RC23262 (97775) March 27, 2000
Computer Science

IBM Research Report

Full System Binary Translation: RISC to VLIW

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

Full System Binary Translation: RISC to VLIW

Erik R. Altman
Kemal Ebcioğlu
IBM T.J. Watson Research Center

Abstract

We describe our experiences with **DAISY** (**D**ynamically **A**rchitected **I**nstruction **S**et from **Y**orktown). **DAISY** dynamically translates code for a RISC processor into code for an underlying VLIW processor. This translation is done piecewise — when a fragment of code is first encountered for execution, it is translated into code for the underlying VLIW machine and saved. This translation process begins with firmware executed by the RISC processor at boot time, continues through a full operating system boot, user login, and **X-Windows**, under which a variety of applications are run. The translated code is executed under simulation thus guaranteeing the correctness of the whole process.

In executing this translated code, numerous "difficult" situations emerge:

- Support of precise exceptions for the RISC processor in VLIW code that is drastically reordered. Such exceptions include page faults, alignment exceptions, and protection violations.
- Support for virtual address translation.
- Ability to deal with raw I/O devices such as networks, keyboards, and graphics adapters. These devices impose not only a semi-realtime requirement, but care must also be taken that reads and writes to I/O locations are not cached. Speculative accesses to I/O locations must be quashed.

This paper describes the novel ways in which **DAISY** deals with these and other difficulties, and presents some preliminary statistics measuring the effectiveness of full system binary translation.

1 Introduction

VLIW processors have traditionally suffered from compatibility problems — both with existing processors and between generations of VLIW processors. Dynamic binary translation as exemplified by our *DAISY* [8, 9, 10, 12] and Transmeta's *Crusoe* [15] provide a way around these problems by using code for an existing processor (*PowerPC* or *x86*) as a distribution format, and dynamically translating the code into the native VLIW form used by the underlying machine. In contrast to binary translation approaches such as IBM's *Mimic* [14], HP's HP3000 Emulator [4], Compaq's *FX!32* [5] and HP's *Dynamo* [2], both *DAISY* and *Crusoe* make binary translation invisible to the user by emulating the entire processor including "hard" system and privileged operations, exceptions, address translation, etc.. Tandem [1] and Apple [17] also perform full system translation like *DAISY* and *Crusoe* albeit with two significant differences: (1) Tandem/Apple binary translation was used to move from CISC to RISC platforms instead of from RISC/CISC to VLIW, and (2) both Tandem and Apple controlled the operating system run on their machines and used this fact to aid translation.

DAISY dynamically translates from *PowerPC* code to an underlying VLIW machine. We do not currently have hardware for **DAISY**. In order to test the full system capabilities of **DAISY** we simulate it on the bare hardware of an existing RISC machine, thus demonstrating that **DAISY** properly handles exceptions, virtual address translation, I/O with its semi-realtime requirements, and is not dependent on any operating system function. In doing so, we have also gained valuable experience and statistics on quantities important to dynamic binary translation such as code reuse, the amount of static code executed, how often certain architectural features such as *Little Endian Mode* are used (and hence how efficiently they must be supported), and the frequency at which code is overwritten. Overwriting code can signify self-modifying code. It can also occur when the operating system overwrites an existing code page with a new code page.

In this environment **DAISY** successfully boots and runs an unmodified RISC workstation. **DAISY** starts translating and firmware at the `RESET` location `0xFFF00100`. As each fragment of code is translated, it is saved for later use and executed. **DAISY** continues this process through the loading of the operating system (*AIX*), user login, the initiation of *X-Windows*, the execution of a variety of applications under *X-Windows* including *emacs, latex,* the *dbx* debugger, and a user mode version of the **DAISY** simulator.

In the remainder of this paper, we describe the **DAISY** VLIW, the challenges to such full-system binary translation, and the novel ways in which we have solved them. We also present some preliminary statistics on the effectiveness of binary translation.

2 DAISY VLIW

Our **DAISY** VLIW architecture is parameterizable. Its issue width could range from 4 to 16, with 64 integer registers, 64 floating point registers, and 64 condition register bits (16 condition register fields). The first 32 integer registers contain *PowerPC* values, while the upper 32 registers are used for speculative computation or are reserved for translator use. For example, `r3` always contains the value that `r3` would contain in a "normal" *PowerPC* program. The condition bits are similar with the first 32 corresponding to the *PowerPC* condition bits, and the second 32 being available for speculative and scratch computation.

Each **DAISY** instruction can have up to 4-16 operations depending on the machine width, of which half can be loads/stores. Given that **DAISY** is designed as a target for *PowerPC* instruction, its primitive operations are similar to *PowerPC* operations, with the exception that complex *PowerPC* operations such as *update* instructions and string operations are cracked into simpler **DAISY** primitives.

Each primitive ALU and memory operation in a **DAISY** instruction can be predicated on up to 3 condition bits, although predicate bits are associated with an VLIW instruction, not each operation. Hence there are not independent predicates for each operation.

Each **DAISY** VLIW instruction can branch to multiple targets. There are some restrictions on the location of targets, e.g., all in the same L1 ICache line. Such restrictions can be dealt with by an assembler which tries to find an instruction layout obeying these branching restrictions, and which duplicates instructions if necessary in order to accommodate them. This style of VLIW instruction was pioneered by Ebcioğlu [7].

Figure 1 shows the overall framework under which **DAISY** operates. Of particular importance is the fact

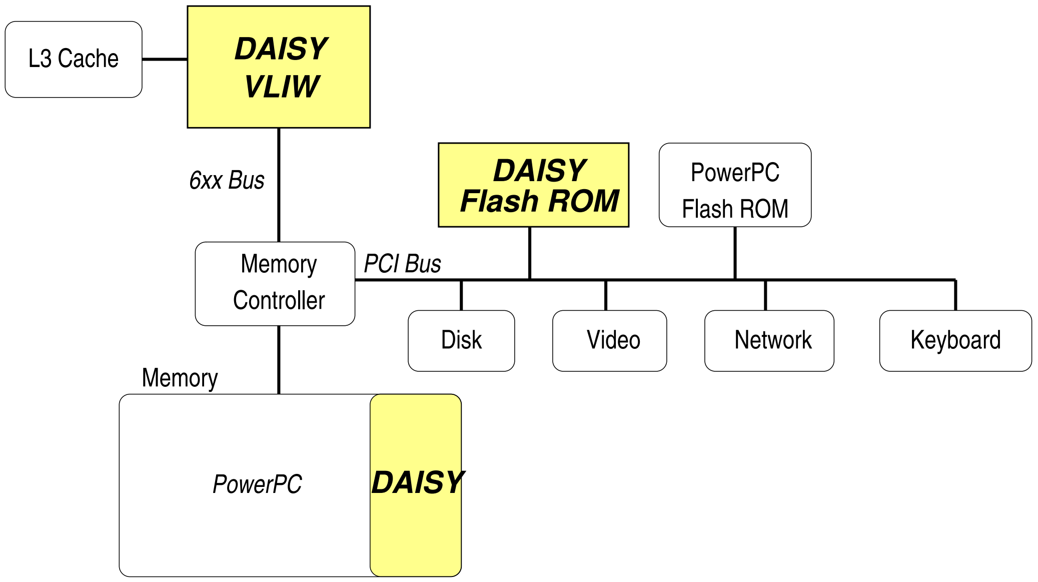

Figure 2: **DAISY** System.

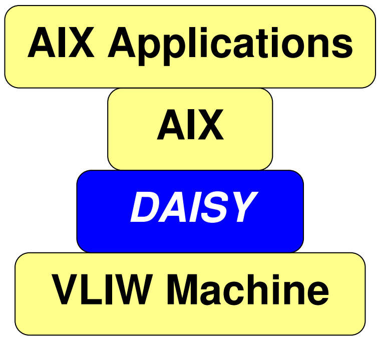

Figure 1: **DAISY** Schematic

that **DAISY** runs directly on the hardware, with no intervening operating system support. This provides the benefit of portability — any operating system from **AIX** to **Linux** to **MacOSX** should run without changes to the **DAISY** system (or the operating system), although to date, we have only tested **DAISY** with **AIX**.

Figure 2 shows a **DAISY** system. The shaded boxes differ from a traditional *PowerPC* system, while the unshaded boxes do not. As can be seen a traditional *PowerPC* processor is replaced by a **DAISY** VLIW processor. The translator and system software for **DAISY** are placed in the **DAISY** Flash ROM. (This code could also be burned into a ROM in the **DAISY** VLIW chip if a pin compatible replacement were desired for an existing and unmodified *PowerPC* system.)

The translator and system software are the only code which are compiled or written natively for **DAISY**. When the machine boots, **DAISY** begins executing code from a preassigned address in the **DAISY** Flash ROM. This boot code performs normal bootup tasks such as probing for the amount of available RAM memory and testing it. It then partitions the memory into *PowerPC* and **DAISY** sec-

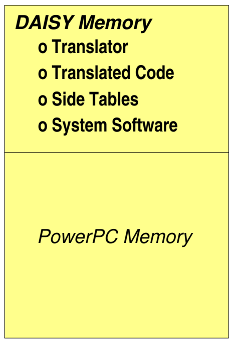

Figure 3: **DAISY** Memory Map

tions, with the *PowerPC* section normally being by far the larger of the two. Later during normal system operation, the *PowerPC* memory will look exactly like it would were a more traditional *PowerPC* processor in use. And, as explained in Section 3, *PowerPC* code will have no access to the **DAISY** code portion of memory, and indeed has no way of knowing that it exists.

The **DAISY** Flash ROM code then copies itself into the **DAISY** portion of memory, possibly uncompressing portions, although compression is probably not necessary, as the code size for the translator and system software is currently only 363 kbytes. At this point, translation of *PowerPC* code begins with the *PowerPC* entry point, `0xFFF00100`, which is contained in the (unmodified) *PowerPC* Flash ROM.

Translated code is kept in the **DAISY** portion of memory. Figure 3 depicts the **DAISY** memory map in slightly more detail. As noted, the **DAISY** portion also contains the **DAISY** system software and translator and other required tables.

3 Data Address Translation

In its current incarnation, **DAISY** is based on 32-bit *PowerPC* [13], under which 32-bit *PowerPC effective addresses* are normally mapped to 52-bit *virtual addresses* via the *PowerPC* segment registers. These 52-bit *virtual addresses* are in turn mapped to 32-bit *real addresses* via the page table or **TLB**. In many *PowerPC* implementations, such as the *604e* this mapping is accomplished with hardware support such as **TLB**s and hardware page-table walks. For high performance, **DAISY** must also provide support for *PowerPC* address translation. Such support includes hardware implementation of **TLB**s, the 16 *PowerPC* `segment registers`, the 16 *PowerPC* `BAT` registers, and the *PowerPC* `SDR1` register indicating the base address and size of the page table.

DAISY handles the bulk of address translation through its Data TLB (**DTLB**), which is illustrated in Figure 4.

There are particular wrinkles in dealing with data **TLB**s (**DTLB**s):

- *PowerPC* address translation is complicated by *Block Address Translation*, (**BAT**) which is controlled by 8 instruction and 8 data `BAT` registers. These `BAT` registers directly map 32-bit *PowerPC effective addresses* to 32-bit *real addresses* without need of a page-table or **TLB** lookup. The range of addresses translated by `BAT` registers is also much larger than the 4 Kbyte page used by the page table. Block sizes are powers of 2 and range from 128 Kbytes to 256 Mbytes.

  For simplicity of hardware and simulation, the **DAISY DTLB** always uses a 4 Kbyte granularity. Thus the 128 Kbyte region translated by a single `BAT` register requires 32 entries in the **DAISY DTLB**. (Accesses which use `BAT` translation turn out to be relatively rare in **RS/6000** firmware and **AIX** 4.1.5, thus supporting `BAT`s in the **DTLB** does not cause excessive overhead.)

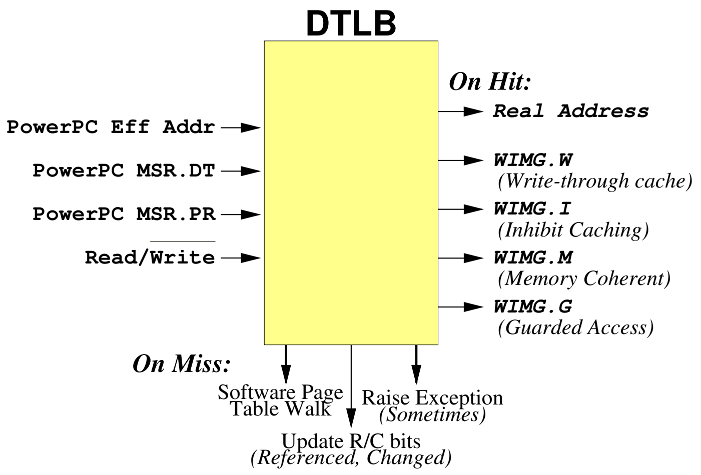

Figure 4: **DAISY** Data TLB (**DTLB**)

- *PowerPC* has multiple forms of memory protection, with certain protections associated with `BAT`s, segment registers, and page table entries. Many of these protections are based on whether the `privilege bit` (`MSR.PR`) in the *PowerPC* `MSR` is set. In other words, in `supervisor` mode certain pages may be read or written that may not be in non-`supervisor` mode. Even address translation may differ between `supervisor` and non-`supervisor` mode. As noted in the *PowerPC* manual, "*... a supervisor program can use the block address translation mechanism to share a portion of the effective address space with a user program (that uses page address translation for this area).*" [13]. Thus, it is essential to have `MSR.PR` as an input to the **DAISY DTLB**.

  Since pages (or `BAT`s) can be further marked as `read-only` or `read-write`, a `read-only` bit is made an additional part of the valid indicator for each **DTLB** entry. This bit is consulted only on a `store`, and results in an exception if the page is marked `read-only`.

- Not only *PowerPC* addresses translated via `BAT`s or the page table need to be remapped in **DAISY**. Even *PowerPC real accesses* need to be translated, i.e., loads and stores in which the `DT` (`Data Translate`) bit in the *PowerPC* `MSR` is off, and the effective address produced by a program is also the real address.

  The primary reason that such *real accesses* must be translated in **DAISY** is to prevent *PowerPC* code from accessing **DAISY** memory. As was illustrated in Figure 3, **DAISY** occupies the high portion of main memory, while *PowerPC* code occupies the low portion of main memory. If no protection were provided, malicious or buggy *PowerPC code* could corrupt **DAISY**'s memory.

  DAISY handles *PowerPC real accesses* by performing translation differently based on whether the *PowerPC* `MSR.DT` bit is set, as the `MSR.DT` input to the **DTLB** in Figure 4 suggests.

In **DAISY**, all *PowerPC* data references go through the **DTLB**. If *PowerPC* tries to access a page in **DAISY** memory, **DAISY** instead places an entry in the **DTLB** pointing to a special *guard page* in **DAISY** memory containing all `0xFFFFFFFF`s. Since all *PowerPC* references outside the legal *PowerPC* memory space go to this *guard page*, uncontrolled access to **DAISY** memory by *PowerPC* code is prevented. This *guard page* page is marked *read-only* in the **DTLB**. The net result is that memory locations beyond allowed *PowerPC* bounds appear not to exist — they always read `0xFFFFFFFF`, and do not change value when written. Thus any *PowerPC* functions probing for available memory find only that allotted to *PowerPC*.

This scheme does have the drawback that all of real memory is allocated between *PowerPC* and **DAISY** at the time a system is booted. If after running for a period of time, **DAISY** discovers it has more memory than needed for translations, it cannot give the extra back to *PowerPC*. Likewise if **DAISY** discovers that its translation area is too small to hold the working set of translations, it cannot regain memory controlled by *PowerPC*. These are interesting areas for future work, and it may be possible to make **DAISY** work with the *hot swapping* capability of many systems, in which new cards and resources can be added to a running system.

- It is also necessary to keep **DMA** and peripheral devices from accessing **DAISY** memory. In general this requires hardware support. However, we have found in our simulations of booting and running a **DAISY** machine that simply preventing *PowerPC* code from accessing the **DAISY** area is sufficient — since the *PowerPC* cannot tell that the **DAISY** memory exists, it does not tell **DMA** or peripheral devices to write to that memory.

- Another aspect of *PowerPC* address translation modeled by the **DAISY DTLB** are the *PowerPC* **WIMG** bits: **W** = Write any `stores` through the cache hierarchy, **I** = Inhibit any caching, **M** = Memory must be kept coherent for MP accesses, and **G** = Guarded memory, i.e., no speculative accesses are allowed. This **WIMG** information is kept in the **DTLB** with each page and provided to the cache and memory system by the **DTLB** on each access, as illustrated in Figure 4.

  I/O locations typically have the **I** and **G** bits set. For memory regions in which **I** or **G** is set, speculative loads must be quashed. Although **DAISY** does not currently do so, it may be worthwhile to re-translate without load speculation, portions of code which frequently access I/O locations.

- The *PowerPC* page table associates with each page a **Referenced** bit and a **Changed** bit. As the names suggest *PowerPC* sets these bits when a page is *referenced* or *changed*. These bits are typically used by an operating system to determine which pages to swap out, and whether any dirty data needs to be swapped out with them.

  Thus whenever an entry is brought into the **DTLB**, its **Referenced** bit is set in the *PowerPC* page table. If an entry is brought into the **DTLB** because of a `load`, then that entry is marked *read-only* in the **DTLB** even if the *PowerPC* protections indicate that the page is *read-write*. Then if a `store` subsequently occurs to that page, the result is a **DTLB** miss. Upon this miss, **DAISY** sets the **Changed** bit in the page table, and changes the **DTLB** entry to be *read-write*, as suggested in Figure 4. (These actions assume that the *PowerPC* protections allow them. If not, **DAISY** signals an exception.)

4 Exceptions

PowerPC provides precise exceptions except in rare cases such as certain types of machine check exceptions. Thus **DAISY** must also provide for precise *PowerPC* exceptions (and probably also precise native VLIW exceptions as well). How precise exceptions can be realized under dynamic binary translation has been described elsewhere [9]. Two essential points of that approach were (1) that values are placed in memory and in registers in the original program order and (2) that the address of the excepting instruction be readily computable based on a register containing the *effective address* of the current *PowerPC* page entry point and a side table containing the offset within a page of the *PowerPC* operation corresponding to the start of each VLIW instruction.

DAISY deals differently with synchronous and asynchronous exceptions. Asynchronous exceptions (such as *external exceptions* and *decrementer* exceptions in *PowerPC*) occur independently of current instruction execution. By contrast, synchronous exceptions (such as *page faults*, *protection violations*, *illegal instructions*, and *floating point* exceptions) occur as a result of executing particular instructions.

The subset of *PowerPC* instructions which can cause synchronous exceptions is known to the translator. This subset includes all `load` and `store` operations, all floating point operations, as well as `privileged` operations like `MTMSR`. (Even floating point `loads` and `stores` can trigger a *floating point unavailable* exception.) If an exception occurs,

1. The appropriate registers (e.g., `SRR0`, `SRR1`, `DSISR`, and `DAR`) are set. `SRR0` contains the *effective address* of the excepting *PowerPC* instruction. As outlined above, a register is kept with the *effective address* of the *PowerPC* page on which this exception occurred. The offset within the page of each VLIW instruction is saved in a side table. The **DAISY** system software finds the offending operation and its *PowerPC* address by interpreting *PowerPC* operations corresponding to the start of the VLIW instruction with the excepting operation and continues until the offending operation is reached. The values for the other registers are copied from known locations or computed based on values available at exception time such as the register values used in calculating the *effective address* being accessed by a `load` or `store`.

2. The translation of the *PowerPC* exception handler is jumped to if it exists. If it does not exist, then its entry is translated, and then jumped to.

In order to minimize exception processing overhead, we normally enable asynchronous exceptions only at group boundaries. Groups are what we term our units of translation. At the start and end of a group, there are no speculative values, i.e., integer registers `r32-r63`, float registers `fp32-fp63`, and condition register fields `cr8-cr15` are dead. Thus, if an exception occurs at a group boundary, no copying of register values or interpretation of *PowerPC* code is needed, as was the case for synchronous exceptions. Enabling asynchronous exceptions only at group boundaries has the additional advantage, that when the interrupt returns, there is generally a group already at the return point. If asynchronous exceptions were enabled everywhere, then eventually groups would need to begin at every point instruction in the original *PowerPC* code.

There is one problem with this scheme: it is possible that asynchronous exceptions are enabled (via the

EE bit of the *PowerPC* MSR register) in the middle of a group, and disabled prior to exiting the group. If such a group is repeatedly executed in a loop or if a long sequence of such groups is encountered, then asynchronous exceptions are effectively and incorrectly disabled.

To overcome this problem the **DAISY** hardware sets a bit, ASYNC_ON, whenever exceptions are enabled during a group. At the end of each group, if exceptions have been disabled, a counter is incremented to indicate the number of groups which have been executed without handling asynchronous exceptions. If exceptions are enabled, the counter is reset and any pending asynchronous exceptions are handled. In either case the ASYNC_ON bit is reset. If the counter ever reaches a threshold value, *PowerPC* operations are interpreted until exceptions are enabled and any pending asynchronous exceptions are handled.

DAISY uses 10 as a value of this threshold counter, and thus far we have seen very few cases where asynchronous exceptions occurred in the middle of a group. Informal observation suggests that such mid-group asynchronous exceptions occur about once every million group executions. Likewise, in informally examining this phenomenon, we have seen only one fragment of *PowerPC* code which caused such mid-group asynchronous exceptions. The fragment was in the lowest level **AIX** code for swapping in pages.

5 Machine Specific Details

Different chips implementing an architecture generally have small differences reflecting their microarchitectures. For example, implementations may differ in their set of *special purpose registers* (**SPR**s). Many *PowerPC* chips have processor version registers, which provide a value unique to each implementation. The *PowerPC 604e* has two hardware implementation dependent registers, HID0 and HID1. These registers can enable and disable caches, flush caches, enable branch prediction, enable serial execution of instructions, and several other low level micro-architectural functions.

Low level software, and in particular the firmware make use of these registers to perform certain power-on-self-tests, as well as to learn the type of chip on which they are running. On the *604e*, the cache disable function is used early on when accessing **I/O** locations with data address translation off (MSR.DT off). When translation is on, the **WIMG** bits can be used to disable caches when accessing such locations. However when data translation is off, there are no **WIMG** bits, and a default value of **WIMG = 0011** is used. Since caching is enabled, accessing **I/O** locations is problematic on *604e* if the cache disable function of the HID0 register is not used.

This being the case, it is useful to have the **DBT** system mimic a particular chip. For **DAISY**, we chose to mimic the *604e*.

6 Management of Translated Code

DAISY has a limited amount of space in which to keep translated code. Consequently, when this space is exhausted and a new page is translated, some existing translation must be discarded. In addition, the *PowerPC* ICBI (Instruction Cache Block Invalidate) instruction signals when a block of *PowerPC* code is no longer valid and should be flushed from the ICache For **DAISY**, ICBI signals that translations corresponding to the specified *PowerPC* block must be invalidated. ICBI must be used with self-modifying code in *PowerPC* and is more typically used when new code pages are created or swapped in.

In contrast to Dynamo's simple policy of destroying all translations when the translation cache is full [3], **DAISY** adopts a more sophisticated strategy. This complex strategy is useful in managing translations efficiently for both the `ICBI` case and the translation area full case. **DAISY** maintains several data structures to support this management:

- The `blk2grp` structure lists for each *PowerPC* code block, the translated groups which contain it.
- The `grp2blk` structure lists for each group, which *PowerPC* code blocks it contains.
- A two level cache maintains a mapping of *PowerPC* real addresses of code entry points to their corresponding **DAISY** code entry points similar to that employed by **Shade** [6]. We expect the first level to be kept as a hardware structure, with the second level containing all current mappings and managed in a way that is somewhat akin to a page table.

When an `ICBI` is encountered, `blk2grp` is used to immediately find what groups contain the specified block and invalidate them. The `grp2blk` list is also appropriately updated, as are the caches mapping *PowerPC* code addresses to VLIW code addresses.

The 2-level cache used to map *PowerPC* code addresses to translated VLIW code addresses serves two purposes. First, it helps speed lookup of recently accessed code pages. Not only is the first-level cache lookup faster in terms of mean number of operations than a search of the second level cache, but as well we expect it to have a smaller Dcache footprint. As is typical in cache hierarchies, **DAISY** employs

- A small, 128 entry, 4-way associative **L1** cache.
- A large, fully associative, **L2** cache. This **L2** cache is implemented as a software hash table and keeps a list of all current translations. It uses a form of generational garbage collection to determine which translations to purge when the translation area is full, or when the second-level cache itself is full.

  Since we have a fixed amount of space for the translator and its associated management software and structures, and since the ratio of the translated VLIW code size to the original *PowerPC* code size is not constant, it can happen that the second level cache is full, even when there is space left for additional translations. (Although we could start taking some of the space normally allotted for translated code in such cases, for simplicity reasons, we did not.)

  The age of each translation in the 2nd level cache is bumped by one each time there is a miss in the first level cache. Since misses to the first level cache are relatively infrequent, we expect the overhead required to bump the ages of every translation to be manageable. Furthermore, if the overhead ever becomes too high, later versions of **DAISY** could add an intermediate size cache between the current two levels, so as to reduce accesses to the second level cache even more.

  If the area to hold translated code becomes full, or likewise if the **L2** translation cache becomes full, then some translation must be invalidated. This translation is chosen randomly from among all of the oldest translations currently in the **L2** cache.

As noted above, we manage all code translations by *real address*. This way if the same real code is referenced at different effective or *virtual addresses*, only a single translation need be made. Likewise, the invalidation process is simplified by not having to know synonyms for each *effective* or *virtual* code address. Indeed, it is in general not known to the translator and simulator how effective and virtual code addresses are shared. Such information is typically maintained by the operating system, e.g., **AIX**, and is not conveyed to the *PowerPC* architecture, and hence is not conveyed to **DAISY**.

7 Results

We do not have **DAISY** hardware. Hence, we simulate **DAISY** on an RS/6000 Workstation running AIX 4.1.5 and containing a 200 MHz *PowerPC 604e* processor. This simulation is discussed in detail in [16]. In short, the simulation:

- Loads the **DAISY** translator and system software into a known fixed location in the real memory of the RS/6000.
- Beginning with the firmware at the reset location of the processor at `0xFFF00100`, translates *PowerPC* code into **DAISY** code page by page on an as-needed basis.
- Generates *PowerPC* code for each piece of **DAISY** code. This *PowerPC* code simulates the **DAISY** VLIW machine in manner similar to *Shade*. In addition to supporting **DAISY**'s 64 integer registers, the simulation code supports the semantics of the VLIW instructions including the predicated semantics of ALU and memory operations, and the proper ordering of ALU and memory operations within a VLIW instruction. The simulation also raises both synchronous and asynchronous exceptions and simulates the virtual address translation needed in **DAISY**. It likewise supports the **WIMG** bits described in Section 3, and in particular make certain that I/O accesses are not cached.
- Executes the simulation code for each fragment of **DAISY** code.

With all of these capabilities, it is possible to model **DAISY**'s full system binary translation. And even with the 35× to 200× slowdown due to simulation, **DAISY** successfully

- Executes the firmware on the RS/6000.
- Loads an unmodified version of AIX 4.1.5 from the hard disk.
- Loads device drivers for all devices in the system, in particular for the graphics adapter, the keyboard, the mouse, and the LAN adapter.
- Displays the `login` prompt.
- Runs `X-Windows` and a variety of applications on `X-Windows` including *emacs, LaTeX, dbx, ls, w, ping* and many others.

DAISY's success in this endeavor shows the viability of full system binary translation based solely on modeling a processor architecture.

Our simulations have provided us with a variety of (preliminary) data on aspects of system performance important to binary translation. Figures 5, 6, and 7 show a variety of information about instructions. For example, approximately 1.7 billion *PowerPC* instructions are executed in the firmware (as indicated by the first bar of the middle group in Figure 5). By the time that the `login` prompt is reached in AIX, almost 8 billion *PowerPC* instructions have been executed (as indicated by the second bar of the middle group in Figure 5). This 8 billion figure includes the 1.7 billion firmware instructions, hence 6.3 billion instructions are executed in bringing up AIX. The third bar in the series indicates that after logging in, bringing up X-Windows, and running

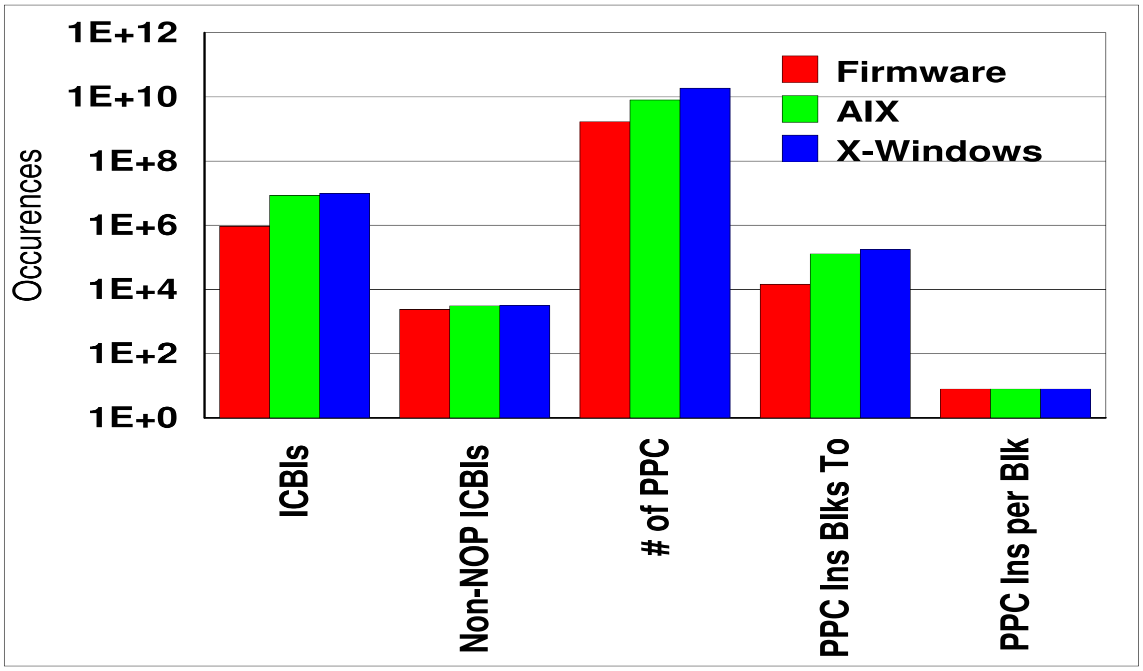

Figure 5: Instruction Counts in **DAISY**.

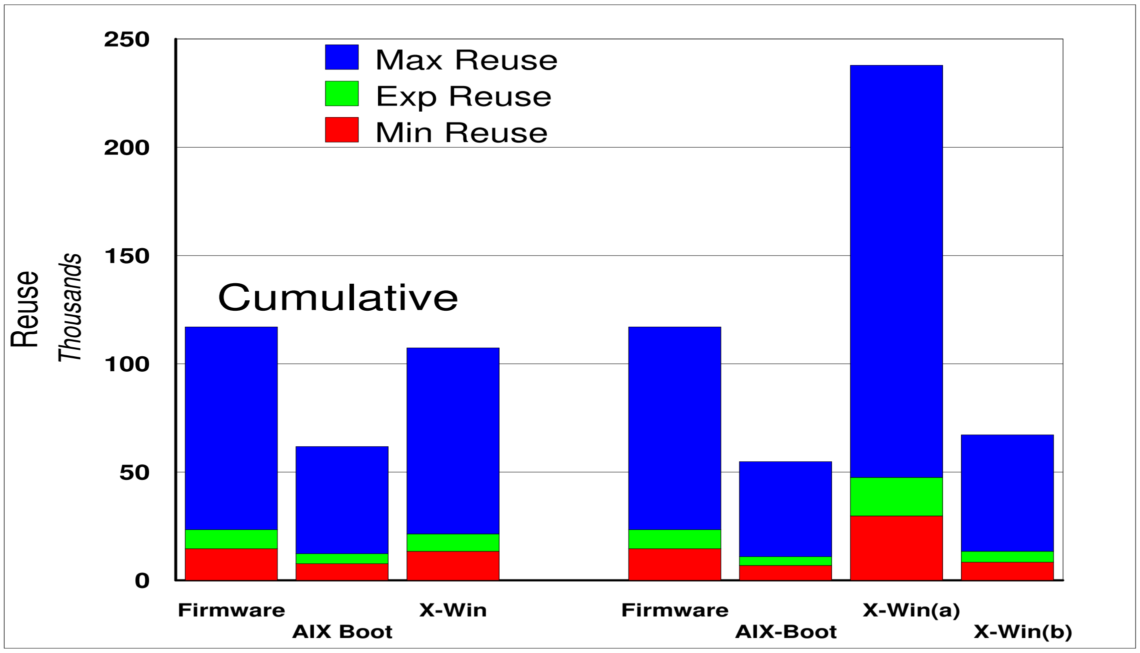

Figure 6: *PowerPC* Instruction Reuse

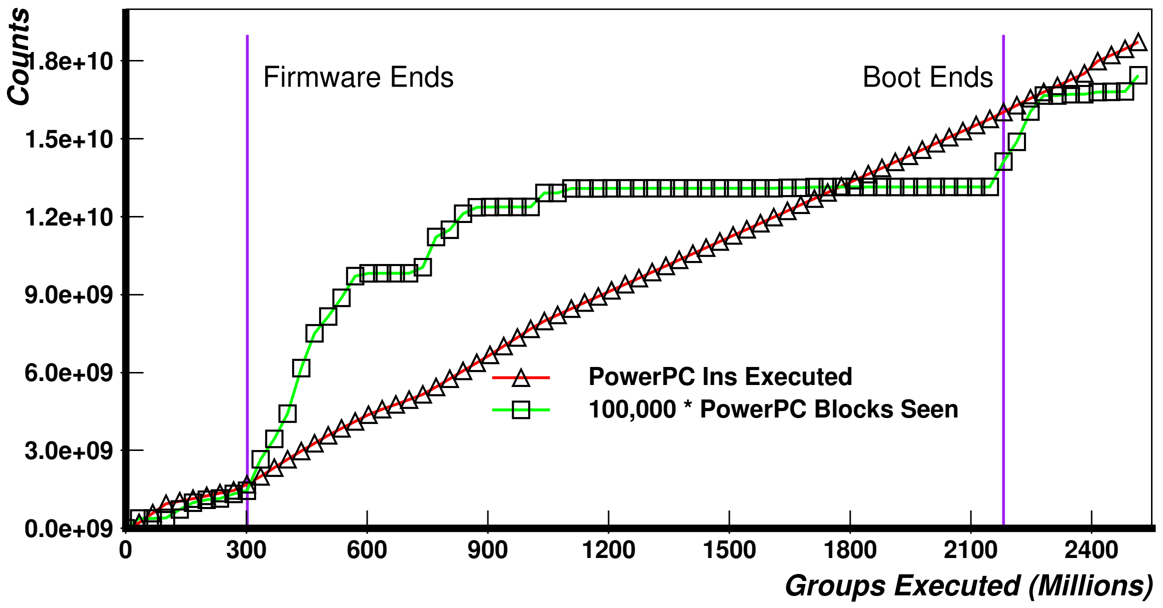

Figure 7: Instruction Counts over time in **DAISY**.

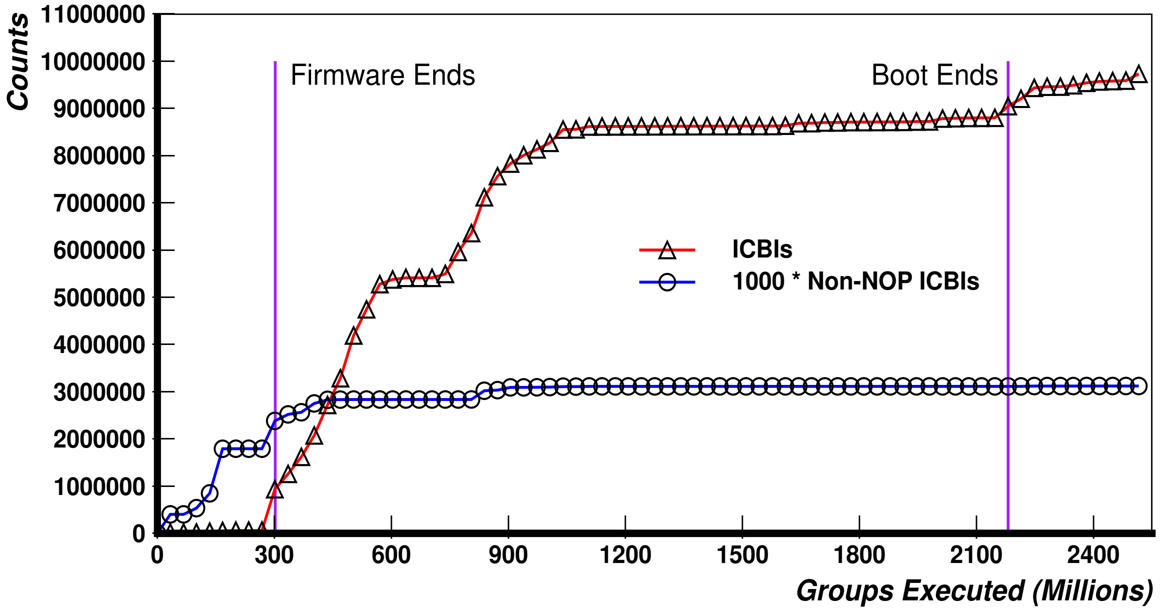

Figure 8: `ICBI`s over time in **DAISY**.

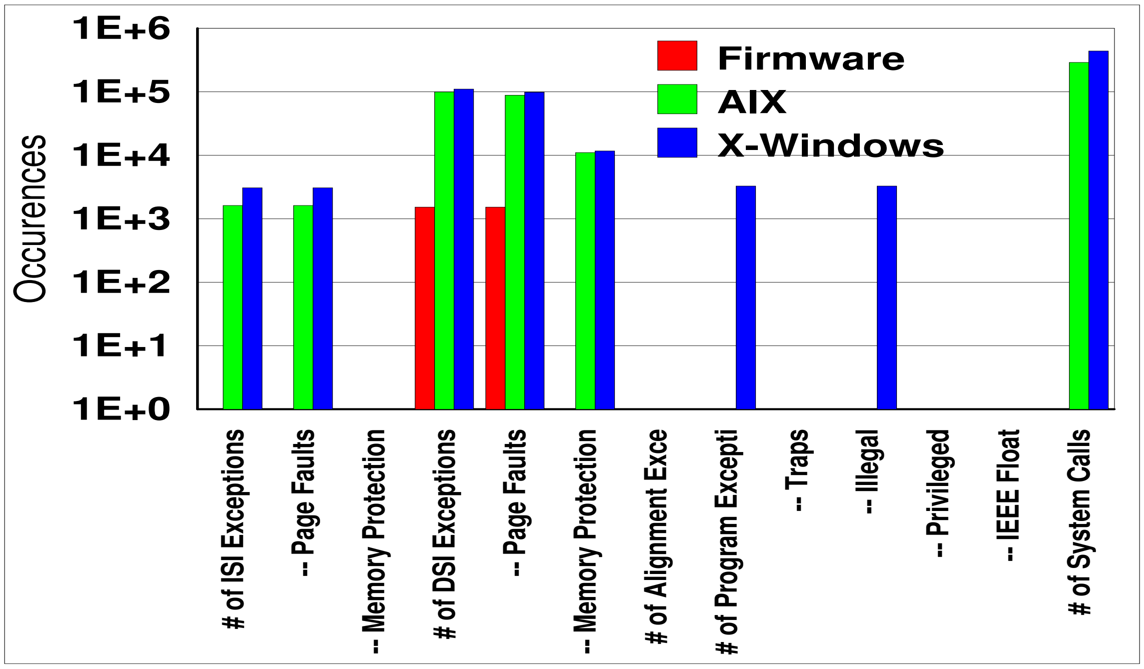

Figure 9: Number of Exceptions in **DAISY**.

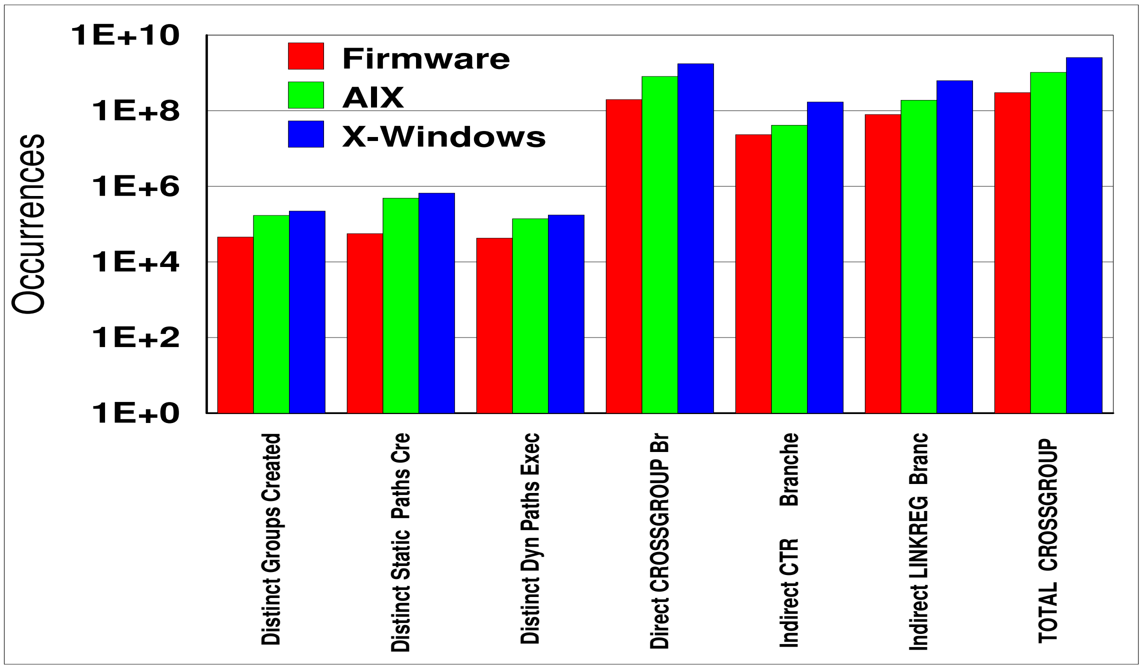

Figure 10: **DAISY** Translation Statistics

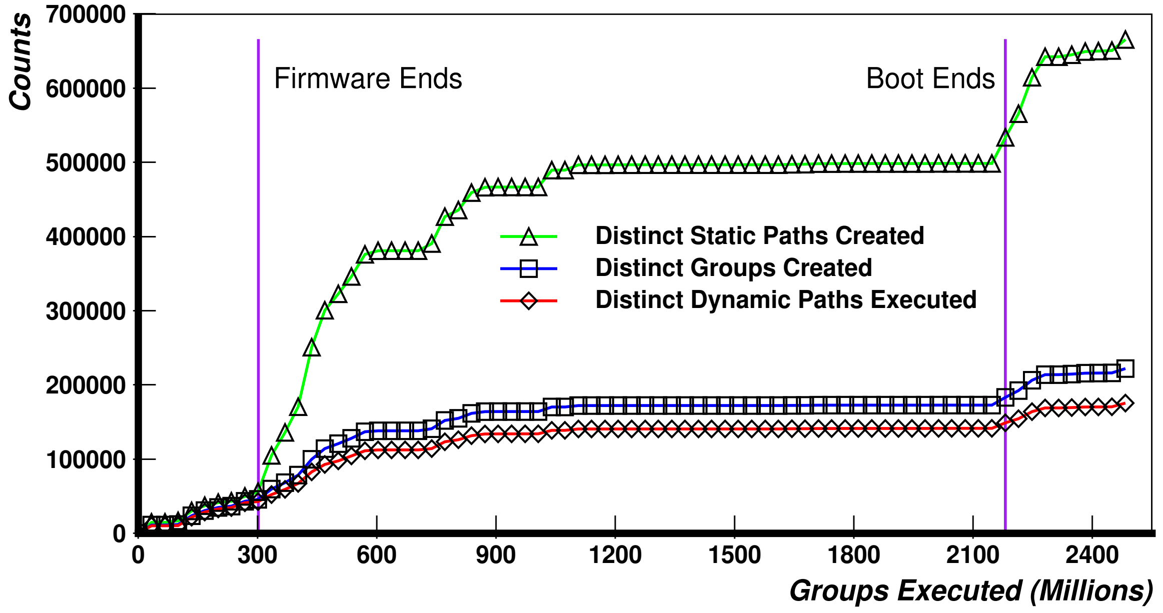

Figure 11: **DAISY** Translation Statistics over Time

a few utilities, almost 19 billion *PowerPC* instructions have been executed. As before, 19 billion is cumulative, hence the number of instructions spent in `X-Windows` and the utilities is approximately 11 billion. The utilities include `ls, w, grep, vmstat`, and `hostname`.

Figure 5 also provides an estimate of the amount of static *PowerPC* code executed. The **DAISY** simulator groups *PowerPC* memory into 8 *PowerPC* instruction chunks, and notes how many 8 instruction chunks have at least one of their 8 instructions executed [1]. In firmware, a little over 14,000 such chunks have at least one instruction executed. In other words, the static size of the code executed in firmware is between 14,000 and $8 \times 14,000 = 112,000$ *PowerPC* instructions. In other experiments we have found that somewhat over half the instructions in an 8 instruction block are executed on average, suggesting that perhaps 70,000 static instructions are executed at some point in the firmware. As with other quantities, Figure 5 also provides cumulative counts after AIX boots and after `X-Windows`. These value gives an indication of how many instructions need be translated by **DAISY**.

Figure 6 uses the number of static and dynamic *PowerPC* instructions in Figure 5 to estimate how many times each *PowerPC* instruction is executed. This is important for **DAISY**'s binary translation, as high reuse enables translation costs to be amortized, while low reuse does not. We estimate that **DAISY** takes approximately 4000 cycles to translate each *PowerPC* instruction to **DAISY** code. Thus any reuse substantially higher than 4000 allows amortization of translation costs. As just discussed, we do not have a precise count of instructions executed, only the number of 8-instruction blocks, thus Figure 6 shows the possible range of reuse. Figure 6 also shows the expected amount of reuse if on average, 5 of the 8 *PowerPC* instructions in a block are actually executed, which is what we have found in other experiments.

Cumulatively (the left part of Figure 6) reuse is expected to be about 23,000 in *firmware*, 12,000 in *firmware* plus *AIX*, and 21,000 in *firmware*, *AIX*, and *X-Windows*. Since AIX does not reuse any firmware code, it is possible to compute reuse during only the AIX portion of the boot. As shown on the right of Figure 6, expected reuse is 11,000. `X-Windows` *does* reuse some code from the AIX boot, although it is not clear how much. Thus Figure 6 shows reuse if no AIX code is reused in the second rightmost bar *(X-Win(a)),* and reuse if all AIX code is reused in the rightmost bar *(X-Win(b)).* The difference is dramatic: 47,000 versus 13,000 respectively.

Although these numbers reflect moderate reuse, Figure 7 suggests that they underestimate actual reuse. The plot of *PowerPC Blocks Seen* in Figure 7 shows that new *PowerPC* instructions are seen mostly at the start of each phase – *firmware, AIX,* and *X-Windows*. All of these phases are run in immediate succession. Although this constant invocation of new code may occur in some systems, we believe that it is not typical. As reflected by the last 60% of the AIX boot time in Figure 7, most applications reach a steady state after which they invoke little new code. Figure 7 also shows the dynamic number of instructions executed, which as expected increase linearly with time.

Returning to Figure 5 the number of `ICBI` (*Instruction Cache Block Invalidate*) instructions encountered is also depicted. The number of `ICBIs` is important because in general translations must be invalidated each time an `ICBI` is encountered. The larger the number of `ICBIs`, the more work (and time) that is required in the translator. Figure 5 indicates that firmware has almost 1 million `ICBIs` and by the time that `X-Windows` has run, this num-

[1]Maintaining counts on individual instructions requires prohibitively large space in our system.

ber jumps to almost 10 million. These numbers are not encouraging: 1 million `ICBI`s in 1.7 billion instructions of firmware corresponds to an `ICBI` every 1700 instructions.

Luckily, however, most of these `ICBI`s are `NOP`s. A `NOP ICBI` is an `ICBI` issued for an address for which there is currently no **DAISY** translation. As can be seen in Figure 5, there are vastly fewer `NOP ICBI`s than `ICBI`s as a whole. In firmware there are only 2381 `ICBI`s which are not `NOP`s, meaning that over 700,000 instructions occur between times when translations must be invalidated.

We have identified two reasons for the large number of `NOP ICBI`s. AIX appears to do an `ICBI` whenever a page is swapped in, regardless of whether it contains instructions. Likewise, it issues 128 `ICBI`s per page since the size of an `ICBI` block is normally 32 bytes and a page has 4096 bytes. In our current implementation, the first `ICBI` done on a page invalidates all **DAISY** translations on that page. As with *PowerPC* instructions, a more precise breakdown of the number of `ICBI`s executed over time is given in Figure 8. Most non-`NOP ICBI`s occur early — prior to starting the AIX portion of the boot. This suggests that in booting and starting `X-Windows`, there is no thrashing of the code working set.

Figure 9 indicates the number of times that a variety of *PowerPC* exceptions occur. Like Figure 5, these counts are cumulative going from firmware to AIX to `X-Windows`. The most common type of exception is the **System Call**, which is the way in which AIX kernel functions are normally invoked. AIX incurs almost 300,000 **System Call** while booting, while `X-Windows` makes about 150,000 more. Luckily **System Calls** are not really an "exceptional" condition in **DAISY**, but more akin to a cross-group branch to the **DAISY** translation of the *PowerPC* **System Call** code.

The next most common type of exception is **DSI** (**D**ata **S**torage **I**nterrupt), which has two common subtypes – page faults and protection violations, with page faults being by far the more common. On a data page fault, **DAISY** does a software page table walk and updates the **DTLB** as described in Section 3.

ISI (**I**instruction **S**torage **I**nterrupts) are orders of magnitude less frequent than **DSI**, and we have observed no instruction protection violations, only instruction page faults. No **Program Exceptions** occur until after `X-Windows` executes. All of the **Program Exceptions** are for **Illegal Instructions** – interestingly no traps are observed. All **Illegal Instructions** that we have examined come from old library code and are *Power* operations such as `lscbx` that are not supported in the *PowerPC* architecture, but seem to be emulated by AIX after are encountered on an **Illegal Instruction** exception. The relative infrequency of traps, illegal instructions, protection violations, as well as IEEE Floating Point Exceptions suggest that **DAISY** has some margin for laxity in dealing with them.

Figures 10 and 11 show several additional statistics related to how **DAISY** performs translation. The *Distinct Groups Created* is the number of *PowerPC* code fragments translated by **DAISY**. The *Distinct Static Paths Created* is the total number of exits from the code fragments translated by **DAISY**. For example, if **DAISY** translated a single basic block, it would have one *Distinct Group Created* and one *Distinct Static Path Created.* If **DAISY** translated a code fragment with a single conditional branch, it would have one *Distinct Group Created* and two *Distinct Static Paths Created.* Interestingly the ratio of *Distinct Static Paths Created* to *Distinct Groups Created* is about 3 for AIX and `X-Windows`, but only about 1 for the firmware. This difference is likely due to the fact that a great deal of the firmware is interpreted *Forth* code. This code has many indirect branches and other code which serializes in this version of **DAISY**. In general, we have concentrated on the functional correctness of **DAISY** to this point, and not on obtaining the highest performance.

Figure 10 also shows the number of *Distinct Dynamic Paths Executed*. Comparing this number to the number of *Distinct Static Paths Created* gives an idea of how much code is translated unnecessarily by **DAISY**. As expected from the fact that the firmware translations contain little branching, 75% of the translated firmware code is actually executed. However, this number drops to a bit less than $\frac{1}{3}$ for AIX and `X-Windows`.

Finally, Figure 10 shows the number of times that **DAISY** branches between translation fragments. These branches are further broken down into *direct, Link Register,* and *Counter Register* branches. These statistics give us an idea of where **DAISY** should be tuned for good performance. Since there are many more *direct* branches than those of the other types, it is especially important that they be executed efficiently. We re-emphasize, however, that these numbers were not obtained with the full aggressive optimization that we feel is possible in **DAISY**.

We reported **DAISY**'s potential performance in terms of ILP in [10]. For **Specint95**, an 8-issue version of **DAISY** achieves an average ILP of about 2.4 *PowerPC* instructions per cycle — including cache and TLB effects, as well as translation and other binary translation overhead. For **TPC-C**, **DAISY** still achieves an ILP of 1.5 *PowerPC* instructions per cycle. Of course **DAISY** normally issues many more than 2.4 or 1.5 instructions in each cycle, however many are speculative and their results are not used. In addition, some *PowerPC* operations are cracked into multiple **DAISY** primitives.

8 Conclusions

We have described **DAISY**, our full system binary translator. There are many difficulties in efficiently performing full system translation, including support for precise exceptions, support for virtual address translation, and the ability to deal with raw I/O devices. **DAISY** deals with all of these and successfully boots an RS/6000 workstation and runs `X-Windows` under translation. We have gathered a variety of preliminary statistics from this work, in particular the fact that translated code is invalidated at a very slow rate, thus allowing time to amortize the cost of translation. Other exceptions and difficult cases also occur with suitably low frequency as to make full system binary translation practical.

Acknowledgements

We thank **Michael Gschwind** and **Sumedh Sathaye** for their numerous useful suggestions and early critiques of this work.

References

[1] Kristy Andrews and Duane Sand, *Migrating a CISC computer Family onto RISC via Object Code Translation*, ASPLOS V (ACM SIGPLAN Notices, Vol. 27, No. 9), pp. 213-222 (Sep. 1992).

[2] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, *Transparent Dynamic Optimization: The Design and Implementation of Dynamo*, HP Labs Report 1999-78, June 1999, Available at http://www.hpl.hp.com/techreports/1999/HPL-1999-78.html

[3] Sanjeev Banerjia, Vasanth Bala, Evelyn Duesterwald, *Efficient Memory Management*

in a Practical Dynamic Optimizer, Proceedings of 1999 Workshop on Binary Translation, Newport Beach, California, October 1999.

[4] Arndt Bergh, Keight Keilman, Daniel Magenheimer, James Miller, *HP3000 Emulation on HP Precision Architecture Computers*, Hewlett-Packard Journal, December 1987.

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

[6] R. Cmelik and D. Keppel, *Shade: A Fast Instruction-Set Simulator for Execution Profiling*, Proceedings of SIGMETRICS'94 Conference on Measurement and Modeling of Computer Systems, Santa Clara, CA, May 1994, pp.128-137.

[7] 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.

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

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

[10] K. Ebcioğlu, E.R. Altman, S. Sathaye, and M. Gschwind, *Execution-Based Scheduling for VLIW Architectures*, Proceedings of EuroPar'99, Toulouse, France, September 1999, pp.1269-1280.

[11] 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.

[12] M. Gschwind, K. Ebcioglu, E. Altman, and S. Sathaye, *Binary Translation and Architecture Convergence Issues for IBM System/390*, Proceedings of ICS-2000, Santa Fe, New Mexico, May 2000.

[13] IBM and Motorola, *The PowerPC Microprocessor Family: The Programming Environments Manual for 32-Bit Microprocessors*, www.mot.com/SPS/PowerPC/teksupport/teklibrary/manuals/pem32b.pdf.

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

[15] Alexander Klaiber, *The Technology Behind Crusoe^{TM} Processors*, January, 2000. Available at `http://www.transmeta.com`

[16] Erik Altman and Kemal Ebcioğlu, *Simulation and Debugging of Full System Binary Translation*, Proceedings of the 13th International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV, August 8-10, 2000, pp.446-453

[17] Apple Computer, *50 Years of Computing Symposium*, IBM T.J. Watson Research Center, 1995.