DAISY: Dynamic Compilation for 100% Architectural Compatibility

Kemal Ebcioğlu and Erik R. Altman
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598
{kemal,erik}@watson.ibm.com

Abstract

Although VLIW architectures offer the advantages of simplicity of design and high issue rates, a major impediment to their use 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 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 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.

Keywords: INSTRUCTION-LEVEL PARALLELISM, OBJECT CODE COMPATIBLE VLIW, DYNAMIC COMPILATION, BINARY TRANSLATION, SUPERSCALAR

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 keynote speech [11], 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 [10]. One has been the object code translation approach (e.g. [20, 21, 22, 23]), 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 using it as a branch target address, 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 ([22, 23]), 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 binary compatible with the old one by hardware design, but does not run with the best performance on old binaries, ([20, 21]), 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 [9, 14, 19]. 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? We attack this problem in the current paper.

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 compiler techniques for attaining high ILP are unacceptably slow for dynamic parallelization, which requires real-time performance from a compiler, in order to make the overhead imperceptible to the user. To this end, we describe a new, significantly faster parallelization technique. 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 give an example illustrating the new fast dynamic compilation algorithm used by **DAISY**. Next, various architectural features to support high performance translation are described. We then describe the dynamic translation mechanism whereby **DAISY** runs the old software with minimal hardware support. Next we discuss the mapping mechanisms from the old code to VLIW code and back. We then provide some experimental results and conclude.

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 [20]. The base architecture could be any architecture, but we will be giving examples mostly from the *IBM PowerPC*.

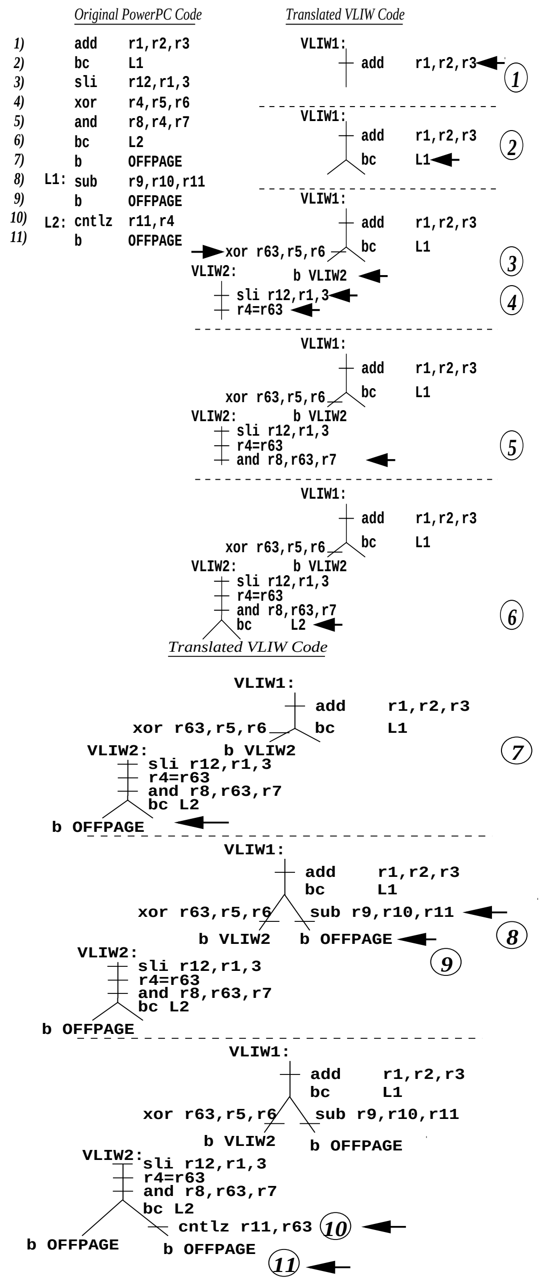
Traditional caching emulators may spend under 100 instructions to translate a typical base architecture instruction (depending on the architectural mismatch and complexity of the emulated machine). So caching emulators are very fast, but do not do much optimization nor ILP extraction. Traditional VLIW compiler techniques, on the other hand, extract considerable ILP at the cost of much more overhead.

The 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. Unlike traditional VLIW scheduling, **DAISY** examines each operation in the order it occurs in the original binary code, converting each into RISC primitives (if a CISCy operation). As each RISC primitive is generated, **DAISY** immediately finds a VLIW instruction in which it can be placed, while still doing VLIW global scheduling on multiple paths and across loop iterations and while maintaining precise exceptions.

Figure 1 shows an example of *PowerPC* code and its conversion to VLIW code [1]. This conversion uses the algorithm described in [6]. However, the discussion below should make the main points clear. We begin with four major points:

- Operations 1–11 of the original *PowerPC* code are scheduled in sequence into VLIW's. It turns out that two VLIW's suffice for these 11 instructions, yielding an ILP of 4, 4, and 3.5 on the three possible paths through the code.
- 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

[1]For an animated version of this example, please visit `http://www.research.ibm.com/daisy`.



```
1)       add    r1,r2,r3
2)       bc     L1
3)       sli    r12,r1,3
4)       xor    r4,r5,r6
5)       and    r8,r4,r7
6)       bc     L2
7)       b      OFFPAGE
8)  L1:  sub    r9,r10,r11
9)       b      OFFPAGE
10) L2:  cntlz  r11,r4
11)      b      OFFPAGE
```

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

non-renameable destinations are placed at the end of the last VLIW on the current path. In this way, precise exceptions can be maintained.

For example, assume an external interrupt occurs immediately after `VLIW1` finishes executing, and prior to the start of `VLIW2`. The interrupt handler is just an incrementally compiled version of the standard *PowerPC* interrupt handler. Hence it looks only at *PowerPC* architected registers. These registers appear as if instruction 2, `bc` has just completed execution and control is about to pass to instruction 3, `sli`. (The instruction address register is even set to the proper *PowerPC* address of the `sli`, as will be described in Section 3.3.) Since `VLIW2` expects the value of the speculatively executed `xor` to be in non-architected register `r63`, it is not a valid entry point for the interrupt handler to return to: the value of `r63` is not saved by the *PowerPC* interrupt handler, and hence its value may be corrupted upon return from the interrupt. Thus the **VMM** must either (1) interpret *PowerPC* instructions starting from instruction 3, `sli`, until reaching a valid entry into VLIW code (which depends only on values in *PowerPC* architected registers), or (2) it must compile a new group of VLIW's starting from instruction 3, so as to create a valid entry point.

- VLIW instructions are trees of operations with multiple conditional branches allowed in each VLIW [5]. 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.

Note that this approach works even if data is misinterpreted as code and aggressively scheduled. As we discuss more in Section 2.1, no side effects to architected resources occur until the point in the original program at which they would occur. However, we can and do limit scheduling of data by halting translation at unconditional branches. If the address following such a branch is indeed code, it must be reachable in one or more of the following ways: (1) by some other path on the page, (2) some branch from another page, or (3) by an indirect jump. In case (1), the address will be properly translated when encountered on the other path. For cases (2) and (3), the address will be translated when the executing program actually attempts to branch to that address. In Section 3.2, we discuss VLIW branch instructions which facilitate cases (2) and (3).

As this example suggests, the instruction set of the *migrant VLIW architecture* should be a superset of the *base architecture* for efficient execution. [2] 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 Section 3.

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. [13, 12, 20, 5, 8]). Furthermore, data intermingled with code, and (unknowingly) scheduled as code must not cause errant execution. In order to keep the paper self contained, we briefly mention here how we address these problems.

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 (in the present implementation, operations that set a

[2] Complex and/or infrequent instructions such as Load Multiple Registers or Decimal Divide can be decomposed into simpler primitives.

non-architected register such as r63 will be speculative). 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  r63<-[Addr]
    bc    L1                 bc    L1
    load  r3<-[Addr]         copy  r3<-r63
```

Register r63 is not architected in the *base architecture*. Hence when it is loaded, no exception occurs, even if this load would normally cause a page fault or segmentation violation. Instead the exception tag bit of r63 is set. If the bc falls through, the attempt to copy r63 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. This mechanism allows the **VMM** to move loads aggressively above conditional branches without changing the exception behavior of the original program.

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 dynamic 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.

Finally, for this paper, we assume that the *base architecture* machine is a "standard" register oriented machine with sequential semantics. This need not be the case. For example, [7] outlines how the **DAISY** approach could be implemented for a *Java* stack machine. If the *base architecture* is a VLIW with parallel semantics, where `x = y; y = x;` means exchange x and y, the code must first be converted to sequential code by introducing temporaries. The exchange then becomes the sequential code,
`t = x;   x = y;   y = t;`.

3 Page and Address Mapping Mechanisms

In this section, 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 **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 2. 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 2, 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 first mapping in the Sections below. However, the second approach can be extended in a straightforward manner to accomplish the actions described.

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 $n \times N +$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&`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 ($n_0 \times N +$ VLIW_BASE) is not mapped, and therefore a "VLIW translation missing" exception occurs, which is handled by the **VMM**. The **VMM** creates a translation for the *base architecture* physical page at physical address n_0, and makes the corresponding translated code area page map 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 ($n \times N +$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 2, 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`).
- The VLIW translation of `0x2100` 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` and writes it into the VLIW 16K page frame at (say) VLIW real address `0x02000000 - 0x02003fff`,
- Finally the **VMM** 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.

We emphasize that the contents of the upper portion of the VLIW Real Memory are managed by the **VMM** and are not paged. When room is needed for newly translated code, the **VMM** discards the least recently used translation to make space. At some point it may be desirable to page translations to disk so as to save them for later use, but initially we wish to develop an architecture that requires no base software or system changes.

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 Section 2, so this is possible.) The **VMM** then performs interrupt actions required by the *base architecture*, such as putting the address of the interrupted `base architecture` instruction in a specific register. 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.

We still need to address the problem 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 3.2. 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

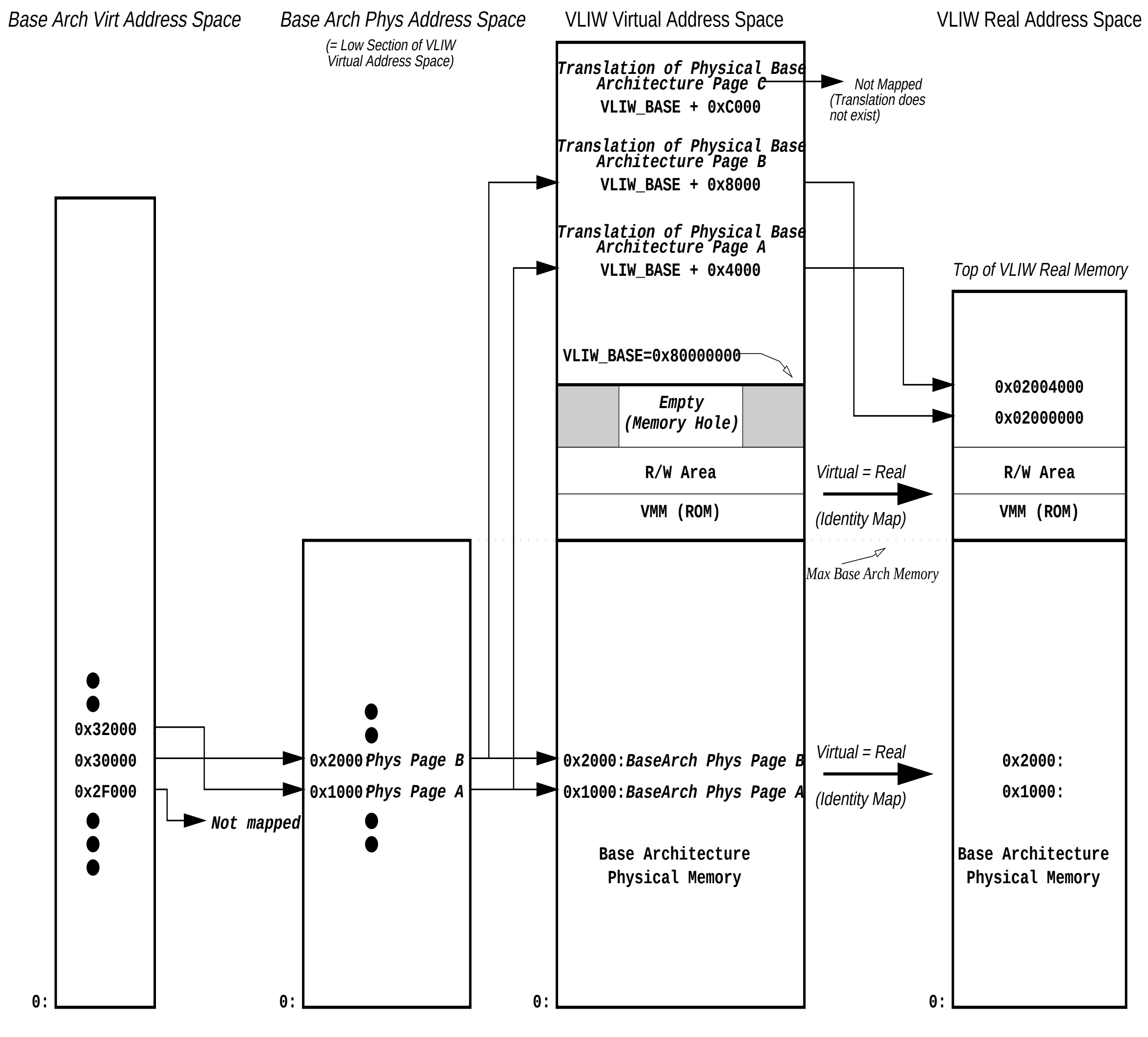

Base Arch Virt Address Space
Base Arch Phys Address Space
(= Low Section of VLIW Virtual Address Space)
VLIW Virtual Address Space
VLIW Real Address Space
Translation of Physical Base Architecture Page C
VLIW_BASE + 0xC000
Not Mapped (Translation does not exist)
Translation of Physical Base Architecture Page B
VLIW_BASE + 0x8000
Translation of Physical Base Architecture Page A
VLIW_BASE + 0x4000
Top of VLIW Real Memory
VLIW_BASE=0x80000000
0x02004000
0x02000000
Empty (Memory Hole)
R/W Area
Virtual = Real
(Identity Map)
VMM (ROM)
Max Base Arch Memory
0x32000
0x30000
0x2F000
Not mapped
0x2000 Phys Page B
0x1000 Phys Page A
0x2000:BaseArch Phys Page B
0x1000:BaseArch Phys Page A
0x2000:
0x1000:
Base Architecture Physical Memory
0:

Figure 2: VLIW Address Space Layout

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.

A final major concern is self modifying code. 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*, ≥ 2 bytes for *S/390*, ≥ 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.

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 3.2.

3.2 VLIW Branch Instructions

We described in Section 3.1 how one could find the translation of a *base architecture* instruction at physical address n, by branching to VLIW virtual address $n \times N + \mathsf{VLIW\_BASE}$. 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, a fact with which cross-page branches must deal. There are multiple ways of implementing cross-page branches, and we first describe a high-performance alternative. We then describe two lower performance and cheaper alternatives.

High Performance Branches

The VLIW primitive to perform a cross-page branch is:
`(GO_ACROSS_PAGE reg offset)`
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 here is 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).

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.)

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 cross-page branch, ITLB, 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.

```
  ; reg + offset is translated to physical address n.
  ; The VLIW real address for VLIW virtual address
  ; n*N + VLIW_BASE is then placed in lr1.
(LRA reg offset (lr1))

  ; Goto VLIW real addr in lr1. Check if valid entry
(GO_ACROSS_PAGE lr1)
```

Simpler Cross Page Branches

As noted earlier, 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.

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

  ; Load ptr to real addr of VLIW code into lr1.
(LOAD_REAL t1 0 (lr1))

  ; Goto VLIW code. Make no valid entry chk
(GOTO lr1)
```

For additional simplicity, we could even give up the ITLB and simulate a big direct mapped ITLB in VLIW real memory by software. In many cases the operations for doing the cross-page branch may be moved up into previous VLIW's, and their latencies hidden.

3.3 Finding Exception-Causing Instruction in Base Architecture

As we mentioned in Section 3.1, 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.

A **Virtual Page Address (VPA)** register is maintained. The **VPA** contains the address of the current page in the original code, and is updated in the translated code whenever a cross-page branch is executed. 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 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. For example, the offset within a page could be kept in a 10-bit field in each VLIW instruction. (10 bits assumes a 4096 byte page aligned on a 4-byte boundary.)

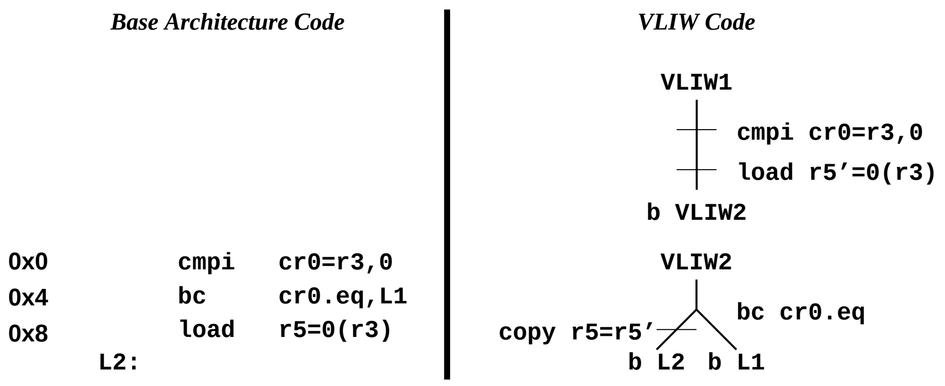
If the VLIW has an exception semantics where the entire VLIW appears not to have executed, whenever an error condition is detected in any of its parcels, then the offset identifies where to continue from in the base code. Interpreting a few base instructions may be needed before identifying the interrupting 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 “VLIW's") so that all parcels that logically preceded the exception causing one have executed when an 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 extra fields, 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. This scheme relies on the fact that the entry point of the group of VLIW's, 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. 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 VLIW's 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: **VPA** $+$ `VLIW1`$_{\text{offset}}/4$, if the code has $4\times$ 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.



```
0x0        cmpi   cr0=r3,0
0x4        bc     cr0.eq,L1
0x8        load   r5=0(r3)
      L2:
```

```
cmpi cr0=r3,0
load r5'=0(r3)
bc cr0.eq
```

Figure 3: Finding the *base architecture* instruction responsible for an exception

4 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 the **SPECint95** benchmarks, as well as a few AIX utilities, and an Erastothenes’ Sieve program for finding prime numbers (a Stanford integer benchmark).

Since our implementation of **DAISY** 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* [3] for simulating our VLIW machine on the RS/6000. During transitions between VLIW’s, 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 the current implementation. But since many 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 implementation.

Table 1 contains the pathlength reductions achieved on the **SPECint95** benchmarks for an 8-issue machine. These numbers were obtained by executing the benchmarks with the full **SPECint95** reference inputs, with a resulting execution of more than 500 billion *PowerPC* operations, or about 200 billion VLIW instructions, as can be discerned from the mean infinite cache pathlength reduction of 2.5. (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.

The VLIW machine has primitives similar to the *PowerPC*, but with 64 integer and floating point registers, rather than 32 in *PowerPC*. Operation latencies are the same as in the *PowerPC-604*. A total of 8 operations (out of which 4 can be memory accesses), and a total of 3 conditional branches (4-way branching) can be executed in each VLIW, which follows the tree instruction model. Efficient hardware implementations of the tree VLIW have been described elsewhere (e.g. [5]). 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.

Program	*PowerPC* Ins *per VLIW*	Average Size of Translated Page
compress	2.6	9K
gcc	2.3	24K
go	2.4	36K
ijpeg	2.7	14K
li	2.5	10K
m88ksim	2.1	14K
perl	2.4	11K
vortex	2.9	12K
MEAN	**2.5**	**16K**

Table 1: Pathlength reductions and code explosion moving from *PowerPC* to VLIW.

DAISY’s 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. Furthermore, we found that **DAISY** comes within 20% of a more traditional static VLIW compiler [16] which implements a large number of sophisticated optimizations and takes approximately 100,000 instructions to compile each instruction. By way of comparison, the *gcc* compiler executes an average of 65,000 RS/6000 instructions to generate each machine instruction in its output. Finally, our implementation is a research prototype intended for flexible experimentation. We expect to reduce our translation overhead significantly with straightforward tuning, and further with an eventual rewrite of the incremental compiler, when the de-

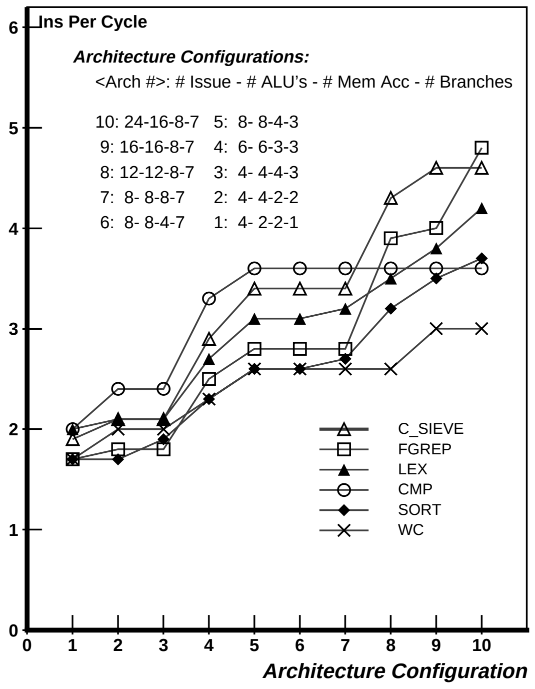


Figure 4: Pathlength reductions for Different Machine Configurations

sign matures. As a rough guess, under 1000 base instructions per base instruction seems achievable for implementing our aggressive compiler techniques.

Code explosion statistics for the benchmarks are also in Table 1. The average code expansion per actually translated page is 16K/4K = 4× (this is just the VLIW code size; empty wasted space on pages due to the 4× 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 translates all pages of the executable.

Figure 4 indicates, for several small utilities [3], 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, with 4-wide VLIW instructions, 2 of whose ops may be ALU operations, and 2 of which may be memory ops, with only 1 branch allowed per cycle. Performance diverges for a 24-wide high end machine, with ILP of close to 5 achieved for `fgrep`.

Another measure of interest is the number of crosspage branches executed. As discussed in Section 3.2 crosspage branches can be expensive, particularly in low-end implementations of the VLIW. Table 2 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.

[3] Time constraints did not permit us to obtain the corresponding numbers for **SPECint95**.

# 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 3: Overhead of Dynamic Compilation

Analysis of Compiler Overhead

Table 3 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 3 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. [6]. By *reuse*, we mean the number of times each translated instruction executes.

Table 4 gives an idea of the reuse factor for large programs such as those in the **SPEC95** benchmark suite. Table 4 indicates that they have very high reuse factors with a mean of over 500,000. The static code sizes were obtained on an *RS/6000* using the installed C compiler. [4].

A final example further supports the fact that dynamic compilation can be practical. 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.

5 Previous Work

Virtual machine concepts have been used for many years, for example in IBM's VM operating systems [2], 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

[4] We thank Mark Charney, Tom Puzak, and Ravi Nair for these numbers and constructing the tools with which to obtain them.

Program	*Branch Type* Direct	via `Linkreg`	via `Counter`	**Total**	Total VLIW's Exec / Total Crosspage Branches
compress	456M	1149M	96	**1605M**	*11*
gcc	771M	620M	69M	**1461M**	*10*
go	1325M	949M	39K	**2273M**	*15*
ijpeg	218M	179M	20M	**416M**	*60*
li	25M	31M	1M	**57M**	*468*
m88ksim	1326M	1072M	667M	**3065M**	*12*
perl	434M	376M	30M	**840M**	*25*
vortex	2076M	1515M	12M	**3603M**	*8*

Table 2: Number of crosspage branches in different benchmarks.

	Dynamic Ins Executed	Static Size *Ins Words*	Ins Reuse Factor
INTEGER			
go	80.8 billion	135,852	*594,764*
m88ksim	74.3 billion	84,520	*879,081*
cc1	34.1 billion	357,166	*95,474*
compress	46.4 billion	52,172	*889,366*
li	66.7 billion	67,084	*994,276*
ijpeg	68.5 billion	88,834	*771,101*
perl	48.7 billion	138,603	*351,363*
vortex	81.8 billion	212,052	*385,754*
FLOATING POINT			
tomcatv	19.8 billion	81,488	*243,003*
swim	23.2 billion	81,041	*287,324*
su2cor	24.9 billion	94,390	*263,911*
hydro2d	35.1 billion	95,668	*367,106*
mgrid	52.1 billion	83,119	*626,519*
applu	36.2 billion	99,526	*363,890*
turb3d	61.1 billion	90,411	*675,320*
apsi	21.2 billion	119,956	*176,690*
fpppp	98.0 billion	91,000	*1,076,624*
wave5	25.3 billion	120,091	*210,390*
MEAN	**49.9 billion**	**116,276**	***513,998***

Table 4: Reuse factors for **SPEC95** benchmarks

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 [10]. 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 [15] and Rau's work [18]), 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 Conte and Sathaye [4] who 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 by hardware to an internal VLIW-like representation at Icache miss time [9, 14, 19, 17] achieves a similar purpose, but requires complex Icache miss processing hardware and more hardware design investment, and does not allow sophisticated compiler techniques that can be implemented in software. Static translation of executable modules such as *FX!32*, was done in [20, 22, 23]. However, static translation does not address the problem of achieving 100% compatibility with the old architecture, including operating system code, debuggers, device drivers, etc.. 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.

6 Conclusion

We have described **DAISY**, an approach for making VLIW's and other novel architectures fully compatible with existing software for *base architecture(s)*. **DAISY** achieves this without any special hardware for emulation. Our approach could be important in the future for making an ultimate open system, where a single hardware chip can run multiple operating systems and *base architectures*. It is only necessary that the chip be constructed with an appropriate superset of the primitive operations of the multiple *base architectures*, e.g. *x86*, *PowerPC*, and *S/390*. A similar technique can be applied to aid migration to other new ILP architectures, that would otherwise break binary compatibility.

Although space constraints do not permit us to elaborate here, a novelty of **DAISY** is that it affords a practical means to achieve oracle parallelism (at high compilation cost): 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 VLIW's, 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 executes VLIW code solely. Further details may be found in [6]

Acknowledgements

We wish to thank the anonymous referees for their many useful suggestions. We also wish to thank Jaime Moreno, Mayan Moudgill, Arkady Polyak, and many of our other colleagues at IBM for their observations and discussion about this work.

References

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

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

[3] 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`

[4] 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.

[5] K. Ebcioğlu, *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. Available at `http://www.research.ibm.com/vliw`

[6] K. Ebcioğlu and E.R. Altman, **DAISY:** *Compilation for 100% Architectural Compatibility*, Report No. RC 20538, IBM T.J. Watson Research Center. Available at `http://www.watson.ibm.com:8080`

[7] K. Ebcioğlu, E.R. Altman, and E. Hokenek, *A JAVA ILP Machine Based on Fast Dynamic Compilation*, Proceedings of (IEEE MASCOTS) International Workshop on Security and Efficiency Aspects of Java, Eilat, Israel, January 9-10, 1997, Available at `http://www.watson.ibm.com:8080`

[8] K. Ebcioğlu 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. Available at `http://www.research.ibm.com/vliw`

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

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

[11] 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`

[12] 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.

[13] 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

[14] 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.

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

[16] M. Moudgill, J.H. Moreno, K. Ebcioğlu, E.R. Altman, S.K. Chen, and A. Polyak, *Compiler/Architecture Interaction in a Tree-Based VLIW Processor*, Report No. RC 20694, IBM T.J. Watson Research Center. Available at `http://www.watson.ibm.com:8080`.

[17] R. Nair and M. Hopkins, *Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups*, In Proceedings of the 24th Annual ACM/IEEE International Symposium on Computer Architecture, Denver, Colorado, June 2-4, 1997.

[18] B.R. Rau, *Dynamically Scheduled VLIW Processors*, Proc. MICRO-26, IEEE Press, December 1993.

[19] 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.

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

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

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

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