RC 21665 (97500) 16 December 1999 **Computer Science**

IBM Research Report

BOA: The Architecture of a Binary Translation Processor

E. Altman, M. Gschwind, S. Sathaye, S. Kosonocky, A. Bright, J. Fritts
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

P. Ledak, D. Appenzeller, C. Agricola, Z. Filan
IBM Burlington
Essex Junction, VT 05452

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

Abstract

High frequency design and instruction-level parallelism (ILP) are two keys to high performance microprocessor implementation. To achieve these sometimes competing goals, the **B**inary-translation **O**ptimized **A**rchitecture (BOA) aims to bring code translation techniques based on continuous profiling into the mainstream. Initially, code is interpreted to detect code hot spots and gather profile information to guide dynamic optimizations. To achieve compatibility with the established *PowerPC* architecture, a binary translation layer translates *PowerPC* instructions into simple VLIW operation primitives. These primitives are then scheduled using VLIW scheduling techniques to a variable length, six issue VLIW/EPIC processor. Binary translation eliminates the binary compatibility problem faced by other processors, while dynamic recompilation enables adaptive re-optimization of critical program code sections and eliminates the need for dynamic scheduling hardware.

As a result, the BOA execution platform can be designed for multiple Gigahertz operation. The hardware execution platform includes novel microarchitectural features to eliminate complex stall and exception logic. Special support is also provided for binary translation in the form of several primitives designed for system-level binary translation functions. The data types of the binary translation processor are similar to that of the emulated *PowerPC* architecture to eliminate data representation issues which could necessitate potentially expensive data format conversion operations. In this work we examine the implications of binary translation on processor architecture and software translation and how we support a very high frequency *PowerPC* implementation via dynamic binary translation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High frequency design and instruction-level parallelism are two keys to high performance microprocessor implementation. The **B**inary-translation **O**ptimized **A**rchitecture (BOA) is an implementation of the IBM *POWER* processor family which tries to combine these techniques based on a binary translation and dynamic optimization framework.

In this context, binary translation and dynamic optimization are used to achieve 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. [1, 2]

Previous processors such as *Pentium Pro* and *POWER4* implementations have tried to achieve the high frequency and instruction-level parallelism goals using a hardware cracking scheme, where an instruction decoder in the pipeline generates multiple micro-operations which can then be scheduled out of order.

We explore an alternative software approach to both decompose complex operations and generate schedules. 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.

Our work was inspired by our previous experience with **DAISY** [3, 4, 5, 6, 7] which uses binary translation for scheduling *PowerPC* code to a VLIW. However, the machine described here is narrower, with primary importance given not to minimizing CPI (*Cycles Per Instruction*), but to maximizing processor frequency. The

resulting smaller size has the benefit of allowing multiple cores to be placed on a single integrated circuit.

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 EPIC architectures, (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 EPIC 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 this described approach offers significant advantages over a purely static compilation approach as exemplified by Intel and HP's *IA-64* architecture. From what has been disclosed, *IA-64* relies purely on static profiling which makes it impossible to adapt to program usage.

In addition to purely 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.

The remainder of this report describes our approach in designing a high frequency *PowerPC* compatible microprocessor through dynamic binary translation. Chapter 2 describes how the processor builds *traces* of *PowerPC* operations and the translation process into the BOA architecture, as well as optimization and scheduling of operations in a VLIW BOA *trace*. Chapter 3 describes the BOA instruction set architecture and details the high frequency implementation. Chapter 4 gives experimental microarchitectural performance results, and Chapter 5 concludes and describes future work.

Chapter 2

Binary Translation Strategy

In BOA, binary translation is a transparent process: As depicted in Figure 2.1, when a system based on the BOA architecture boots, control transfers to the VMM (**V**irtual **M**achine **M**anager), which implements the binary translation system. The VMM is part of a BOA system firmware, although it is not visible to the software running on it, much like microcode is not visible in a microcoded machine.

After BOA VMM initialization, the BOA VMM interpreter initiates the *PowerPC* boot sequence. In other words, a *PowerPC* system built on a BOA 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, i.e., `0xFFF00100`.

Similar to a native *PowerPC* system, a *PowerPC* boot ROM is located at that address, which will be executed under control of the BOA VMM. When the boot ROM initialization has completed after loading a kernel, and control passes to that kernel, the BOA 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 BOA 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 *traces*. *Traces* 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.

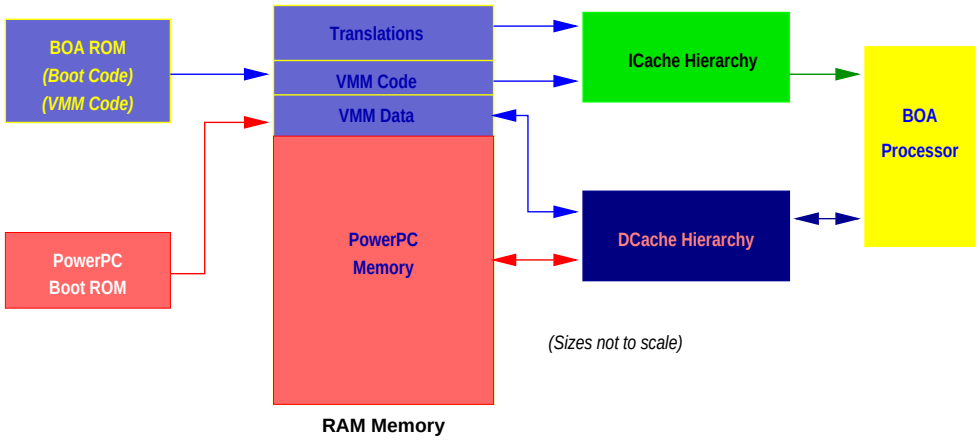

Figure 2.1: Components of a BOA System.

In fact, no code can even determine that it is hosted upon a layer of code implemented by the BOA VMM.

When the BOA 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 threshold, before it is translated into BOA machine code.

Interpretation serves multiple purposes: first, it serves as a filter for rarely executed code, i.e., much initialization code is executed only a few times, i.e., it 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 subsequence executions.

Interpretation also allows for the collection of profiling data, which is used to guide optimization. Currently, we use this information to determine the path of translation *traces* generated by the BOA VMM, as well as for branch prediction. Other uses are possible and planned for the future, such as guiding optimization aggressiveness, control and data speculation, and value prediction.

Traces have two types of exits: *side exits* which represent a mispredicted branch, and *end of trace exits*, which represent translation stopping points. Wise choice of such stopping points can limit *trace* size and help improve Instruction Cache (ICache) performance.

BOA gathers *PowerPC* operations from a single straightline path or *trace* in the

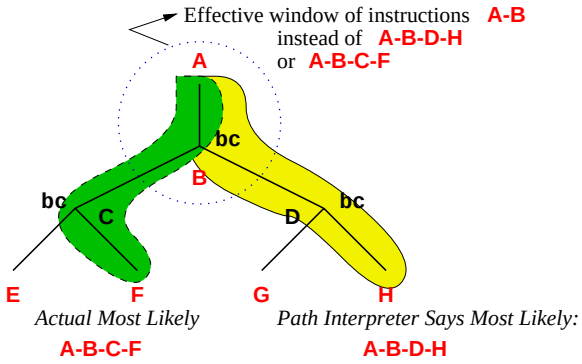

Figure 2.2: Effective window of operations can be small if interpreter does not correct predict the most-likely path through a group of *PowerPC* operations.

PowerPC code and puts them in a group. Examples of straightline *traces* are **A-B-C-E** and **A-B-C-F** in Figure 2.2. A group with both **A-B-C-E** and **A-B-C-F** would *not* be a straightline *trace*. Use of straightline *traces* simplifies many areas of scheduling and optimization, thus helping to meet BOA's real-time requirements. For example, there is at most one reaching definition for each value allowing use of inexpensive algorithms and data structures. In addition translated BOA code for a *trace* can be laid out contiguously in memory, which may require that the sense of some conditional branches be inverted from the original *PowerPC* code. This contiguous layout improves ICache packing and improves the ability to fetch instructions quickly, similar to static code layout techniques [8].

On the downside if the most likely *trace* during interpretation turns out not to be the most likely *trace* later, then there is a smaller effective instruction window from which BOA can extract operations to run in parallel. For example *trace* **A-B-D-H** in Figure 2.2 might be most likely during interpretation, while *trace* **A-B-C-F** turns out to be most likely later. Instead of being able to take advantage of any parallelism among *PowerPC* operations on the long path **A-B-D-H**, only parallelism among the smaller number of operations along the **A-B** path is usefully exploited.

2.1 Trace Formation

BOA instruction *traces* are formed along a single path after interpreting the entry point of a *PowerPC* operation sequence several times. We decided to follow single

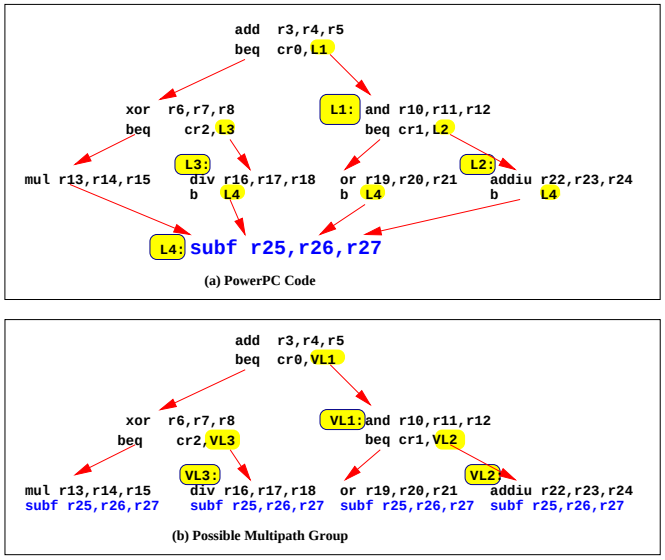

Figure 2.3: “Code Explosion” possible from multipath groups.

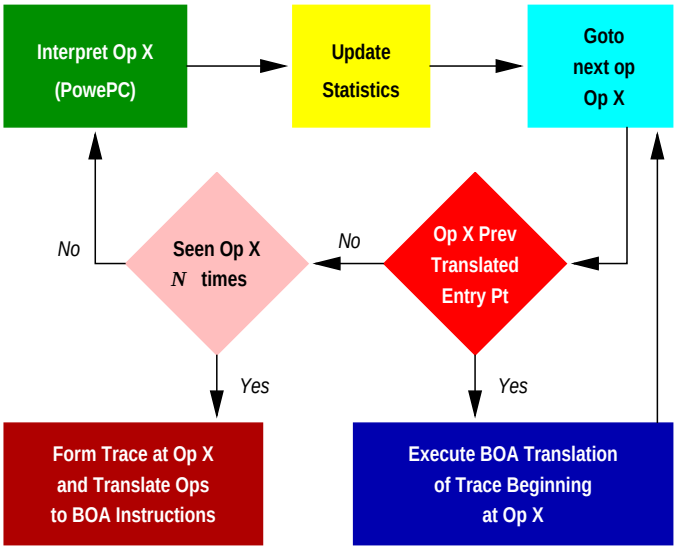

Figure 2.4: Flowchart of BOA Operation.

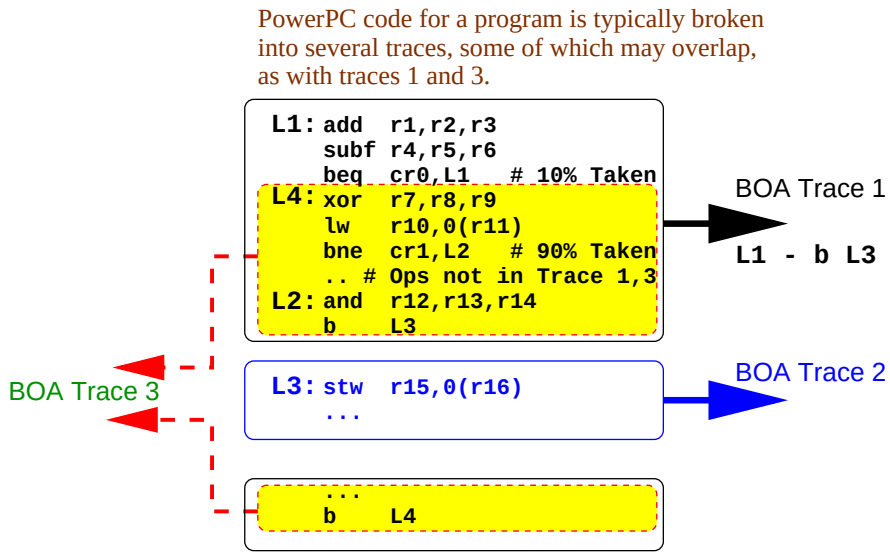

Figure 2.5: *PowerPC* code for a program yields several BOA groups.

paths for two reasons: (1) Instructions can be fetched more quickly and easily if they typically are grouped consecutively in memory. If the most likely direction of a *PowerPC* branch is taken, then the sense of the branch is inverted and the target code added to the trace as straightline code. (2) The number of operations can easily increase exponentially over the number of native *PowerPC* operations if multiple paths are followed. In Figure 2.3, the `subf` occurs only once in the original *PowerPC* code, but could occur 4 times in a multipath group.

During BOA's interpretation phase, statistics are kept on the number of times each conditional branch is executed as well as on the total number of times it is taken, thus allowing a dynamic assessment of the probability the branch is taken. Similar information is also kept about the targets of register branches.

Figure 2.4 gives a flowchart of how BOA operates. As can be seen in Figure 2.4, once the *trace* entry has been seen beyond a threshold number of times, the code starting at the entry point is assembled into a *PowerPC trace*, and translated into a BOA instruction *trace* for efficient execution on the underlying hardware. At each conditional branch point, the most likely path is followed. As each conditional branch is reached during the translation, the probability of reaching this point from the start of the *trace* decreases. When the probability goes below a threshold value, the *trace* is terminated.

A *trace* can also be terminated if the total number of operations in it exceeds a threshold or if the number of store operations in it exceeds the number of entries in the store buffer, as detailed in Section 3. Register branches can optionally end the *trace*.

Alternatively a register branch such as `blr` can be replaced by a test of the most likely branch target, followed by a conditional branch:

```
cmpi  cr_X,LR,<MOST_LIKELY_LR_VALUE>
bne   EXIT_TRACE
<>    # Code Translation from
      # <MOST_LIKELY_LR_VALUE>
...
EXIT_TRACE:
blr
```

2.2 Code optimization and Scheduling

As *traces* 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.

Optimization is particularly useful for dealing with legacy code, but it also brings significant performance improvements to already optimized code. This is possible because unlike a static compiler, the dynamic optimizer does not need to consider the entire control flow graph in making optimization decisions. Instead, short traces are “carved” out of the control flow graph, eliminating all control flow joins.

Eliminating control flow joins opens up numerous optimization possibilities because data usage only has a single reaching definition for each use. We can take advantage of this by performing constant propagation, copy propagation, combining, strength reduction, load/store telescoping, and loop unrolling. Figure 2.6 shows how carving two traces from a control flow graph eliminates multiple reaching definitions into an operation, thereby opening optimization and scheduling opportunities. It is difficult to profitably perform these optimizations in a traditional static compiler, since there are exponentially many paths through code and it is difficult to know which to choose. Profiling an execution of an application and using the results in the next compilation mitigates this problem to some degree, although such profiling is infrequently used for real applications, as we noted earlier. Even if

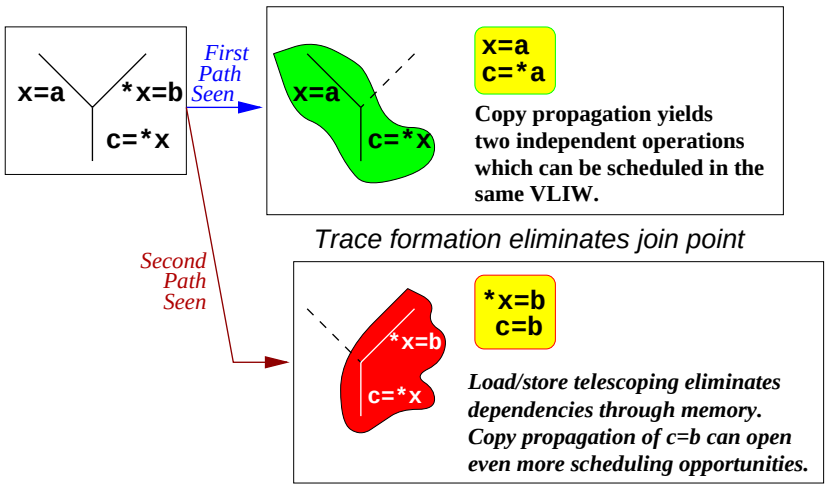

Figure 2.6: Dynamic optimization opens new optimization opportunities. Two definitions reaching use of $\star$x in original program can be carved into two *traces* in BOA binary translation.

profiling is done, it yields a single answer, and cannot adapt to changes in program behavior.

In addition to their own benefit, these optimizations also reduce dependencies and thereby reduce schedule height, allowing more operations to be scheduled in parallel so as to exploit BOA's parallel execution units more effectively.

BOA operations are scheduled to maximize ILP opportunities, taking advantage of speculation possibilities supported by the underlying architecture. The current scheduling approach is greedy, i.e., each operation is scheduled to execute at the *earliest possible time* when (1) all input operands are available, (2) there is a function unit available on which to execute the operation, and (3) there is a free register in which to put the result. In determining this *earliest possible time*, BOA makes use of copy propagation, load-store telescoping, and the other optimizations mentioned above. 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, it must be possible to generate the proper *PowerPC* register and memory values when an exception occurs. Memory ordering is guaranteed by scheduling stores in their original program order. We have looked at several methods to compute the correct

register values in the presence of exceptions.

A first approach consisted of using a modification of **DAISY**'s approach of placing any out-of-order results in registers not architected in *PowerPC* (e.g., R32-R63). **DAISY** then inserted a COPY operation at the original location to copy the value to its proper *PowerPC* register. The number of such COPY operations can be high. BOA attempts to reduce this number by performing COPY operations only at *trace* exits. All values not in their architected *PowerPC* register are copied to the appropriate *PowerPC* register on *trace* exit.

An alternate approach is to copy all register contents to backup registers upon entering a *trace*. Values are computed directly into their *PowerPC* destination register. If the *trace* incurs any exception, the backup registers are restored, and the *trace* is interpreted. Stores go only to a store buffer in this scheme, and hence are never truly reflected to the memory. If no exception occurs by *trace* end, the store buffer contents are flushed to memory. This approach has the disadvantage that if the exception occurs near the end of the *trace*, all the useful work that was done prior to that point must be discarded.

Finally, we chose a hybrid hardware/software approach based on maintaining precise checkpoints at *trace* transition boundaries and the ability to roll back to such a checkpoint otherwise.

Checkpointing is achieved by copying all registers to a set of backup registers at *trace* transitions. Within a *trace*, instructions are scheduled out of order and registers are renamed to support speculative execution. Store operations are executed in original program order, but are marked as `pending` so they can be revoked if an exception occurs. At basic block boundaries within a *trace* (i.e., at any point were control can transfer out of a *trace* through a *side exit*), all architected *PowerPC* registers are in their *home locations*, e.g., R3 is in R3 not in R45.

When a *trace* is exited during the course of normal execution, the *PowerPC* registers are committed into the checkpoint registers, `pending` stores are marked `definite`, and execution continues with the next *trace*. (Control between *traces* is occurs via a `checkpoint and branch` operation.) When an exception occurs, the working registers and all `pending` stores are discarded and the processor state is recovered from the checkpoint registers.

2.3 System issues

Out-of-order loads must be treated specially during scheduling (and execution) so as to conform with *PowerPC* memory ordering semantics. Each LOAD and STORE

in a *trace* is assigned a number indicating its sequence in the *trace*, e.g., the first LOAD/STORE is assigned 1, the second 2, etc. If the hardware detects (1) that a LOAD with a later sequence number has executed earlier than a STORE with an earlier sequence number and (2) that they are to overlapping addresses, an exception is signaled, with the result that the problem LOAD (and any subsequent operations dependent upon it) are eventually re-executed, so as to receive the proper values. Compared to an approach such as LOAD-VERIFY [9], this approach requires fewer memory ports, as most loads are executed only once, instead of twice - once speculatively and for verification.

In a multiprocessor system, *PowerPC* memory semantics also require that if two load operations occur to the same location, the second load cannot receive a value older than the first load. With our scheduling approach, this could happen if the second load were speculatively placed before the first. To detect when errors arise from this problem, the hardware uses the sequence numbers to check not only if a speculative LOAD has an address overlapping with a STORE address, but also if it has passed another LOAD with an overlapping address. Either case triggers an exception.

Another issue with speculative load operations occurs when a speculative load attempts to access non-cacheable memory, such as an I/O location. Such operations cannot be allowed to complete, as they can have side effects. For example, many I/O devices use a single location for all reads, and sequentially place a new value at the location each time it is read. To avoid this problem, BOA has special hardware to detect and quash any non-cacheable load operations. Detection is relatively simple since we are translating *PowerPC* code and uncacheable memory areas are designated by the TLB. While speculative loads are squashed silently, non-speculative memory operations raise an exception and require special handling by the VMM. Code segments which frequently trigger exceptions due to memory operations into I/O space can be translated using serial semantics to reduce excessive overhead. This approach meets the double requirements of preserving semantic correctness in all cases, and ensuring good performance on actual device driver code performing operations in the I/O space.

BOA uses an **LRA** (**L**oad-**R**eal-**A**ddress) operation when branching between *traces* of translated instructions.. **LRA** operations are placed at the start of each *trace* by the scheduler. When executed, **LRA** checks that the TLB and page tables still map the virtual address for the start of this *trace* in the same way that they did when this *trace* was originally translated. If not, a trap occurs, and the BOA system software destroys this *trace* and begins interpreting at the proper address.

We have investigated whether **LRA** and its accompanying operations should

be scheduled as a separate “prologue” to the *trace* or whether they should be scheduled in the *trace* itself. Scheduling as a prologue has the advantage that another *trace* on the same page can branch directly to the “real” code for the *trace* and skip the prologue, since the **LRA** check was already made by this prior *trace* (or perhaps some *trace* prior to it). Scheduling **LRA** and its accompanying operations into the *trace* has the advantage that their operation can often be overlapped with normal *trace* execution. In general we found this second scheme to work better, although typically by less than 5%.

Chapter 3

BOA Architecture Support for Binary Translation

BOA is specifically architected to reduce control logic and facilitate simple high frequency implementation, even if some operations require additional cycles to execute or some *PowerPC* operations need to be cracked into multiple simpler BOA operations.

3.1 Instruction Set Architecture

BOA is an unexposed architecture with an instruction set specifically designed to support binary translation. As such, the architecture is not intended as a platform for handwritten user code, but instead provides a number of primitives and resources to make it a good target for binary translation. These primitives include operations to support the efficient execution of the translated code and the binary translation firmware.

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

The BOA architecture provides extra machine registers to support efficient code

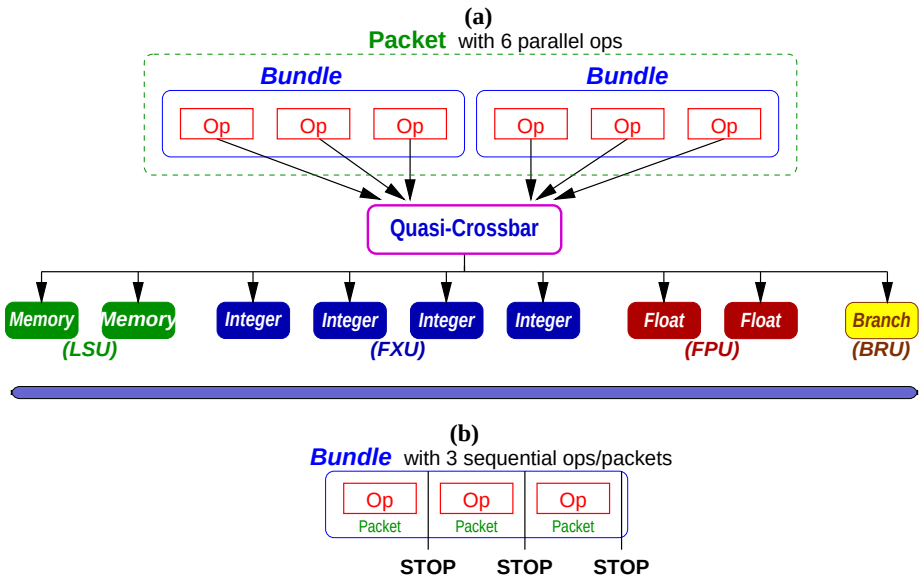

Figure 3.1: BOA instruction formats.

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. Speculation is further supported by speculative state bits associated with each register, such as the `carry`, `overflow` and `exception` bits. These state bits enable state changes from speculative operations to be renamed in conjunction with the speculative destination register until such point that the state change would occur in the original in-order *PowerPC* program.

BOA uses a statically-scheduled, compressed instruction format, similar to an EPIC (**E**xplicitly **P**arallel **I**nstruction **C**ode) architecture, or a variable length VLIW (**V**ery **L**ong **I**nstruction **W**ord) architecture. A parallel instruction, hereafter referred to as a **packet**, can simultaneously issue up to six operations per cycle as illustrated in Figure 3.1(a). Code generation guarantees that no dependencies exist between operations in a *packet* , so they can safely be issued in parallel. As also illustrated in Figure 3.1(a), the six issue slots can contain operations for up to nine different execution units: two memory units, four fixed-point units, two floating-point units, and one branch unit. Any combination of operations can be issued in a *packet*, but to simplify instruction decoding and dispatch, operations must be encoded in this

order in a *packet*.

To ensure efficient memory layout, operations are packed into 128-bit **bundles** containing three operations. Each operation contains 39 bits and one stop bit, using a total of 120 bits among the three operations, leaving 8 bits for future system enhancements, such as support for predication, or additional system functions. Stop bits are used to delineate the *packets* of parallel operations. A de-asserted stop bit indicates the current operation and the next operation belong to the same *packet*, while an asserted stop bit indicates the current operation is the last operation of a *packet* and the next operation begins a new *packet*. *Bundles* are distinct from *packets* in that a *bundle* defines a group of three not-necessarily parallel operations aligned on 128-bit boundaries, while a *packet* defines a variable-sized group of parallel operations (up to six) that is not aligned in memory. Figure 3.1(a) depicts a *packet* consisting of 2 *bundles* and 6 independent operations. Figure 3.1(b) depicts one *bundle* with 3 *packets* and 3 operations which must be executed sequentially.

To simplify decoding and instruction fetch, a restriction was placed on branch targets, requiring them to be aligned on double *bundle* (two *bundles*) boundaries. Prepare-to-branch operations are available for prefetching operations using static branch prediction.

According to its statically-scheduled nature, operation latencies are exposed in the BOA architecture. All branch and fixed-point operations have a single cycle latency, memory accesses require three cycles for performing address generation (AGEN), cache access, and TLB access (TLB), and floating point operations have multi-cycle latencies. Additionally, each operation has an additional one cycle latency penalty because no bypassing is provided within the BOA architecture. One cycle must elapse before a result may be used by a successor operation. The lack of bypass is due to high frequency wire delay costs of broadcasting each result to all execution units. A full pipeline stage, broadcast (BC), is required for supporting this wire delay.

3.2 Implementation

Figure 3.2 depicts a possible implementation of the BOA architecture. For achieving high frequency, the processor assumes a simple hardware design with a medium-length pipeline. The basic processor provides stall-on-use capability, dynamic support for out-of-order loads and stores, decoupled fetch and execute pipelines, and a commit/recirculate scheme for pipeline control. The pipeline consists of two fetch stages (F1 and F2), one instruction decode stage (D1), one issue stage (I1), a

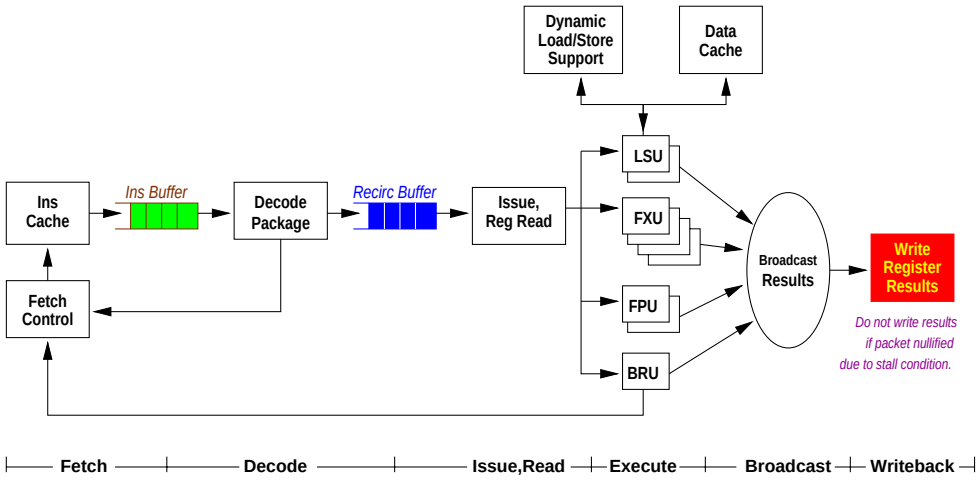

Figure 3.2: The BOA Processor.

register access stage (RF), one to four execution stages (EX), a broadcast stage (BC), and a writeback stage (WB). Only static branch prediction is provided, with a branch misprediction penalty of 7 cycles.

While dynamic scheduling and dynamic branch prediction are considered effective techniques for increasing the execution rate of a processor, designing for high frequency allows only limited dynamic processing support. The first support mechanism uses register scoreboarding to attempt to lessen the impact of stalled load operations, enabling in-order issue to continue in the presence of non-dependent memory stalls. The processor also provides load and store queues for checking for address conflicts between loads and stores which have been reordered during translation, as described in Section 2.3. The third dynamic support mechanism is the use of instruction buffers to decouple the fetch pipeline from the execute pipeline, which is effective at hiding some of the instruction fetch stall penalties, improving overall performance.

The final dynamic support mechanism is a novel pipeline control method that enables the pipeline to automatically advance on each processor cycle [10]. Assuming the code is scheduled properly, only memory stalls should be capable of holding up the execution pipeline. Instead of checking for the existence of a stall before proceeding, the pipeline is automatically advanced every cycle. Upon issuing a new *packet*, the *packet* is both issued and copied into the recirculation buffer (see Figure 3.2), which holds a copy of the contents of every *packet* currently exe-

Cache	Bytes	Line Size	Assoc	Hit Latency
I1	256K	256	4	1
D1	64K	128	2	4
Shared L2	4M	128	8	14

Table 3.1: BOA Cache Hierarchy

cuting. The existence of a stall in the execution pipeline may then be determined late in the execution process and indicated to the appropriate *packets* prior to their committing results during the writeback stage. The dependent *packet* and all subsequent *packet*s are canceled and then reissued from the recirculation buffer. The recirculating *packet*s will repeat the process of issuing, progressing down the execution pipeline. While the stall condition remains during reissue, the *packet*s are continually canceled and reissued from the recirculation buffer until the processor stall completes. This assumed pipeline advancement strategy simplifies pipeline control.

The processor contains separate first-level data and instruction caches, and a joint second level cache. Cache hierarchy details are shown in Table 3.1.

3.3 High Frequency Design Considerations

While a simple instruction set is a requirement for achieving high frequency, many additional factors go into the design of high frequency processors. One significant performance limitation is processor control logic. In previous processor generations, logic delay was the primary limiting factor of processor speed, while wire delay had minimal impact. Newer technologies show wire delay becoming the more significant factor. Novel microarchitectural design and circuit techniques are necessary to meet the frequency challenge presented by the 1999 **ITRS** (**I**nternational **T**echnology **R**oadmap for **S**emiconductors).

The 1999 ITRS calls for achieving a high performance microprocessor with a 2 GHz on-chip clock by the middle of the next decade, while CMOS technology is predicted to provide only $1.9\times$ increase in FET performance, with an $8\times$ increase in numbers of transistors per chip over typical 1999 $0.18\mu m$ CMOS capabilities. To meet and exceed these cycle time goals, the BOA microarchitecture is designed to allow a worst case cycle time of 700ps in a current $0.18\mu m$ CMOS bulk technology under nominal process and temperature conditions. This cycle time target represents more then 50% improvement over reported $0.25\mu m$ designs [12, 13] scaled

to 0.18μm based on the 1997 **ITRS** roadmap and should allow operation in excess of 2 GHz in the next few years.

The BOA microarchitecture also accommodates the expected relative increase in wire delay of future advanced CMOS processes by allowing a full cycle to transmit data across the CPU core, as indicated by the Broadcast Results stage of the BOA pipeline in Figure 3.2. With these constraints, the BOA pipeline and control were designed to use static scheduling, in-order execution, and a stall-on-use scoreboarding method to hide load latencies.

The BOA general purpose register file is implemented as replicated two-read, six-write register file. Each of the four fixed point units and two load/store units has its own copy. Coherency is maintained by simultaneous write-back of results to all six copies.

Floorplanning studies also show a wire length benefit from having separate GPR copies for each unit.

Chapter 4

Experimental Results

At the outset, we note that all CPI measurements in this paper refer to *PowerPC* equivalent CPI. Thus, if a *PowerPC* program executes 200 operations, and BOA requires 100 cycles to execute this program, then the CPI would be 0.50. Our simulation environment utilized a set of trace-based tools for performance analysis. Both **SPECint95** and **TPC-C** traces were analyzed. Each **SPECint95** trace was composed of 100M operations consisting of 50, 2M operation segments representing the critical code loops. The **TPC-C** trace was composed of 170M operations.

4.1 Overhead Calculations and Reporting Technique

In order to get a detailed understanding of BOA architectural performance, a number of factors were taken into account. These factors included results obtained directly from the tool set as well as additional CPI performance adders arising from the binary translation technique. These overheads are dependent on key measurements of both the hardware and attributes of the executing code and are reported in table 4.1.

The overall CPI was calculated from nine components: the base CPI represents the execution time of executing VLIW instructions in translated trace groups. Branch misprediction is modeled separately, and included as branch misprediction penalty. Cache memory adders account for the cost of cache misses, broken down as instruction and data cache misses at the first level, and misses from the joint second level cache. In addition, TLB miss traffic is modeled. The VMM cost is broken down into the cost of profiling and interpretation, translation, and system cost for achieving precise exceptions. This cost includes the adverse effects that

Variable	Description	Value
R	Reuse Rate: precomputed average number of times a given instruction is reused in a program	100000
T	Retranslation Rate: average number of times an instruction is translated	calculated at runtime
$CPI_{Translator}$	Translator CPI: estimated average number of cycles to translate a single instruction by software	2500
$CPI_{Interpreter}$	Interpreter CPI: estimated average number of cycles to interpret a single instruction	20 (30 if looking for exit points after RFI)
E	Exception Rate: estimated average rate of synchronous exceptions encountered during execution	1 exception / 20000 instructions
ICPT	I-Cache Pollution Impact per Translation: we estimate every translation flushes the first-level instruction cache	2048 lines/cache * 10 cycles/line = 20480 cycles/cache
P	Path Length: number of sequential PowerPC instructions comprising a translation	computed at runtime
Pt	Profile Threshold: number of times a profile block must be interpreted before it triggers translation	15
$CPI_{Profiler}$	Profiler CPI: estimated average number of cycles to update/create profile block, per interpreted instruction	100 cycles / 5 instructions = 20 cycles/instruction

Table 4.1: Parameters for computing binary translation overhead

VMM execution has on the memory hierarchy be expelling data and instruction from caches to hold code from the BOA VMM.

The base CPI is computed as the number of cycles spent executing in translation groups:

$$CPI_{Base} = \frac{\text{Cycles Executing BOA Groups}}{\textit{PowerPC} \text{ Instructions in Those Groups}}$$

The cost of translation overhead is computed by amortizing the retranslation rate T, i.e., the number of times each instruction is translation, over the reuse rate R and then multiplying it with the cost of translating a single instruction. Translation causes additional penalties by displacing already translated VLIW code from the translated program parts from the cache as the translator executes. This event event occurs in for each translation which is created, but since multiple instructions are created at the same time, it is amortized over the reuse of all instructions which have been translated in a group:

$$CPI_{Translation} = \frac{T}{R} \times CPI_{Translator} + \frac{ICPT \times T}{R \times P}$$

Detailed analysis of **SPECint95** benchmarks as well as real-time instrumentation of server workloads (such as databases and webserving applications) suggest code reuse rates are substantially greater than our lower bound average of $R = 100,000$. However, there are workloads which have significantly lower reuse rates, thereby increasing the translation component of CPI.

Interpretation and profiling CPI is computed by estimating that for each translation T created, the instructions has previously been interpreted a threshold number *Pt* and the amortize it over the reuse rate R:

$$CPI_{Interp\ and\ Profile} = \frac{T \times Pt}{R} \times (CPI_{Interpreter} + CPI_{Profiler})$$

When the system returns from servicing an exception, a new group is not formed. Instead, execution is started interpretatively, until an existing group is found. This is intended to prevent the formation of groups at random starting addresses in response to exceptions. The cost of this interpretation is a function of the exception rate E and the average distance of an excepting instruction from the beginning of the next group approximated as $P/2$:

$$CPI_{Exception} = \frac{E \times P}{2} \times (CPI_{Interpreter} + CPI_{Interpreter\ with\ Exit\ Check})$$

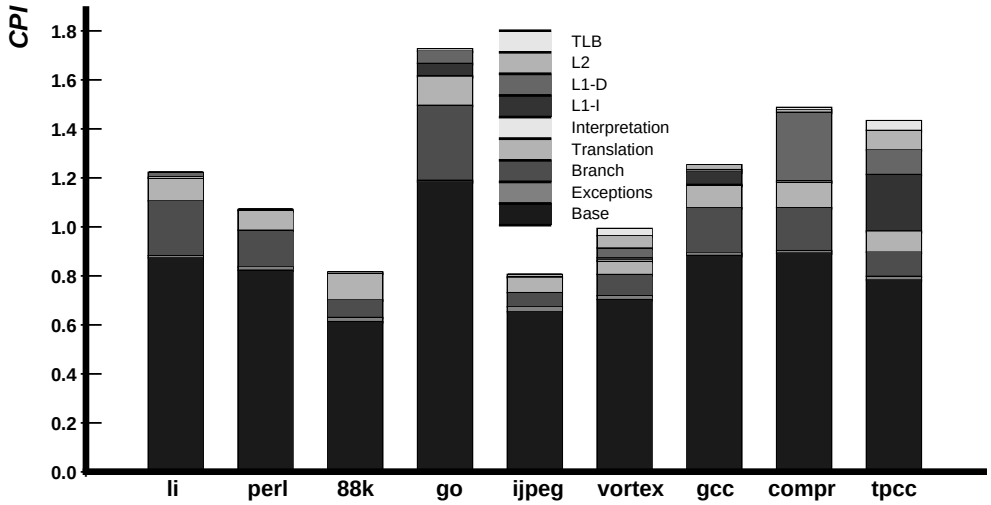

Figure 4.1: BOA Baseline CPI

The memory CPI components are computed by cache simulation, and the branch misprediction penalty is derived by the frequency of the misprediction rate of the static branch prediction used in BOA.

4.2 Data and Results

Figure 4.1 details the preliminary BOA CPI performance for the **SPECint95** suite as well as **TPC-C** *(DB2)*. These results provide the basis for comparison on subsequent data charts. Notice that the two largest components are the *Base CPI* and the branch overhead.

In our experimentation, we found quickly that the largest component, the *Base CPI*, was a function of achievable dynamic *trace* size. The reason for this relationship was that the scheduler needed enough operations to optimize effectively. In order to obtain larger average dynamic *trace* size, we found the two most relevant properties were: (1) the static length and (2) "quality" of the *traces* where quality refers to a measure of how likely control flow is to reach the end of the *trace*.

Clearly, there is a balance to be obtained as statically long *traces* offer opportunity to perform more scheduling, however longer *traces* also increase the likelihood of exiting the *trace* early. An early exit from a *trace* further detracts from CPI in

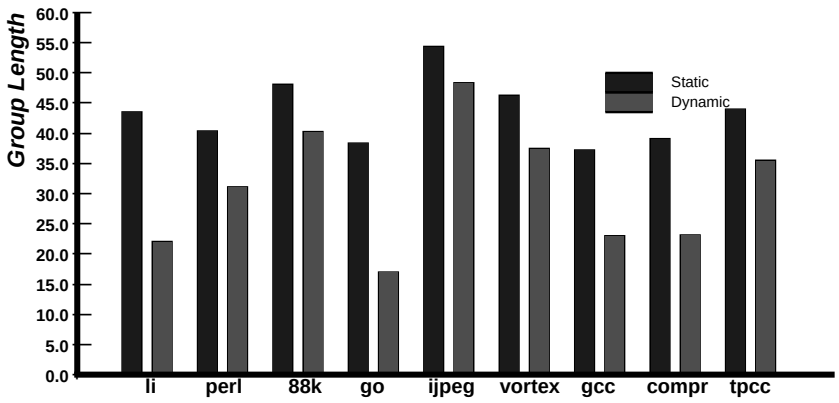

Figure 4.2: Average BOA Trace Sizes

that scheduling opportunity has been lost, since operations from the wrong stream were executed speculatively. Dynamic *trace* sizes observed from the experiment above are shown in Figure 4.2.

Premature group exit has additional associated penalties due to branch mispredictions and instruction cache effects. Since side exit branches within a trace are predicted to be not taken, branch misprediction occurs on premature group exit. Branch repair in the current architecture costs 7 cycles, and thus can degrade performance substantially in the case of frequent branch mispredictions. Overly long static traces also use up instruction cache space, leading to less dense packing because infrequently executed code at the end of trace segments is interspersed with more frequently executed code in the same cache line. Thus, fewer frequently executed instructions can be placed in the instruction cache and higher cache miss rates result.

We next conducted experiments to increase the static predictability of branches and thereby enhance the "quality" of *traces*. This was achieved by profiling branches, and using selective *trace* extension: only if a branch was likely to be taken a certain percentage of the profiled iterations would the *trace* be extended by appending the operations at the target of the branch. Otherwise, the *trace* would be terminated and a new *trace* would be started. We expected this approach to drastically reduce the number of early exits (i.e., branches which are taken out of the *trace* before the *trace* end is reached). Figure 4.3 shows the reduction in early exits achieved for bias values of 8, 12, and 15 out of 15 iterations. Note how higher bias values for extending the *trace* improve the quality of the *traces*.

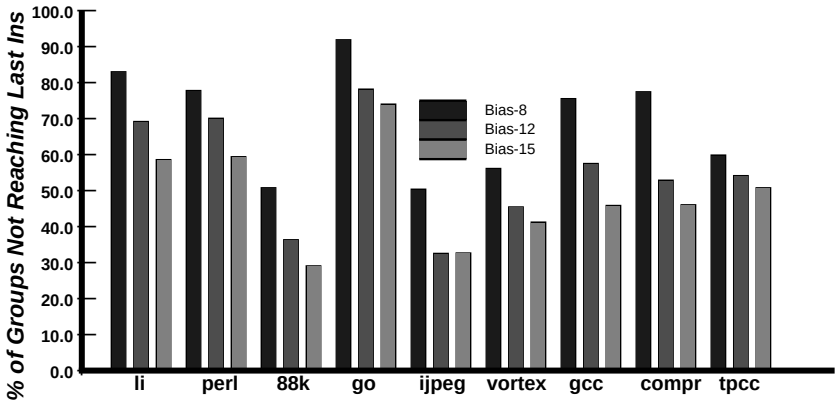

Figure 4.3: Effect of Bias on BOA Early Exits

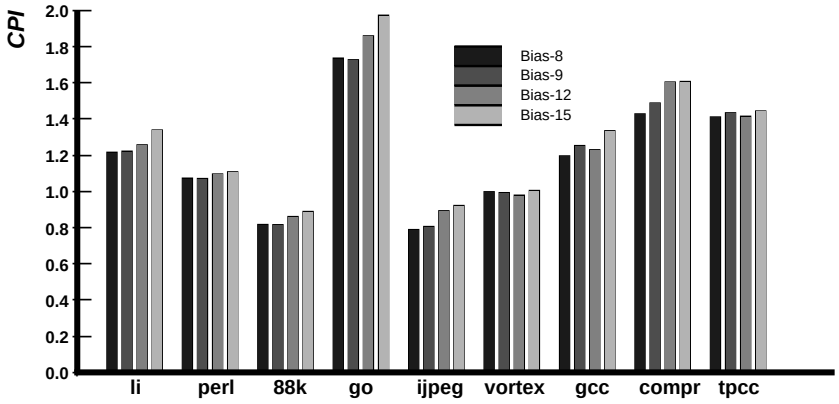

Figure 4.4: Effect of Bias on BOA CPI

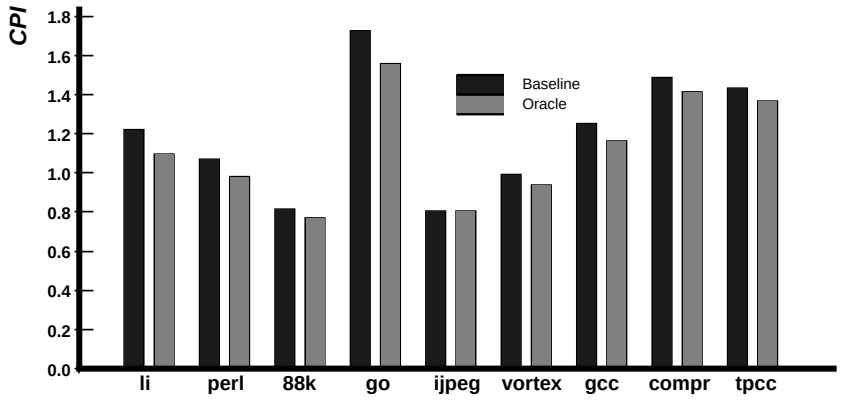

Figure 4.5: "*Oracle*" Static Conditional Branch Prediction

By reducing the number of early exits, we hoped to improve the CPI of the system under evaluation. However, Figure 4.4 demonstrates that the reduced number of early exits and misspeculation cannot compensate for reduced static *trace* length, and as a result the overall CPI suffered. The lesson to draw from this is that static *trace* length is overall a more important contributor to the average dynamic *trace* length and performance than the number of early exits. It is noteworthy that longer static *trace* length does result in more code duplication which reduces instruction cache effectiveness. This effect is responsible for the reduced instruction cache CPI adder for higher bias values.

Since our results showed that *trace* length was key to performance, we proposed an additional experiment where we approximated an upper bound value for static prediction, which we refer to as "*Oracle Static Prediction*." Oracle Static Prediction is based on perfect static branch prediction using profile-directed feedback from the same run of the trace by processing the trace twice. A first scan through the trace selects the most likely direction for each branch. This direction was used in building the *traces* on a second iteration. Figure 4.5 compares the results obtained for this oracle prediction with our initial baseline results, based on profiling performed during the interpreted executions of each branch.

Chapter 5

Conclusion

We have described the BOA architecture, and how dynamic binary translation can be used to make it compatible with *PowerPC*. We have also outlined how BOA can be implemented in a very high speed design, so as to be able to run at a speed of more than 2 GHz by the middle of the next decade, assuming that the assumptions in the *ITRS Roadmap* hold true. The CPI of BOA is around 1, but varies somewhat across benchmarks in a manner roughly proportional to the dynamic *trace* size, thus substantiating the expected result that a larger window of operations permits better CPI. A CPI of 1 is comparable to that reported for many modern-day superscalar processors, but is a bit worse than that reported for other experimental architectures. However, BOA is backed up by an extremely high frequency design, which can compensate for some CPI loss.

Binary translation has been shown to be a viable technique for generating competitive performance on existing workloads. Our analysis suggests that the critical performance sensitivities are minimizing instruction cache overheads, maximizing dynamic *trace* lengths, and effectively managing the amortization of translation overhead.

The placement of optimized traces in memory allows for efficient prefetching and helps keep down instruction cache penalties. Saving only the most likely branch path code minimizes the potential explosive bloating of trace code in memory (and the subsequent expensive retranslation). We believe that this approach has managed the instruction cache penalty to the extent that it may be relaxed to obtain improvements in other areas.

Dynamic *trace* length optimization appears to be the main source of improvements in CPI. We have shown that with relatively simple profiling and scheduling we can obtain traces of reasonable length and CPI. Although care must be taken

to avoid code bloat and attendant instruction cache penalties, we believe that further balanced improvements in profiling and scheduling may lead to overall performance gains and are an area for future investigation.

Acknowledgments

The authors would like to thank Albert Chang, Kemal Ebcioğlu, Martin Hopkins, and Patrick Bohrer for their critical contributions to the work presented.

Bibliography

[1] 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.

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

[3] K. Ebcioğlu and E. Altman. DAISY: dynamic compilation for 100% architectural compatibility. Research Report RC 20538, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1996.

[4] 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.

[5] K. Ebcioğlu, J. Fritts, S. Kosonocky, M. Gschwind, E. Altman, K. Kailas, and T. Bright. An eight-issue tree-VLIW processor for dynamic binary translation. 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.

[6] 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.

[7] K. Ebcioğlu, E. Altman, S. Sathaye, and M. Gschwind. Optimizations and oracle parallelism with dynamic translation. In *Proc. of the 23nd ACM/IEEE*

International Symposium on Microarchitecture, pages 284–295, Haifa, Israel, November 1999. ACM, IEEE, ACM Press.

[8] Karl Pettis and Robert Hanson. Profile guided code positioning. In *Proc. of the 1990 Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, NY, June 1990. ACM.

[9] J. Moreno and M. Moudgill. Method and apparatus for reordering of memory operations in a processor. US Patent No. 5,758,051, May 1998.

[10] M. Gschwind. Pipeline control mechanism for high-frequency pipelined designs. Filed for U.S. Patent, January 1999.

[11] J. Silberman, N. Aoki, D. Boerstler, J. Burns, S. Dhong, A. Essbaum, U. Ghoshal, D. Heidel, P. Hofstee, K. Lee, D. Meltzer, H. Ngo, K. Nowka, S. Posluszny, O. Takahashi, I. Vo, and B. Zoric. A 1.0GHz single-issue 64b PowerPC integer processor. In *Proc. of the 1998 Internation Solid State Circuits Conference*, pages 230–231, San Francisco, CA, February 1998.

[12] G. Northrop, R. Averill, K. Barkley, S. Carey, Y. Chan, Y.H. Chan, M. Check, D. Hoffman, W. Huott, B. Krumm, C. Krygowski, J. Liptay, M. Mayo, T. McNamara, T. McPherson, E. Schwarz, L. Sigall, T. Slegel, C. Webb, D. Webber, and P. Williams. 600MHz G5 S/390 microprocessor. In *Proc. of the 1999 International Solid-State Circuit Conference*, pages 88–89, San Francisco, CA, February 1999.

[13] S. Fischer, R. Senthinathan, H. Rangchi, and H. Yazdanmehr. A 600MHz IA-32 microprocessor with enhanced data streaming for graphics and video. In *Proc. of the 1999 International Solid-State Circuit Conference*, pages 98–99, San Francisco, CA, February 1999.