RC22025 (98128) 18 July 2000 **Computer Science**

IBM Research Report

Dynamic Binary Translation and Optimization

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

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

Abstract

We describe a VLIW architecture designed specifically as a target for dynamic compilation of an existing instruction set architecture. This design approach offers the simplicity and high performance of statically scheduled architectures, achieves compatibility with an established architecture, and makes use of dynamic adaptation. Thus, the original architecture is implemented using dynamic compilation, a process we refer to as DAISY (Dynamically Architected Instruction Set from Yorktown). The dynamic compiler exploits runtime profile information to optimize translations so as to extract instruction level parallelism. This work reports different design trade-offs in the DAISY system, and their impact on final system performance. The results show high degrees of instruction parallelism with reasonable translation overhead and memory usage.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Instruction level parallelism (ILP) is an important enabler for high performance microprocessor implementation. Recognizing this fact, all modern microprocessor implementations feature multiple functional units and attempt to execute multiple instructions in parallel to achieve high performance. Today, there are essentially two approaches which can be used to exploit the parallelism available in programs:

dynamic scheduling In dynamic scheduling, instructions are scheduled by issue logic implemented in hardware at program run time. Many modern processors use dynamic scheduling to exploit multiple functional units.

static scheduling In static scheduling, instruction ordering is performed by software at compile time. This approach is used by VLIW architectures such as the TI C6x DSP [1] and the Intel/HP IA-64 [2].

Dynamic scheduling has been attractive for system architects to maintain compatibility with existing architectures. Dynamic scheduling affords the opportunity to use existing application code and execute it with improved performance on newer implementations.

While compatibility can be maintained, the cost of this can be high in terms of hardware complexity. This leads to error-prone systems with high validation effort. Legacy instruction sets are often ill suited to the needs of a high-performance ILP architecture which thrives on small atomic operations which can be reordered to achieve maximum parallelism. This is typically dealt with by performing instruction cracking in hardware, i.e., complex

instructions are decomposed into multiple, simpler micro-operations which afford higher scheduling freedom to exploit the available function units.

In contrast, static scheduling is performed in software during compilation time, leading to simpler hardware designs and allowing processors with wider issue capabilities. Such architectures are referred to as very long instruction word (VLIW) architectures. While VLIW architectures have demonstrated their performance potential in scientific code, some issues remain unresolved for their widespread adoption.

1.1 Historical VLIW Design

VLIW architectures have historically been a good execution platform for ILP-intensive programs since they offer a high number of uniform execution units with low control overhead. Its performance potential has been demonstrated by superior performance on scientific code.

However, extracting instruction level parallelism from programs has been a challenging task. Early VLIW architectures were targeted at highly regular code, typically scientific numeric code which spent the major execution time in a few loops which were highly parallelizable [3][4][5]. Integer code with control flow has been less amenable to efficient parallelization due to frequent control transfers.

In the past, adoption of VLIW has been hampered by perceived problems of VLIW in the context of code size, branch-intensive integer code and inter-generational compatibility.

Code-size issues have been largely resolved with the introduction of variable length VLIW architectures.

In previous work [6], we have presented an architectural approach called "tree VLIW" to increase the available control transfer bandwidth using a multi-way branching architecture. Multiway branching capabilities have been included in all recently proposed architectures. [7][8][9]

Much work has been performed in the area of inter-generational compatibility. In hardware implementations, scalable VLIW architectures allow the specification of parallelism independent of the actual execution target. The parallel instruction word is then divided into chunks which can be executed simultaneously on a given implementation [10][11][9]. In software implementations, dynamic rescheduling can be used to adapt code pages at program

load or page fault time to the particular hardware architecture. [12]

1.2 Compatibility Issues and Dynamic Adaptation

Years of research have culminated in renewed interest in VLIW designs, and new architectures have been proposed for various application domains, such as DSP processing and general purpose applications.

However, two issues remain to be resolved for wide deployment of VLIW architectures:

compatibility with legacy platforms A large body of programs exists for established architectures, representing a massive investment in assets. This results in slow adoption of any new system designs which break compatibility with the installed base.

dynamic adaptation Statically compiled code relies on profiling information gathered during the program release cycle and cannot adapt to changes in program behavior at run time.

We address these issues critical to widespread deployment of VLIW architectures in the present work. The aim is to use a transparent software layer based on dynamic compilation above the actual VLIW architecture to achieve compatibility with legacy platforms and to respond to dynamic program behavior changes. This effectively combines the performance advantages of a low complexity statically scheduled hardware platform with wide issue capabilities with the benefits of dynamic code adaptation.

The software layer consists of dynamic binary translation and optimization components to achieve compatibility and dynamic response. We refer to this approach as DAISY for **D**ynamically **A**rchitected **I**nstruction **S**et from **Y**orktown [7, 8].

The DAISY research group at IBM T.J. Watson Research Center has focused on bringing the advantages of VLIW architectures with high instruction-level parallelism to general purpose programs. The aim is to achieve 100% architectural compatibility with an existing instruction set architecture by transparently executing existing executables through the use of dynamic compilation. While we describe DAISY in the context of PowerPC implemen-

tation, this technique can be applied to any ISA. In fact, multiple ISAs, such as PowerPC, Intel x86 or IBM System/390, can be implemented using a single binary translation processor with appropriate personalization in the translation firmware.

In this paper, the "*base architecture*" [13, 14] refers to the architecture with which we are trying to achieve compatibility, e.g., *PowerPC*, *S/390* [15], or a virtual machine such as JVM [16]. The DAISY VLIW which emulates the old architecture we called the *migrant architecture*, following the terminology of [14]. In this paper, our examples will be from *PowerPC*. To avoid confusion, we will refer to *PowerPC* instructions as *operations*, and reserve the term *instructions* for VLIW instructions (each potentially containing many *PowerPC*-like primitive *operations*).

The remainder of the paper describes our approach in designing a high performance *PowerPC* compatible microprocessor through dynamic binary translation. A number of difficulties are addressed, such as self modifying code, multi-processor consistency, memory mapped I/O, preserving precise exceptions while aggressively re-ordering VLIW code, and so on. We give an overview of the DAISY target architecture in Chapter 2. Chapter 3 describes how the processor forms *tree groups* of *PowerPC* operations and the translation process into the DAISY architecture. Chapter 4 gives an overview of the adaptive group formation and instruction scheduling performed during translation. Chapter 5 gives experimental microarchitectural performance results. We discuss related work in chapter 6 and draw our conclusions in chapter 7.

Chapter 2

Architecture Overview

The target architecture for the DAISY binary translation system is a clustered VLIW processor. Each cluster contains 4 execution units and either one or two load/store units. Within a cluster dependent operations can be issued back-to-back, but when a cluster is crossed a one cycle delay is incurred. This basic cluster building block can be used to build configurations ranging from a single-cluster 4-issue architecture to a 16-issue processor consisting of four clusters.

The preferred execution target of the DAISY binary translation system is a clustered VLIW architecture with 16 execution units configured as 4x4 clusters (this corresponds to configuration 16.8 in table 5.1). This configuration offers high execution bandwidth while maintaining high frequency by limiting the size of bypassing wire length to local clusters.

The DAISY architecture defines execution primitives similar to the *PowerPC* architecture in both semantics and scope. However, not all *PowerPC* operations have an equivalent DAISY primitive. Complex *PowerPC* operations (such as "Load Multiple Registers") are intended to be layered, i.e., implemented as a sequence of simpler DAISY primitives to enable an aggressive high-frequency implementation. To this end, instruction semantics and data formats in the DAISY architecture are similar to the *PowerPC* architecture to eliminate data representation issues which could necessitate potentially expensive data format conversion operations.

The DAISY architecture provides extra machine registers to support efficient code scheduling and aggressive speculation using register renaming. Data are stored in one of 64 integer registers, 64 floating point registers,

and 16 condition code registers. This represents a twofold increase over the architected resources available in the *PowerPC* architecture. The architecture supports renaming of the carry and overflow bits in conjunction with the general purpose register. Thus, each register has extra bits to contain carry and overflow. This rename capability enables changes to global state (such as the carry and cumulative overflow information) to be renamed in conjunction with the speculative destination register until the point where the state change would occur in the original in-order *PowerPC* program.

The DAISY VLIW also has the usual support for speculative execution in the form of non-excepting instructions which propagate and defer exception information with renamed registers [6][17][13][14][18]. 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, speculative operations are identified by using a set of registers known to receive speculative operations as result register). A speculative operation that causes an error does not cause an exception, it just sets the exception tag bit of its result register and resets it if no exception is encountered. 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 mechanism allows the dynamic compiler to schedule instructions which may encounter exceptions aggressively above conditional branches without changing the exception behavior of the original program. [18][6][17]

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.

Efficient control flow operations are supported by the tree VLIW concept. Based on this architecture, each instruction has the ability to perform a 4-way multiway branch [6].

The cluster concept is also applied to the caches. L1 data caches are duplicated in each cluster, but stores are broadcast to all copies. A cache miss during a memory access will stall the entire processor until the appropriate data are retrieved from memory. While stall-on-use implementations can provide better CPI, they also result in more complex designs and impact

operating frequency.

Instruction caches are partitioned to achieve small physical cache sizes to achieve high clock frequency. Thus, each pair of execution units is connected to a slice of the instruction cache (termed "mini-Icache") which provides 1/8 of a VLIW instruction supplying a pair of ALUs.

Thus, in table 2.1, the L2 mini-ICache size is conceptually 128K instead of 1M, and the mini-ICache linesize is 256 bytes instead of 2K bytes. But because each such mini-ICache has to have a redundant copy of the branch fields to reduce the wire delays, the physical size is larger than the logical size. The instruction cache hierarchy supports history-based prefetching of the two most probable successor lines of the current cache line to improve instruction cache hit rate.

The DAISY VLIW also offers support for moving loads above stores optimistically, even when static disambiguation by the dynamic compiler is not possible. [6, 17, 19, 14, 20, 18]. If the current processor or some other processor alters the memory location referenced by a speculative load, between the time the speculative load is executed and the time that load result is committed, an exception is raised when the load result is committed. The DAISY software component then takes corrective actions, and may also retranslate the code which contained the mis-speculation. 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).

To avoid spurious changes in attached I/O devices, I/O references should not be executed out of order. While most such references can be detected at dynamic compile time (by querying the system memory map) and scheduled non-speculatively, it is not always possible to identify all load operations which refer to I/O space at dynamic compile time. To ensure correct system operation, speculative load operations to memory-mapped I/O space are treated as no-op by the hardware, and the exception tag of the result register of the load operation is set. When the load is committed, an exception will occur and the load will be re-executed — non-speculatively this time. Frequent occurrence of mis-speculations due to previously undetected I/O semantics of particular memory regions can be remedied by retranslating the relevant code.

The DAISY VLIW supports a memory hierarchy which is very similar to the emulated PowerPC architecture. This choice reduces the cost of im-

Cache	**Size / Entries**	**Line Size**	**Assoc**	**Latency**
L1–I	*32K*	1K	**8**	1
L2–I	*1M*	2K	**8**	3
L1–D	*32K*	256	**4**	2
L2–D	*512K*	256	**8**	4
L3	*32M*	2048	**8**	42
Memory	–	–	–	150
DTLB1	*128 entries*	–	**2**	2
DTLB2	*1K entries*	–	**8**	4
DTLB3	*8K entries*	–	**8**	10
Page Table	–	–	–	90

Table 2.1: Cache and TLB Parameters.

plementing operations accessing the memory. Our experience indicates that if memory operations have to be implemented using a sequence of operations to emulate the memory management structure of the base architecture, severe performance degradation can result. If multiple platforms are to be supported by a *common* core, then this core must provide an efficient way of emulating the memory management structure of all supported base architectures appropriately. In particular, it is important to ensure that frequently executed memory operations do not incur emulation burden, whereas updates to the memory map (such as changes to the page tables, segment registers, etc.) might require additional logic to update the DAISY VLIW core's native memory management system. A more detailed description of supporting multiple base architectures on a common core can be found in [15].

The DAISY VLIW processor contains hardware support for profiling in the form of an 8K entry *8-way* set associative (hardware) array of cached counters indexed by the exit point id of a tree region. These counters are automatically incremented upon exit from a tree region and can be inspected to see which tips are consuming the most time. They offer the additional advantages of not disrupting the data cache and being reasonably accurate. [21]

Chapter 3

Binary Translation Strategy

In this chapter, we describe the execution-based dynamic compilation algorithm used in DAISY. Compared to hardware cracking schemes such as employed by the Pentium Pro/II/III and POWER4 processors, software allows more elaborate scheduling and optimization than hardware, yielding higher performance. At the same time complex control hardware responsible for operation decomposition is eliminated from the critical path. Thus, a binary translation-based processor implementation is able to achieve maximum performance by enabling high frequency processors while still exploiting available parallelism in the code.

In looking forward to future high performance microprocessors, we have adopted the dynamic binary translation approach as it promises a desirable combination of (1) high frequency design, (2) greater degrees of parallelism, and (3) low hardware cost. Unlike native VLIW architectures such as the Intel/HP IA-64, (1) the dynamic nature of the compilation algorithm presented here allows the code to change in response to different program profiles and (2) compatibility between VLIW generations is provided by using *PowerPC* as the binary format for program distribution.

Dynamic optimization and response to changing program profiles is particularly important for wide issue platforms to identify which operations should be executed speculatively. Dynamic response as inherent in the DAISY approach offers significant advantages over a purely static compilation approach as exemplified by Intel and HP's *IA-64* architecture. Current compilers for *IA-64* rely purely on static profiling which makes it impossible to adapt to program usage.

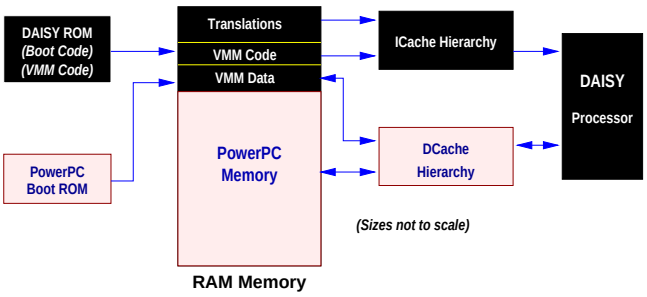

Figure 3.1: Components of a DAISY System.

In addition to performance limitations and technical hurdles, the *IA-64* static profiling approach requires that extensive profiling be performed on products by Independent Software Vendors (ISVs), and that they generate differently optimized executables corresponding to each generation of the processor. Given the reluctance of ISVs to ship code with traditional compiler optimizations enabled, it may be difficult to induce ISVs to take the still more radical step of profiling their code.

DAISY achieves hardware simplicity by bridging a semantic gap between the *PowerPC* RISC instruction set and even simpler hardware primitives, and by providing the ability to extract instruction-level parallelism by dynamically adapting the executed code to changing program characteristics in response to online profiling.

From the actually executed portions of the *base architecture* binary program, the dynamic compilation algorithm creates a VLIW program consisting of *tree regions*, which have a single entry (root of the tree) and one or more exits (terminal nodes of the tree). The choice of translation unit is described below.

3.1 System Operation

In DAISY, binary translation is a transparent process: As depicted in Figure 3.1, when a system based on the DAISY architecture boots, control transfers to the DAISY software system. We refer to the DAISY software component as VMM (**V**irtual **M**achine **M**onitor), since the software is re-

sponsible for implementing a virtual PowerPC architecture on top of the high-performance DAISY VLIW processor. The virtual machine monitor is part of DAISY system firmware, although it is not visible to the software running on it, much like microcode is not visible in a microcoded machine.

After DAISY VMM initialization, the DAISY VMM interpreter initiates the *PowerPC* boot sequence. In other words, a *PowerPC* system built on a DAISY architecture executes the same steps as it would on a native *PowerPC* implementation. Thus, the architected state of the virtualized *PowerPC* is initialized, and then *PowerPC* execution starts at the bootstrap address of the emulated *PowerPC* processor.

Similar to a native PowerPC system, a PowerPC boot ROM is located at the standard fixed address (0xfff00100). The PowerPC code in the boot ROM will be interpreted, translated and executed under control of the DAISY VMM. When the boot ROM initialization has completed after loading a kernel, and control passes to that kernel, the DAISY VMM in turn starts the interpretation and translation of the kernel, and after that has been initialized, of the user processes.

Actual instruction execution always remains under full control of the DAISY VMM, although the locus of control does not necessarily have to be *within* the VMM proper, i.e., the *interpreter, translator, exception manager,* or *memory manager*. If the locus of control is not within the VMM nucleus, it will be within VMM-generated translation *tree groups*. *Tree groups* are translated carefully so as to only transfer control to each other, or back to the VMM as part of a service request, such as translating previously untranslated code, or handling an exception.

This determinism in control transfer guarantees system safety and stability. No *PowerPC* code can ever access, modify or inject new code into the translated code. In fact, no code can even determine that it is hosted upon a layer of code implemented by the DAISY VMM.

3.2 Interpretation

When the DAISY VMM first sees a fragment of *PowerPC* code, it interprets it to implement *PowerPC* semantics. During this interpretation, code profile data is collected which will later be used for code generation. Each code piece is interpreted several times, up to a given interpretation thresh-

old, before it is translated into DAISY machine code. As *base architecture* instructions are interpreted, the instructions are also converted to execution primitives (these are very simple RISC-style operations and conditional branches). These execution primitives are then scheduled and packed into VLIW tree regions which are saved in a memory area which is not visible to the *base architecture*.

Any untaken branches, i.e., branches off the currently interpreted and translated trace, are translated into calls to the binary translator. Interpretation and translation stops when a stopping condition has been detected. (Stopping conditions are elaborated in section 3.4.) The last VLIW of an instruction group is ended by a branch to the next tree region.

Then, the next code fragment is interpreted and compiled into VLIWs, until a stopping condition is detected, and then next code fragment, and so on. If and when program decides to go back to the entry point of a code fragment for which VLIW code already exists, it branches to the already compiled VLIW code. Recompilation is not required in this case.

Interpretation serves multiple purposes: first, it serves as a filter for rarely executed code such as initialization code, which is executed only a few times and has low code re-use. Thus, any cost expended on translating such code would be wasted, since the translation cost can never be recuperated by the faster execution time in subsequent executions.

Interpretation also allows for the collection of profiling data, which can be used to guide optimization. Currently, we use this information to determine *tree group* formation used by the DAISY VMM. Other uses are possible and planned for the future, such as guiding optimization aggressiveness, control and data speculation, and value prediction [22].

3.3 Translation Unit

The choice of translation unit is critical to achieving good performance since scheduling and optimizations are performed only at the translation unit level. Thus, a longer path within a translation unit usually achieves increased instruction level parallelism.

In this work, we use tree groups as translation unit. As their name suggests, tree groups have a single entry point, and multiple exit points. No control flow joins are allowed *within* a tree group, control flow joins can

only occur on group transitions. Tree groups can span multiple processor pages, include register-indirect branches and can cross protection domain boundaries (user/kernel space).[1] We term the leaves of a tree "tip". Since *groups* are trees, knowing by which *tip* the group exited, fully identifies the control path executed from the *group* entrance (or tree root).

Using tree groups simplifies scheduling and many optimization algorithms, since there is at most one reaching definition for any value. Since any predecessor VLIW instruction dominates all its successors, scheduling instructions for speculative issue is simplified.

A downside of using tree groups is code space expansion due to tail duplication. This duplicated code beyond join points can result in VLIW code that is many times larger than the code for the base architecture. To counterbalance these effects, a number of design decisions have been made to reduce code expansion.

In its original version, DAISY used processor pages as unit of translation. When a page was first entered, a translation was performed for the entire code page, following all paths reachable from the entry point. As additional entry points were discovered, pages were retranslated to accommodate additional page entries.

Thus if execution reached a previously unseen page **P**, at address **X**, then all code on page **P** reachable from **X** — via paths entirely within page **P** — was translated to VLIW code. Any paths within page **P** that went offpage or that contained a register branch were terminated. At the termination point was placed a special type of branch that would (1) determine if a translation existed for the offpage/register location specified by the branch, and (2) branch to that translation if it existed, and otherwise branch to the translator. Once this translation was completed for address **X**, the newly translated code corresponding to the original code starting at **X** was executed.

This could lead to significant code expansion when paths were translated

[1] Crossing protection boundaries involves either an explicit change to a machine state register (e.g., move to machine state register operation in PowerPC), and or an implicit change, which is usually accompanied by a direct or indirect branch (e.g., system call, return from interrupt, program exceptions in PowerPC). The translation of these operations will need to set up a hardware global register representing the new memory protection state. Whenever it is beneficial, loads can be moved speculatively above logically preceding protection domain changes, as long as there is a corresponding load-verify/commit [23] executed in the original sequential order, using the new protection state.

which were rarely, if ever, executed. Thus, processor pages were appropriate for the original VLIW targets of modest width and large instruction caches. However, for very wide machines, page crossings and indirect branches limited ILP. [7, 8]

In contrast, tree groups are built incrementally to limit code expansion, and can cross pages, indirect branches and protection domains to overcome these ILP limitations and attack the code explosion problem. [24]

In related work about the BOA project, we have described the use of straightline *traces* corresponding to a single path through the executed code as translation units in the context of binary translation [25]. Unlike DAISY, which primarily focuses on the extraction of instruction-level parallelism, BOA was primarily focused on achieving very high processor frequency [26][27].

The achievable branch predictability puts an inherent limit on the achievable average *dynamic* group size, i.e., the number of instructions which will actually be executed before control exits a translation unit through any side exit. This limit is $\sum_{i=0}^{n} \textit{prediction accuracy}^{i} * \textit{basic block size}$, where n is a limit on the static trace length (expressed in basic blocks) to restrict code expansion. As a result, the dynamic window size for BOA was between 15 and 40 instructions for SPECint95 benchmarks, and 22 for TPC-C.

In comparison, DAISY achieves an average dynamic group size of about 100 instructions for SPECint95 benchmarks and 60 instructions for TPC-C. The achievable dynamic group size in DAISY is only limited by the tolerated code expansion, since a tree group can encompass an arbitrary number of paths.

Managing code duplication is a high priority for DAISY, since uncontrolled code duplication results in large working sets with bad locality, hence resulting in significant instruction cache miss penalties. DAISY includes a number of strategies to limit code size, such as (1) using initial interpretation to reduce the number of tree groups with low reuse, (2) initial translation with conservative code duplication limits, and (3) stopping points chosen to represent natural control flow joins to reduce code expansion across such points.

Looking at Figure 3.2(a), if the program originally took path `A` through a given code fragment (where `cr1.gt` and `cr0.eq` are both false), and if the same path `A` through the code fragment (tree region) is followed during the second execution, the program executes at optimal speed within the code fragment — assuming a big enough VLIW and cache hits.

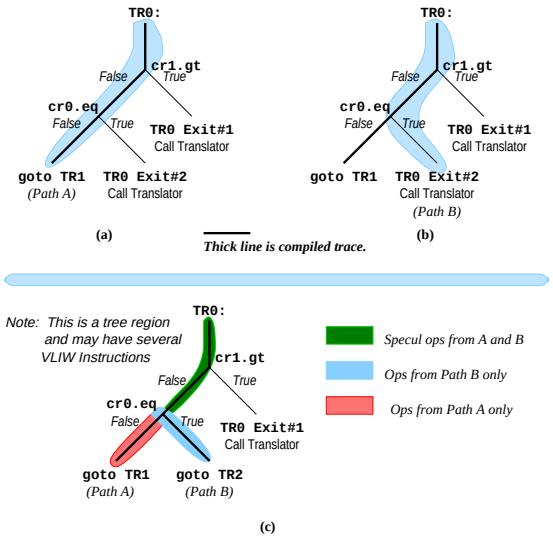

Figure 3.2: Tree regions and where operations are scheduled from different paths.

If at a later time, when the same tree region labeled `TR0` is executed again, the program takes a different path where `cr1.gt` is false, but `cr0.eq` is true (labeled path `B`), it branches to the translator, as seen in Figure 3.2(b). The translator may then start a new translation group at that point, or instead extend the existing tree region by interpreting *base architecture* operations along the second path `B` starting with the target of the conditional branch `if cr0.eq`. The *base architecture* operations are translated into primitives and scheduled into either the existing VLIWs of the region, or into newly created VLIWs appended to the region, as illustrated in Figure 3.2(c).

Assuming a VLIW with a sufficient number of functional units and cache hits, if the program takes path `A` or `B`, it will now execute at optimal speed within this tree region `TR0`, regardless of the path. This approach makes the executed code resilient to performance degradation due to unpredictable branches.

The compilation of the tree region is necessarily never complete. It may have "loose ends" that may call the translator at any time. For instance, as seen in Figure 3.2(c), the first conditional branch `if cr1.gt` in tree region `TR0` is such a branch whose off-trace target is not compiled. Thus, dynamic

compilation is potentially a never-ending task.

Tree groups can cross page boundaries and indirect branches, by making use of run time information to convert each indirect branch to a set of conditional branches. There is no limit on the number of pages a translated code fragment may cross. Only interrupts and code modification events are serializers.

3.4 Stopping Points for Paths in Tree Regions

Finding appropriate stopping points for a tree region is crucial for achieving high ILP, as well as for limiting the size of the generated VLIW code and translation time required for translation. Currently we consider ending a tree region at two types of operations:

- The **target** of a *backward branch*, typically a loop starting point, or
- a **subroutine entry** or **exit**, as detected heuristically through *PowerPC* `branch and link` or register-indirect branch operations.

Stopping (and hence starting) tree regions only at well-defined potential stopping points is useful, since if there was no constraint on where to stop, code fragments starting and ending at arbitrary *base architecture* operations could result, leading to unnecessary code duplication and increasing code expansion. Establishing well-defined starting points increases the probability of finding a group of compiled VLIW code when the translator completes translation of a tree region.

We emphasize that encountering one of the *stopping points* above does *not* automatically end a tree region. To actually end a tree region at a stopping point, at least one of the following *stopping conditions* must previously have been met:

- The desired ILP has been reached in scheduling operations, or
- the number of *PowerPC* operations on this path since the beginning of the tree region entry has exceeded a maximum *window size*.

The purpose of the ILP goal is to attain the maximum possible performance. The purpose of the *window size* limit is to limit code explosion — a high ILP goal may be attainable only by scheduling an excessive number of operations into a tree region. Both the ILP goal and the maximum window size are adjusted dynamically in response to the frequency of execution of particular code fragments.

This approach implicitly performs loop unrolling as execution follows the control flow through the loop several times until a stopping condition has been met.

3.5 Translation Cache Management

Translated tree regions are stored in the translation cache. The translation cache is a memory area reserved for storing translations and not accessible to the system which is hosted above the DAISY VMM.

Preliminary experiments on large multi-user systems indicate that a translation space of 2K-4K *PowerPC* pages is sufficient to cover the working set for code. With a code expansion factor of 1.8×, such large multi-user systems would likely require a translation cache with 15 – 30 Mbytes to hold dynamically generated VLIW code. In our implementation, the translation cache is a memory area allocated from main memory, which will not be made accessible to the system executing under the DAISY VMM.

When a translation is first generated, it is allocated memory from the translation cache pool. A group can be ejected from the translation cache either when the underlying page for which it contains a translation is modified, or when cache space is reclaimed to free space for new translations.

While using the same architecture facilities, a number of events can cause a code page to change, e.g., when a program is terminated and a new program is loaded into the same physical address space, a page is paged out and replaced by some other page, or actual *in situ* code modification. When such events are detected, all translation groups which include modified code are detected and ejected from the translation cache. This does not require actual removal of the group, but changing all control transfers to such a group into control transfers to the translator, and ensuring that a new translation will be generated when the modified code is re-executed.

When the translation cache is full, a number of translation cache man-

agement strategies might be employed in a dynamic binary translation system, e.g., space can be reclaimed either incrementally by garbage collecting previously invalidated or little used translations. Alternatively, the whole cache can be invalidated resulting in retranslation of all translation units. In DAISY, we implement generational garbage collection which provides a simple, low-overhead management technique [28].

Experiments in the context of other projects (such as DIF [29] and Dynamo [30]) indicate that there is some performance benefit in invalidating the entire cache as a means of performing translation cache garbage collection. Invalidating the translation cache allows groups to adapt faster to changing profiles. This benefit is mostly derived from preventing premature group exits which are particularly costly for straightline trace groups as used in BOA [31, 25, 32], DIF [29], or Dynamo [30]. We expect this to be of less use in DAISY, since tree-groups are more resilient to such profile shifts as additional paths can be included in a translation group, thereby eliminating the cost of premature group exits.

Chapter 4

Group Formation and Scheduling

The group formation strategy is an essential point in generating tree groups which expose parallelism to be exploited by the target VLIW architecture. Successive steps then perform parallelism-enhancing optimizations, and exploit the parallelism by generating appropriate schedules and performing speculation.

As *tree groups* are formed, the VLIW operations are passed to the code optimizer and scheduler to generate native VLIW code. Complex operations are cracked at this point, and multiple simple VLIW operations are passed to the optimization and scheduling step.

DAISY VLIW operations are scheduled to maximize ILP opportunities, taking advantage of speculation possibilities supported by the underlying architecture. The current scheduling approach is greedy, as described in more detail in section 4.2. In determining the *earliest possible time*, DAISY makes use of copy propagation, load-store telescoping, and other optimizations which are described in more detail in [33]. Thus scheduling, optimization, and register allocation are all performed at once, i.e., operations are dealt with only once.

Since this scheme can schedule operations out of order, and since we wish to support precise exceptions for the underlying (*PowerPC*) architecture, we need some way to generate the proper *PowerPC* register and memory values when an exception occurs. The architected register state is obtained by renaming all speculative results and committing those values to the ar-

chitected register state in program order. Memory ordering is guaranteed by scheduling stores in their original program order. An alternative mechanism to implement precise exceptions which does not require all results to be committed in-order is based on a state repair mechanism described in [34].

If special attention is not paid, page crossings — either by direct branch or by falling through to the next page — can cause difficulties. When a group contains base architecture code crossing a base architecture page boundary, a check must be made to ensure that the new code page is still mapped in the *PowerPC* page tables, and that its translation from *effective* to *real* address has not changed since the translation was created. Unlike code modification events, page table changes do not cause the destruction of translations since they usually occur when an operating system performs a task switch, and are later restored when the task is scheduled again.

A probe operation like `LOAD_REAL_ADDRESS_AND_VERIFY` (`LRAV`) suffices for the task of testing for changes in the memory map. `LRAV` makes use of a Virtual Page Address register (VPA), which is maintained in a non-*PowerPC* register and indicates the *effective* address starting the current *PowerPC* code page. `LRAV <VPA>,<DISP>,<EXPECTED_VAL>` works as follows:

1. Computes the *effective* address of the new page as `r36+<DISP>`,
2. Translates the *effective* address to a *real PowerPC* address (if this fails, a trap occurs to the VMM which passes control to the PowerPC instruction page fault handler),
3. Compares the *real* address to `EXPECTED_VAL`,
4. If they are equal, `r36` is updated with the value `r36+<DISP>`, and execution continues normally. *This is the normal case* with most operating systems.
5. Otherwise a trap occurs, and the binary translation software makes the proper fixup. *This is a very unusual case.*

4.1 Branch conversion

Register-indirect branches can cause frequent serializations in our approach to dynamic binary translation (in which there is no operating system support

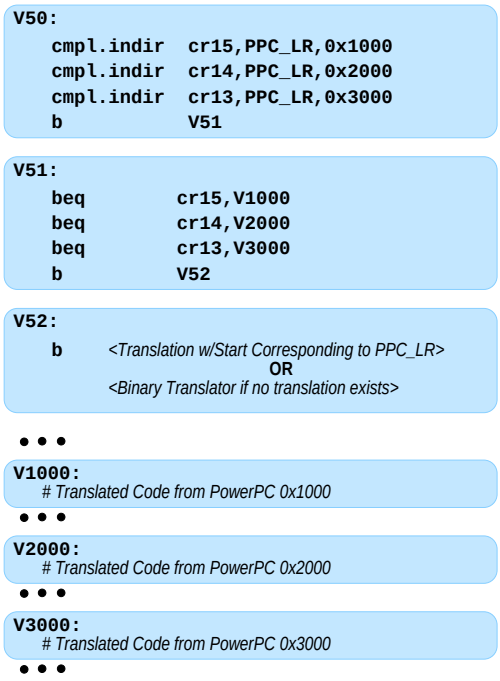

Figure 4.1: Translation of Indirect Branch

for binary translation and all code from the original architecture is translated including OS code and low-level exception handlers). Such serializations can significantly curtail performance, and hence it is important to avoid them.

This can be accomplished for indirect branches by converting them into a series of conditional branches backstopped by an indirect branch. This is similar to the approach employed in Embra [35]. However, Embra checked only a single value for the indirect branch, whereas we check multiple values.

For example, consider a *PowerPC* indirect branch `blr`, (Branch to `Link Register`) which, the first 100 times it is encountered, goes to 3 locations in the original code, `0x1000`, `0x2000`, and `0x3000`. Then the binary translated code for this `blr` might be as depicted in Figure 4.1.

We make several points about Figure 4.1:

- The `PPC_LR` value is kept in an integer register such as `r33` that is not architected in *PowerPC*.

- Translated operations from `0x1000`, `0x2000`, and `0x3000` can be speculatively executed prior to `V1000`, `V2000`, and `V3000` respectively as resource constraints permit. Such speculative execution can reduce critical pathlengths and enable better performance.

- If additional return points such as `0x4000` are discovered in the future, the translated code can be updated to account for them — up to some reasonable limit on the number of immediate compares performed.

- A special form of compare, `cmpl.indir` is used because in *PowerPC* and most architectures, the register used for the indirect branch (e.g., `PPC_LR`) holds an *effective* (or *virtual*) address, whereas for reasons outlined in [7], it is important to reference translations by *real* address. The `cmpl.indir` operations in `V50` in Figure 4.1 translate the `PPC_LR` value to a real address before comparing it to the immediate (real address) value specified.

- It is also helpful if the `cmpl.indir` operation can specify a 32 or 64-bit constant, so as to avoid a sequence of instructions to assemble such an immediate value.

The use of the `cmpl.indir` is an optimization. Alternatively, a comparison of the effective address can be performed, followed by the code for page transitions using an `LRAV` instruction if a page crossing occurs.

4.2 Scheduling

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 (for complex operations). As each RISC primitive is generated, DAISY immediately finds a VLIW instruction in which it can be placed, while still performing VLIW global scheduling on multiple paths and across loop iterations and while maintaining precise exceptions.

For ease of understanding, we begin by providing a brief review of our scheduling algorithm, which is described in more detail in [7, 8]. Our algorithm maintains a **ready** time for each register and other resources in the system. This **ready** time reflects the earliest time at which the value in that register may be used. For example if the instruction `addi r3,r4,1` has *latency* 1 and is scheduled at time 5, then the **ready** time for `r3` is $5 + 1 = 6$. When an operation such as `xor r5,r3,r9` is scheduled, our algorithm computes its earliest possible time $t_{earliest}$ as the *maximum* of the ready times for `r3` and `r9`. Our algorithm is greedy, and so searches forward in time from $t_{earliest}$ until a time slot with sufficient resources is available in which to place the instruction.

Resources fall into two broad types. First the appropriate type of functional unit (e.g., integer ALU) must be available on which to execute the instruction. Second, a register must be available in which to place the result of the operation. If the operation is scheduled in order (i.e., after predecessor operations in the original code have written their results to their original locations), then the result is just placed where it would have been in the original code. For example if `addi r3,r4,1` is scheduled in order, the result is placed in `r3`. If the operation is executed speculatively (i.e., out of order) then its result is first renamed into a register not visible to the base architecture, and then copied into the original destination register of the operation, in the original program order. For example, if `addi r3,r4,1` is executed speculatively, it might become `addi r63,r4,1`, with `copy r3,r63` placed in the original location of the `addi`. No side effects to architected resources occur until the point in the original program at which they would occur. Clearly the target architecture must have more registers than the original base architecture under this scheme.

Figure 4.2 shows an example of *PowerPC* code and its conversion to VLIW code. We begin with four major points:

- Operations 1–11 of the original *PowerPC* code are scheduled in sequence into VLIWs. It turns out that two VLIWs 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

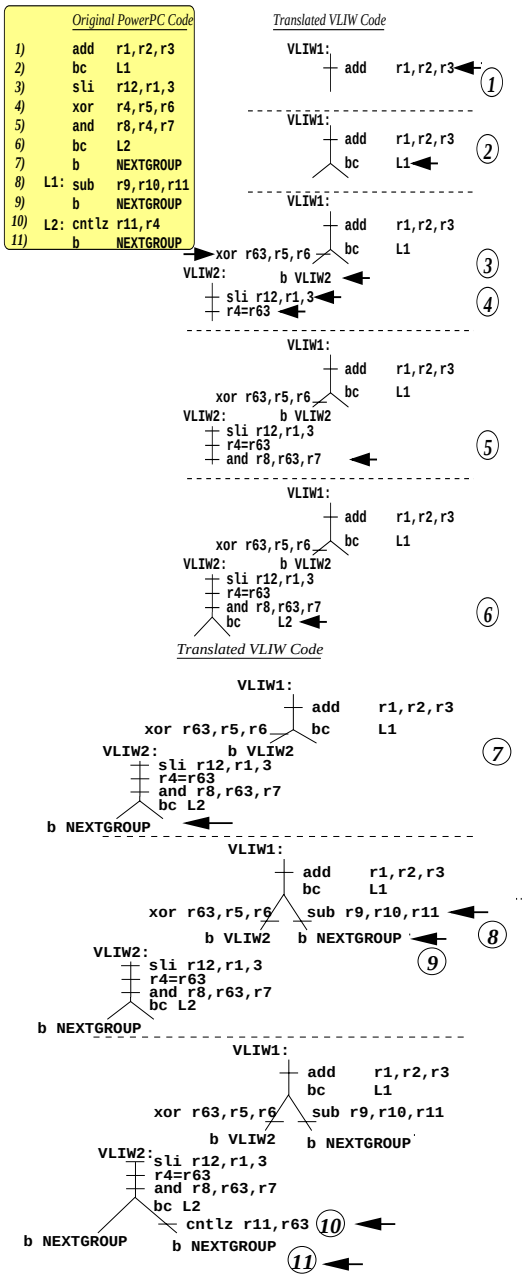

Figure 4.2: Example of conversion from *PowerPC* code to VLIW tree instructions.

the last VLIW, then the operation is performed as early as possible with the result placed in a renamed register (that is not architected in the original architecture). The renamed register is then copied to the original (architected) register at the end of the last VLIW. This is illustrated by the `xor` instruction in step 4, whose result is renamed to `r63` in VLIW1, then copied to the original destination `r4` in VLIW2. By having the result available early in `r63`, later instructions can be moved up. For example, the `cntlz` in step 11 can use the result in `r63` before it has been copied to `r4`. (Note that we use parallel semantics here in which all operations in a VLIW read their inputs before any outputs from the current VLIW are written.)

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

- As noted earlier, VLIW instructions are trees of operations with multiple conditional branches allowed in each VLIW [6]. 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.

4.3 Adaptive Scheduling Principles

To obtain the best possible performance, group formation and scheduling are adaptive and a function of execution frequency and execution behavior.

To conserve code space and reduce code duplication, tree groups are formed initially with modest ILP and window size parameters. If this region eventually executes only a few times, this represents a good choice for conserving code size and compile time.

The generated code is then profiled as described in chapter 2. If it is found that the time spent in a tree region tip is greater than a threshold of the total cycles spent in the program, the group is extended. Group extension forms translations using a significantly higher ILP goal and larger window size. Thus, if there are parts of the code which are executed more

frequently than others (implying high re-use on these parts), they will be optimized very aggressively. If, on the other hand, the program profile is flat and many code fragments are executed with almost equal frequency, then no such optimizations occur, which could be good strategy for preserving instruction cache resources and translation time.

In addition to group formation, the actual schedule can also be adapted in response to program behavior. When a tree group is first translated, load instructions are speculated aggressively even if disambiguation is not successful. Ambiguous load instructions are verified at the in-order point, and if speculation resulted in incorrect execution, a DAISY-level exception is raised and corrective actions are performed to determine the in-order load value and recompute all dependent operations. [23]

This behavior is profiled by the DAISY VMM using counters which determine the nature and frequency of misspeculations. If frequent mis-speculation results in performance degradation, then the offending tree group is rescheduled conservatively, and frequently mis-speculated load instructions are performed in-order.

Other code generation issues can be treated similarly, to detect and reschedule speculative load operations which have an inordinate number of data cache misses. Since the DAISY architecture uses a stall-on-miss policy, stalling on speculative loads which may not contribute to program progress is prohibitive. The concept of adaptive code generation can also be applied to other optimizations, for example in the context of value prediction to recompile code with high misprediction rates.

4.4 Implementing Precise Exceptions

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. Exceptions such as TLB misses that hit in the original architecture's page table, simple storage operand misalignments, and code modification events (discussed further in section 4.6) 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, as described below. 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.

For example, assume an external interrupt occurs immediately after `VLIW1` of figure 4.2 finishes executing, and prior to the start of `VLIW2`. The interrupt handler is just a dynamically 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`.

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. Note that 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 VLIWs starting from instruction 3, so as to create a valid entry point.

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 group is entered or a page boundary is crossed within a group. 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. (This assumes a 4096 byte page with base architecture instructions being aligned on a 4-byte boundary.)

If the VLIW has atomic 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 sequential exception semantics (like an in-order superscalar, where independently executable operations have been grouped together in "VLIWs" [36][2]) so that all parcels that logically preceded the exception causing parcel 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.

4.5 Communicating Exceptions and Interrupts to the OS

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

Notice that the mechanism described here does not require any changes to the *base architecture* operating system. The net result is that all existing

software for the *base architecture*, including both the operating system and applications, runs unchanged, by dint of the **VMM** software.

4.6 Self-modifying and self-referential code

Another concern is self-referential code such as code that takes the checksum of itself or code with floating point constants intermixed with real code or even PC-relative branches. These are all transparently handled by the fact that all registers architected in the *base architecture* — including the *program counter* or *instruction address register* — contain the values they would contain were the program running on the *base architecture*. The only means for code to refer to itself is through these registers, hence self-referential code is trivially handled.

A final major concern is self modifying code. Depending on the architecture semantics, different solutions are possible. Many recent architectures, such as the PowerPC, require an instruction to synchronize data and instruction caches by invalidating the instruction cache. In such architectures, it is sufficient to translate such instruction cache invalidation instructions to invalidate all translation groups containing the respective address in the translation cache.

However, some older architectures, e.g., Intel x86 require automatic synchronization between data and instruction caches. Efficient implementation of such architecture requires some hardware support. To solve this, we add a new read-only bit to each "unit" of *base architecture* physical memory, which is not accessible to the *base architecture*. (The unit size is ≥ 2 bytes for *S/390* and ≥ 1 byte for *x86*, but it could also be chosen at a coarser granularity, e.g., a cache line.) Whenever the **VMM** translates any code in a memory unit, it sets its read only bit. 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 group(s) 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 resuming interpretation-translation-execution at the base architecture instruction immediately following the one

that modified the code.

Chapter 5

Performance Evaluation

To demonstrate the feasibility of the presented approach, we have implemented a prototype which translates PowerPC code to a VLIW representation, and then uses a compiled simulation approach to emulate the VLIW architecture on a PowerPC-based system. The prototype was able to boot the unmodified AIX 4.1.5 operating system. A detailed description of this translator and its simulation methodology may be found in [28, 37].

To achieve a more detailed understanding of the contribution of each component in a binary translation system, we have used a trace-based evaluation methodology. We use this approach in conjunction with an analytic model to report results for SPECint95 and TPC-C. The traces for these benchmarks were collected on **RS/6000** *PowerPC* machines. Each SPECint95 trace consists of 50 samples containing 2 million PowerPC operations, uniformly sampled over a run of the benchmark. The TPC-C trace was obtained similarly, but contains 170 million operations.

The performance evaluation tools implement the dynamic compilation strategy using a number of tools:

- A tree-region former reads a *PowerPC* operation trace and forms tree-regions according to the strategy described in this paper. The initial region formation parameters were: an infinite resource ILP goal of 3 instructions per cycle and a window size limit of 24 operations. When 1% of the time is spent on a given tree region tip, the tip is aggressively extended with an infinte resource ILP goal of 10 and a window size limit of 180. An 8K entry, 8-way associative array of counters

were simulated, to detect the frequently executed tree region tips, as described in section 4.3.

- A VLIW scheduler schedules the *PowerPC* operations in each tree region and generates VLIW code according to the clustering, functional unit and register constraints, and determines the cycles taken by each tree region tip.

- A VLIW instruction memory layout tool lays out VLIWs in memory according to architecture requirements.

- A multi-level instruction cache simulator determines the instruction cache CPI penalty using a history-based prefetch mechanism.

- A multi-level data cache and data TLB simulator. The data references in the original trace are run through these simulators for hit/miss simulation. To account for the effects of speculation and joint cache effects on the off-chip L3, we multiplied the data TLB and data cache CPI penalties by a factor of *1.7* when calculating the final CPI. This factor is based on speculation penalties we have previously observed in an execution-based model [7]. To account for disruptions due to execution of translator code, we flush on-chip caches periodically based on a statistical model of translation events.

From the number of VLIWs on the path from the root of a tree region to a tip, and the number of times the tip is executed, we can calculate the total number of VLIW cycles. Empty VLIWs are inserted for long latency operations, so each VLIW takes one cycle. The total number of VLIWs executed, divided by the original number of *PowerPC* operations in the trace, yields the infinite cache, but finite resource CPI.

Stall cycles due to caches and TLBs, are tabulated using a simple *stall-on-miss* model for each cache or TLB miss. In the *stall-on-miss* model everything in the processor stops when a cache miss occurs, or data from a prior prefetch is not yet available.

To model translation overhead, we first define **re-use rate** as the ratio between the number of PowerPC instructions executed over a program run and the number of unique instruction addresses referenced at least once. **Reuse rates** are shown in the last column of Table 5.2, and are in millions.

SPECint95 rates were measured through an interpreter based on the reference inputs although operations in library routines were not counted. The TPC-C rate was obtained from the number of code page faults in a benchmark run.

Re-use rates may be used to estimate **translation overhead** (in terms of CPI) as follows:

- $\#P$ = number of Times an Operation undergoes *Primary* Translation
- $\#S$ = number of Times an Operation undergoes *Secondary* Translation
- CP = cycles per *Primary* Translation of an Operation
- CS = cycles per *Secondary* Translation of an Operation

Then

$$\textbf{Overhead} = \frac{\#P \times CP + \#S \times CS}{\textbf{Re-use Rate}}$$

The translation (or *primary* translation) of a *PowerPC* operation occurs when it is being added to a tree region for the first time. A *secondary* translation of an operation occurs when it is already in a tree region while new operations are being added to the tree region. In this study we have used an estimate of 4000 cycles for a *primary* translation and 800 cycles for a secondary translation. Our experience with DAISY indicates about 4000 PowerPC operations to translate one *PowerPC* operation [8]. Secondary translation merely requires disassembling VLIW code and reassembling it, something we estimate to take about 800 cycles.

The translation event can have an additional effect on the translated code in terms of flushing the caches, so when the translated code is executed after a translation, it is almost a cold-start of the caches. To simulate this effect, the instruction and data caches are partially flushed (corresponding to the translator's memory footprint) to simulate interference from the translator.

A first study was conducted to measure the performance improvements possible from various system design options. The issues studied covered a wide range:

- The impact of terminating groups at page boundaries and register-indirect branches, as was the case in the original DAISY design [8, 7]. Note that these numbers are not actually using page-based translation,

but instead terminate trace-based groups at page boundaries. Amongst other things, this reports a lower translation cost than a truly page-based translation system, since only the cost for the actually executed code is charged.

- The impact of variable-width and fixed-width VLIW instruction set design points on the instruction cache performance.
- The impact of using an interpreter to filter groups which are executed fewer than 30 times. This has the overall impact of reducing the translation cost and the size of the translation cache.
- Using some metrics to limit code duplication and expansion by capping the expansion ratio for any given page of original PowerPC code.
- The impact of prefetching from the instruction stream to improve performance of the instruction cache hierarchy.

Figure 5.1 gives an overview of the CPI achieved when adding these different options to the VLIW system. The components of performance are broken down (bottom to top) as infinite resource CPI, finite resource adder, instruction cache adder, data cache adder, TLB adder, translation cost adder, interpreter adder and exception handling adder.

For each benchmark, we simulated the following configurations based on the preferred machine configuration and a 1% threshold:

page_fixed Encode VLIW instructions using a fixed width format and terminate translation groups at page boundaries.

page_base Encode VLIW instructions using a variable width format and terminate translation groups at page boundaries.

trace_fixed Encode VLIW instructions using a fixed width format.

trace_base Encode VLIW instructions using a variable width format.

trace_pre Adds pre-interpretation as a filter to avoid translating infrequently executed code to trace_base.

trace_expl Limits group formation when excessive code expansion is detected for trace_pre.

trace_pf Adds hardware instruction prefetching capability to trace_expl.

The *Infinite Resource* CPI column of Table 5.2 describes the CPI of a machine with infinite registers and resources, constrained only by serializations between tree regions, and and PowerPC 604-like operation latencies. The *Finite Resource CPI Adder* describes the extra CPI due to finite registers and function units, as well as clustering effects, and possibly compiler immaturities. *Infinite Cache CPI* is the sum of the first two columns. The ICache, DCache and DTLB CPI describe the additional CPI penalties incurred due to instruction cache, data cache, and data TLB misses, assuming the *stall-on-miss* machine model described above. *Translation Overhead* is determined using the formulas and values above.

We have also measured the dynamic group length which is a measure of the code quality achieved by the translator and determines the amount of usable ILP (see figure 5.2). We have also studied the impact of these different options on code expansion as depicted in Figure 5.3, which gives the ratio of actually referenced DAISY code pages to the actually referenced PowerPC code pages.

We found that depending on the benchmark, terminating groups at page crossings and register-indirect branches can have a significant impact on the achievable dynamic pathlength with a significant reduction in several cases. Dynamic pathlength (average dynamic number of PowerPC instructions between group-to-group transitions) corresponds strongly to achievable infinite resource CPI, since longer groups give the translator more possibility to speculatively issue instructions. However, these speculative instructions are only useful if they lie on the taken path, hence if groups are frequently exited prematurely, there is less benefit from speculative execution.

While tree groups can reduce the cost of branch misprediction by incorporating multiple paths which can be scheduled and optimized as a unit, excessive mispredictions reduce the dynamic group pathlength as the number of paths exceeds what can be reasonably covered by tree-structured groups. An example of this is the go SPECint95 benchmark, which is known for its low branch predictability which translates directly into short dynamic group length and high infinite resource CPI. Experiments with varying thresholds reported below show that lowering the group extension threshold does result in more paths being covered to lower infinite resource CPI, but at the cost of significant code expansion to cover all possible paths. This leads directly

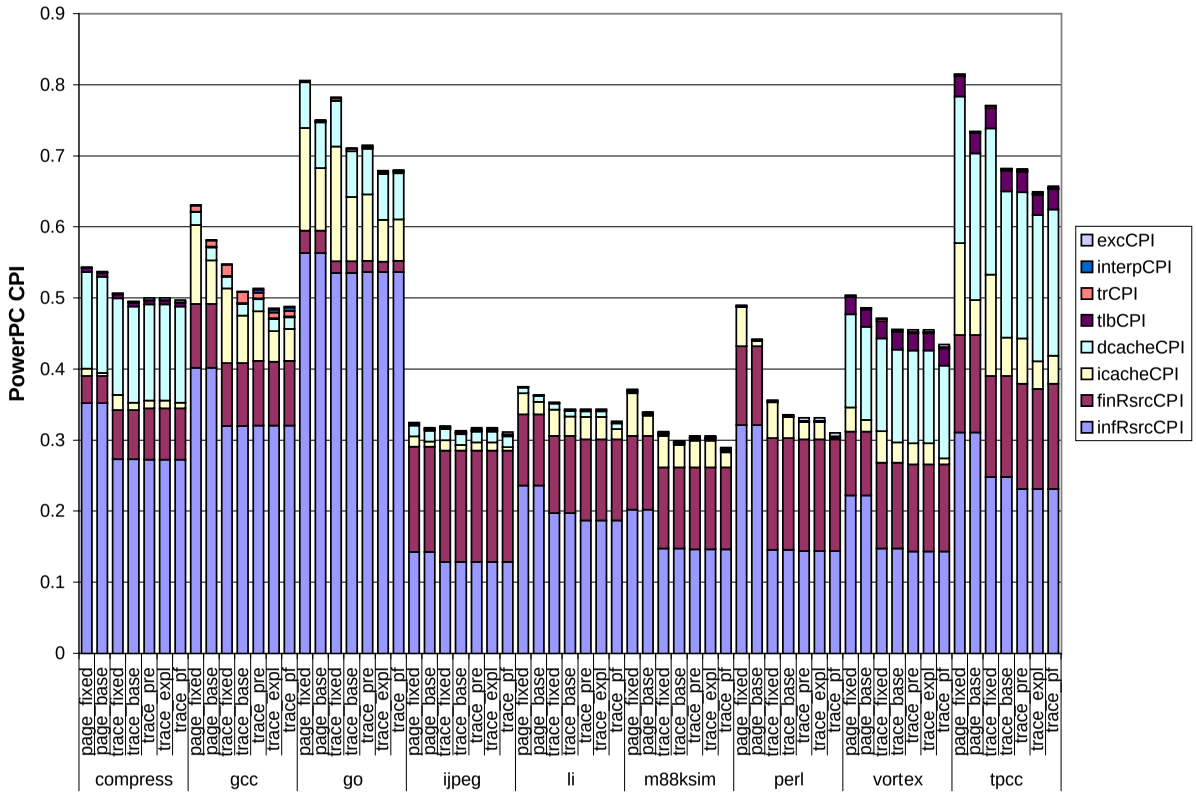

Figure 5.1: CPI for system design points of a DAISY system.

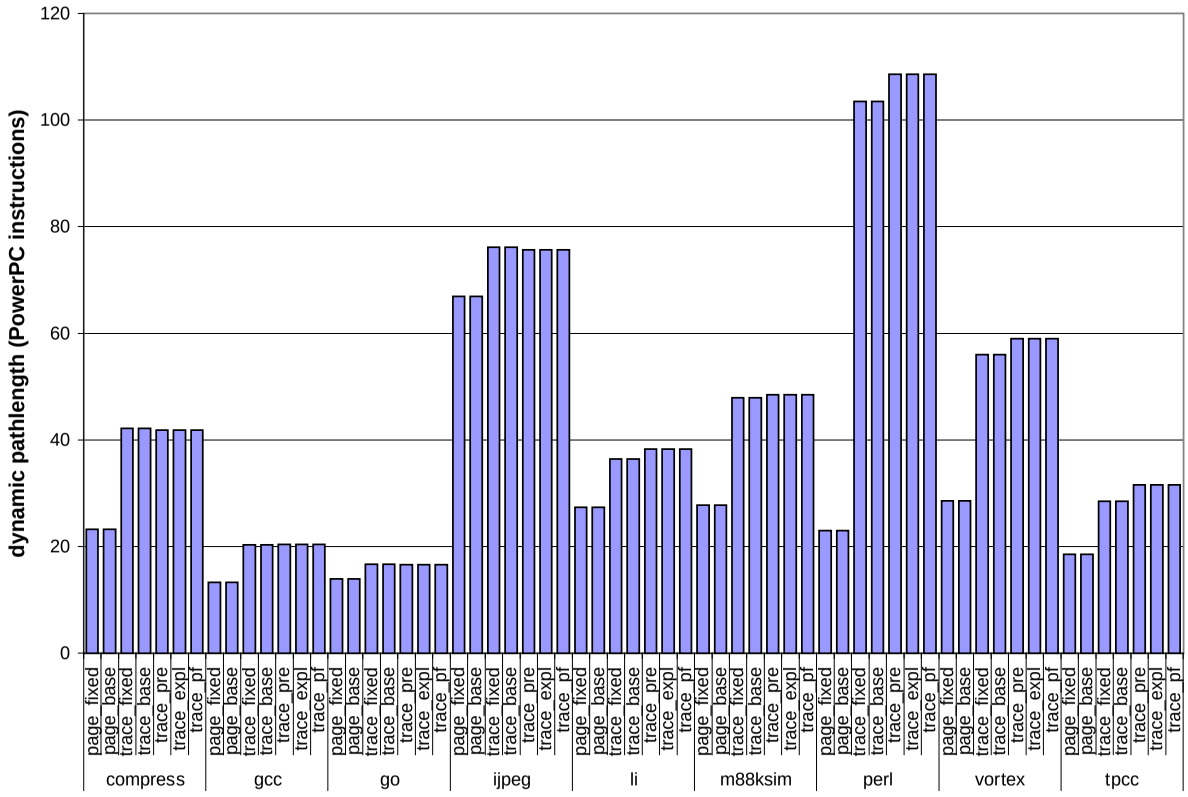

Figure 5.2: Dynamic group path length for system design points of a DAISY system.

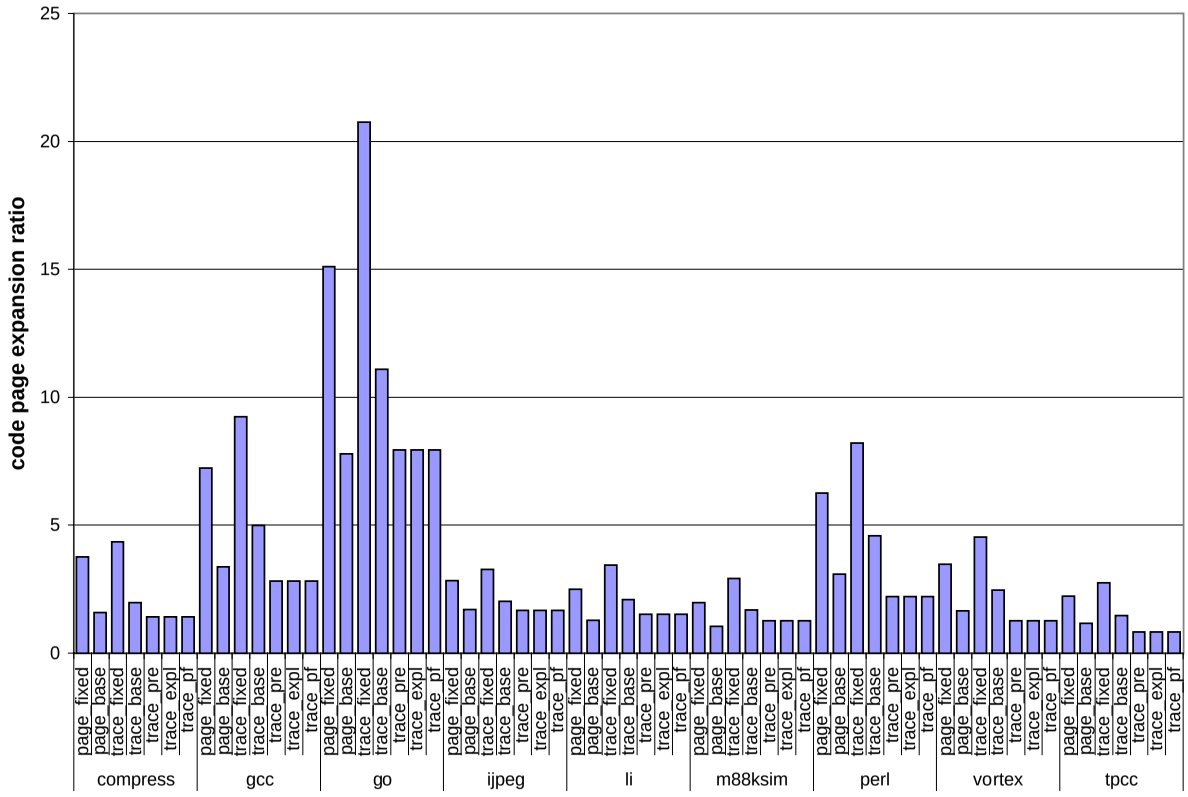

Figure 5.3: Code page expansion ratio for system design points of a DAISY system.

to a significant instruction cache penalty offsetting all gains in infinite cache CPI.

Code expansion numbers are presented in figure 5.3 and are important for two reasons. First, they indicate the amount of memory which must be set aside for holding translation (Tcache), and it also serves as indicator of the pressure on the instruction cache. Thus, higher code expansion usually entails higher instruction cache penalty. To reduce the code expansion, we experimented with using initial interpretation as a filter to reduce the amount of code being translated.[1]

Preinterpretation resulted in a minor increase of pathlength, since the elimination of preinterpreted groups from the group formation process resulted in lowering the overall bar to extend groups. Preinterpretation had beneficial impact on the code expansion by removing significant portions of infrequently executed code from the translation cache. Consequently, the code expansion declined with preinterpretation. This did however not result in better instruction cache performance, since the code in hot regions now got extended more easily and hot regions were clearly not affected by preinterpretation. Thus, the working set for those regions actually increased, resulting in a larger instruction cache performance component than in experiments without preinterpretation.

The impact of group extension and reoptimization policies was another interesting issue. To see what an appropriate threshold fraction of overall execution time would be appropriate for triggering group extension to achieve best performance, we experimented with threshold values from 0.1% to 5% for the preferred DAISY configuration (see figure 5.4).

As expected, lowering the threshold for group reoptimization results in improved dynamic group path length (see figure 5.5 and infinite cache CPI. However, this also lead to significant code expansion of the translated code (see figure 5.6) and instruction cache penalty. In addition, frequent group extension lead to increases in the translation cost overhead.

Another experiment explored the impact and mix of execution units on performance. We experimented with a set of clustered machines having 4 execution units up to 16 units. The machine configurations are reported in table 5.1. All machines are clustered with 4 execution units per cluster. All

[1]Unlike our work reported in [25], preinterpretation served solely as a filter and was not used to collect statistics used to guide group formation.

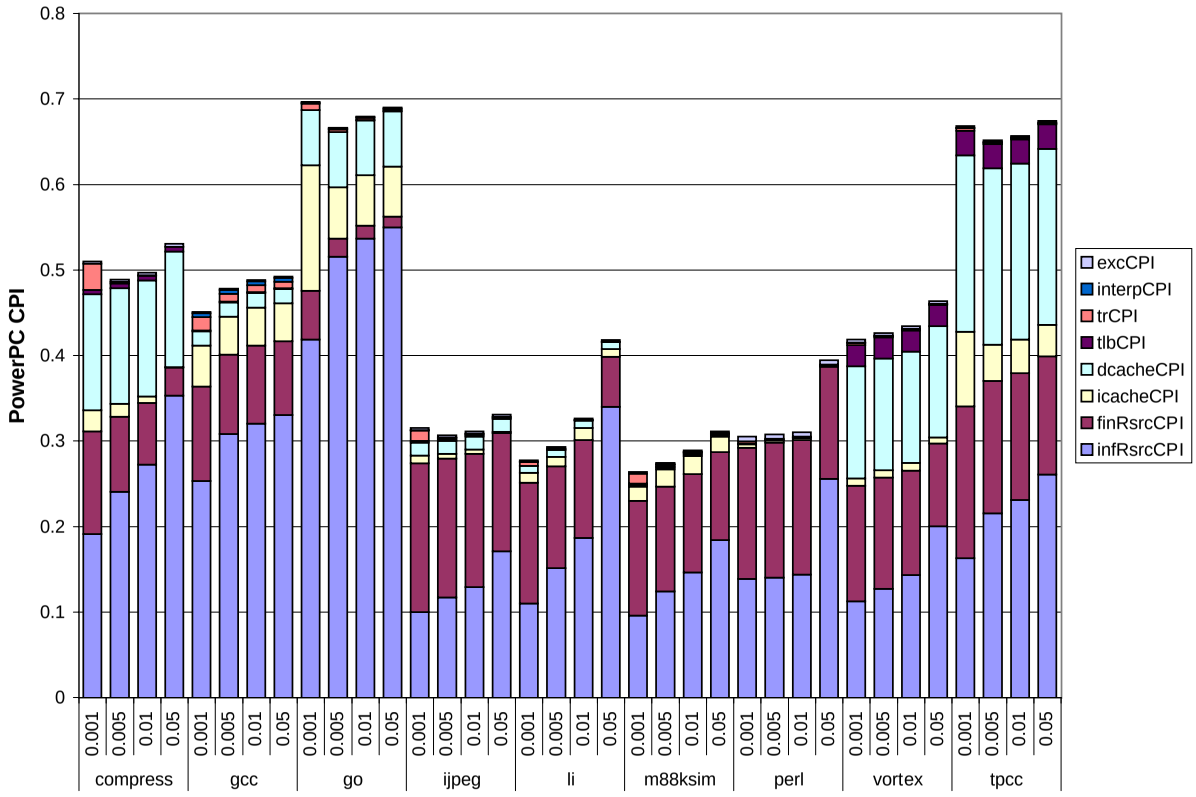

Figure 5.4: CPI for different threshold values for the preferred DAISY configuration.

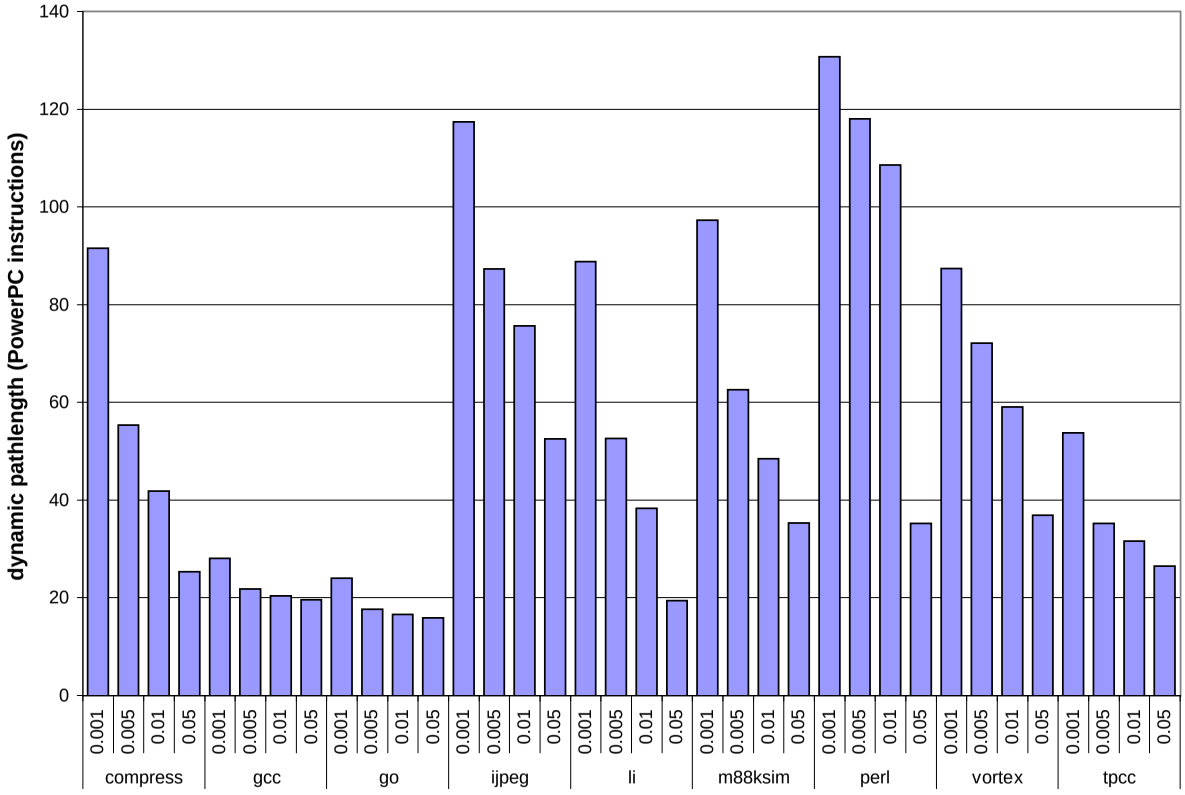

Figure 5.5: Dynamic group path length for different threshold values for the preferred DAISY configuration.

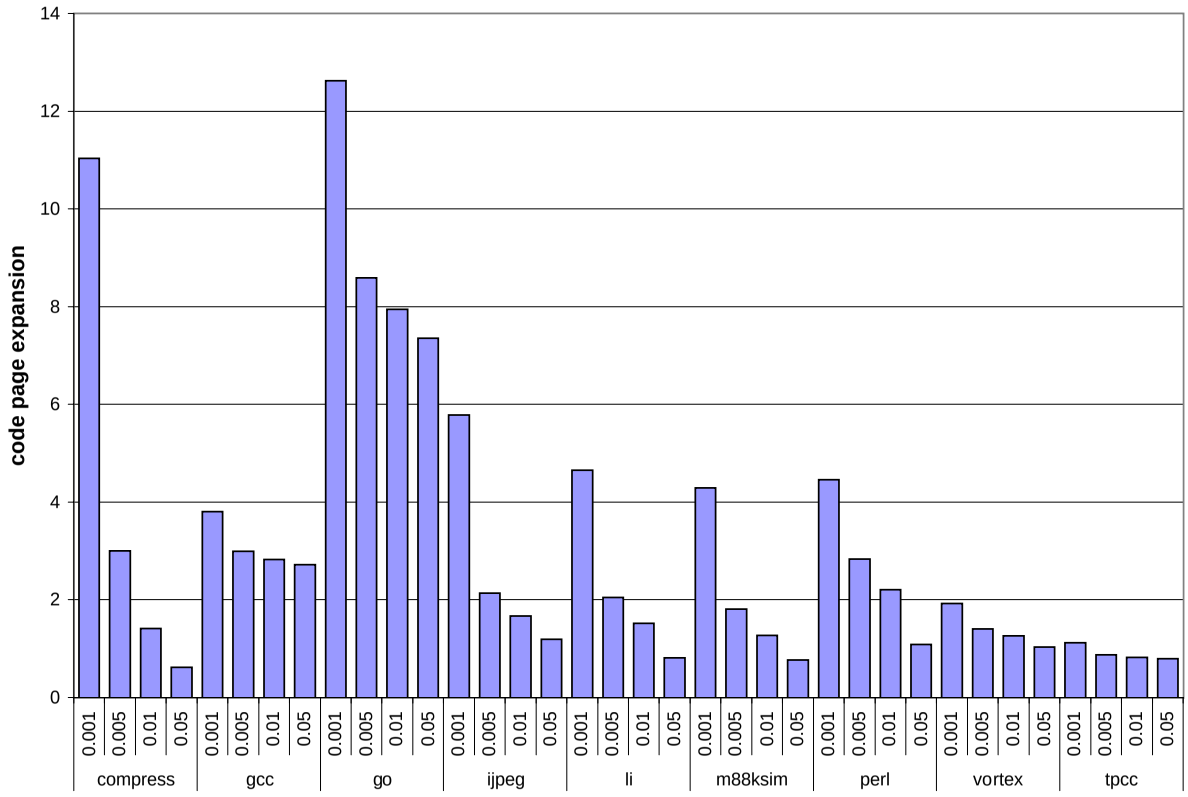

Figure 5.6: Code page expansion ratio for different threshold values for the preferred DAISY configuration.

Configuration	4.1	4.2	8.2	8.4	16.4	16.8
No. clusters	1	1	2	2	4	4
No. ALU/cluster	4	4	4	4	4	4
No. LS/cluster	1	2	1	2	1	2
No. branches	1	1	2	2	3	3
I-cache	8K	8K	16K	16K	32K	32K

Table 5.1: This table lists the resources available in the machine configurations explored in this article.

execution units can perform arithmetic and logic operations, and depending on the configuration, either one or two units per cluster can perform memory operations. A cross-cluster dependency incurs a delay of one additional cycle.

Depending on the machine width, each VLIW can contain up to three branch instructions. The memory hierarchy was similar, but the instruction line size was reduced for narrower machines due to the mini-ICache configuration requirements. Because the critical paths cannot accommodate driving larger arrays, this resulted in reduced first level instruction cache sizes.

Figure 5.7 shows the machine CPI for different cluster configurations as shown in table 5.1. Even narrow machines offer good performance since translation overhead is low and thus the resulting machines offer very good CPI compared to today's superscalar machines. Their hardware simplicity should also result in very high frequency implementations. Wider machines offer a very significant improvement over current architectures and high frequency.

We have also explored the impact of register file size on performance. Since register set pressure may prevent additional beneficial speculation, it may throttle performance. To test if 64 registers were sufficient for preventing unnecessary performance bottlenecks, we explored register file sizes from 64 to 256 registers for the preferred machine configuration 16.8 using a 1% threshold. Figure 5.8 shows the impact of using more registers. These do not in general improve performance although some benchmarks may derive small gains. Also, in some cases, more registers actually resulted in slightly worse performance since register pressure prevented performance-degrading overspeculation.

Table 5.2 reports the final performance for the preferred hardware and software configurations.

Final CPI is then the sum of the *Infinite Cache CPI*, ICache, DCache,

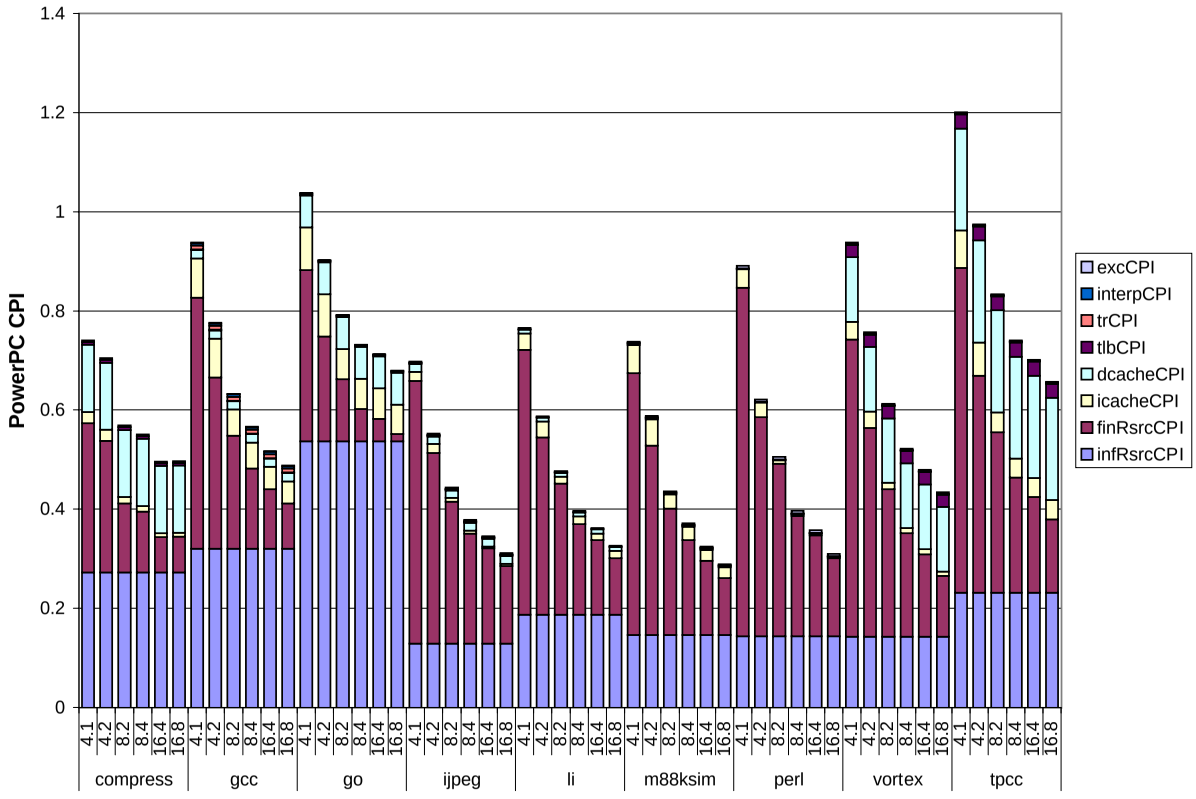

Figure 5.7: CPI for different machine configurations for the preferred DAISY configuration.

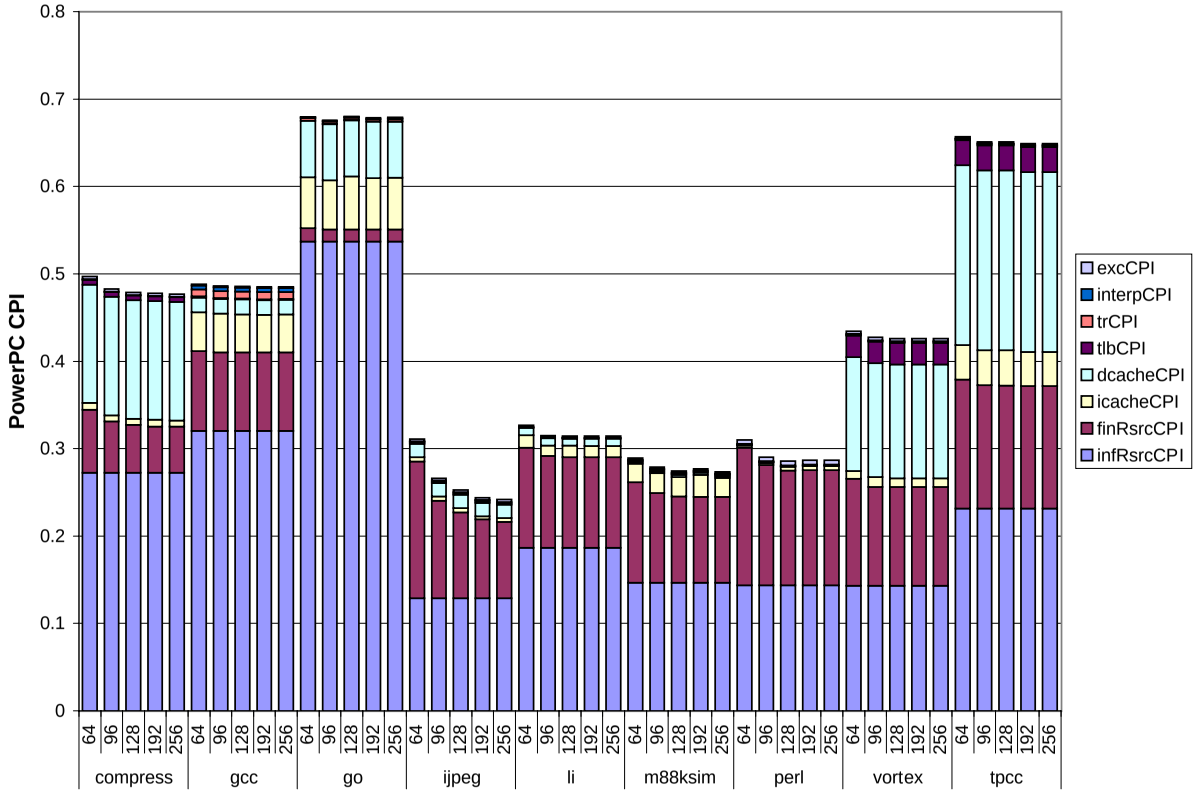

Figure 5.8: CPI for varying register file size for the preferred DAISY configuration.

Program	Inf Resrc CPI	Resrc CPI Adder	Inf Cache CPI	*CPI Adders* ICache	DCache	TLB	Xlate Overhd (CPI)	**Final CPI**	Code Expn pPg	Avg. Path len
compress	0.27	0.07	0.34	0.01	0.14	0.01	0.00	0.50	1.41	41.85
gcc	0.32	0.09	0.41	0.04	0.02	0.00	0.01	0.49	2.82	20.42
go	0.54	0.02	0.55	0.06	0.06	0.00	0.00	0.68	7.95	16.59
ijpeg	0.13	0.16	0.29	0.00	0.02	0.00	0.00	0.31	1.67	75.68
li	0.19	0.11	0.30	0.01	0.01	0.00	0.00	0.33	1.52	38.27
m88ksim	0.15	0.11	0.26	0.02	0.00	0.00	0.00	0.29	1.27	48.45
perl	0.14	0.16	0.30	0.00	0.00	0.00	0.00	0.31	2.21	108.57
vortex	0.14	0.12	0.27	0.01	0.13	0.02	0.00	0.43	1.27	59.02
tpcc	0.23	0.15	0.38	0.04	0.21	0.03	0.00	0.66	0.82	31.56

Table 5.2: Performance on SPECint95 and TPC-C for a clustered DAISY system with 16 execution units arranged in 4 clusters.

D-TLB, and *Overhead* columns. The initial and exception recovery interpretation overhead are insignificant. Also, there are no branch stalls, due to our zero-cycle branching technique [6, 38].

Unlike previous infinite cache VLIW studies, our model takes into account all the major CPI components. However, while we did examine the critical paths in detail, we have not built hardware for this particular VLIW machine. The hardware design process is likely to involve some design trade-offs that may be hard to foresee. Hence performance could fall short of the numbers presented here. However, our model also omits some potential performance enhancers, such as software value prediction, software pipelining, tree-height reduction, and data cache latency tolerance techniques.

The *Average Window Size* in Table 5.2 indicates the average dynamic number of *PowerPC* operations between tree region crossings. The *Code Explosion* indicates the ratio of translated VLIW code pages to *PowerPC* code pages. Our mean code expansion of 1.8 is more than 2× better than the previous page-based version of DAISY. This improvement has come about largely because of our use of adaptive scheduling techniques and the fact that

only executed code is translated.

Chapter 6

Related Work

Before the inception of the DAISY project, no machines have been designed exclusively as target platforms for binary translation. The DEC/Compaq Alpha was however designed to ease migration from the VAX architecture, and offered a number of compatibility features. These include similar memory management capabilities to ease migration of the VAX/VMS operating system, and support for VAX floating point formats. DEC's original transition strategy called for static binary translators to support program migration. Two translators supported these migration strategy: *VEST* for VAX/VMS migration to Alpha/OpenVMS and *mx* for migration from DEC Ultrix on the MIPS architecture to OSF1 on DEC Alpha [39]. Later, the FX!32 dynamic binary translator was added to ease migration from Windows on x86 to Windows on Alpha.

Recently, Transmeta has announced an implementation of the Intel x86 processor based on binary translation to a VLIW processor [40]. The processor described is based on a VLIW with hardware support for checkpointing architected processor state to implement precise exceptions using a rollback/commit strategy. Rollback of memory operations is supported using a gated store buffer [41].

Previous work in inter-system binary translation has largely focused on easing migration between platforms. To this end, problem state executables were translated from a legacy instruction set architecture to a new architecture. By restricting the problem domain to a single process, a number of simplifying assumptions can be made about execution behavior and the memory map of a process. Andrews and Sand [42] and Sites et al. [39] re-

port the use of static binary translation for migrating user programs from CISC-based legacy platforms to newer RISC-based platforms.

Andrews reports that many users of the Tandem Himalaya series use code generated by the Accelerator translator, and that many customers have never migrated to native RISC-based code and prefer to use CNS-based infrastructure such as debuggers. This demonstrates the viability of using binary translation for providing commercial solutions, provided the infrastructure and user interfaces continue to operate unchanged.

The first dynamic binary translator reported in the literature is Mimic, which emulates user-level System/370 code on the RT/PC, a predecessor of the IBM PowerPC family. May [43] describes the foundations of further work in the area of dynamic binary translation, such as dynamic code discovery and the use of optimistic translations which are later recompiled if the assumptions are not satisfied. The use of dynamic binary translation tools for performance analysis is explored in [44].

Dynamic binary translation of programs as a translation strategy is exemplified by caching emulators such as FX!32 [45]. 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). FX!32 emulates only the user program space and depends on support from the OS (Microsoft Windows NT) to provide a native interface identical to that of the original migrant system.

The presented approach is more comparable to full system emulation, which has been used for performance analysis (e.g., SimOS [46], [35]) and for migration from other legacy platforms as exemplified by Virtual PC, SoftPC/SoftWindows and to a lesser extent WABI, which intercepts Windows calls and executes them natively. Full system simulators execute as user processes on top of another operating system, using special device drivers for *virtualized* software devices. This is fundamentally different from our approach which uses dynamic binary translation to implement a processor architecture. Any operating system running on the emulated architecture can be booted using our approach.

For an in-depth comparison of several binary translation and dynamic optimization systems, in particular the Transmeta Crusoe processor, the HP Dynamo dynamic optimization system, and the Latte JavaVM just-in-time compilation system, we refer the reader to [47].

The present approach is different from the DIF approach of Nair and

Hopkins [29]. DAISY schedules operations on multiple paths to avoid serializing due to mispredicted branches. Also, in the present approach, there is virtually no limit to the length of a path within a tree region or the ILP achieved. In DIF, the length of a (single-path) region is limited by machine design constraints (e.g., 4-8 VLIWs). Our approach follows an all software approach as opposed to DIF which uses a hardware translator. This all-software technique allows aggressive software optimizations hard to do by hardware alone. Also, the DIF approach involves almost three machines: the sequential engine, the translator, and the VLIW engine. In our approach there is only a relatively simple VLIW machine.

Trace processors [48] are similar to DIF except that the machine is out-of-order as opposed to a VLIW. This has the advantage that different trace fragments do not need to serialize between transitions between one trace cache entry and another. However, when the program takes a path other than what was recorded in the trace cache, a serialization can occur. The present approach solves this problem by incorporating an arbitrary number of paths in a software trace cache entry, and by very efficient zero overhead multiway branching hardware [38]. The dynamic window size (trace length) achieved by the present approach can be significantly larger than that of trace processors, which should allow better exploitation of ILP.

Chapter 7

Conclusion

The DAISY architecture is a binary-translation based system which achieves high performance by dynamically scheduling base architecture code (such as PowerPC, System/390 or x86 code) to a VLIW architecture optimized for high frequency and ILP. DAISY addresses compatibility with legacy systems and intergenerational compatibility by using object code in legacy format as distribution format.

The key to DAISY's success is ILP extraction at execution time, which allows to achieve high performance through the dynamic adaptability of code. Unlike traditional, static compilers for VLIW architectures, DAISY can use execution-profile data to drive the aggressiveness of optimizations. This dynamic adaptability to profile changes is a major advantage over static VLIW compilers which have to make tradeoffs based on heuristics or possibly unrepresentative profiles collected during profile runs.

We have conducted a range of experiments to determine the impact of various system design considerations on the performance of a binary-translation-based system. We have also explored the stability of this approach as a result to changes in workload and its behavior.

ILP extraction based on the dynamic binary translation and optimization exposes significant ILP by using runtime information to guide program optimization, with values reaching almost 2.5 instructions per cycle even after accounting for cache effects.

Bibliography

[1] MPR. TI's new 'C6x DSP screams at 1600 MIPS. *Microprocessor Report*, 7(2):February, 1997.

[2] Intel. *IA-64 Application Developer's Architecture Guide*. Intel Corp., Santa Clara, CA, May 1999.

[3] R. P. Colwell, R.P. Nix, J. J. O'Donnel, D. P. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, 37(8):318–328, August 1988.

[4] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer: design philosophies, decisions, and trade-offs. *IEEE Computer*, 22(1):12–35, January 1989.

[5] B. R. Rau and J. A. Fisher, editors. *Instruction-level parallelism*. Kluwer Academic Publishers, 1993. Reprint of The Journal of Supercomputing, 7(1/2).

[6] K. Ebcioğlu. Some design ideas for a VLIW architecture for sequential-natured software. In M. Cosnard et al., editor, *Parallel Processing*, pages 3–21. North-Holland, 1988. (Proc. of IFIP WG 10.3 Working Conference on Parallel Processing).

[7] K. Ebcioğlu and E. Altman. DAISY: dynamic compilation for 100% architectural compatibility. Research Report RC20538, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1996.

[8] K. Ebcioğlu and E. Altman. DAISY: dynamic compilation for 100% architectural compatibility. In *Proc. of the 24th Annual International*

Symposium on Computer Architecture, pages 26–37, Denver, CO, June 1997. ACM.

[9] Intel. *IA-64 Application Developer's Architecture Guide*. Intel Corp., Santa Clara, CA, May 1999.

[10] B. R. Rau. Dynamically scheduled VLIW processors. In *Proc. of the 26th Annual International Symposium on Microarchitecture*, pages 80–92, Austin, TX, December 1993. ACM.

[11] J. Moreno, M. Moudgill, K. Ebcioğlu, E. Altman, C. B. Hall, R. Miranda, S.-K. Chen, and A. Polyak. Simulation/evaluation environment for a VLIW processor architecture. *IBM Journal of Research and Development*, 41(3):287–302, May 1997.

[12] T. Conte and S. Sathaye. Dynamic rescheduling: A technqiue for object code compatibility in VLIW architectures. In *Proc. of the 28th Annual International Symposium on Microarchitecture*, pages 208–217, Ann Arbor, MI, November 1995. ACM.

[13] G. M. Silberman and K. Ebcioğlu. An architectural framework for migration from CISC to higher performance platforms. In *Proc of the 1992 International Conference on Supercomputing*, pages 198–215, Washington, DC, July 1992. ACM Press.

[14] G. M. Silberman and K. Ebcioğlu. An architectural framework for supporting heterogeneous instruction-set architectures. *IEEE Computer*, 26(6):39–56, June 1993.

[15] M. Gschwind, K. Ebcioğlu, E. Altman, and S. Sathaye. Binary translation and architecture convergence issues for IBM System/390. In *Proc. of the International Conference on Supercomputing 2000*, Santa Fe, NM, May 2000. ACM.

[16] K. Ebcioğlu, E. R. Altman, and E. Hokenek. A JAVA ILP machine based on fast dynamic compilation. In *IEEE MASCOTS International Workshop on Security and Efficiency Aspects of Java*, January 1997.

[17] K. Ebcioğlu and R. Groves. Some global compiler optimizations and architectural features for improving the performance of superscalars. Research Report RC16145, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1990.

[18] K. Ebcioglu and G. Silberman. Handling of exceptions in speculative instructions. US Patent 5799179, August 1998.

[19] S. A. Mahlke, W. Y. Chen, W.-m. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 238–247, Boston, MA, October 1992.

[20] V. Kathail, M. Schlansker, and B.R. Rau. HPL PlayDoh architecture specification: Version 1. Technical Report 93-80, HP Laboratories, Palo Alto, CA, March 1994.

[21] E. Altman, K. Ebcioğlu, M. Gschwind, and S. Sathaye. Method and apparatus for profiling computer program execution. Filed for US Patent, August 2000.

[22] H. Chung, S.-M. Moon, and K. Ebcioğlu. Using value locality on VLIW machines through dynamic compilation. In *Proc. of the 1999 Workshop on Binary Translation*, IEEE Computer Society Technical Committee on Computer Architecture Newsletter, pages 69–76, December 1999.

[23] M. Moudgill and J. Moreno. Run-time detection and recovery from incorrectly ordered memory operations. Research Report RC20857, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1997.

[24] K. Ebcioğlu, E. Altman, S. Sathaye, and M. Gschwind. Execution-based scheduling for VLIW architectures. In *Euro-Par '99 Parallel Processing – 5th International Euro-Par Conference*, number 1685 in Lecture Notes in Computer Science, pages 1269–1280. Springer Verlag, Berlin, Germany, August 1999.

[25] M. Gschwind, E. Altman, S. Sathaye, P. Ledak, and D. Appenzeller. Dynamic and transparent binary translation. *IEEE Computer*, 33(3):54–59, March 2000.

[26] M. Gschwind. Pipeline control mechanism for high-frequency pipelined designs. US Patent 6192466, February 2001.

[27] M. Gschwind, S. Kosonocky, and E. Altman. High frequency pipeline architecture using the recirculation buffer. in preparation, 2001.

[28] E. Altman and K. Ebcioğlu. Full system binary translation: RISC to VLIW. in preparation.

[29] R. Nair and M. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *Proc of the 24th Annual International Symposium on Computer Architecture*, pages 13–25, Denver, CO, June 1997. ACM.

[30] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical Report 99-78, HP Laboratories, Cambridge, MA, June 1999.

[31] S. Sathaye, P. Ledak, J. LeBlanc, S. Kosonocky, M. Gschwind, J. Fritts, Z. Filan, A. Bright, D. Appenzeller, E. Altman, and C. Agricola. BOA: Targeting multi-gigahertz with binary translation. In *Proc. of the 1999 Workshop on Binary Translation*, IEEE Computer Society Technical Committee on Computer Architecture Newsletter, pages 2–11, December 1999.

[32] E. Altman, M. Gschwind, and S. Sathaye. BOA: the architecture of a binary translation processor. Research Report RC21665, IBM T.J. Watson Research Center, Yorktown Heights, NY, March 2000.

[33] K. Ebcioğlu, E. Altman, S. Sathaye, and M. Gschwind. Optimizations and oracle parallelism with dynamic translation. In *Proc. of the 32nd ACM/IEEE International Symposium on Microarchitecture*, pages 284–295, Haifa, Israel, November 1999. ACM, IEEE, ACM Press.

[34] M. Gschwind and E. Altman. Optimization and precise exceptions in dynamic compilation. In *Proc. of the 2000 Workshop on Binary Translation*, Philadelphia, PA, October 2000. also in: Computer Architecture News, December 2000.

[35] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proc. of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 68–79, Philadelphia, PA, May 1996. ACM.

[36] J. Moreno, K. EBcioğlu, M. Moudgill, and D. Luick. ForestaPC user instruction set architecture. Research Report RC20733, IBM T.J. Watson Research Center, Yorktown Heights, NY, February 1997.

[37] E. Altman and K. Ebcioğlu. Simulation and debugging of full system binary translation. In *Proc. of the 13th International Conference on Parallel and Distributed Computing Systems*, pages 446–453, Las Vegas, NV, August 2000.

[38] K. Ebcioğlu, J. Fritts, S. Kosonocky, M. Gschwind, E. Altman, K. Kailas, and T. Bright. An eight-issue tree-VLIW processor for dynamic binary translation. In *Proc. of the 1998 International Conference on Computer Design (ICCD '98) – VLSI in Computers and Processors*, pages 488–495, Austin, TX, October 1998. IEEE Computer Society.

[39] R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[40] A. Klaiber. The technology behind Crusoe processors. Technical report, Transmeta Corp., Santa Clara, CA, January 2000.

[41] E. Kelly, R. Cmelik, and M. Wing. Memory controller for a microprocessor for detecting a failure of speculation on the physical nature of a component being addressed. US Patent 5832205, November 1998.

[42] K. Andrews and D. Sand. Migrating a CISC computer family onto RISC via object code translation. In *Proc. of the 5th International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 213–222, 1992.

[43] C. May. Mimic: A fast S/370 simulator. In *Proc. of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques*, volume 22 of *SIGPLAN Notices*, pages 1–13. ACM, June 1987.

[44] R. Cmelik and D. Keppel. Shade: a fast instruction-set simulatore for execution profiling. In *Proc. of the 1994 Conference on Measurement and Modeling of Computer Systems*, pages 128–137, Nashville, TN, May 1994. ACM.

[45] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. FX!32–a profile-directed binary translator. *IEEE Micro*, 18(2):56–64, March 1998.

[46] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, 3(4):34–43, Winter 1995.

[47] E. Altman, K. Ebcioğlu, M. Gschwind, and S. Sathaye. Advances and future challenges in binary translation and optimization. *Proc. of the IEEE*, 2001. submitted.

[48] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, Research Triangle Park, NC, December 1997. IEEE Computer Society.