

Inherently Lower Complexity Architectures using Dynamic Optimization

Michael Gschwind, Erik Altman

{*mikeg,erik*}@watson.ibm.com

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

Based on the conviction that modern superscalar out-of-order designs squander useful resources for little incremental gain, the BOA team embarked on a design effort to develop an architecture where computational elements dominated the design. At the same time, we wanted to preserve the ability to adapt to changing workload behavior dynamically, but without the overhead inherent in traditional out-of-order designs. We turned to maturing dynamic compilation technology to achieve dynamic adaptability, while keeping core complexity low.

1 Introduction

Modern architectures are characterized by the ability to issue multiple instructions out-of-order (OOO) to achieve high instruction-level parallelism by exploiting dynamic program information about program behavior.

While these techniques improve instruction-level parallelism, their impact on designs skews the distribution of logic towards predictors, complex issue queues with out-of-order wake-up and issue logic,

register mapping tables, completion tables, and so forth. Thus, a significant fraction of logic which may otherwise be spent executing a program is dedicated to holding data which does not directly contribute to program progress.

The impact of these typical OOO structures goes beyond their appetite for transistors which reduce what can be applied towards program execution, and which may easily be overlooked. While OOO structures do not contribute to advancing program execution, they complicate design and verification significantly and are often the source of difficult to find bugs and concurrent schedule slips. In a marketplace characterized by Moore's Law, a slip in design schedule will cause a design fall behind the expected performance curve and hence penalize a design's performance relative to competitors entering the market at the same time. This can easily wipe out all the performance gains attributable to out-of-order architectures *relative* to those of other designs.

Schedule slip	Relative performance
1 month	4%
3 month	12%
6 month	26%
9 month	41%
12 month	59%
18 month	100%

Beyond the distribution of logic capacity and schedule impact, many of the traditional out-of-order components are also among the most power-inefficient components in a microprocessor design. This is largely due to the fact that they are operating in every cycle, and by their nature cover a large set of data being queried to operate on a single element. Power-aware microarchitecture research will eventually address these issues ([1], [2], [3]), but not without introducing additional design complexity if the overall architectural approach remains unchanged.

We concluded that while exploiting instruction-level parallelism was desirable, extracting it should not come at the expense of huge increases in design complexity. The solution we identified was to leverage advanced compilation technology to accomplish ILP extraction in software. [4, 5])

In the process, design simplicity allows shorter latency from pipeline entry to exit as a function of overall FO4 (as opposed to super-deep pipelines which increase frequency but not performance).

The main goal in the use of the dynamic optimizer is to identify frequently executed program regions and optimize them using information about the specific workload behavior. Although in this specific instance, we implemented the BOA target architecture as a VLIW platform, we could also have chosen a simple in-order superscalar PowerPC implementation.

The BOA architecture¹ traces its roots to the IBM

¹Originally, the architecture name “Boa Constrictor” described the very stringent design constraints of conceptual clarity and simplicity we had defined. The acronym “Binary transla-

VLIW project [7, 8], and more particularly the Dynamically Architected Instruction Set from Yorktown (DAISY) [9, 10, 11, 5, 4]. While the VLIW project had previously demonstrated the performance potential of VLIW architectures for general-purpose sequential-natured software, our DAISY experience had given us the ability to achieve compatibility with the PowerPC architecture and to optimize code dynamically.

2 ILP Extraction

Modern microprocessors extract instruction-level parallelism in a series of steps. Instruction fetch determines the likely path of execution under the guidance of sophisticated predictors. The architectural instruction set is decomposed into primitives which can be pipelined efficiently. Depending on the complexity of the instruction, this can be performed either by hardwired logic (“instruction cracking”), or by transferring control to a microcode ROM. The primitives are then grouped into instruction groups so they can be tracked throughout the design, stored in the issue queue where they are selected, and issued by out-of-order control logic. To resolve data hazards and ensure correct sequential in-order program semantics, register renaming is performed and results are retired in-order to the architected register file. ([12]).

The resulting design is characterized by deep pipelines where several pipeline stages are dedicated to performing the above steps. This in turn triggers several self-reinforcing effects, where a more complex model forces ever more complex solutions. For example, because the deep pipeline exacerbates branch mispredictions by increasing the branch

tion and Optimization Architecture”, or “Binary translation Optimized Architecture”, was derived to describe the techniques which allowed us to achieve this goal.

penalty, more sophisticated and complex branch prediction logic becomes necessary.

In BOA, we opted for a much simplified pipeline model combined with the latest advances in dynamic compilation [4, 5, 13]. In this approach, software would be responsible for analyzing application behavior, decomposing PowerPC instructions into pipeline primitives, scheduling them into instruction groups, renaming registers, and assembling traces of instructions likely to be executed along a path based on the execution history of the program. The prescheduled traces – together with the runtime environment – also contain all logic necessary to preserve in-order semantics. The trace fragments created by software are tightly packed as they follow the most likely path of execution, and hence exploit the instruction caches more efficiently. This is similar to the advantages achieved by the use of trace caches, but achieved with much simpler hardware for both assembling the traces and selecting the particular trace to be executed.

We decided to preserve only two basic functions in support of ILP extraction in hardware. First is a software-managed checkpointing mechanism, which saves the current state when transitioning between groups of instructions scheduled by the software dynamic optimizer. In the event of synchronous exceptions or memory semantics violations due to speculation, this checkpointing mechanism allows BOA to rollback to a good state, and then sequentially proceed past the problem point without speculation.

The other component of hardware support for ILP extraction is a set of load/store order tables. These tables allow software to aggressively schedule memory operations which may be aliased with other memory operations.

Beyond these operations in support of dynamic optimization, BOA is a simple variable length VLIW architecture with 64 registers and the ability to issue up to six instructions to any combination of 9 execu-

tion pipelines [14].

3 Hardware Design Simplicity

The BOA processor is depicted schematically in Figure 1. The processor issues atomic instruction groups – referred to as packets – containing up to six operations to a subset of the 9 execution pipelines. Packets can contain from one up to six operations and are delimited by stop bits. The operations which form the packets are encoded in bundles of 3 operations, using a 128 bit memory word. Using this encoding, a packet can contain operations from up to 3 bundles, and a bundle can contain operations from up to 3 packets, delimited by stop bits. (see Figure 2)

BOA’s pipeline architecture reflects the goal of conceptual simplicity. Instructions are issued by the issue logic as packets when the inputs to all operations within a packet are ready. Input operand readiness is determined using a scoreboard architected in the issue stage.

Results from simple fixed point operations are available after two cycles, reflecting a one cycle computational latency, and an additional cycle for results to be broadcast across the core.

To simplify the scoreboard architecture, scoreboards are updated in the cycle *after* an instruction has been scheduled. This reduces the crucial time to read the scoreboard bits, analyze them, and update them. Since the register files are distributed, scoreboards are also distributed, and an update to remote a scoreboard is assumed to take a cycle. Thus, dependent operations cannot be scheduled by the dynamic optimizer in back-to-back packets, reducing the need to deal with wire delay in a moderately aggressive ILP architecture without frequency penalty [15]. The net effect is that predictable latencies are dealt with by software, but unpredictable latencies

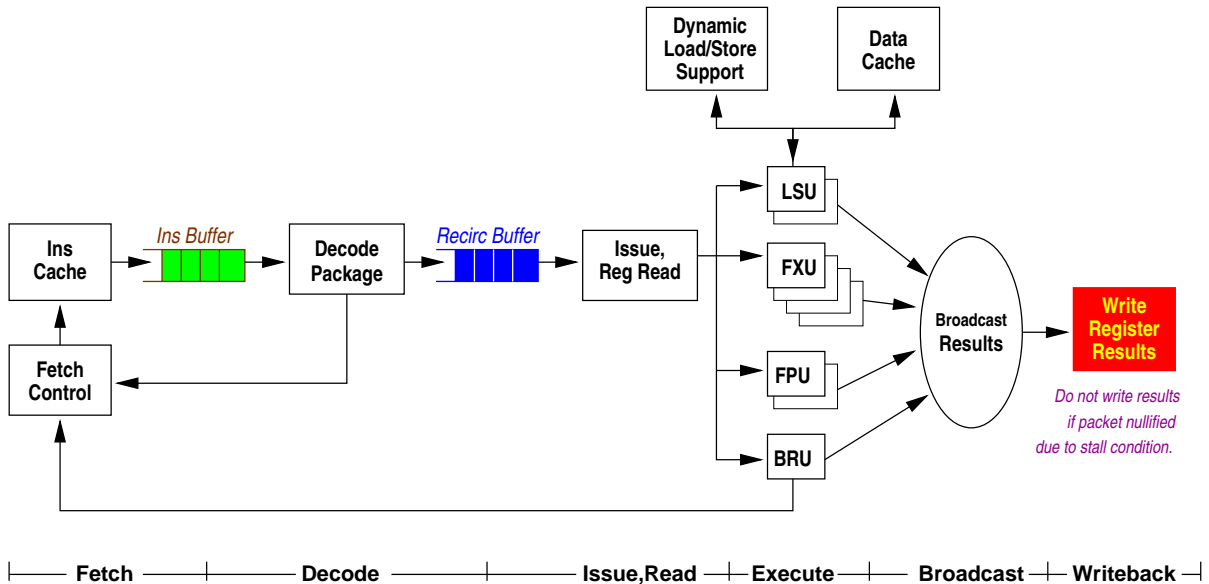


Figure 1: The BOA processor

such as memory access do not cause a *stall-on-miss*² to the L1 data cache, but only when (and if) there is an attempt to use the loaded data before it has been retrieved from the cache. There is likewise no stall after variable-length floating point operations — if and until an attempt is made to use the result and it is not ready.

To avoid the need to perform state rollback to the previous checkpoint on frequent events such as TLB misses, the BOA architecture offers precise behavior on most memory faults. To this end, the pipelines are architected to perform address generation and TLB access before a packet containing memory operations is committed (see Figure 3). Thus, if such a packet incurs a TLB miss (or page fault), the entire packet can be canceled and a TLB reload can be

²Stall-on-miss requires global synchronization and introduces additional circuit complexity, hence it was not desirable regardless of the CPI penalty of this policy.

attempted.

Pipeline control logic was simplified by choosing a simple *recirculation*-based approach to resolving stall conditions and the pipelines are implemented as simple dataflow elements. Correct operation is achieved through the use of a recirculation buffer [16]. The recirculation buffer is managed as a circular buffer containing instructions executing in each pipeline stage.

In our approach, 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 in Fig-

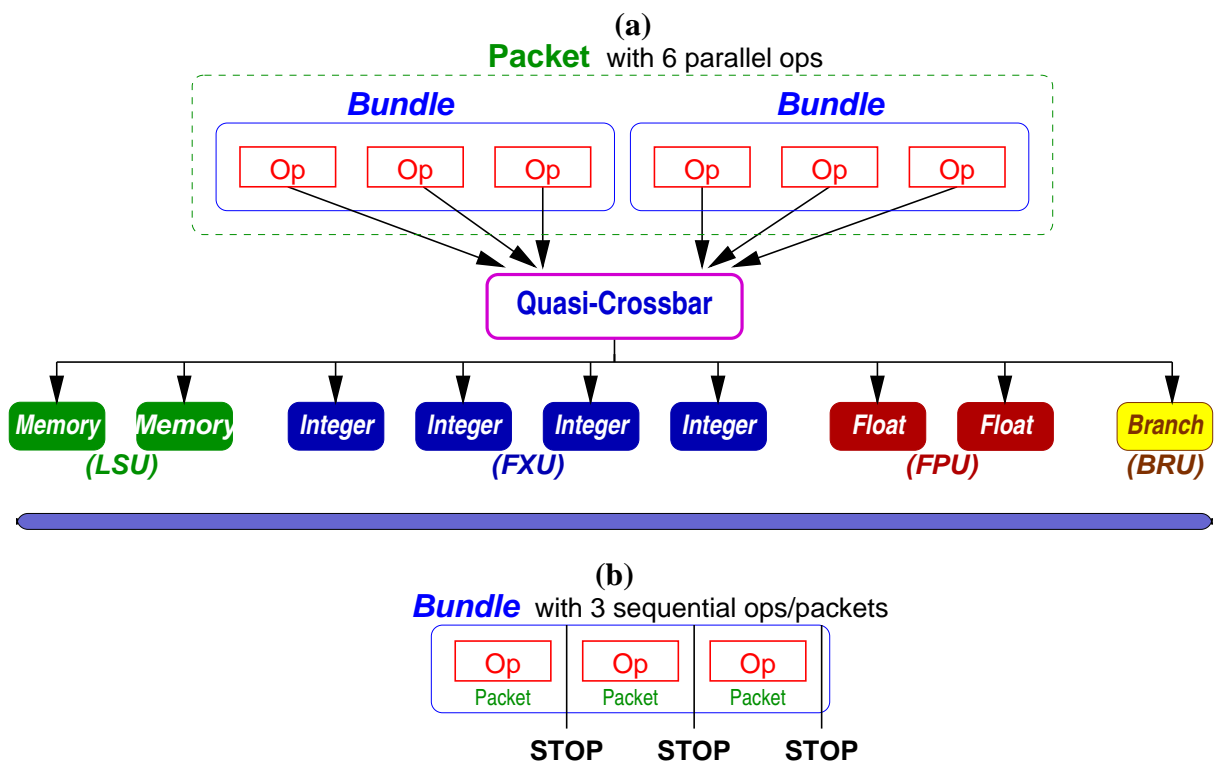


Figure 2: BOA Instructions

Fixed-Point	Load	Store
Fetch 1	Fetch 1	Fetch 1
Fetch 2	Fetch 2	Fetch 2
Decode	Decode	Decode
Issue	Issue	Issue
GPR Read	GPR Read	GPR Read
Execute	AGEN	AGEN
Broadcast	TLB Access	TLB Access
Writeback	Mem Access 1	S-CAM
	Mem Access 2	
	Broadcast	
	Writeback	

Figure 3: The integer pipelines are matched in depth such that any exception conditions are detected before a packet is committed to the processor state. This decision eliminates history buffers and other mechanisms to resolve race conditions between different pipeline depths and ensure in-order semantics.

ure 3. The dependent packet and all subsequent packets are canceled and then reissued from the recirculation buffer. The recirculating packets then repeat the process of issuing and progressing down the execution pipeline. If the stall condition remains during reissue, the packets are continually canceled and reissued from the recirculation buffer until the processor stall completes. To reduce the cost of reissuing instructions while known long-latency operations are in flight (e.g., an L2 data cache miss), recirculation can be suspended for a predetermined, event-dependent number of cycles. We found that by eliminating global pipeline control, we significantly simplified the BOA core architecture.

After address translation is performed in the seventh stage of the Load and Store pipelines in Figure 3, a load operation is enqueued into a decoupling FIFO at the head of the load store units.³ At this point, the operation is architecturally considered as having executed, even if the result is not available (which would cause a recirculation condition for any dependent packet). When a memory slot becomes available, load operations are resolved from the memory hierarchy, and store results are stored in the store CAM which implements a multiprocessor-capable gated store buffer.

By using the ability to recover state using the software-controlled checkpointing mechanism, it would be possible to implement precise exceptions even if the hardware did not support precise exceptions. However, architecting precise exceptions did not introduce significant overhead and we felt that a significant body of code, such as BOA’s dynamic optimizer, would run natively and would benefit from the existence of precise exceptions.

The dynamic optimizer offers enhanced flexibility over the stringent, cycle-time limited instruction

³The instruction **Issue** unit only issues packets if there is space in the decoupling FIFOs.

selection and issue logic implemented by out-of-order superscalar processors in hardware. For example, issue logic can optimize resource usage such as scheduling for a limited number of register file ports. While this optimization could conceivably also be implemented in hardware, it would lead to a significant increase in instruction issue complexity, likely offsetting any complexity reduction achieved in the register file by reducing the number of register file ports.

Since this scheduling step is performed in software, during trace group formation, this is a highly effective trade-off to reduce register file complexity. Preliminary experiments suggest that with almost no loss in performance, the number of register ports can be reduced from the maximum needed if the six worst-case operations (from a register port perspective) are combined in one instruction packet [17]. The dynamic optimizer can ensure that no such packets are generated.

BOA’s software dynamic optimizer provides the ability to map the PowerPC architecture to a simpler, streamlined hardware base in other areas as well:

- BOA uses dynamic optimization and binary translation to emulate PowerPC processors. PowerPC has a condition register which can be accessed in 3 ways: (1) as a 32-bit register, (2) as 8, 4-bit condition register fields, and (3) as 32, 1-bit registers. Tracking renames for these multiple modes of access is quite complicated in hardware. BOA’s software optimization handles this complexity, allowing for significantly simpler hardware.
- Streamlining of instruction idioms and special purpose register usage. For example, in “address calculation” instructions (including the add instruction) in the PowerPC architecture, the register specifier 0 is treated as the literal

Cache	Bytes	Line Size	Assoc	Latency
<i>L1-Ins</i>	256K	256	4	1
<i>L1-Data</i>	64K	128	2	4
<i>L2-Shared</i>	4M	128	8	14

Table 1: BOA Cache Hierarchy

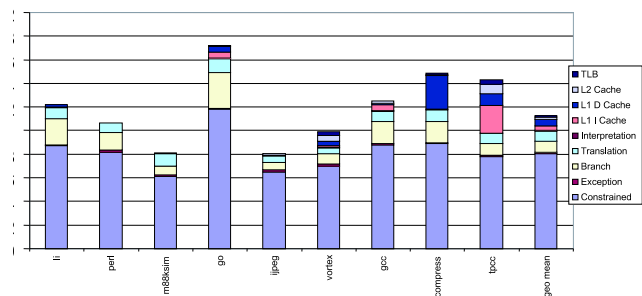


Figure 4: BOA CPI

value 0, instead of the contents of register R0. This difference between R0 and the other integer registers R1-R31, complicates hardware. The base address register in BOA always refers to a register, never literal 0. BOA’s dynamic optimizer and translator efficiently handles PowerPC code with a 0 base register: By software convention, BOA dedicates one of its registers to hold the value 0, and substitutes this register whenever R0 as literal 0 is encountered in PowerPC code. Similarly, the CTR, LR, M/Q special purpose registers of the POWER/PowerPC architecture are mapped to general purpose registers.

4 Experiments

We have conducted experiments to explore several design points for a simple EPIC-style architecture supporting dynamic optimization. BOA’s dynamic

optimizer initially executes programs using interpretation. to gather information about program behavior. It then uses this information to perform scheduling and other optimizations. Figure 4 gives the CPI for our baseline configuration with the cache sizes depicted in Figure 1.

While dynamic optimization offers the desirable features of reducing hardware complexity and being able to adapted to workload behavior, system performance depends on the dynamic optimization cost being amortized over significant execution time. Thus, dynamic optimization cannot respond to cycle-by-cycle behavior changes such as may be exploited by dynamic predictors.

This is of particular concern for the trace formation process, since choosing the right trace paths has a significant impact on system performance. In the BOA dynamic optimizer, trace formation is based on the profiled behavior of branches during the interpretation phase. Thus, branch prediction is “static” in that once a group is formed, the prediction cannot be changed until a group is re-optimized. We compare this with a clairvoyant static predictor (i.e., which can make the statically optimal prediction based on the branch behavior over the entire workload execution period) to determine the maximum performance potential of this approach, and a simple dynamic predictor.

Figure 5 shows that the penalty for using static over dynamic branch prediction is significant with misprediction rates of 11.6% and 4.6%, respectively. However, degradation from clairvoyant static to history-based static prediction is less significant, increasing misprediction rate by only 2.2%.

In an attempt to recuperate some of the performance lost due to static branch prediction, we included simple path prediction into our model. This simple path predictor make the prediction depending on the address of the previous branch instruction. This simple path predictor reduced branch

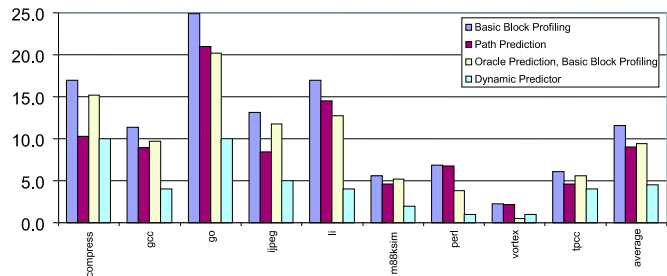


Figure 5: Branch misprediction rates

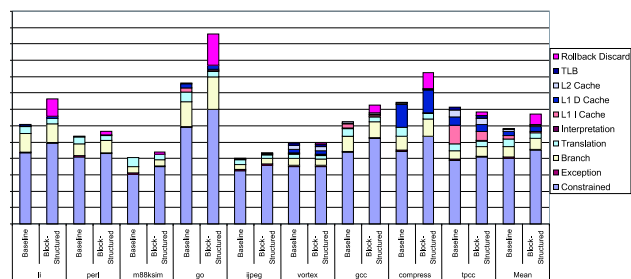


Figure 6: CPI for a translation strategy based on Block Structured ISA

mispredictions to 9.0%. Despite this improvement in misprediction rates, the geometric CPI mean of SPECint and TPC-C benchmarks remained virtually unchanged at 1.15.

Since the dynamic optimizer is a mutable piece of software (typically a part of the firmware in a configuration Flash ROM), it can be updated and modified to incorporate new optimizations, and support new trace group formation and instruction scheduling policies. In one experiment, we have extended the dynamic optimizer to implement a version of the block-structured ISA [18] as the basis for trace formation.

In the block structured ISA approach, an entire path of instructions is either executed atomically, or

all state changes discarded when a path misprediction is encountered. The aim of this policy is to achieve complete optimization and scheduling freedom along the predicted path, as side exits from the trace fragment need not be considered. When the trace fragment corresponds to a mispredicted path, a recover trap is raised and control transfers to a code block implementing an alternative path.

Figure 6 shows the CPI has degraded for SPECint benchmarks due primarily due to an increase in the constrained machine CPI and the additional rollback discard penalty which accounts for a block being discarded if the path was incorrectly predicted. At the same time, the TPC-C result improved due to a decrease in the L1 instruction cache adder.

Both the increase in the constrained instruction cache adder for SPECint benchmarks and the reduction in the L1 instruction cache adder are due to a change in group formation policy. While the baseline results require only a certain threshold of predictability for a basic block to be included in a trace path, the block structured ISA configuration required 100sampled period in order to reduce the cost of aborted atomic blocks. This reduced the overall length of the traces which were formed, and since trace path length has a significant impact on CPI [19], this caused constrained machine CPI degradation. At the same time, this also reduce code duplication and hence reduction instruction cache pressure for the TPC-C workload.

5 Related Work

BOA is a variable-width VLIW leveraging the design experience of previous VLIW research at IBM [7, 8, 20, 21, 9, 10, 11, 5, 4]. Many of the features developed over the course of IBM’s VLIW research effort have found their way into other VLIW architectures, such as scalable variable-width VLIW in-

structions [22], the handling of exceptions in speculatively executed instructions based on deferred-exception indicator bits [23] and hardware support for alias detection during aggressive speculation of memory operations [24].

The two systems most closely related to BOA are the IBM DAISY and Transmeta Crusoe processor. Like BOA, these systems implement a host architecture on top of a VLIW specifically designed as a target for binary translation and dynamic optimization. The Transmeta Crusoe processor has architecture support for dynamic optimization which is of similar scope as BOA, but optimized for x86 architectural compatibility and with narrow issue width [25, 26]. No microarchitectural details have been published for the Crusoe processors.

While BOA uses special-purpose hardware support in the form of the checkpointing and rollback facilities for the architecting of precise exceptions, the DAISY uses in-order software-managed commit operations. This allows to take exceptions without rolling back the processor state to the previous checkpoint by determining the corresponding original program point for any optimized trace fragment. To ensure correct correspondence of the program state between the optimized code fragments and the original program, DAISY has to compute the entire state and commit it in original program order. Because DAISY was targeted at wide high-performance VLIW architectures [27], this did not result in performance degradation. Later work extends this approach to allow for the elimination of dead state during the optimization of trace fragments [28, 29], and thus allows to perform more aggressive optimization of trace fragments on architectures with more limited issue bandwidth, such as for PowerPC-to-PowerPC dynamic optimization.

The present approach is different from the DIF approach of Nair and Hopkins [30], and trace processors [31]. By choosing to implement the trace for-

mation and scheduling in software, BOA can generate perform more extensive profiling to determine which trace paths are frequently executed, assemble longer traces, perform more aggressive optimizations and generate better schedules. Also, the underlying hardware only has to support a single execution mode whereas DIF and trace processors require almost three machines: a frontend processor used when executing normal code which has not been collected and preprocessed, a system for preparing, cracking and pre-scheduling the traces to be stored in the trace cache, and the execution engine optimized for executing code from the trace cache.

Our trace-based dynamic optimization is related to the idea of optimizing for the most likely execution path described by Fisher [32]. Trace scheduling requires to estimate the likelihood of a given path being executed and thus requires information about the runtime behavior of programs. Modern compilation systems attempt to address this issue by collecting execution profiles to be used by the compiler. Alas, this profile-directed feedback approach does not allow to optimize the program for different execution profiles according to specific workloads, or for phased program behavior. Unlike the static compilation techniques assumed in this work, dynamic optimization profits from the ability to collect and process profile information at runtime and to react to execution profile changes.

6 Future Directions and Conclusions

Some possible extensions to the current BOA architecture include support for dynamic optimization code executing in a parallel thread. This would allow to use empty cycles to further optimize traces. Since the dynamic optimizer is a trusted system thread executing at the hypervisor level, the second thread

can execute with a much simpler machine model, wherein not all resources have to be duplicated. For example, the support for the second thread can omit any support for the virtual memory system, since the dynamic optimizer will always reside in main memory executing with physical addresses.

While we have chosen a simple, streamlined VLIW architecture as the target of the BOA dynamic optimizer, the optimizations performed during optimization are similarly application to superscalar processors [33] in a dynamic optimizer performing PowerPC-to-PowerPC dynamic optimization.

We feel that dynamic optimization is a nascent technology which allows to capture many of the benefits found in out-of-order superscalar processors and incorporate many leading edge architectural ideas.

Acknowledgments

The definition of a new system is never the work of a few individuals, but of an entire team. The authors would like to thank Ai Chang, Kemal Ebcioglu, Marty Hopkins, Craig Agricola, Dave Appenzeller, Arthur Bright, Jason Fritts, Steve Kosonocky, Paul Ledak, Sumedh Sathaye, and Zac Filan.

References

- [1] P. Bose, D. Brooks, A. Büyüktosunoğlu, P. Cook, K. Das, P. Emma, M. Gschwind, H. Jacobson, T. Karkhanis, P. Kudva, S. Schuster, J. Smith, V. Srinivasan, V. Zyuban, D. Albonesi, and S. Dwarkadas. Early-stage definition of LPX: A low power issue-execute processor. In *Proc. of PACS'02 held in conjunction with HPCA*, Cambridge, MA, 2002.

- [2] A. Büyüktosunoğlu. *Power-Efficient Issue Queue Design*. PhD thesis, University of Rochester, Rochester, NY, 2002. to appear.
- [3] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *Proc. of the 28th International Symposium on Computer Architecture*, pages 230–239, June 2001.
- [4] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 2001. in press.
- [5] K. Ebcioglu, E. Altman, S. Sathaye, and M. Gschwind. Optimizations and oracle parallelism with dynamic translation. In *Proc. of the 32nd ACM/IEEE International Symposium on Microarchitecture*, pages 284–295, Haifa, Israel, November 1999. ACM, IEEE, ACM Press.
- [6] B. R. Rau and J. A. Fisher, editors. *Instruction-level parallelism*. Kluwer Academic Publishers, 1993. Reprint of *The Journal of Supercomputing*, 7(1/2).
- [7] K. Ebcioglu. Some design ideas for a VLIW architecture for sequential-natured software. In M. Cosnard et al., editor, *Parallel Processing*, pages 3–21. North-Holland, 1988. (Proc. of IFIP WG 10.3 Working Conference on Parallel Processing).
- [8] G. M. Silberman and K. Ebcioglu. An architectural framework for supporting heterogeneous instruction-set architectures. *IEEE Computer*, 26(6):39–56, June 1993.
- [9] K. Ebcioglu and E. Altman. DAISY: dynamic compilation for 100% architectural compatibility. Research Report RC20538, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1996.
- [10] K. Ebcioglu 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.
- [11] K. Ebcioglu, 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.
- [12] J.M. Tandler, J.S. Dodson, J.S. Fields, , H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, January 2002.
- [13] G. M. Silberman and K. Ebcioglu. An architectural framework for migration from CISC to higher performance platforms. In *Proc of the 1992 International Conference on Supercomputing*, pages 198–215, Washington, DC, July 1992. ACM Press.
- [14] M. Gschwind, E. Altman, S. Sathaye, P. Ledak, and D. Appenzeller. Dynamic and transparent binary translation. *IEEE Computer*, 33(3):54–59, March 2000.
- [15] M. Gschwind. Method and apparatus for the selective scoreboarding of computation results. *Research Disclosures*, 2001. in press.
- [16] M. Gschwind. Pipeline control mechanism for high-frequency pipelined designs. US Patent 6192466, February 2001.

- [17] E.R. Altman, J. Moreno, and M. Moudgill. Method and Apparatus for Reducing the Number of Ports to Shared Resources in a Processor. US Patent Filing, July 2000.
- [18] E. Hao, P.-Y. Chang, M. Evers, and Y. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. In *Proc. of the 29th annual IEEE/ACM international symposium on Microarchitecture*, pages 191–200, Paris, France, 1996.
- [19] S. Sathaye, P. Ledak, J. LeBlanc, S. Kosonocky, M. Gschwind, J. Fritts, Z. Filan, A. Bright, D. Appenzeller, E. Altman, and C. Agricola. BOA: Targeting multi-gigahertz with binary translation. In *Proc. of the 1999 Workshop on Binary Translation*, IEEE Computer Society Technical Committee on Computer Architecture Newsletter, pages 2–11, December 1999.
- [20] J. Moreno, M. Moudgill, K. Ebcioğlu, E. Altman, C. B. Hall, R. Miranda, S.-K. Chen, and A. Polyak. Simulation/evaluation environment for a VLIW processor architecture. *IBM Journal of Research and Development*, 41(3):287–302, May 1997.
- [21] J. Moreno, K. Ebcioğlu, M. Moudgill, and D. Luick. ForestaPC user instruction set architecture. Research Report RC20733, IBM T.J. Watson Research Center, Yorktown Heights, NY, February 1997.
- [22] J. Moreno. Object-code compatible representation of very long instruction word programs. US Patent 5669001, September 1997.
- [23] K. Ebcioğlu and G. Silberman. Handling of exceptions in speculative instructions. US Patent 5799179, August 1998.
- [24] K. Ebcioğlu, E. Kronstadt, and M. Kumar. Method and apparatus for improving performance of out of sequence load operations in a computer system. US Patent 5542075, July 1996.
- [25] E. Kelly, R. Cmelik, and M. Wing. Memory controller for a microprocessor for detecting a failure of speculation on the physical nature of a component being addressed. US Patent 5832205, November 1998.
- [26] A. Klaiber. The technology behind Crusoe processors. Technical report, Transmeta Corp., Santa Clara, CA, January 2000.
- [27] 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.
- [28] M. Gschwind and E. Altman. Optimization and precise exceptions in dynamic compilation. In *Proc. of the 2000 Workshop on Binary Translation*, Philadelphia, PA, October 2000. also in: *Computer Architecture News*, March 2001.
- [29] M. Gschwind and E. Altman. Precise exceptions in dynamic compilation. In *Proc. of the 2002 Workshop on Compiler Construction*, LNCS, Grenoble, France, 2002.
- [30] R. Nair and M. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 13–25, Denver, CO, June 1997. ACM.

- [31] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. US Patent 5381533, January 1995.
- [32] J.A. Fisher. Trace scheduling: A technique for global microcode compaction. *Transactions on Computers*, C-30(7):478–490, July 1981.
- [33] K. Ebcioglu and R. Groves. Some global compiler optimizations and architectural features for improving the performance of superscalars. Research Report RC16145, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1990.