

BOA: Targeting Multi-Gigahertz with Binary Translation

Sumedh Sathaye¹, Paul Ledak², Jay LeBlanc², Stephen Kosonocky¹, Michael Gschwind¹, Jason Fritts¹, Zachary Filan², Arthur Bright¹, David Appenzeller², Erik Altman¹, Craig Agricola²

Abstract

This paper presents BOA (Binary-translation Optimized Architecture), a processor designed to achieve high frequency by using software dynamic binary translation. Processors for software binary translation are very conducive to high frequency because they can assume a simple hardware design. Binary translation eliminates the binary compatibility problem faced by other processors, while dynamic recompilation enables re-optimization of critical program code sections and eliminates the need for dynamic scheduling hardware. 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.

1.0 Introduction

The design of processors with clock speeds of 1 GHz or more has been a topic of considerable research in both industry and academia. Binary translation presents an interesting alternative for processor design as it enables good performance on simple processor designs. Processors for binary translation achieve maximum performance by enabling high frequency processors while still exploiting available parallelism in the code. The effect of both of these optimizations is to minimize overall execution time [1]:

$$Execution\ Time = (Number\ Of\ Instructions) \left(\frac{Cycles}{Instruction} \right) \left(\frac{Seconds}{Cycle} \right)$$

One method of minimizing execution time is instruction set simplicity which can enable high frequency design. Many approaches can be taken to achieve high frequency while maintaining compatibility with existing code. Three popular possible methods include:

- **Hardware Cracking:** Hardware cracking of complex instructions into simpler "micro" operations is frequently used (e.g., Intel processors such as Pentium Pro, Pentium II, and Pentium III). Nair and Hopkins' DIF processor [2] also uses a hardware approach, to crack and schedule simple PowerPC operations into groups for high-speed execution on a VLIW (Very-Long Instruction Word) processor. Cracked code can be stored in the cache, or cracking can occur on the fly (in the pipeline), but either way, it consumes time and transistors. This limits the achievable frequency.
- **Microcode Emulation:** Microcode can also be used to maintain compatibility, with each instruction of the original architecture going to a fixed microcode routine for emulation on the new (high frequency) architecture. However, if the new instruction set architecture does not closely resemble the original architecture, the total number of instructions executed is much higher than on a direct implementation of the original machine. In addition, there is no opportunity to overlap the execution of instructions from the original machine, thus preventing the exploitation of any intrinsic parallelism.
- **Binary Translation:** A third alternative is dynamic binary translation via software as exemplified by FX!32 [3] or DAISY [4]. Since the translation is done by software, no extra transistors are needed for implementation -- the same resources used to execute the application code are used by the translation software. Assuming that the high frequency processor for a binary translation machine is designed with the original in mind, the high frequency primitive operations can efficiently execute instructions for the original machine. Unlike microcode, the high frequency primitives from several original instructions can be overlapped so as to exploit parallelism. Another advantage of dynamic binary translation over hardware cracking and microcode is adaptability -- the code generated can be adapted to efficiently handle special cases discernible only at runtime. For example if fixed parameters are always passed to a subroutine, the dynamic translation software can generate code which checks for these fixed parameters, and then jump to a highly optimized translation of the subroutine if these parameters are encountered³. Similarly, translations can be optimized to varying degrees depending upon the execution frequency of the code section.

¹ IBM T. J. Watson Research Center, Yorktown Heights, NY, USA

² IBM Server Division, Burlington, VT, USA

³ Our current implementation does not support this optimization.

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. In this paper, we describe the application of dynamic binary translation to the design of BOA (Binary-translation Optimized Architecture) a high-frequency EPIC-style implementation of the PowerPC architecture.

The remainder of the paper describes our approach in designing a high frequency PowerPC compatible microprocessor through dynamic binary translation. Section 2 describes how the processor builds groups of PowerPC instructions and the translation process into the BOA architecture. Section 3 describes the BOA instruction set architecture and details the high frequency implementation. Section 4 gives experimental microarchitectural performance results, and Sections 5 and 6 conclude and describe future work.

2.0 Binary Translation Strategy

Our overall strategy of binary translation is to translate a PowerPC instruction flow into a set of BOA instruction groups which are then stored in memory and executed natively. The system must start the process by interpreting the PowerPC stream of instructions and make intelligent decisions on when and how to translate these instructions into BOA groups. This process is described in Section 2.1.

Once groups have been formed, the translation process continues by scheduling the BOA groups for performance. These optimizations are covered in Section 2.2 where again, intelligent tradeoffs must be made between code optimization, code reuse, and software optimization costs.

Code execution is extremely fast when the processor is able to use BOA groups in memory. These optimized groups are generally much larger than a basic block and can contain many branches. The last instruction in each BOA group is a direct branch to the next group stored in memory. As long as a translation exists for the address pointed to by the final branch in a group, the new groups is fetched and executed. If the code does not exist, the machine re-enters interpretation mode to slowly execute PowerPC instructions and possibly build more BOA groups. Likewise, if a branch is encountered within the BOA group that is mispredicted, native execution must be halted, and a branch is taken back to the interpreter. It is therefore critical that BOA groups not only be as long as possible (to give ample opportunity for code optimization), but also that they be as high quality as possible (to avoid calls back to the interpreter).

2.1 Group Formation

BOA instruction groups are formed along a single path after interpreting the entry point of a PowerPC instruction sequence 15 times. In the PowerPC interpretation process, we decided to only follow single paths of execution as to support very high-frequency hardware (described in Section 3). This single path approach is somewhat similar to that used by DIF [2], except that DIF requires special hardware to form groups and does not interpret PowerPC groups. DAISY [4, 5] by contrast forms groups from operations along multiple paths, and thus requires somewhat more time-consuming group formation and scheduling heuristics.

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.

As noted, once the group entry has been seen 15 times, the code starting at the entry point is assembled into a PowerPC group, and translated into BOA instruction groups for efficient execution on the underlying hardware. At each conditional branch point, the most likely path is followed. For efficient execution, it is best if the translated path always falls through at conditional branch instructions. In some cases, this requires that the sense of the branch be changed. As each conditional branch is reached during the translation, the probability of reaching this point from the start of the group decreases. When the probability goes below a threshold value, the group is terminated.

For example, if a particular branch goes less than 12 times in its more likely direction, we might decide to terminate the group. Later

in the paper, we refer this as "bias-12", i.e. the branch must go in one direction at least 80% (12 of 15) of the time. Since we are applying this strategy in a runtime environment, quick group formation is essential, even if it can produce sub-optimal results. For example, consider the following PowerPC code:

```

A    blt  cr0,L1    # 60% taken
B    bgt  cr4,L4    # 0% taken
C    bgt  cr5,L5    # 0% taken
D    nop
...
E  L1:  blt  cr1,L2    # 60% taken
...
F  L2:  blt  cr2,L3    # 60% taken
...
G  L3:  nop

```

Following the most likely branch direction in each case, our strategy will form a group with the instructions A,E,F,G even though there is only a 21.6% (0.63) chance of executing G. On the other hand, the path A,B,C,D also has 3 conditional branches, but has a 40% chance of being taken. A,B,C,D is not chosen because the branch condition at A is more likely to be taken than fall-through as is required for A,B,C,D.

A group can also be terminated if the total number of operations in it exceeds a certain value or if the number of store operations in it exceeds the number of entries in the store buffer, as detailed in Section 2. Register branches can optionally end the group. Alternatively a register branch such as "blr" can be replaced by a sequence similar to:

```

cmpi cr_X,LR,<MOST_LIKELY_LR_VALUE>
bne  EXIT_GROUP
<> # Code Translation from <MOST_LIKELY_LR_VALUE>
...
EXIT_GROUP:
Blr

```

(This sequence is slightly simplified for clarity, and does not deal with issues such as how to perform an immediate compare with a 64-bit constant, or the mechanics of indirect branching on the underlying machine.)

Other issues must be addressed during the interpretation and group formation process. For example, there might be two common paths from a particular entry point, with one path likely for certain predecessor paths, and the other path likely for the remaining predecessor paths. Since our underlying architecture is geared towards single-path groups, we cannot put both paths in one group. However, there may be two translations for the same entry point. One is chosen based on what has executed immediately prior.

2.2 Scheduling

The scheduler schedules operations from the group one at a time, and if need be, cracks them into simpler operations supported by the underlying high frequency BOA architecture. The BOA operations are then scheduled greedily, i.e. 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.

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. Memory ordering is guaranteed by scheduling stores in their original program order. We have looked at a couple of ways of getting correct register values.

One is to use 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 group exits. All values not in their architected PowerPC register are copied to the appropriate PowerPC register on group exit.

An alternate approach is to copy all register contents to backup registers upon entering a group. Values are computed directly into their PowerPC destination register. If the group incurs any exception, the backup registers are restored, and the group 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 group end, the store buffer contents are flushed to memory. This approach has the disadvantage that if the exception occurs near the end of the group, all the useful work that was done prior to that point must be discarded.

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 group is assigned a number indicating its sequence in the group, 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 [6], this approach requires fewer memory ports, as most loads are executed only once, instead of twice - once speculatively and once 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 wrinkle with speculative LOAD's occurs when a speculative LOAD attempts to access non-cacheable memory, such as an I/O location. Such LOAD's 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's. Detection is relatively simple since we are translating PowerPC code, detection is relatively simple. PowerPC maintains 4 WIMG bits which define how the page may be referenced. If the I bit is set, caching is inhibited for the page. (The other bits control write-through, coherence, and guardedness properties.)

BOA uses an LRA (Load-Real-Address) operation when branching between groups of translated instructions or when a group spans pages. LRA operations are placed at the start of each group by the scheduler. When executed, LRA checks that the TLB and page tables still map the virtual address for the start of this group in the same way that they did when this group was originally translated. If not, a trap occurs, and the BOA system software destroys this group 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 group or whether they should be scheduled in the group itself. Scheduling as a prologue has the advantage that another group on the same page can branch directly to the "real" code for the group and skip the prologue, since the LRA check was already made by this prior group (or perhaps some group prior to it). Scheduling LRA and its accompanying operations into the group has the advantage that their operation can often be overlapped with normal group execution. In general we found this second scheme to work better, although typically by less than 5%.

3.0 BOA Architecture Support for Binary Translation

This section describes the underlying architecture of BOA (Section 3.1), its implementation (Section 3.2), and how this implementation supports high frequency (Section 3.3). BOA is specifically architected to reduce control logic and facilitate simple high frequency implementation, even at the cost of incurring additional CPI penalties, i.e., the "cycles per instruction" term in the overall runtime defined in the Introduction.

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 instructions to support the efficient execution of the translated code and the binary translation firmware.

BOA uses a statically-scheduled, compressed instruction format, similar to an EPIC (Explicitly Parallel Instruction Code) architecture, or a variable length VLIW (Very-Long Instruction Word) architecture. A parallel instruction, hereafter referred to as a "packet", can simultaneously issue up to six operations per cycle. Code generation guarantees that no dependencies exist between operations in a packet, so they can safely be issued in parallel. 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. For simplified decoding, the architecture requires this order to be followed by operations within 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 predication⁴ 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

⁴ Current software does not generate predicated code, as predicated code requires time-consuming program analysis overhead.

a variable-sized group of parallel operations (up to six) that is not aligned in memory. 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 instructions are available for prefetching instructions 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 successive 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. However, load operations are scoreboardd to allow stall-on-use semantics for memory operations.

The BOA architecture defines execution primitives similar to the PowerPC architecture in both semantics and scope. However, not all PowerPC instructions have an equivalent BOA primitive. Many PowerPC instructions 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, data layout and instruction primitives of the BOA architecture are similar to the PowerPC architecture: primitives are 32-bit data, with support for signed and unsigned byte and halfword data. Comparison operations store results in 4-bit condition registers, whose bits represent ‘less than’, ‘equal to’, ‘greater than’, and overflow.

The BOA architecture provides extra machine registers to support efficient code scheduling and aggressive speculation using register renaming. Data are stored in one of 64 general-purpose 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, and a checkpointing capability for the architected PowerPC machine state. These speculative state bits enable state changes from speculative operations to be saved with the speculative destination register until such point that the state change would occur in the original in-order PowerPC program.

The checkpointing mechanism allows the implementation of precise exceptions by all registers containing the PowerPC machine state. In addition, a gated store buffer provides the capability to undo store operations. This hardware support allows the full re-ordering of operations between checkpoints without further consideration to the implementation of precise exceptions. When an exception does occur, control is transferred to the BOA Virtual Machine Monitor, which restores the last checkpoint, undoes all store operations since that checkpoint, and starts to interpret the PowerPC code in order. Exceptions are then raised in the original program order. Checkpoints are created by a special “checkpoint and branch (conditional)”, and typically correspond to basic block boundaries. Using hardware support for checkpointing is similar to the approach described in [7].

Speculative execution is further supported by speculative load operations, which query the TLB for access mode information. Accesses to locations which may refer to memory mapped I/O locations (such as memory areas which are uncached) are squashed and a speculative exception is indicated for the target register. When such value is later accessed in a non-speculative operation, an exception will be raised and interpretation will start from the last check point. The memory access will then be performed in-order thereby guaranteeing correct semantics. Initial interpretation already detects most such accesses and generates in-order accesses to such locations. This allows good performance without the cost of interpretation for code accessing memory mapped I/O devices, such as frame buffers. Hardware support is necessary to catch memory locations which may have changed their access modes and guarantee correct performance in this case. (Code re-optimization may then be attempted to re-optimize code which triggers this mechanism.)

To enable static reordering of loads, additional hardware support is provided for the two memory ports in the load-store unit. This hardware consists of a 32-entry load reorder queue, a 32-entry store queue, and an 8-entry store miss queue. This support provides detection of load/store collisions, implements store forwarding and ensures proper program ordering of loads. Store operations are dispatched strictly in-order.

A further hardware feature specifically to support binary translation is the LRA instruction. As noted in the previous section, the LRA instructions is used to check whether the page mapping has been modified since the code was generated. This is achieved by translating the effective address of the code page whose translation follows to a physical address. That physical address is then compared to the original page mapping, and if a change in the mapping has occurred an exception transfers to the translator.

3.2 Implementation

A possible implementation of the BOA architecture is shown in Figure 1. For achieving high frequency, the processor assumes a simple hardware design with a medium-length pipeline. The basic processor provides stall-on-use capability,

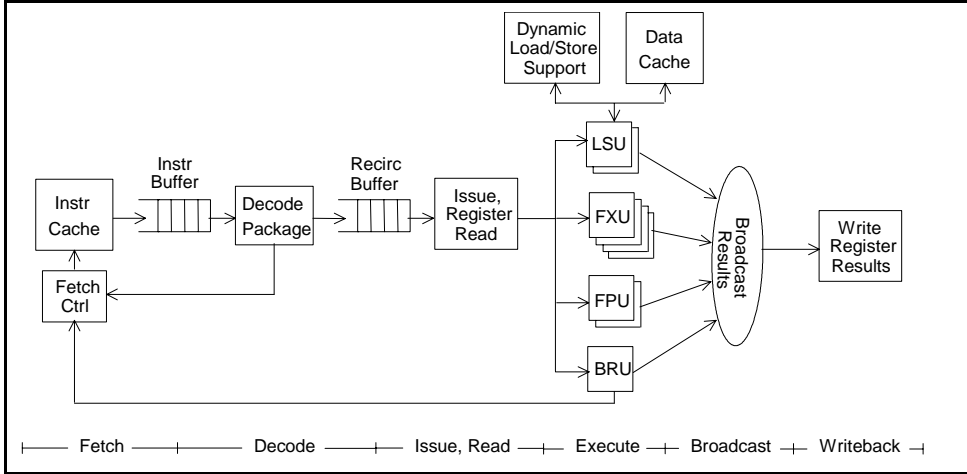


FIGURE 1: The BOA Processor

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 2.2. 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. 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, which holds a copy of the contents of every packet currently executing. 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 packets are canceled and then reissued from the recirculation buffer. The recirculating packets will repeat the process of issuing, progressing down the execution pipeline. While the stall condition remains during reissue, the packets are continually canceled and reissued from the recirculation buffer until the processor stall completes. This assumed pipeline advancement strategy simplifies pipeline control. [8]

The processor contains separate first-level data and instruction caches, and a joint second level cache. Cache hierarchy details are shown below in Table 1.

Cache	Cache Size / Line Size	Associativity	Hit Latency
L1 I-Cache	256K / 256B	4W	1
L1 D-Cache	64K / 128B	2W	4
L2 Shared Cache	4M / 128B	8W	14

TABLE 1: BOA Cache Hierarchy

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 SIA National Semiconductor Roadmap [9].

The 1997 SIA roadmap calls for achieving a 2GHz microprocessor by the middle of the next decade, but CMOS technology is predicted to provide only 1.8x increase in FET performance and 2.8x increase in density over 1999 typical 0.18um CMOS capabilities [9]. To meet and exceed these goals, the BOA microarchitecture is designed to allow a worst case cycle time of 700ps in a current 0.18um CMOS bulk technology under nominal process and temperature conditions. This cycle time target represents more than 50% improvement over reported 0.25um designs scaled to 0.18um based on the SIA roadmap [10, 11], and should allow operation in excess of 2GHz by the middle of the next decade predicted by the SIA roadmap for technology.

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. With these constraints, the BOA pipeline and control were designed to use static scheduling, in-order execution, and a stall-on-use scoreboard method to hide load latencies. Detailed circuit simulation of critical paths was used to verify the cycle time target. To meet these goals, aggressive dynamic circuit techniques were extensively used as described in [12].

Fast hardware assisted rollback of the architected state is crucial for binary translation performance. This mechanism is implemented in the BOA GPR, a 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. Commit and rollback capabilities are provided for the architected registers by use of a master-slave latch configuration. The slave latch stores the backup data each time a commit occurs, when rollback is required, the slave latch dumps it's data to the master latch, restoring the architected state in a single cycle.

4.0 Experimental Results

This section describes the methodology, calculations, and results obtained through our modeling efforts with the BOA architecture. Section 4.1 describes the tool methodologies used to generate results, and Section 4.2 explains the various components of CPI adders which were analyzed. Finally, Section 4.3 details the set of experimental results obtained in trying to optimize the microarchitectural performance of the BOA processor while attempting to maintain very high frequency. At the outset, however, we note that all CPI measurements in this paper refer to PowerPC equivalent CPI. Thus, if a PowerPC program executes 200 instructions, and BOA requires 100 cycles to execute this program, then the CPI would be 0.50.

4.1 Methodology

Our simulation environment utilized a set of trace-based tools for performance analysis. Both SPECint and TPC-C traces were analyzed. Each SPECint trace was composed of 100M instructions consisting of 50 2M instruction segments representing the critical code loops. The TPC-C trace was composed of 170M continuous instructions.

Once the traces were read, the first step in the Binary translation process was to begin the group formation process. Since our methodology was trace-based, the profiler had knowledge of the exact execution of the code. This enabled the program to partition the traces into executed profile blocks, and then form groups from these blocks when the bias and counters were appropriate as described in Section 2.1. A "tip-trace" (a mapping of code execution from block to block) of the executed blocks was then saved to disk along with a binary file of the groups created. Control was then passed to the scheduler.

The scheduler optimized code blocks, ensured MAP consistency (see Section 2.2), and simulated execution of the groups using the tip-trace to obtain cycle counts of the BOA code. The resulting CPI number was referred to as the "Constrained Machine CPI Adder."

Because BOA was defined as a stall on miss machine, cache adders were calculated and taken into account after simulation of the code. In the case of the I-Cache, the scheduler was able to assign I-Cache addresses during simulation and build a separate I-Cache trace. Once the simulation was completed, this I-Cache trace was submitted to the I-Cache simulator to obtain the L1 I-Cache adder. D-Cache adders required a separate program which re-consumed the original PowerPC trace, searching for loads and stores. These were then submitted to a D-Cache simulator to obtain the D-Cache CPI adder. To generate the L2 CPI adder, all instruction addresses were sent to the cache simulator, followed by all data references. Although this did not provide exact intermixing of data and instructions in the L2 cache, with low miss rates, this was a good approximation.

4.2 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 were dependent on key measurements of both the hardware and attributes of the executing code and are shown below in Table 2.

Symbol	Description	Value
R	Reuse Rate : Precomputed average number of times a given instruction is reused in a SPECint95 program.	100,000 ⁵
T	Retranslation Rate : Average number of times a given instruction must be translated.	Calculated at runtime
CPI _{Translator}	Translator CPI : Estimated average number of cycles taken to translate 1 instruction by software.	2,500
CPI _{Interpreter}	Interpreter CPI : Estimated average number of cycles required for software to interpret 1 instruction	20 (30 if looking for exit points after an RFI instruction)
E	Exception Rate : Estimated average rate synchronous exceptions are encountered in the execution of code.	1 exception /20,000 instructions
ICPT	I-Cache Pollution Impact Per Translation : We estimate every translation flushes the I-Cache	2048 lines/cache *10 cycles/line = 20,480 cycles/cache
P	Path Length : Maximum number of sequential PPC instructions that comprise a single translation.	Calculated at runtime
Pt	Profile Threshold : Number of times a profile block must be interpreted before it starts a translation.	15
CPI _{Profiler}	Profiler CPI : Estimated number of cycles to update/create profile block, per instruction interpreted.	100 Cycles / 5 Instructions = 20 Cycles/Instruction

TABLE 2: Binary Translation Overheads

The overall CPI was calculated from nine components as follows:

$$CPI_{Overall} = CPI_{ConstrainedBoa} + CPI_{Translation} + CPI_{Interpretation \& \text{ Profiling}} + CPI_{Exception} + CPI_{D1Cache} \\ + CPI_{I1Cache} + CPI_{L2Cache} + CPI_{TLB} + CPI_{BranchMisprediction}$$

Where {

$$CPI_{ConstrainedBoa} = \frac{\text{Cycles Executing BOA Groups}}{\text{BOA Instructions in those Groups}}$$

$$CPI_{Translation} = \left[\frac{1}{R} CPI_{Translator} T \right] + \left[\frac{ICPT \cdot T}{R \cdot P} \right]$$

$$CPI_{Interpretation \& \text{ Profiling}} = \frac{1}{R} Pt \cdot T (CPI_{Interpreter} + CPI_{Profiler})$$

$$CPI_{Exception} = E \left[CPI_{Interpreter} \frac{P}{2} + CPI_{Interpreter_with_exit_check} \frac{P}{2} \right]$$

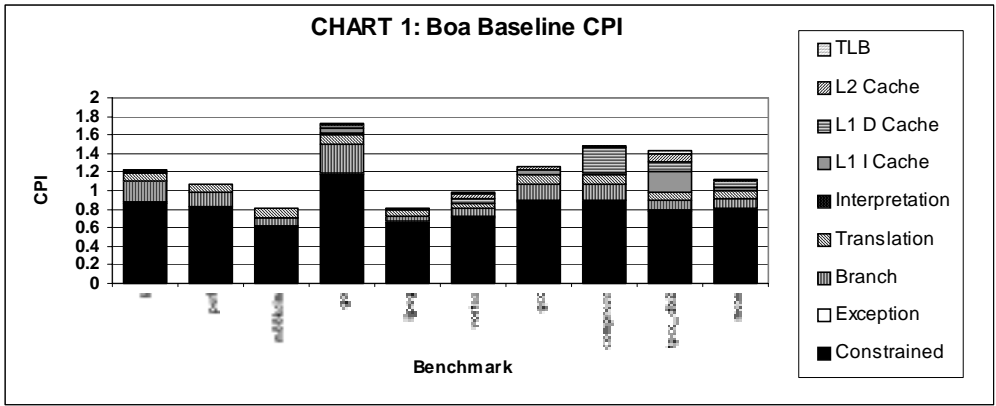
}

4.3 Data and Results

The following chart 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 adders are the constrained machine CPI and the branch overhead. In our experimentation, we found quickly that the largest adder, the constrained machine CPI, was a function of achievable dynamic trace group size. The reason for this relationship was that the scheduler needed enough instructions to optimize effectively. In order to obtain larger average dynamic trace group size, we found the

⁵ Detailed analysis of SPEC benchmarks as well as real-time instrumentation of server workloads (HTTP, Notes) suggest code reuse rates are substantially greater than lower bound average of 100,000. It should be noted that there are workloads which have reuse rates which can be significantly lower, increasing the translation component of CPI.

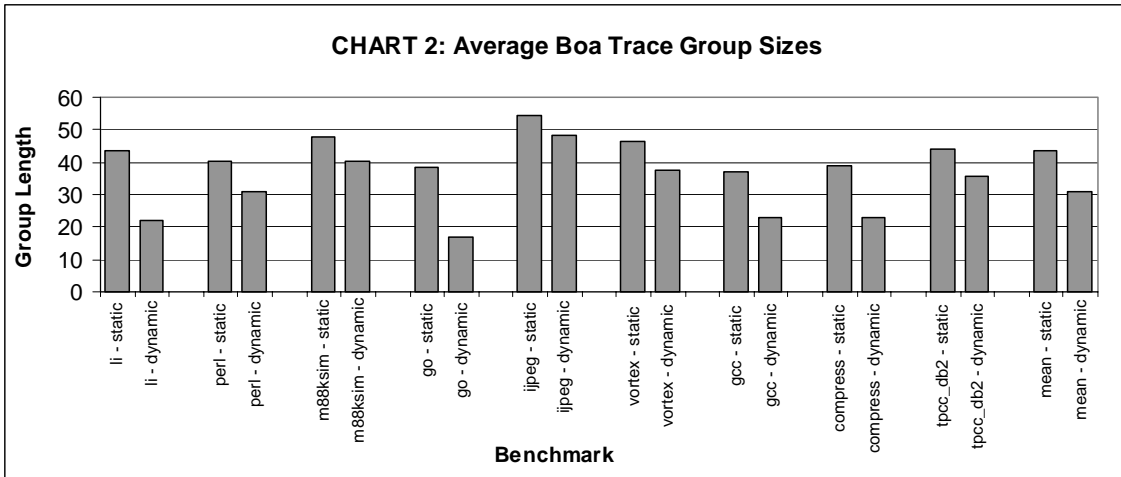
two most relevant properties were: (1) the static length and (2) “quality” of the groups where quality refers to a measure of how likely control flow is to reach the end of the group.



Clearly, there is a balance to be obtained as statically long groups offer opportunity to perform more scheduling, however longer groups also increase the likelihood of exiting the group early and sending the program back to the interpreter. In this case, a branch misprediction occurs and expensive branch repair has to be performed in the pipeline (7 cycles for BOA - which gives some insight into the branch

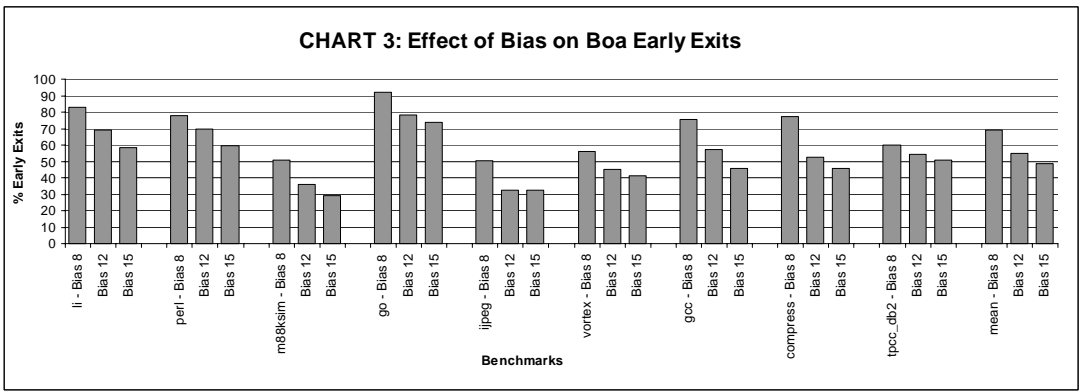
adders as well). An early exit from a group further detracts from CPI in that scheduling opportunity has been lost, since instructions from the wrong stream were executed speculatively. Dynamic group sizes observed from the experiment above are shown in Chart 2.

We next conducted experiments to increase the static predictability of branches and thereby enhance the “quality” of groups. This was achieved by profiling branches, and using selective group extension: only if a branch was likely to be taken n times



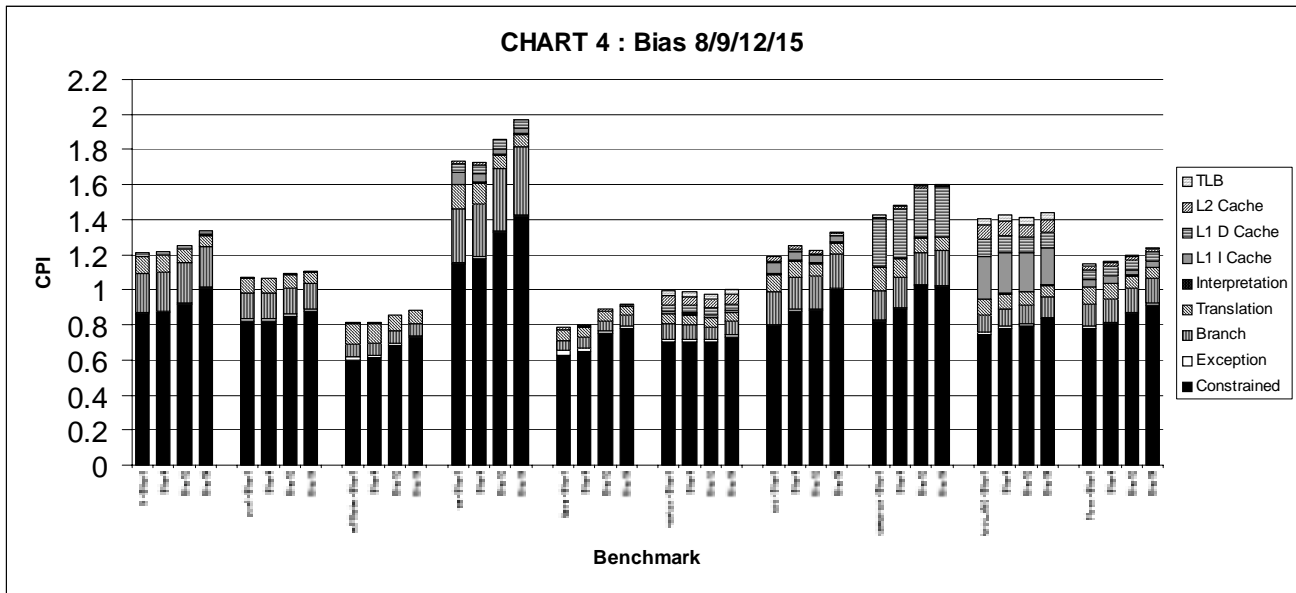
out of the 15 profiled iterations would the group be extended by appending instructions at the target of the branch. Otherwise, the group would be terminated and a new group was started. We expected this approach to greatly reduce the number of

early exits (i.e., branches which are taken out of the group before the group end is reached). Chart 3 shows the reduction in early exits achieved for bias values of 8, 12, and 15. Note how higher bias values for extending the group improve the quality of the groups.

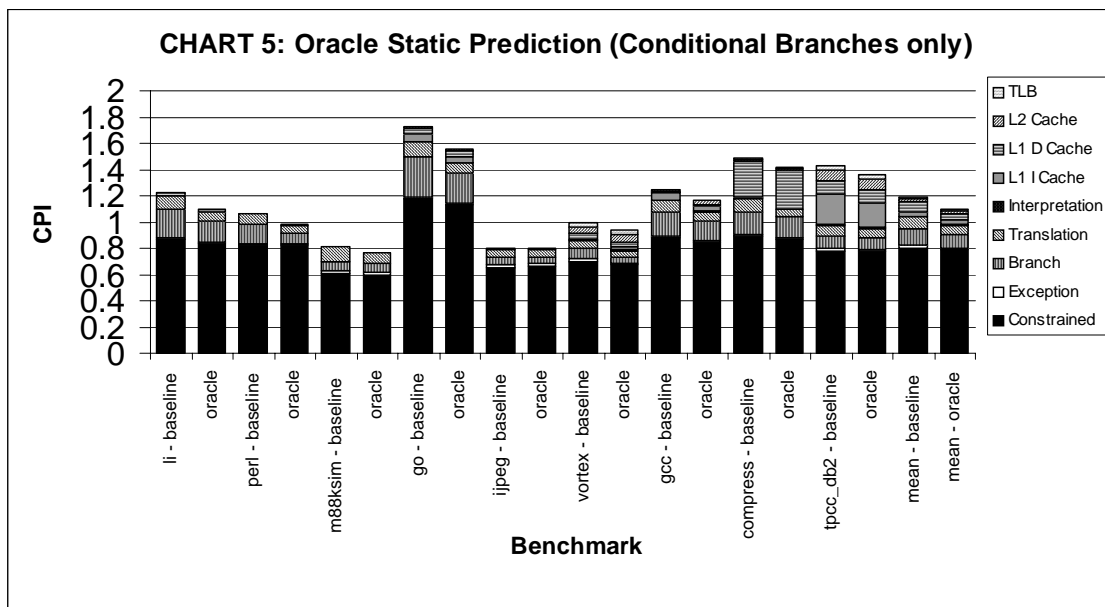


By reducing the number of early exits, we hoped to improve the CPI of the system under evaluation. However, Chart 4 demonstrates that the reduced number of early exits and misspeculation cannot compensate for reduced static group length, and as a result

the overall CPI suffered. The lesson to draw from this is that static group length is overall a more important contributor to the average dynamic group length and performance than the number of early exits. It is noteworthy that longer static group 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 group 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 groups on a second iteration. Chart 5 compares the results obtained for this oracle prediction with our initial baseline results, based on profiling performed during the first 15, interpreted executions of each branch.



5.0 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 SIA roadmap hold true. The CPI of BOA is around 1, but varies somewhat across benchmarks in a manner roughly proportional to the dynamic group size, thus substantiating the expected result that a larger window of instructions 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 I-Cache overheads, maximizing dynamic group lengths, and effectively managing the amortization of translation overhead.

The placement of optimized traces in memory allows for efficient prefetching and helps keep down I-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 I-Cache penalty to the extent that it may be relaxed to obtain improvements in other areas.

Dynamic group 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 I-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.

6.0 Future Work

The success of Binary Translation rests fundamentally on the group quality. A characteristic of a good group is a low occurrence of early exits. To this end, there is significant experimentation left that can be done with different modifications to the binary translation system software. Currently the binary translation approach is constrained to building groups that follow a single path of execution, based on statistics gathered about the branches during the interpretation phase. Identifying multiple paths (emanating from “fickle” branches) that would be appropriate to include in the group is not a difficult task; this functionality is already available in our system. The slightly more difficult (and therefore still unimplemented in our system) task is to schedule these multiple paths into the same group of native machine code. This can be done in either of two ways. The first and most intuitive is to generate a group which can branch locally within the group so as to allow non-straight line control flow through a scheduled group. This has the downfall of adding branch latencies to the compiled group. This can be avoided by using the second approach, which is to add support for predication to the architecture, and allow multiple paths of control flow from the original code to be scheduled into the same VLIWs, predicated such that only the path that is being followed will execute.

Additional sources of scheduling improvement might come with additional profiling data of the executed code. Repeated load values might suggest targeted use of value prediction. Repeated cache access patterns might suggest the scheduling of data prefetch or other cache management instructions. The coupling of static profiling information with dynamic hardware structures (such as branch predictors) might allow better optimization of the executing code.

Methods for better managing the overhead associated with translation are required for workloads which do not exhibit high reuse rates. Potential techniques exploring methods to reduce the cost of translation include hardware assist, simultaneous multithreaded translation, block translations, and garbage collection.

Lastly, better understanding of these techniques on a broader spectrum of workloads is required. The goal of the BOA project was to understand binary translation in a SMP commercial server environment. The use of these techniques in client workloads or scientific/engineering was not studied. These workloads likely have different characteristics and might suggest different optimization points.

7.0 Acknowledgments

The authors would like to thank the following people for their critical contributions to the work presented: Albert Chang, Kemal Ebcioğlu, Marty Hopkins (IBM T. J. Watson Research Center, Yorktown Heights, NY, USA), and Patrick Bohrer, (IBM Server Division, Austin, TX, USA).

8.0 References

- [1] J. Hennessey and D. Patterson, "Computer Architecture: A Quantitative Approach", Second Edition, Morgan Kaufmann Publishers, 1996.
- [2] R. Nair and M. Hopkins, "Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups", Proceedings of the 24th Annual International Symposium on Computer Architecture, pages 13-25, Denver, CO, June 1997. ACM.
- [3] R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson, "Binary Translation", Digital Technical Journal, Vol. 4, No. 4, Digital Equipment Corporation, Maynard, MA, 1992.
- [4] K. Ebcioglu and E. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility", Proceedings of the 24th Annual International Symposium on Computer Architecture, pages 26-37, Denver, CO, June 1997. ACM.
- [5] K. Ebcioglu, E. Altman, S. Sathaye, and M. Gschwind, "Execution-based Scheduling for VLIW Architectures", Proceedings of Europar '99, Lecture Notes in Computer Science 1685, pages 1269-1280, Toulouse, France, August/September 1999. Springer Verlag.
- [6] M. Moudgill and J. Moreno, "Method and apparatus for reordering of memory operations in a processor", US Patent No. 5,758,051, May 1998.
- [7] 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 5,832,205, November 1998.
- [8] M. Gschwind, "Pipeline Control Mechanism for High Frequency Pipelined Designs", US Patent Application, January 1999.
- [9] SIA, "National Technology Roadmap for Semiconductors: Technology Needs, 1997 Edition", online document located at URL: <http://notes.sematech.org/ntrs/PublNTRS.nsf>.
- [10] G. Northrop, R. Averill, K. Barkley, S. Carey, Y. Chan, Y.H. Chan, M. Check, D. Hoffmann, W. Huott, B. Krumm, C. Krygowksi, J. Liptay, M. Mayo, T. McNamara, T. McPherson, E. Schwarz, L. Siegel, C. Webb, D. Webber, and P. Williams, "600MHz G5 S/390 Microprocessor", Proceedings of the 1999 International Solid-State Circuit Conference, pages 88-89, San Francisco, CA, February 1999.
- [11] S. Fischer, R. Senthinathan, H. Rangchi, and H. Yazdanmehr, "A 600MHz IA-32 Microprocessor with Enhanced Data Streaming for Graphics and Video", Proceedings of the 1999 International Solid-State Circuit Conference, pages 98-99, San Francisco, CA, February 1999.
- [12] 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", Proceedings of the 1998 International Solid-State Circuit Conference, pages 230-231, San Francisco, CA, February 1998.