COVER FEATURE

Dynamic and Transparent Binary Translation

BOA's dynamic optimization offers significant advantages over purely static compilation approaches like those Intel and Hewlett-Packard currently propose for the IA-64 architecture.

Michael Gschwind
Erik R. Altman
Sumedh Sathaye
IBM T.J. Watson Research Center

Paul Ledak
David Appenzeller
IBM Burlington

High-frequency design and instruction-level parallelism (ILP) are two keys to high-performance microprocessor implementations. The Binary-translation Optimized Architecture (BOA), an implementation of the IBM PowerPC family, combines binary translation with dynamic optimization. We use these techniques to simplify the hardware by bridging a semantic gap between the PowerPC RISC (reduced instruction set computer) instruction set and even simpler hardware primitives.

Processors like the Pentium Pro and Power4 have tried to achieve high frequency and ILP by implementing a cracking scheme in hardware: An instruction decoder in the pipeline generates multiple micro-operations that can then be scheduled out of order. BOA relies on an alternative software approach to decompose complex operations and to generate schedules.

Software allows more elaborate scheduling and optimization than hardware. Thus, you can use software to eliminate complex control hardware so that a processor implementation based on binary translation can achieve maximum performance by enabling high-frequency processors while still exploiting available parallelism in the code.

Our work on BOA was inspired by earlier binary translation work such as FX!32[1]—and especially Daisy,[2,3] which uses binary translation for scheduling PowerPC code to a very long instruction word (VLIW) processor. However, the machine described in this article is narrower, with priority given not to minimizing cycles per instruction (CPI) but to maximizing processor frequency. By limiting the size of individual processor cores, multiples of them can be placed on a single die for SMP-on-a-chip configurations.

BOA's dynamic optimization offers significant advantages over purely static compilation approaches like those Intel and Hewlett-Packard currently propose for the IA-64 architecture. Reliance on purely static profiling makes it impossible to adapt to changes in program usage. In addition, static profiling requires that independent software vendors (ISVs) perform extensive profiling and generate different executables optimized for a particular processor generation.

Given the reluctance of ISVs to ship code with traditional compiler optimizations enabled, it may be difficult to induce them to take the still more radical step of profiling. None of these problems arise with BOA's dynamic approach, which performs invisibly.

BOA'S TRANSLATION STRATEGY

In BOA, binary translation is transparent: As Figure 1 shows, when BOA boots, control transfers to the virtual machine manager (VMM), which implements the binary translation system. The VMM is part of BOA firmware, although invisible to the software running on it.

After the BOA VMM initialization, the BOA VMM interpreter initiates the PowerPC boot sequence. Specifically, a PowerPC system built on top of BOA executes the same steps as it would on a native PowerPC implementation. Actual instruction execution always remains under full control of the BOA VMM, although the locus of control need not necessarily be within the VMM proper, which includes the interpreter, translator, exception manager, and memory manager. If the locus of control is not in the VMM, it is within VMM-generated traces that have been translated carefully so as to transfer control only to each other or back to the VMM.

When the BOA VMM first sees a fragment of PowerPC code, it interprets it to implement PowerPC semantics. During this interpretation, BOA collects code

0018-9162/00/$10.00 © 2000 IEEE

profile data that will later be used for code generation. Interpretation also serves as a filter for rarely executed code, which cannot amortize its translation cost.

Trace formation

After interpreting the entry point of a PowerPC operation sequence, BOA gathers PowerPC operations from a single path through the PowerPC code and places them in a group. Translated BOA code for a trace can be laid out contiguously in memory. In PowerPC code, code layout is a function of program structure and compile-time decisions. BOA, in contrast, uses dynamic runtime information to generate traces corresponding to the most likely path of program execution. This contiguous layout improves instruction-cache packing and the ability to fetch instructions quickly.[4]

Figure 2 shows the formation of a trace along the path ABDH. BOA places this path into contiguous memory locations to improve instruction cache packing and instruction fetch. While forming traces is advantageous, it can also result in a degradation of performance if BOA were to predict the path incorrectly. Figure 2 shows how the effective window size for exploiting parallelism can be reduced by incorrectly predicting the most likely path.

PowerPC code can be broken into several traces, some of which may overlap. Traces have two types of exits: side exits that represent a mispredicted branch and end-of-trace exits that represent translation stopping points. Choosing stopping points wisely can limit the number of translated traces and help improve cache performance.

Figure 3 shows how BOA operates. Once BOA sees the trace beyond a threshold number of times, it assembles the code starting at the entry point into a PowerPC trace and translates it into a BOA instruction trace for execution on the underlying hardware. At each branch point, BOA follows the most likely path. As the trace moves to each conditional branch during the translation, the probability of reaching each point from the start of the trace decreases. When the probability drops below a threshold value, BOA terminates the trace.

Code optimization and scheduling

Optimization, particularly useful for dealing with legacy code, can also improve the performance of already-optimized code. Unlike a static compiler, the dynamic optimizer need not consider the entire control flow in making optimization decisions. Instead, short traces can be carved out of the control-flow graph, eliminating all control-flow joins. Eliminating control-flow joins opens up numerous optimization possibilities as shown in Figure 4.

Performing BOA optimizations in a traditional static compiler presents a special challenge because of

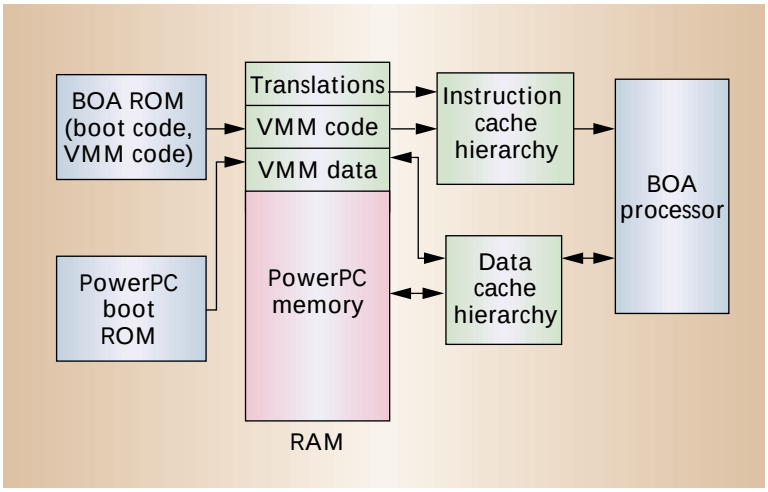

Figure 1. The components of a BOA system.

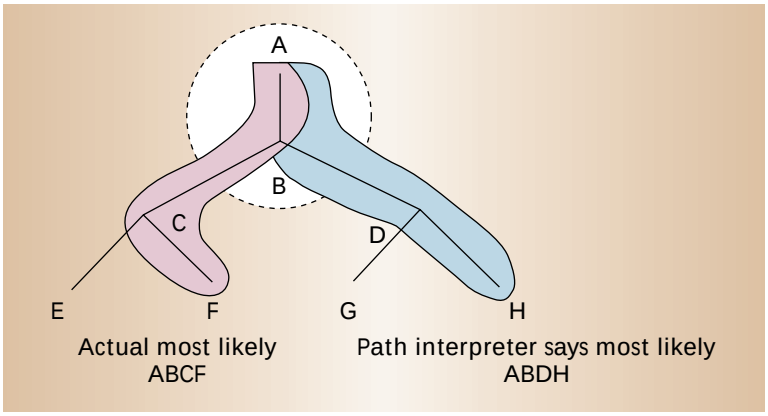

Figure 2. The formation of a trace along the path ABDH. The effective window of operations can be small if the interpreter does not correctly predict the most likely path through a group of PowerPC operations. The dotted circle indicates how the effective window of operations is truncated to path AB by path mispredication, instead of exploiting a long path such as ABDH or ABCF.

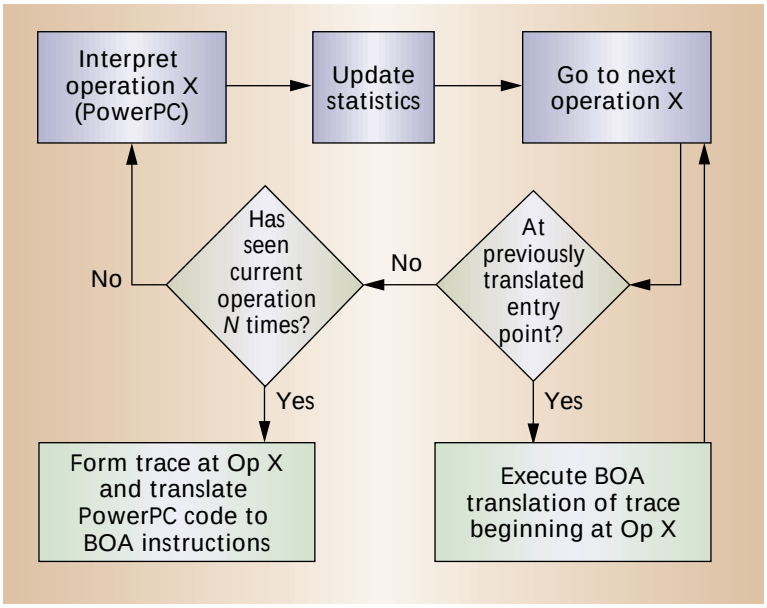

Figure 3. How BOA operates.

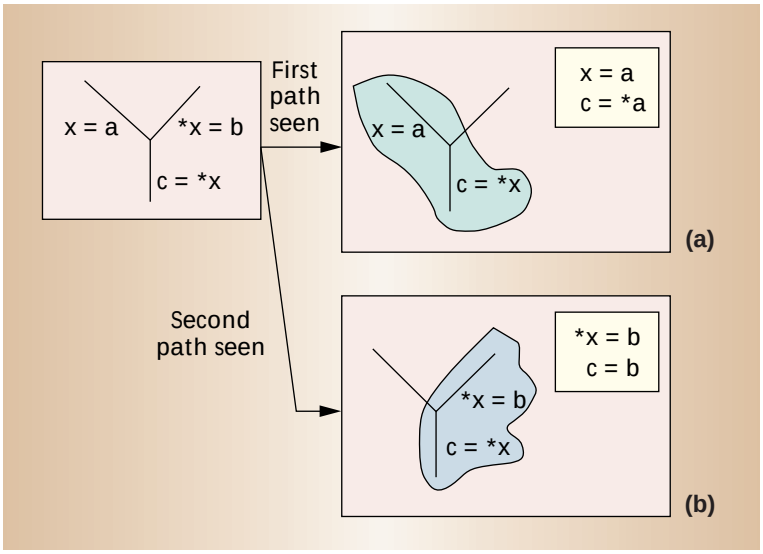

Figure 4. Dynamic optimization can open new optimization opportunities. In (a), copy propagation yields two independent operations that can be scheduled in the same VLIW. In (b), load-store telescoping eliminates dependencies through memory. Copy propagation of c = b can open even more opportunities.

the difficulty in choosing from among the exponentially many paths through the available code. Profiling an application's execution and using the results in the next compilation mitigates this problem to some degree. In addition to their own benefit, these optimizations reduce dependencies and thereby reduce schedule height, which allows more operations to be scheduled in parallel to exploit BOA's parallel execution units more effectively.

BOA's design schedules operations to maximize ILP opportunities and take advantage of the speculation capabilities supported by the underlying architecture. The current scheduling approach is greedy: Each operation executes at the earliest possible time when

- all input operands are available,
- a function unit on which to execute the operation is available, and
- there is a free register in which to put the result.

In determining this earliest possible time, BOA makes use of several optimization techniques.

BOA simultaneously performs scheduling, optimization, and register allocation. The out-of-order scheduling approach used by the BOA translator makes it difficult to maintain the precise exceptions required by the PowerPC architecture. To solve this problem, BOA uses a hybrid hardware-software approach based on maintaining precise checkpoints at trace transition boundaries and the ability to roll back to an earlier checkpoint.

To initiate a checkpoint procedure, BOA copies all registers to a set of backup registers. Within a trace, BOA schedules instructions out of order and renames the registers to support speculative execution. BOA executes store operations in original program order but labels them pending so they can be revoked if an exception occurs.

When a trace is exited during the course of normal execution, the Power PC registers are committed into the checkpoint registers, pending stores are marked definite, and execution continues with the next trace. When an exception occurs, the system discards working registers and all pending stores and recovers the processor state from the checkpoint registers. The BOA VMM then enters interpretative mode and executes in-order instructions until the cause and location of the exception are discovered.

System issues

Out-of-order loads must be treated specially during scheduling and execution to conform with PowerPC memory-ordering semantics. BOA assigns each load and store in a trace a number indicating its sequence in the trace. If the hardware detects that a load with a later sequence number has executed earlier than a store with an earlier sequence number, BOA signals an exception, restores the last checkpoint, and enters interpretative mode. The result is that BOA re-executes the problem load to receive the proper values.

Attempts to access noncacheable memory, such as an I/O location, are also problematic. Such operations cannot be allowed to complete because they can have side effects such as changing the hard disk contents. To avoid this problem, BOA uses hardware to suppress and detect any noncacheable loads that require special handling in software.

When branching between traces of translated instructions, BOA places load real address (LRA) operations at the start of each trace that is by the scheduler. When executed, the LRA operation checks that the translation lookaside buffer and page tables still map the virtual address for the start of the trace in the same way that they did when the trace was originally translated. If a change in the page tables might affect the validity of a translation, BOA initiates a trap, destroys the trace, and begins interpreting at the proper address.

ARCHITECTURE AND IMPLEMENTATION

We created BOA to be an unexposed architecture with an instruction set specifically designed to support binary translation. We did not intend the architecture to be a platform for handwritten user code, but instead to provide several primitives and resources to make it a good target for binary translation.

Instruction set architecture

In a sense, BOA instructions are the equivalent of microcode operations, which represent the real machine language used to implement the public ISA. The BOA instruction set is not accessible from PowerPC user or supervisor mode, and may change from implementation to implementation.

BOA's primitives resemble the PowerPC's in both semantics and scope. However, not all PowerPC operations have an equivalent BOA primitive. Many PowerPC operations are intended to be layered as a sequence of simpler BOA primitives to enable high-frequency imple-

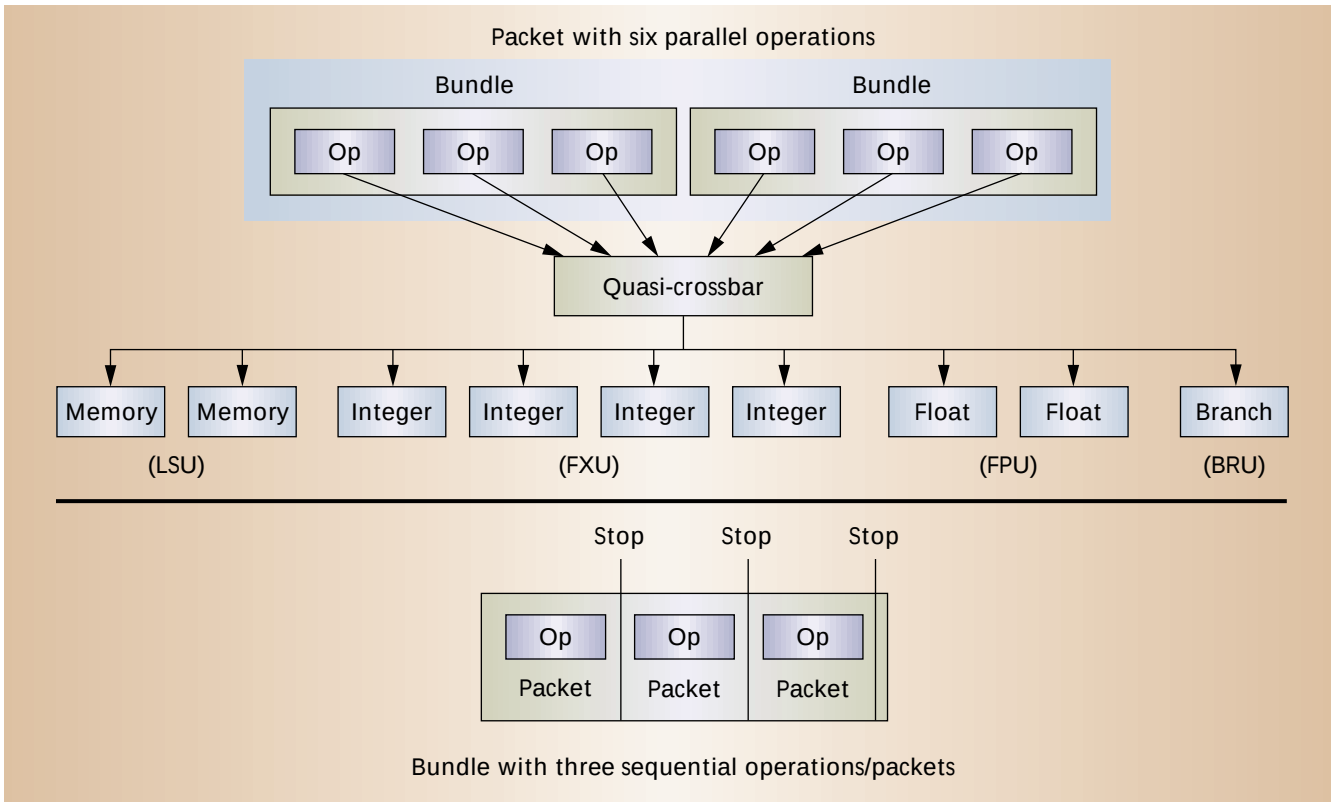

Figure 5. BOA instruction formats.

mentations. BOA instruction semantics and data formats also resemble the PowerPC's, eliminating the need for expensive data format conversions. To support code scheduling and speculation using register renaming, BOA provides twice as many machine registers for each class of PowerPC registers.

BOA uses a statically scheduled, compressed instruction format similar to the IA-64 architecture. A parallel instruction can simultaneously issue up to six operations per cycle, as Figure 5 shows. To ensure efficient memory layout, BOA packs operations into 128-bit bundles that contain three operations each. Each operation contains 39 bits and one stop bit. This brings the total to 120 bits among the three operations, leaving 8 bits for future system enhancements.

As Figure 5 shows, 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, four integer, two floating-point, 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.

All operations effectively have an additional latency cycle because BOA provides no bypassing. One cycle must elapse before a result can be used by a successor operation. Eliminating result bypass permits reduced cycle time by allowing a full pipeline stage to broadcast results to each execution unit.

Implementation

Figure 6 shows the contents of the BOA processor box depicted 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 for loads, allowing instruction execution to continue in the presence of cache misses during memory accesses. This allows memory accesses and independent instructions to overlap efficiently until a dependent operation forces a stall or BOA retrieves ROM memory. The processor also provides dynamic support for out-of-order loads and stores, decoupled fetch-and-execute pipelines, and a commit-recirculate scheme for pipeline control.

Although dynamic scheduling and branch prediction have proven their value on superscalar processor implementations, we feel that they can limit frequency if not used sparingly. We limited BOA hardware to four dynamic processes:

- register scoreboarding lets in-order instruction issues continue in the presence of nondependent memory stalls,
- load and store queues check for address conflicts between loads and stores reordered during translation,
- instruction buffers decouple the fetch pipeline from the execute pipeline to hide some instruction fetch stalls, and
- a pipeline control method enables the pipeline to advance on each processor cycle, which simplifies pipeline control.

Assuming proper code scheduling, only memory stalls will hold up the execution pipeline. Instead of checking for the existence of a stall before proceed-

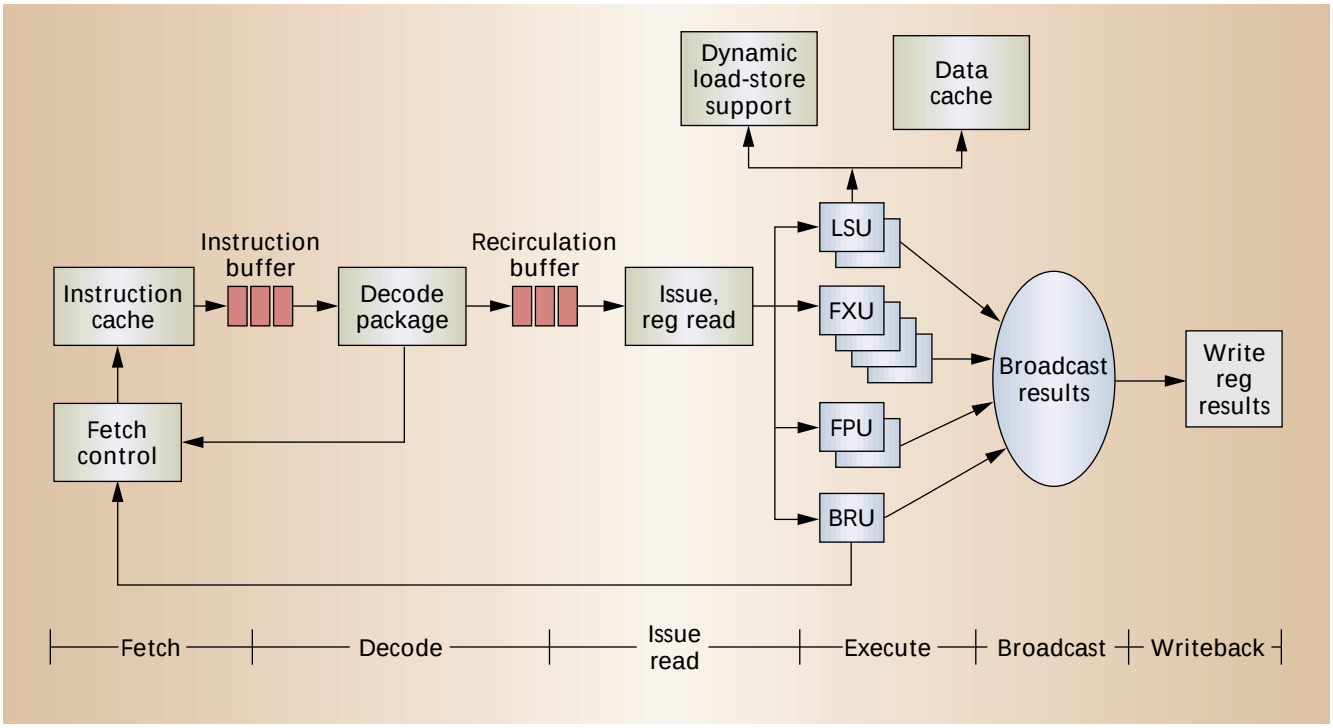

Figure 6. The BOA processor executes instructions concurrently in two load-store units (LSUs), four fixed-point units (FXUs), two floating-point units (FPUs), and a branch resolution unit (BRU).

ing, the pipeline advances every cycle. When a packet is issued, it is also copied into the recirculation buffer shown in Figure 6. The recirculation buffer holds a copy of every packet currently executing.

Stalls need not be noticed until late in the execution process. For example, if the data cache misses and does not return a value, the packet with the corresponding load operation is marked null, as are any packets issued after this load packet. The recirculation buffer then reissues the load packet and all subsequent packets. If the reissued packets also incur a cache miss, the process repeats. These repeats occur until all stalls have been eliminated.

Newer technologies show wire delay replacing logic delays as the limiting factor in clock speed. For BOA, this means paying special attention to processor control logic. The BOA microarchitecture 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.

Figure 7 shows BOA performance as cycles per PowerPC instruction on the SPECint95 and TPC-C system benchmarks, and details the contribution of all system aspects.

You can find detailed information about different system configurations and their performance impact elsewhere.[5,6] The performance numbers demonstrate that binary translation is an interesting implementation choice for future high-performance systems. As described in the sidebar "High-Frequency Processing," we expect BOA soon to achieve a 2-GHz clock frequency. ✸

High-Frequency Processing

Stephen Kosonocky, IBM T.J. Watson Research Center

The 1999 International Technology Roadmap for Semiconductors (http://notes.sematech.org/ntrs/PublNTRS.nsf) calls for achieving a high-performance microprocessor with a 2-GHz on-chip clock by the middle of this decade, while CMOS technology will provide only a 190 percent increase in Field Effect Transistor performance. The numbers of transistors will increase per chip over typical 1999 0.18-µm CMOS capabilities. To meet and exceed these cycle time goals, the BOA microarchitecture allows a worst-case cycle time of 700 picoseconds in a current 0.18-µm CMOS bulk technology under nominal process and temperature conditions. This cycle time target represents more than a 50 percent improvement over reported 0.25-µm designs scaled to 0.18 µm and should allow operation in excess of 2 GHz in the next few years.

Acknowledgments

Modern processor design is the work of many individuals. Albert Chang, Kemal Ebcioğlu, Martin Hopkins, Craig Agricola, Patrick Bohrer, Arthur Bright, Zachary Filan, Jason Fritts, and Jay LeBlanc have all been instrumental in the BOA project.

References

1. R. Sites et al., "Binary Translation," *Digital Technical J.*, Dec. 1992, pp. 137-152.
2. K. Ebcioğlu and E. Altman, "DAISY: Dynamic Compilation for 100 Percent Architectural Compatibility," *Proc. ISCA24*, ACM Press, New York, 1997, pp. 26-37.
3. K. Ebcioğlu et al., "Execution-Based Scheduling for VLIW Architectures," *Proc. Europar99*, Lecture Notes

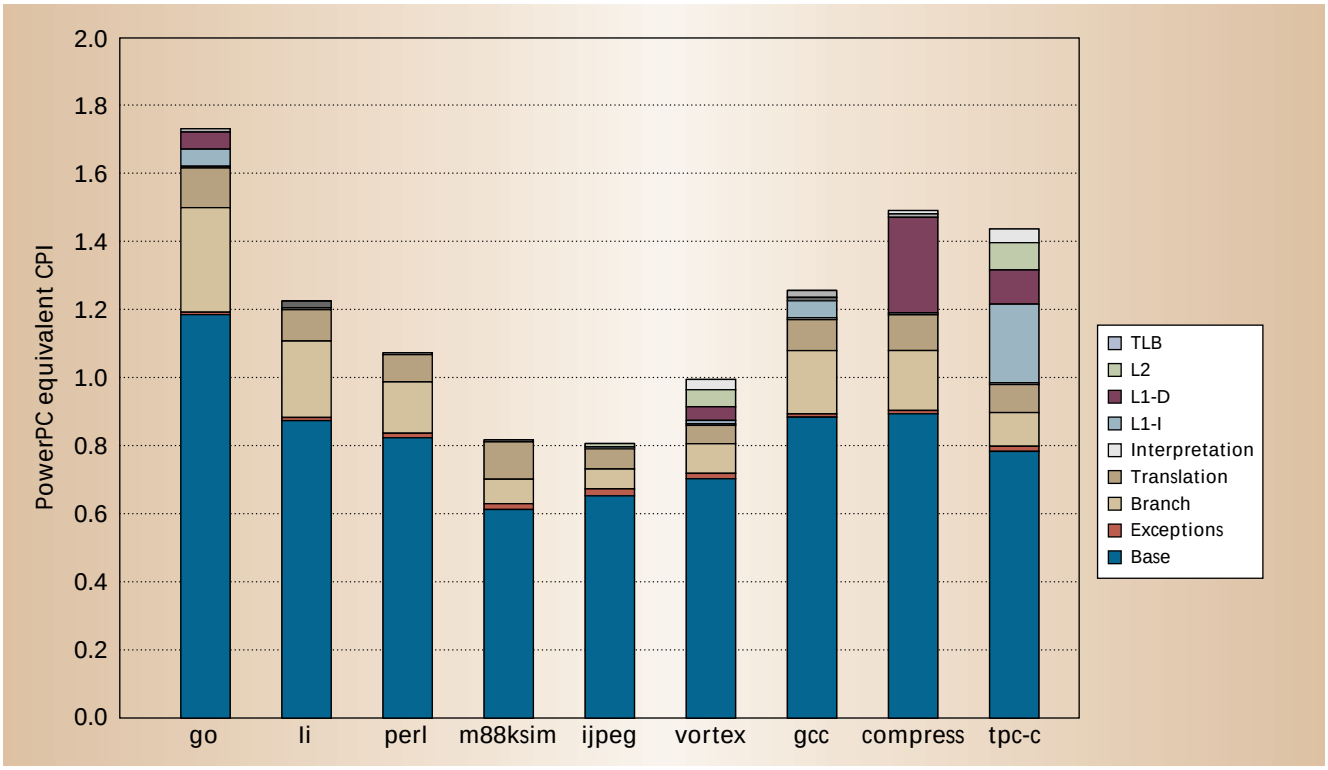

Figure 7. This chart shows the number of cycles necessary to execute an average PowerPC instruction for each of the SPECint95 benchmarks and for the TPC-C database benchmark. The base component represents the cycle time of each PowerPC instruction without any overhead, such as translation cost or cache misses. The additional stacked bars represent this overhead. In the key, TLB represents the cost of TLB misses. L2, L1-D, and L1-I represent the cost for cache misses in the L2, L1 data, and L1 instruction caches, respectively. Finally, "Interpretation" reflects the time necessary for initial interpretation and profiling, "Translation" represents the cost of translating from PowerPC to BOA code, "Branch" represents branch penalties, and "Exceptions" represents the cost of implementing precise exceptions using a rollback scheme.

in Computer Science 1685, Springer Verlag, Berlin, 1999, pp. 1269-1280.

4. K. Pettis and R.C. Hanson, "Profile Guided Code Positioning," *Proc. 1990 SIGPLAN PLDI*, ACM Press, New York, 1990, pp. 16-27.
5. S. Sathaye et al., "BOA: Targeting MultiGigahertz with Binary Translation," *IEEE TCCA Newsletter*, Fall 1999, pp. 2-11.
6. E. Altman et al., *BOA: The Architecture of a Binary Translation Processor*, IBM Research Report RC21665, IBM, Yorktown Heights, N.Y., 2000.

Michael Gschwind is in the high-performance VLSI architecture group at IBM T.J. Watson Research Center. He received a PhD in computer science from Technische Universität Wien, Austria. His research interests include compilers, binary translation, computer architecture, hardware-software codesign, and application-specific processors. Contact him at mikeg@watson.ibm.com.

Erik R. Altman is in the high-performance VLSI architecture group at IBM T.J. Watson Research Center and was one of the originators of the DAISY project. He received a PhD in computer science from McGill University. Contact him at erik@watson.ibm.com.

Sumedh Sathaye is a research staff member at IBM T.J. Watson Research Center. His research interests include computer architecture and microarchitecture, instruction-level parallelism, and binary translation. He received a PhD in computer engineering from North Carolina State University. Contact him at sathaye@watson.ibm.com.

Paul Ledak is vice president of IBM Server Architecture Development. He has spent most of his career managing and contributing to IBM microprocessor development for PowerPC and x86 architecture processors.

David Appenzeller is a manager for PowerPC microprocessor development at IBM Burlington. He received a BS in electrical engineering from Drexel University.