5.1 Analysis of Compiler Overhead

Table 5.8 indicates the extra runtime of a two second program, due to dynamic compilation, assuming a VLIW machine running at 1 GHz and assuming that both the incremental compiler and the program have an average ILP of 4 instructions per cycle.

Table 5.8 was devised using a rough formula for relating the amount of reuse needed of each page (or instruction) in order to make a VLIW with an incremental compiler faster than the *base architecture* in executing a particular program. Let

P_V	$= 4$	= Avg ILP achieved by VLIW
P_R	$= 1.5$	= Avg ILP achieved by Base Architecture
g		= Number of pages touched during program execution
t		= Time (in cycles) to translate one page
T_V		= Time to execute VLIW code
T_R		= Time to execute Base Architecture code
r		= Reuse factor (average) per page

$$i \quad = 1024 \quad = \text{Number of instructions per page}$$

Then

$$\frac{r \times g \times i}{P_R} = T_R \quad \text{(Time to execute Base Architecture code)}$$

$$\frac{r \times g \times i}{P_V} + g \times t = T_V \quad \text{(Time to execute VLIW code)}$$

We want the value of r when $T_R = T_V$:

$$\frac{r \times g \times i}{P_V} + g \times t = \frac{r \times g \times i}{P_R} \tag{5.1}$$

or

$$t = r \times i \times \left(\frac{1}{P_R} - \frac{1}{P_V}\right) \tag{5.2}$$

If, as above $i = 1024$, $P_R = 1.5$, and $P_V = 4.0$, then

$$t = 427 \times r \tag{5.3}$$

Since,

- We require 3900 instructions to translate and schedule one instruction.
- There are $i = 1024$ instructions per page.
- We assume the translator has parallelism $P_V = 4$.

the amount of time t taken to translate one page is

$$t = \frac{3900 \times 1024}{4} = 998,400$$

Plugging this in Equation (3) yields the reuse r needed to match the *base architecture* with the "realistic" assumptions above:

$$998,400 = 427 \times r \qquad \text{or} \qquad r = 2340$$

In a multiuser system with N users running identical (but separate) programs, the amount of time required for an individual user's program to complete must account for the compilation time required by all users. Whereas before we had $\frac{r\times g\times i}{P_V}+g\times t = T_V$, now we have

$$\frac{r \times g \times i}{P_V} + g \times t \times N = T_V \quad \text{(Time to execute VLIW code)}$$

This in turn changes Equation (2) to

$$t = \frac{r \times i}{N} \times \left(\frac{1}{P_R} - \frac{1}{P_V}\right)$$

In other words N times as much reuse is needed to break even with the original machine. We found that a reuse factor of $r = 2340$ was sufficient for a single user in the example above. Were this a $N = 10$-user machine a reuse of $r = 23,400$ would be needed. This is perhaps unnecessarily pessimistic however, as on large multiuser machines, the large majority of users are typically executing a few shared programs, not N individual custom applications.

Alternatively, we can obtain a rough lower bound on the reuse r by making optimistic assumptions about the VLIW and translator, and pessimistic assumptions about the *base architecture*. Let

$$\begin{array}{lll} P_R & = 1.5 & \text{(low ILP for Base Architecture)} \\ P_V & = \infty & \text{(infinite ILP for VLIW)} \\ 200 & & \text{(number of instructions to translate one instruction)} \end{array}$$

Then from Equation (2)

$$\begin{aligned} t &= r \times 1024 \left(\frac{1}{1.5} - \frac{1}{\infty}\right) && (5.4) \\ &= 683 \times r && (5.5) \end{aligned}$$

The amount of time taken to translate one page is now

$$t = \frac{200 \times 1024}{5} = 40,960$$

assuming that although the ILP of the application is $P_V = \infty$, the ILP of the compiler is only 5. Plugging $t = 40,960$ into (5),

$$r = 60$$

	Dynamic Ins Executed	Static Code Size in Ins Words	Ins Reuse Factor
INTEGER			
go	28,484,380,204	135,852	209,672
m88ksim	74,250,235,201	84,520	878,493
cc1	530,917,945	357,166	1,486
compress95	46,447,459,568	52,172	890,276
li	67,032,228,801	67,084	999,228
ijpeg	23,240,395,306	88,834	261,616
perl	31,756,251,781	138,603	229,117
vortex	81,194,315,906	212,052	382,898
FLOATING POINT			
tomcatv	19,801,801,846	81,488	243,003
swim	23,285,024,298	81,041	287,324
su2cor	24,910,592,778	94,390	263,911
hydro2d	35,120,255,512	95,668	367,106
mgrid	52,075,609,242	83,119	626,519
applu	36,216,514,505	99,526	363,890
turb3d	61,056,312,213	90,411	675,320
apsi	21,194,979,390	119,956	176,690
fpppp	97,972,804,125	91,000	1,076,624
wave5	25,265,952,275	120,091	210,390
MEAN	**41,657,557,272**	**116,276**	**452,420**

Table 5.9: Reuse factors for SPEC95 benchmarks

i.e. in a very optimistic case, a reuse factor of at least $r = 60$ is needed to make the VLIW time faster than the Base Architecture.

However, this is not a problem for two reasons. First, as detailed in Table 5.9, large programs such as those in the SPEC95 benchmark suite have very high reuse factors with a mean of over 450,000 [2].

A final illustrative example illustrates the second reason. Consider a worst case program that jumps from page to page, never repeating code. If the number of unique code pages executed is reasonable (say 200), the large percentage increase in time is probably imperceptible, as we expect only a millisecond will be required to translate each page. If the number of unique code pages is large, the overhead is likely to be dominated by the *base architecture* OS paging activity. Of course, thrashing due to a translated code area that is not large enough, will lead to extreme slowdown, and must be prevented.

[2]The static code sizes were obtained on a *PowerPC* using the installed C compiler. The dynamic execution counts are actually an underestimate, as only a subset of the complete SPEC95 reference input was used for most benchmarks. We thank Mark Charney, Tom Puzak, and Ravi Nair for these numbers and constructing the tools with which to obtain them.

Chapter 6

Ideas for Reaching Oracle Parallelism

The dynamic compilation ideas described here can also be used to measure and approach *oracle parallelism*, i.e. the amount of parallelism possible in a machine with unlimited resources and which schedules every operation at the earliest possible time allowed by control and data dependences. Earlier approaches by contrast have collected a trace of program execution and *ex post facto* scheduled all operations from the trace into the earliest cycle allowed by control and data dependences [TheobaldEtAl92, Wall91].

In Chapter 2 (and Appendix A) the guessing of branch directions implicitly assumed either compiler prediction based on heuristics [BallLarus93] or traditional profile directed feedback (e.g. passed to the **VMM** through static prediction bits in *PowerPC* branches or otherwise). If, instead, the compiler were interpreting each instruction after decoding it, then a potentially more accurate form of branch prediction can be obtained. Notice that since we are decoding the *base architecture* instructions, interpreting them at that point would add only a small overhead.[1]

In **DAISY**'s interpretive compilation approach, the first time an entry point to a page is encountered, the instructions in the page starting at the entry point are interpreted and the execution path revealed by the interpretation (say path A) is compiled into VLIWs, until a stopping point is encountered on path A. If the group is entered again, and it takes the same path A, performance will be high since it

[1]Our interpretive compilation idea in this section was inspired by a suggestion by Ravi Nair [NairHopkins]. Related ideas have been also been used in caching emulators [Halfhill94]

executes VLIW code solely. If on a third time entry, the code takes a different path B emanating from the previous path A, the instructions of path B at the exit of the path A are also interpreted, and scheduled into existing VLIWs of the same group whenever there are resources, until a stopping point is encountered on path B. So the next time the same VLIW group is entered. it will try to execute operations speculatively from both paths A and B. The incremental interpretive compilation can be repeated with different inputs to the program, until the compiled VLIW gets more stable– i.e. gets into interpretive execution less frequently. The aggressiveness of dynamic compilation parameters may be reduced when "shipping" the translation of a program that was trained this way, to reduce potential compilation overhead in the field if the program takes a new path it has never seen before.

We are currently already following multiple paths and scheduling them to the same VLIWs. However, with the interpretation approach, since it focuses on the executed instructions and ignores those that are never executed, we can afford a larger window size and may hit code explosion limits later than the static (i.e. non-interpretive) compilation approach.

Also, interpretation has advantages in compiling indirect branches. If an indirect branch is encountered to the link register `lr`,

```
 (GOTO lr)
```

and the current value of lr is 1000, then the current code can be scheduled as:

```
(IF (lr==1000) goto 1000)
(GOTO lr)
```

In the rescheduled code, no serialization occurs on the indirect branch. While techniques already exist for avoiding serialization on indirect branches for performing procedure returns (e.g. plain constant propagation will turn the indirect branch for a return into a direct branch, in an "inlined" routine, when incremental compilation is allowed to pass procedure boundaries), this approach could be useful, e.g. for avoiding serialization due to virtual calls in object oriented C++ programs, and optimizing the most common cases for a C switch statement.

Also, if cache misses are also simulated during interpretation (assuming we are willing to risk the overhead), and a cache miss is detected in a load instruction, a touch instruction can be placed an appropriate number of VLIWs ahead of this instruction. So memory latencies may be reduced. This is important for transaction processing code where much time is spent in cache misses.

We are already optimistically moving loads above stores, even if there is a chance of overlap, but with run-time information such guesses can be more accurate.

Some of the same results discussed here may also be obtained by excellent branch prediction. But interpretation during compilation leads to an especially unexpected advantage, not shared by any branch prediction mechanism (unless with infinite history): If one completely interpreted the entire trace (ignoring page boundaries) and compiled it into VLIW code, and the VLIW had sufficiently large resources and registers, then oracle parallelism can be achieved during the second execution of that program with the same input. With different inputs, more interpretation and compilation may occur, to accommodate the different traces into the VLIWs. Oracle parallelism can actually be achieved for small programs, and the present proposal may be the most practical way to achieve it (among the more theoretical alternatives. e.g. [Wall91]), because of the low overhead in the generation of the execution trace by interpretation, followed by fast scheduling of the operations on the trace for maximum ILP.

However, oracle parallelism may require a very large code size and long compilation time. So what are the practical intermediate points on the way to oracle level parallelism?

One method is to have an ILP goal, and each time a potential stopping point is reached on a path (e.g. a loop header), stop if the ILP goal has been achieved and the ILP has stopped increasing since the last potential stopping point on the path. Here the risk is that in the future the ILP may start increasing again.

For a given number of resources, even the oracle parallelism will be limited. We plan to study these methods to increase parallelism, and compare them to the actual oracle parallelism limits in the near future.

Chapter 7

Comparison to Previous Work

Virtual machine concepts have been used for many years, for example in IBM's VM operating systems [BuzenGagliardi73], but virtual machines have so far implemented a virtual architecture on almost the same architecture (e.g. *S/360* on *S/370*, *8086* on *486*, whereas in **DAISY** we support a very different virtual architecture on a VLIW. Caching emulators are commonly used for speeding up emulation. For example, each instruction is individually translated and the translation is cached for re-use when the instruction is emulated again [Halfhill94]. However, in this approach, there is no sophisticated reordering, and thus no consequent difficult issues to deal with, for maintaining precise exceptions. We are also inspired by VLIW compiler research (e.g. the Moon-Ebcioglu compiler techniques [MoonEbcioglu92]), but in this paper we have proposed a new dynamic compilation algorithm that is much faster than existing VLIW compilation techniques, and which achieves good run-time performance.

Our initial page-based translation ideas were inspired by the work of [ConteSathaye95] which proposed a translation at page fault time. However, their approach is intended for achieving object code compatibility between different generations of the same family of VLIW machines, and is not intended for emulating an existing architecture. Conte and Sathaye's approach has a clever encoding which guarantees that the size of the code does not change during translation. However this guarantee does not hold for general virtual machine implementations. Dynamic translation to an internal VLIW representation at Icache miss time [FranklinSmotherman94, MelvinEtAl88, RotenbergEtAl96, NairHopkins] achieves a similar purpose, but requires complex Icache miss preprocessing hardware, and does not allow sophisticated compiler techniques that can be done in software. Static translation of executable modules was done in [SilbermanEbcioglu93, Sites93]. However, static translation does not address the problem of achieving 100% compatibility with the old architecture, including operating system code. So, although there are many influences to our line of thought, we believe that the combination of the ideas presented here constitute a new solution for an important compatibility problem.

Bibliography

[AustinSohi95] T.M. Austin and G.S. Sohi *Zero-Cycle Loads: Microarchitecture Support for Reducing Load Latency*, Proc. MICRO-28, pp. 82-92, 1995.

[BallLarus93] T. Ball and J. Larus *Branch Prediction for Free* Proc. PLDI '93, pp. 300-313, June 1993.

[BuzenGagliardi73] J.P. Buzen and U.O. Gagliardi. *The Evolution of Virtual Machine Architecture* National Computer Conference, pp.291-299, 1973.

[CmelikKeppel93] R.F. Cmelik and D. Keppel, *Shade: A Fast Instruction-Set Simulator for Execution Profiling*, Technical Report UWCSE 93-06-06, University of Washington Computer Science and Engineering Department, 1993, `http://www.cs.washington.edu/research/compiler/papers.d/shade.html`

[ConteSathaye95] T.M. Conte and S.W. Sathaye *Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures* Proc. MICRO-28, pp. 208-217, 1995.

[Ebcioglu88] K. Ebcioglu, *Some Design Ideas for a VLIW Architecture for Sequential-Natured Software*, In Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing), edited by M. Cosnard et al., pp. 3-21, North Holland.

[EbciogluGroves90] K. Ebcioglu and R. Groves, *Some Global Compiler Optimizations and Architectural Features for Improving the Performance of Superscalars*, Report No. RC 16145, IBM T.J. Watson Research Center.

[EggersEtAl96] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers and B.N. Bershad, *Fast, Effective Dynamic Compilation*, PLDI '96.

[Ellis86] Ellis, J., *Bulldog: A Compiler for VLIW Architectures*, Ph.D. Dissertation, Department of Computer Science, Yale University (also MIT Press, 1986).

[FranklinSmotherman94] M. Franklin and M. Smotherman. *A Fill-unit Approach to Multiple Instruction Issue* Proc. MICRO-27, 1994.

[Halfhill94] T.R. Halfhill, *Emulation: RISC's Secret Weapon* BYTE, April 1994.

[Hwu94] W.M. Hwu *VLIW: Is It For Real This Time?* Keynote Speech in MICRO-27, November 1994. The foils are currently in: `http://american.cs.ucdavis.edu/Micro27`

[Kathail94] V. Kathail, M. Schlansker, and B.R. Rau. *HPL PlayDoh Architecture Specification Version 1.0*, Technical report HPL-93-80, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304, Feb. 1994.

[MahlkeEtAl92] Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Rau, and Micheal S. Schlansker, *Sentinel Scheduling for VLIW and Superscalar Processors*, Proceedings of the Fifth Int'l Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-V), Boston, MA, Oct. 12-15, 1992, pp.238-247

[MelvinEtAl88] S. Melvin, M. Shebanow, and Y. Patt, *Hardware Support for Large Atomic Units in Dynamically Scheduled Machines*, In Proceedings of the 21st Annual International Symposium on Microarchitecture, December 1988.

[MoonEbcioglu92] S.M. Moon and K. Ebcioglu, *An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors*, Proc. MICRO-25, pp. 55-71, IEEE Press, December 1992.

[Moudgill96] M. Moudgill, J. Moreno *???? Patent Application on Load-Verify Idea*, IBM T.J. Watson Research Center. 1996.

[NairHopkins] R. Nair and M. Hopkins, *Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups*, Report in Preparation, IBM T.J. Watson Research Center, 1996

[NakataniEbcioglu89] T. Nakatani, and K. Ebcioglu, *"Combining" as a Compilation Technique for a VLIW Architecture*, In Proceedings of the 22nd Annual International Workshop of Microprogramming and Microarchitecture, ACM and IEEE, pp. 43-55.

[RotenbergEtAl96] E. Rotenberg, S. Bennett, and J.E. Smith, *Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching*, In Proceedings of the 29th Annual International Symposium on Microarchitecture, November 1996.

[SilbermanEbcioglu92] G.M. Silberman and K. Ebcioglu, *An Architectural Framework for Migration from CISC to Higher Performance Platforms*, Proc. 1992 International Conference on Supercomputing, pp. 198-215, ACM Press, 1992.

[SilbermanEbcioglu93] G.M. Silberman and K. Ebcioglu, *An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures*, IEEE Computer, Vol. 26, No. 6, June 1993, pp. 39-56.

[Sites93] R. Sites et al. *Binary Translation*, CACM, Vol. 36, no.2, pp. 69-81, Feb. 1993.

[TheobaldEtAl92] K. B. Theobald, G. R. Gao, and L. J. Hendren, *On the Limits of Program Parallelism and its Smoothability*, Proc. MICRO-25, pp. 10-19, IEEE Press, December 1992.

[Thompson96] T. Thompson *An Alpha in PC Clothing* BYTE, February 1996.

[Wall91] David W. Wall, *Limits of Instruction Level Parallelism*, Proc. 4th ASPLOS, 1991.

Appendix A

The Compilation Algorithm

In this appendix, we first describe six functions that are at the heart of the compilation algorithm for converting code from the *base architecture* to VLIW. Note that these functions are greedy – always scheduling operations as early as possible, and never perform any backtracking. Such limitations seem necessary in order to minimize the overhead and make the proposed approach practical. We then provide additional detail on the actual creation of VLIW's in Section A.1.

The `CreateVLIWGroupForEntry` function in Figure A.1 follows *base architecture* code from some entry point, scheduling each instruction in a VLIW as it is seen. The `DecodeAndScheduleOneInstr` function in Figure A.2 checks if the current instruction is a stopping point (e.g. the end of the page). If it is not, the instruction is decoded and scheduled into a VLIW by a routine specific to the instruction type, for example `ScheduleThreeRegOp` for `add`.

The `ScheduleThreeRegOp` in Figure A.3 and its subroutines `ScheduleThreeRegOp_OutOrder` in Figure A.4 and `ScheduleThreeRegOp_InOrder` in Figure A.5, find the earliest VLIW on the current path, at which data dependences allow an instruction to be scheduled. It then moves forward on the path looking for the first VLIW in which sufficient resources are available to insert the instruction. If that VLIW is earlier than the last VLIW on this path, the result is placed in a VLIW register that is not architected in the *base architecture*. For example, a result may be placed in `r63` if the *base architecture* is *PowerPC* with only 32 integer registers. Then in the latest VLIW on this path, a copy operation is inserted to move the value from the non-architected register to the architected register.

The `ScheduleBranchCond` function in Figure A.6 handles conditional branches. It destroys the current path, and in its place inserts entries in the `PathList` corre-

```
void CreateVLIWGroupForEntry (x);
/* Create a group of VLIWs for group of base arch instructions
   starting at address x */
{
   /* Create an empty path with continuation x, put it in PathList  */
   PathList = CreatePath (/*continuation=*/x, /*probability=*/1.0,
                          /*nextpath=*/NULL);
   while( PathList!=NULL ) {
      /* First entry in the pathlist is the most probable path */
      /* DecodeAndScheduleOneInstr may change the pathlist */
      DecodeAndScheduleOneInstr (PathList);
   }
}
```

Figure A.1: The function `CreateVLIWGroupForEntry`.

sponding to the target and fall-through of the conditional branch. The target and fall-through are assigned execution probabilities, so that instructions from more probable paths can be scheduled first.

A.1 Actual creation of VLIWs

In order to keep compilation time down, the translator works on only a small chunk of the *base architecture* program at a time. This helps limit the amount of code that is needlessly translated, i.e. that is translated but never executed by the program. When the program jumps to an untranslated location, another small chunk is translated starting at the jumped to location. As will be explained in more detail in Chapter 3, one page is a suitable chunk and is easily supported by simple hardware structures.

To reduce slowdown effects due to branch mispredictions, the translation algorithm in Figure 2.1 pursues multiple paths. The multiple paths through an individual VLIW tree instruction provide for this, as does the fact that VLIW instructions starting from a page entry point themselves form a tree. The root VLIW instruction of the tree of VLIW instructions corresponds to the page entry point. The algorithm in Figure 2.1 for creating the tree of VLIWs maintains a list of paths, `Pathlist`, that is sorted in decreasing probability order.

Initially there is one dummy path in `Pathlist`, whose continuation is the entry base address (`EntryAddr`), and whose probability is 1.0. Later, because of the sched-

```
void DecodeAndScheduleOneInstr (T_PATH *p)
{
   /* If path at stopping point stop extending this path */
   if (IsStoppingPoint (p,p->continuation)) {
      RemoveFromPathlist(p);
      /* If continuation of the path is in the current page, add
         its continuation address to worklist (if not already there) */
      if (p->continuation!=UNDEFINED    &&
         InCurrentPage(p->Continuation) &&
         WasNeverInWorklist(p->continuation))
        AddToWorklist(p->continuation);
      return;
   }
   else {            /* Continuing to extend the probable path... */
      /* Fetch instruction */
      ins = *(p->continuation);
      /* Decode instruction, and call a scheduler routine for each RISC
         primitive resulting from the instruction */
      switch (OPCODE(ins)) {
         case ...: ...
         case OP_ADD: ScheduleThreeRegOp(p,OP_ADD,RT(ins),RA(ins),RB(ins));
                break;
         ...
         case OP_BC: ScheduleBranchCond(p,OP_BC,TEST(ins),CC(ins),TARGET(ins));
                break;
         ...
      }
   }
}
```

Figure A.2: The function DecodeAndScheduleOneInstr.

```
void ScheduleThreeRegOp (T_PATH *p,int opc,int rt,int ra,int rb)
{
   /* Scheduling rt=opc(ra,rb) */
   /* Find the first VLIW v where both operands are available   */
   /* p->avail[r] gives the earliest VLIW # on this path, where
      an instruction using register r can be scheduled */
   v = MAX (p->avail[ra], p->avail[rb]);

   /* If v > LastVLIW on this path, open new empty VLIWs so LastVLIW<=v */
   while (p->LastVLIW <=v) OpenNewVLIW (p,p->lastVLIW);

   /* v is the earliest VLIW where op can be scheduled.
      Starting from v, look for a VLIW on the path that has:
      (1) enough resources to accommodate the op,
      (2) a nonarchitected destination register that is free
      until the end of the path */

   while(v<p->LastVLIW && !(AluResourceOk(p->vliw[v]) &&
          (p->vliw[v]->FreeGprsUntilEnd)) ) v++;

   /* v is the VLIW where op will be legally scheduled */
   if (v < p->LastVLIW)
       ScheduleThreeRegOp_OutOrder (p, v, opc, rt, ra, rb);
   else ScheduleThreeRegOp_InOrder (p,    opc, rt, ra, rb);

   /* Increment the continuation address of this path */
   p->continuation += 4;
   return;
}
```

Figure A.3: The function ScheduleThreeRegOp.

```
void ScheduleThreeRegOp_OutOrder (T_PATH *p, v, opc, rt, ra, rb)
{
   /* Scheduling rt=opc(ra,rb) out of order at VLIW v on path p*/
   /* Pick a suitable non-architected register to use as dest. reg*/
   dst=CountLeadingZeros(p->vliw[v]->FreeGprsUntilEnd)+FIRST_NONARCH_REG;

   /* Need to have op use non-architected dest. reg dst, and
   possibly renamed source regs. p->vliw[v]->map[ra] is the
   name of ra in VLIW # v*/
   AddToVLIWTip(p->vliw[v], opc, dst,
                p->vliw[v]->map[ra],  p->vliw[v]->map[rb]);
   IncrementAluResource(p->vliw[v]);

   /* Now schedule the commit dst->rt in the last VLIW,
      (or one VLIW after it, if last vliw is full) */
   if (!(AluResourceOk (p->vliw[p->LastVLIW])) OpenNewVliw(p);

   AddToVLIWTip(p->vliw[p->LastVLIW],OP_COMMIT,rt,dst)
   IncrementAluResource(p->vliw[p->LastVLIW]);

   /* rt is mapped (renamed) to dst after v until the last vliw  */
   /* dst is not free after v until the last vliw */
   for (v1=v+1; v1 <= p->LastVLIW; v1++) {
      p->vliw[v1]->map[rt]   = dst;
      p->vliw[v1]->FreeGprs &= ~(0x80000000>>(dst-FIRST_NONARCH_REG));
   }

   /* Update the regs that are free in each vliw
   until the end of the path */
   for (t = -1, v1 = p->LastVLIW; v1 >= 0; v1--) {
      t = t & p->vliw[v1]->FreeGprs;
      p->vliw[v1]->FreeGprsUntilEnd = t;
   }

   /* The destination register rt is available after v */
   p->avail[rt] = v+1;
}
```

Figure A.4: The function ScheduleThreeRegOp_OutOrder.

```
void ScheduleThreeRegOp_InOrder (T_PATH *p, opc, rt, ra, rb) {
   /* Schedule op rt=opc(ra,rb) in order (after all
   logically preceding ops have committed results)*/
   /* Commit result directly to architected reg. rt*/

   /* If the last VLIW cannot accommodate op, open a new VLIW */
   if (!(AluResourceOk (p->vliw[p->LastVLIW])))
     OpenNewVliw(p);
   /* Add op to last VLIW */
   /* Source registers may be renamed, dest reg is not */
   AddtoVLIWTip (p->vliw[p->LastVLIW], opc, rt,
                 p->vliw[p->LastVLIW]->map[ra],
                 p->vliw[p->LastVLIW]->map[rb]);
   IncrementAluResource (p, p->LastVLIW);

   /* FreeGprs not updated, map not updated */
   /* rt is available after last VLIW */
   p->avail[rt]=p->LastVLIW+1;
}
```

Figure A.5: The function ScheduleThreeRegOp_InOrder.

```
void ScheduleBranchCond (T_PATH *p, opc, test, cc, target)
/* Schedule a conditional branch */
{
   /* Branches, stores are scheduled in last VLIW -- later, if dep */
   v = MAX (p->avail[cc], p->LastVLIW);

   /* If branch has to be scheduled after the last VLIW,
      open a new VLIW  and update last VLIW pointer */
   while (p->LastVLIW < v) OpenNewVliw (p);

   /* if not enough resources in the last VLIW, open a new VLIW */
   if (!BrResourceOk (p, p->LastVLIW)) OpenNewVliw (p);

   /* Clone the path */
   p2 = NewPath ();
   CopyPath (p2, p, p->LastVLIW);

   /* Add the if */
   AddIfToTreePath (p->vliw[p->lastVLIW],
                    test, cc, p2->vliw[p->lastVLIW]);

   /* Guess the branch probability */
   BrProb= GuessBranch (p->continuation);
   p2->prob = p->prob *    BrProb;
   p->prob  = p->prob * (1.0-BrProb);

   /* p continues with fallthrough instr, p2 with branch targ */

   p->continuation += 4;
   p2->continuation = target;
   IncrementBrResource(p, p->LastVLIW);


   /* Add the two new paths to pathlist in probability order */
   RemoveFromPathlist(p);
   AddToPathlist(p);         /* Maintain probability order */
   AddToPathlist(p2);        /* Maintain probability order */

   return;
 }
```

Figure A.6: The function ScheduleBranchCond.

uling of conditional branches, more than one path may be present in the `PathList` at a given time. The parallelization algorithm always picks the most probable path in `Pathlist`, and extends it at the end of the path by adding to it the base instruction that is the continuation of this path.

As noted above, when a conditional branch is scheduled on a path, both the branch target address and fall-through address are added to `Pathlist`. A probability is assigned to the branch. The probability can be a compile time guess, or can be provided by static prediction bits in the *base architecture* binary (like the `Y-bit` in *PowerPC*), or can use some other means like profile directed feedback. The probability of a path is the product of the probabilities of the actions of branches on the path (E.g. if there is a taken branch b_1 followed by an untaken branch b_2 on the path, the probability of the path is $Prob(b_1 \ \text{taken}) \times (1 - Prob(b_2 \ \text{taken}))$, assuming we use simple independent branch probabilities). The probabilities of the two new paths resulting from the conditional branch is computed and the two new paths are inserted into the `Pathlist`, while maintaining its ordering by path probabilities.

By picking the path with the highest probability of execution, the resources of the VLIW's are preferably spent on the operations on the most probable paths, making those paths run faster. Operations scheduled in early VLIW's in the group tend to come from probable paths. Note that our approach does not penalize less probable paths as much as trace scheduling, since operations from a less probable path can be moved into the most probable path.

Another question is how the scheduler avoids entering an infinite loop when scheduling program loops. A conditional branch at the end of a program loop adds two values to the `Pathlist` in Figure 2.1, the fall-through exit of the loop and the top of the loop. Without additional termination conditions, the head of the loop will repeatedly be added to `Pathlist` in a manner corresponding to unlimited unrolling. To avoid this problem, the algorithm terminates for a given entry point when all paths are closed, and `PathList` becomes empty. Closing paths at the following stopping points guarantees that eventually all paths will be closed:

- A cross page branch or indirect branch (mandatory stopping points),
- A join point that we have already visited k times. (This is a throttle on code explosion and guarantees that a base instruction will not belong to more than $k+1$ VLIWs),
- A join point where the number of instructions scheduled on the path since the entry point has exceeded a window size limit (another throttle on code explosion,

as well as register pressure and compilation time),

- A loop header where the ILP has not improved significantly since the last visit to this loop header (to avoid useless unrolling).
- Another rule related to stopping: if a loop exit (a conditional branch inside a loop that goes out of it), or a loop header which is not the same as the entry point of the group, is seen, the remaining window size budget on this path is decreased (in order not to pull in too many operations from the exit of a loop into a loop, or from an inner loop into an outer loop).

Note that loops and join points are identified incrementally as each branch is seen.

As noted, the scheduling of a base instruction at the continuation of a path, consists of converting the instruction into RISC primitives, and then scheduling each one into an existing VLIW on the current path, or, failing that, creating a new VLIW and appending it to the current path, and scheduling the RISC primitive in the new VLIW.

To be more precise, first the base instruction that is the continuation of the path being extended is decoded, and converted into RISC primitives. Then each primitive (say `r1:=r2+r3`) is scheduled as follows: Find the VLIW on this path where `r2` is available (ready), and the VLIW where `r3` available (these VLIWs can be directly accessed by a per-path array that maps each architected register to the sequence number of the VLIW on this path where that register is available). Choose the VLIW that occurs later among these two. Starting from this VLIW, walk forward on the path, until a VLIW that has a free nonarchitected register `r1'`, (to use as a destination register) is found, and that has enough resources (so we are doing register constrained list scheduling on this path). The names of `r2` and `r3` may in general be different than the actual architected `r2` and `r3` in this VLIW. At the end of each path — from the first (root) VLIW to the end of each leaf VLIW — there is a map that indicates the times at which each architected register `r` is defined, possibly into a non-architected register `r'`. The map also defines the time at which `r'` is committed to `r`. Note that this map cannot be kept on a per VLIW basis, as the following code fragment illustrates:

```
     Original Code      |       VLIW Code
                        |
     bc   L1            |  ======================
     add  r5<--r3,r4    |        add  r5' <--r3,r4   o VLIW1:  2 mappings
     b    L2            |        add  r5''<--r2,r4             for r5
L1:  add  r5<--r2,r4    |        bc   VL1
L2:                     |  ======================
                        |  VL1:  copy r5  <--r5'     o VLIW2
                        |        b    VL2
                        |  ======================
                        |  VL2:  copy r5  <--r5''    o VLIW3
                        |  ======================
```

Since `VLIW1` has both `VLIW2` and `VLIW3` as successors, `map[r5]` is multiply defined for `VLIW1`. If `bc L1` is an entry point to the original code, then there are two paths, with PATH1 terminating at `VLIW2` and PATH2 at `VLIW3`. Thus for PATH1, `map[r5] =` {`[t=1,reg=r5'],[t=2,reg=r5]`} and for PATH2, `map[r5] =` {`[t=1,reg=r5''],[t=2,reg=r5]`}, where `t` denotes the time at which the value is ready and we have assumed unit latencies.

Returning to our original example, assume `r2` is renamed to `r2'`, and `r3` is not renamed (its value is in `r3` itself) in the VLIW where we have chosen to schedule the operation. We schedule the operation in the form `r1':=r2'+r3` in this VLIW, and then schedule a `commit` operation `r1:=r1'` in the last VLIW of the path (or in a new VLIW after the last one, if it will not fit in the last VLIW). This use of non-architected registers (in the *base architecture*) to speculatively compute results is a key to obtaining good ILP.

If the operation has to be placed in the last VLIW (or later), the result can be assigned directly to the architected register `r1`, e.g. in the form `r1=r2'+r3`. This reduces code explosion due to having to generate two operations per original operation: one to execute the operation out of order (`r1':=r2'+r3`) and another one to commit the result to the architected register in order (`r1:=r1'`). Stores and conditional branches are always placed in the last VLIW (or in a new VLIW at the end of the path, if they are not ready or there are not enough resources in the last VLIW).

As just described, an operation can be executed out-of-order as soon as its operands are ready, but its result is placed in a non-architected register if it is placed anywhere before the last VLIW. The non-architected register will in turn be committed to the original architected result register of the operation, in the last VLIW or later. This way stores, branches and assignments to architected registers are executed in order, and precise exceptions are easily achieved. That is, at a given point just before a VLIW in the VLIW program, one can identify a *base architecture* instruction I so that all instructions before I have committed their results, and none after I have committed their results to architected registers and memory. Note that even though re-ordering of assignments to architected resources is not allowed, multiple assignments to architected registers, multiple branches, and multiple stores can be executed in a single VLIW, if the resource constraints allow it. So multiple *base architecture* instructions can be completed per cycle.

Appendix B

Supporting Imprecise Interrupts, and Other Optimizations to the VMM Scheme

So far we have concentrated on getting high performance with a fully compatible implementation of the *base architecture*. However, once good performance is achieved in the fully compatible version of the new machine, extra non-compatible performance features can be an advantage. Such features may require small software changes to support them, and may not be adopted immediately since they were not in the *base architecture*, but will likely be source of improved performance over time. In this appendix we will briefly consider some such features.

`Commit/copy` operations can drain machine resources, and reduce performance somewhat. It would be desirable to compile without the copy operations. Also, given a mode where we are allowed more compilation time, it may be desirable to perform more substantial optimizations, that change the order of the computations, and which can make a program almost unrecognizable. Examples of such optimizations are tree height reduction, loop transformations, and even re-implementation of inner loops using new multimedia operations. But in a program compiled without precise exceptions, it is impossible to identify a precise point in the *base program* to restart from, when an unexpected exception such a page fault occurs. Our suggested way to allow imprecise exceptions in **DAISY** is to add a new "restartable" CISC-like instruction called `resume_VLIW` to the *base architecture*, but to always place it in a normally unreachable part of the base code, so old *base architecture* machines never execute it. A procedure generated by a new version of a compiler for the *base architecture*, that is aware of the **VMM**, could start with:

```
    B L1
    Magic number
    Various information to be sent to VMM
    ...
    resume_vliw &save_area
    ...
    Checksum of the information
L1: Normal base architecture code for the procedure
```

The semantics of the `resume_vliw &save_area` instruction is to restore the VLIW extra registers from the save area, and to continue from where the VLIW program left off. In a program compiled without precise exceptions, when an unexpected interrupt such as a page fault or external interrupt occurs, the **VMM** recognizes that there is no precise exception information, but knows the location of the `resume_vliw` instruction and therefore the save area address. The **VMM** then stores the contents of the extra VLIW registers into the save area, as well as the VLIW program counter, and reports the address of the `resume_vliw` instruction as the exception causing base architecture instruction. The save area must have been touched before executing any code with imprecise exceptions, to guarantee that it is present in memory when it is needed. When the operating system services the external interrupt or page fault, it will restart the interrupted program from the `resume_vliw` base instruction. The translation of `resume_vliw` into VLIW code: (1) restores the extra VLIW registers from the save area, and (2) branches to the saved VLIW program counter value. Note that the translation of privileged base instructions always verifies supervisor state in the *base architecture*, so a program in user state will not be able to execute any privileged operations, even if it alters the save area maliciously.

Note that the `resume_vliw` instruction will never be executed by an older version of the same *base architecture*, since it is in a normally unreachable portion of the code. So executables compiled for the new VLIW-**VMM** implementation, will continue to run on older machines without changes. Binaries can be exchanged between old and new implementations of the *base architecture* over a network, without problems.

This technique is a good way to introduce a completely new architecture, that has extra registers, without really exposing it to the user or operating system. It requires only minimal changes in an existing *base architecture* compiler. It also requires no operating system changes in the *base architecture*, unless the operating system examines the bit pattern in the interrupting instruction, and refuses this particular new opcode. To avoid this latter possibility, one could choose the opcode of `resume_vliw` to be equal to that of an existing memory operation, where the `VMM` understands its true meaning by context.

There are many other optimizations that could improve performance in a **VMM** environment. We briefly list some of these below:

- The **VMM** can keep a cache of translated pages that it can quickly look up, before starting a new translation from scratch

- The **VMM** can save the translation cache at power down time on hard disk, and restore it at power up time. This can be done with or without the cooperation of the base operating system.

- The **VMM** can translate procedures spanning multiple pages together

- A new compiler for the *base architecture* can pass useful information to the **VMM** (in the unreachable locations of the code between `B L1` and `L1:`), such as aliasing information, jump tables, list of volatile loads/stores.

- The **VMM** can utilize the wait time of the OS to improve previously generated translations. The obvious procedures to work on are the ones where most of the time is spent.

Appendix C

Example of *PowerPC* to VLIW Conversion

Figure C.1 depicts the example from Section A.1 (Figure 2.2) on translating *PowerPC* to VLIW code. The translation follows the algorithm detailed in Appendix A, and a detailed description of why each step is performed is contained in Figure C.2.

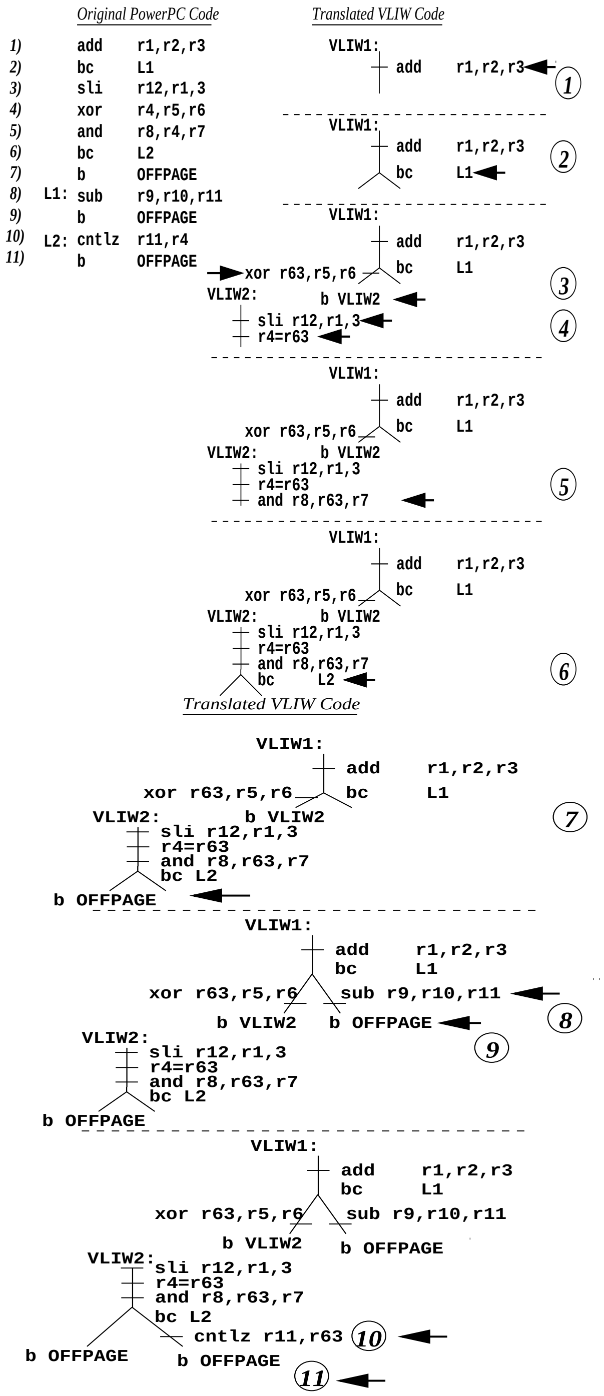

Figure C.1: Example of conversion from *PowerPC* code to VLIW tree instructions.

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(0.7×0.7)], [(1,2),8,0.3], [(1,2,3,4,5,6),10,0.21(0.7×0.3)]`}.
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 C.2: Description of conversion from *PowerPC* to VLIW.

Appendix D

Conversion of *PowerPC* Code

In implementing **DAISY**, we have observed a few details of interest beyond the main points discussed in the body of the paper. First, the *PowerPC* architecture has only two registers, the link register `lr` and the counter registers *ctr* through which indirect jumps may be performed. At an arbitrary point in *PowerPC* code, the values in both may be live. The `bcrl` instruction in *PowerPC* branches to the address contained in `lr` and then sets `lr` to the address of the instruction following `lr`. Since VLIW tree code is not sequential, i.e. the address of the operation following the translation of `bcrl` is not the address that should be placed in `lr`, some other means is needed to set the new value of `lr`. One possibility is to add a 32-bit immediate field to the VLIW version of this instruction, with the field containing the proper *PowerPC* address. However this wastes instruction encoding space, hence we prefer to allow the VLIW to perform indirect jumps through at least one other register. This could be a new special purpose register for this purpose, e.g. `lr2`, or the VLIW could be allowed to perform indirect jumps through any GPR. Since registers R32-R64 are not architected, this would provide many possible choices at any given point.

The *PowerPC* also has many branch instructions which decrement `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 `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.

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 we prefer to support 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 `ai` instruction in particular is heavily used to increment loop index variables. Alas, `ai` not only bumps the value in its destination GPR, it also sets the carry value in `CA`. Unless this `CA` value can be renamed, `ai` 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. [1] It would perhaps be better if compilers used the `cal` instruction instead of `ai` in such cases, but no matter, as a large body of code exists using `ai`. To get around this problem, we place the value of `CA` in an extender bit of the target GPR for an operation such as `ai` — if the `ai` 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

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

committed to the `CA` bit in the *PowerPC* `XER` register. If an `ai` 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.

Appendix E

Conversion of S/390 and x86 code into VLIW: Examples

Here are some code examples from *x86* and *S/390* to give a further flavor of our approach for **DAISY**.

```
         S/390 code                        RISC Primitives
A       L     r10,2892(0)                  l    r10=2892(0)
B       LH    r2,118(0)                    lh   r2=118(0)
C       MVI   552(0),4        (1)          li   r17=4
                              (2)          stb  r17,552(0)
D       STC   r2,288(r10,r2)  (1)          a    r17=r10,r2
                              (2)          stb  r2,288(r17)
E       BASR  R9,0                         la   r9=X'9DA'(rvpa)
                                           ; rvpa = Register containing virtual
                                           ;        address of current page
                                           ;        (Kept on cross page branches)
F       L     R9,1434(R9)                  l    r9=1434(r9)
G       LA    R6,4095(R9)                  la   r6=4095(r9)
                                           ; Result of "la" AND'ed
                                           ; with an "address mask" to
                                           ; implement 31 bit or 24 bit mode
H       L     R5,520(0)                    l    r5=520(0)
I       LCTL  R6,R6,36(R5)   (1)           l    r17=36(r5)
                             (2)           trap_priv
                                           ; Checks for supervisor state in
                                           ; special register
                             (3)           st_real r17,cntlreg6(rra)
                                           ; rra = Pointer to an area of VLIW real
                                           ;       memory used by emulator (VMM)
                                           ;       to keep control registers and
                                           ;       other data structures
J       L     R7,528(0)                    l    r7=528(0)
K       L     R8,548(0)                    l    r8=548(0)
        BCR   15,0                         nop     ; Assume a strongly consistent
                                           ;         memory system, not requiring
                                           ;         stop at a serializing op
L       L     R0,28(R10)                   l     r0=28(r10)
M       LTR   R0,R0                        cmp  cr0=r0,0
                                           ; Operations set 390 condition codes
                                           ; in 390 mode:  Exactly one among the
                                           ; eq,lt,gt,ov bits of the result
                                           ; cr field is set to 1
N       BNE   L1A30                        bf     cr0.eq,L1A30
O       MC    X'428'(0),7 (1)              l_real r17=cntreg8(rra)
                          (2)              and    r18=r17,256
                          (3)              trap   r18!=0
                                           ; Monitor call for event class #8
P       TM    114(r8),8   (1)              lb     r17=114(r8)
                          (2)              tmi    cr0=r17,8
                                           ; tmi performs S/390 TM function
                                           ; and sets condition code
Q       BZ    L13AA                        bt     cr0.eq,L13AA
R       CH    r0,118(r8)   (1)             lh     r17=118(r8)
                           (2)             cmp    cr0=r0,r17
S       BZ    L13AA                        bt     cr0.eq,L13AA
           ...

        L13AA:
T       CLI   540(r7),0    (1)             lb    r17=540(r7)
                           (2)             cmp   cr0=r17,0
U       BNE   L1D30                        bf    cr0.eq,L1D30
V       L     r3,36(r10)                   l     r3=36(r10)
W       LTR   r3,r3                        cmp   cr0=r3,0
X       BZ    L13DE                        bt    cr0.eq,L13DE
          ...
```

Figure E.1: Original *S/390* Code Fragment and Corresponding RISC Primitives.

```
VLIW's are separated by asterisks.
Small   letters indicate S/390 operations executed out of order.
Capital letters indicate S/390 operations committed in    order.

         S/390 code                   VLIW code
(If an exception occurs, restart at S/390 Instruction A)
A       L     r10,2892(0)             l    r10=2892(0)     ; VALID_ENTRY
B       LH    r2,118(0)               lh   r2=118(0)
c       MVI   552(0),4    (1)         li   r17=4
e       BASR  R9,0                    la   r9'=X'9DA'(rvpa)
h       L     R5,520(0)               l    r5'=520(0)
j       L     R7,528(0)               l    r7'=528(0)
k       L     R8,548(0)               l    r8'=548(0)
o       MC    X'428'(0),7  (1)        l_real r17''=cntreg8(rra)
                                      b    v1
*********
v1:    (If an exception occurs restart at C)
C       MVI   552(0),4       (2)      stb  r17,552(0)
d       STC   r2,288(r10,r2) (1)      a    r17=r10,r2
i       LCTL  R6,R6,36(R5)   (1)      l    r17'=36(r5')
f       L     R9,1434(R9)             l    r9''=1434(r9')
l       L     R0,28(R10)              l    r0'=28(r10)
o       MC    X'428'(0),7    (2)      and   r18=r17'',256
p       TM    114(r8),8      (1)      lb    r17'''=114(r8')
r       CH    r0,118(r8)     (1)      lh    r17''''=118(r8')
t       CLI   540(r7),0      (1)      lb    r17'''''=540(r7')
v       L     r3,36(r10)              l     r3'=36(r10)
                                      b     v2
**********
v2:     (If an exception occurs, restart at D)
D       STC   r2,288(r10,r2)  (2)     stb  r2,288(r17)
E       BASR  R9,0                    lr   r9=r9' (dead)
F       L     R9,1434(R9)             lr   r9=r9''
G       LA    R6,4095(R9)             la   r6=4095(r9'')
m       LTR   R0,R0                   cmp  cr0'=r0',0
p       TM    114(r8),8       (2)     tmi   cr0''=r17''',8
r       CH    r0,118(r8)      (2)     cmp   cr0'''=r0',r17''''
t       CLI   540(r7),0       (2)     cmp   cr0''''=r17''''',0
w       LTR   r3,r3                   cmp   cr0'''''=r3',0
                                      b     v3
************
v3:     (If an exception occurs, restart at H)
H       L     R5,520(0)               lr   r5=r5'
I       LCTL  R6,R6,36(r5)    (2)     trap_priv
I       LCTL  R6,R6,36(R5)    (3)     st_real r17',cntlreg6(rra)
J       L     R7,528(0)               lr   r7=r7'
K       L     R8,548(0)               lr   r8=r8'
L       L     R0,28(R10)              lr   r0=r0'
M       LTR   R0,R0                   lr   cr0=cr0'
N       BNE   L1A30                   bf   cr0'.eq,L1A30
O       MC    X'428'(0),7      (3)    trap r18!=0
P       TM    114(r8),8               lr   cr0=cr0''
Q       BZ    L13AA                   bt   cr0''.eq,L13AA
R       CH    r0,118(r8)              lr   cr0=cr0'''
S       BZ    L13AA                   bf   cr0'''.eq,exit1
    exit1:
           ...
    L13AA:
T       CLI   540(r7),0               lr   cr0=cr0''''
U       BNE   L1D30                   bf   cr0''''.eq,L1D30
V       L     r3,36(r10)              lr   r3=r3'
W       LTR   r3,r3                   lr   cr0=cr0'''''
X       BZ    L13DE                   bt   cr0'''''.eq,L13DE
     exit2:                        exit2:
                                      ...
```

Figure E.2: Parallelized VLIW code (25 390 instrucs in 4 VLIWs = 6.25 *S/390* instrucs per VLIW)

```
                  x86 code                               RISC primitives
                                                         ; (all 16 bit ops)
A                 push    bp            (1)              ; st     bp,-2(sp,ss)
                                        (2)              ; ai     sp=sp,-2
B                 mov     bp,sp                          ; lr     bp=sp
C                 push    ds            (1)              ; stseg  ds,-2(sp,ss)
                                        (2)              ; ai     sp=sp,-2
D                 mov     ax,[bp+6]                      ; l      ax=6(bp,ss)
E                 test    ax,1                           ; andil. cr0,scr=ax,1
F                 jnz     short loc_0240                 ; bf     cr0.eq,loc_0240
G                 push    ax             (1)             ; st     ax,-2(sp,ss)
                                         (2)             ; ai     sp=sp,-2
H                 call    sub_0116       (1)             ; li     t1=return1
                                         (2)             ; st     t1,-2(sp,ss)
                                         (3)             ; ai     sp=sp,-2
                                         (4)             ; b      sub_0116
   return1:       ............                           ;
K  loc_0240:      mov     es,ax                          ; descr_lookup es=ax
                                                         ;  searches descriptor
                                                         ;  lookaside buffer
                                                         ; l      t1=data_0391(0,es)
L                 cmp     word ptr es:data_0391e,454Eh
                                         (2)             ; cmp    cr0=t1,0x454e
M                 je      short loc_0245                 ; bt     cr0.eq,loc_0245
N                 mov     es,word ptr cs:[2](1)          ; l      t1=2(0,cs)
                                             (2)         ; descr_lookup es=t1
O                 mov     cx,es:data_0068e               ; l      cx=data_0068(0,es)
P  loc_0241:      jcxz    short loc_0242     (1)         ; cmp    cr1=cx,0
                                             (2)         ; bt     cr1.eq,loc_0242
Q                 mov     es,cx                          ; descr_lookup es=cx
R                 cmp     ax,cx                          ; cmp    cr0=ax,cx
S                 je      short loc_0243                 ; bt     cr0.eq,loc_0243
T                 mov     cx,es:data_0001e               ; l      cx=data_0001e(0,es)
U                 cmp     ax,es:data_0014e    (1)        ; l      t1=data_0014e(0,es)
                                              (2)        ; cmp    cr0=ax,t1
V                 jne     loc_0241                       ; bf     cr0.eq,loc_0241
W                 mov     ax,es:data_0015e               ; l      ax=data_0015e(0,es)
X                 jmp     short loc_0245                 ; b      loc_0245
                  ...........
HH loc_0245:      mov     cx,ax                          ; lr     cx=ax
II                pop     ds                  (1)        ; l      t1=0(sp,ss)
                                              (2)        ; ai     sp=sp,2
                                              (3)        ; descr_lookup ds=t1
JJ                leave                       (1)        ; lr     sp=bp
                                              (2)        ; l      bp=0(sp,ss)
                                              (3)        ; ai     sp=sp,2
KK               retf    2                    (1)        ; l_lr   lr1=0(sp,ss)
                                              (2)        ; l      t2=2(sp,ss)
                                              (3)        ; ai     sp=sp,6
                                              (4)        ; descr_lookup cs=t2
                                              (5)        ; b_across_page cs,lr1
```

Figure E.3: An *x86* routine, with corresponding RISC primitives.

```
VLIW's are separated by asterisks.
Small   letters indicate x86 operations executed out of order.
Capital letters indicate x86 operations committed in    order.

                x86 code                          VLIW Code
v0:  Restart at A if interrupted                ; (All 16-bit ops)
A               push    bp       (1)            ; st     bp,-2(sp,ss)
A               push    bp       (2)            ; ai     sp=sp,-2      /* DEAD         */
B               mov     bp,sp                   ; ai     bp=(old)sp,-2 /* WAS lr bp=sp */
C               push    ds       (1)            ; stseg  ds=-4((old)sp,ss)
C               push    ds       (2)            ; ai     sp=(old)sp,-4
D               mov     ax,[bp+6]               ; l      ax=4((old)sp,ss)
h               call    sub_0116 (1)            ; li     t1=return1
n               mov     es,word ptr cs:[2] (1)  ; l      t1'=2(0,cs)
ii              pop     ds       (1)            ; lr     t1''=ds /* DEAD */
ii              pop     ds       (3)            ; lr     ds'=ds
kk              retf    2        (1)            ; l_lr   lr1=0((old)sp,ss)
kk              retf    2        (2)            ; l      t2=2((old)sp,ss)
                                                ; b      v1
************************************************
v1:  Restart at E if interrupted
E               test    ax,1                    ; andil. cr0,scr=ax,1
k               mov     es,ax                   ; descr_lookup es'=ax
n               mov     es,word ptr cs:[2] (2)  ; descr_lookup es''=t1'
kk              retf    2        (4)            ; descr_lookup cs'=t2
                                                ; b      v2
************************************************
v2:  Restart at F if interrupted
F               jnz     short loc_0240          ; bf     cr0.eq,loc_0240'
G               push    ax       (1)            ; st     ax,-2(sp,ss)
G               push    ax       (2)            ; ai     sp=sp,-2  (dead)
H               call    sub_0116 (2)            ; st     t1,-4((old)sp,ss)
H               call    sub_0116 (3)            ; ai     sp=(old)sp,-4
                                                ; b      sub_0116
                                                ;##### (ILP = 8/3 = 2.7) #####
                                                ;
K  loc_0240':   mov     es,ax                   ; lr     es=es'
L               cmp     word ptr es:data_0391e,454Eh (1)
                                                ; l      t1=data_0391e(0,es')
o               mov     cx,es:data_0068e        ; l      cx'=data_0068e(0,es'')
                                                ; b      v3
************************************************
v3:  Restart at L if interrupted                ; cmp    cr0=t1,0x454e
L               cmp     word ptr es:data_0391e,454Eh (2)
p               jcxz    short loc_0242     (1)  ; cmp    cr1=cx',0
q               mov     es,cx                   ; descr_lookup es'=cx'
r               cmp     ax,cx                   ; cmp    cr0'=ax,cx'
                                                ; b      v4
************************************************
v4:  Restart at M if interrupted
M               je      short loc_0245          ; bt     cr0.eq,loc_0245'
N               mov     es,word ptr cs:[2] (2)  ; lr     es=es''
O               mov     cx,es:data_0068e        ; lr     cx=cx'
P               jcxz    short loc_0242     (2)  ; bt     cr1.eq,loc_0242
Q               mov     es,cx                   ; lr     es=es'
R               cmp     ax,cx                   ; lr     cr0=cr0'
S               je      short loc_0243          ; bt     cr0'.eq,loc_0243
T               mov     cx,es:data_0001e        ; l      cx=data_0001e(0,es')
U               cmp     ax,es:data_0014e   (1)  ; l      t1=data_0014e(0,es')
w               mov     ax,es:data_0015e        ; l      ax'=data_0015e(0,es')
                                                ; b      v5
HH loc_0245:                                    ; loc_0245':
                mov     cx,ax                   ; lr     cx=ax
II              pop     ds       (2)            ; ai     sp=(old)sp,2 (dead)
II              pop     ds       (3)            ; lr     ds=ds'
JJ              leave            (1)            ; lr     sp=bp
JJ              leave            (2)            ; lr     bp=bp (dead)
JJ              leave            (3)            ; ai     sp=(old)sp,4 (dead)

KK              retf    2        (3)            ; ai     sp=(old)sp,10
KK              retf    2        (4)            ; lr     cs=cs'
KK              retf    2        (5)            ; b_across_page cs',lr1
                                                ;##### (ILP = 13/5 = 2.6) #####
************************************************
v5:  Restart at U if interrupted
U               cmp     ax,es:data_0014e   (2)  ; cmp    cr0=ax,t1
                b       v6
************************************************
v6:  Restart at V if interrupted
V               jne     loc_0241                ; bf     cr0.eq,loc_0241
W               mov     ax,es:data_0015e        ; lr     ax=ax'
HH              mov     cx,ax                   ; lr     cx=ax'
II              pop     ds       (2)            ; ai     sp=sp,2 (dead)
II              pop     ds       (3)            ; lr     ds=ds'
JJ              leave            (1)            ; lr     sp=bp
JJ              leave            (2)            ; lr     bp=bp (dead)
JJ              leave            (3)            ; ai     sp=(old)sp,4 (dead)
KK              retf    2        (3)            ; ai     sp=(old)sp,10
KK              retf    2        (4)            ; lr     cs=cs'
KK              retf    2        (5)            ; b_across_page cs',lr1
                                                ;##### (ILP = 24/7 = 3.4) #####
************************************************
```

Figure E.4: Parallelized VLIW code: 24 *x86* instructions in 7 VLIWs (3.4X speedup), on path A-F, K-X, HH-KK.