

List of Figures

1.1	Example stages of instruction execution.	14
1.2	Decoupled view of the processor: a fetch engine produces instructions, and an execution engine consumes them.	14
1.3	Fetch performance factors: memory latency, fetch width, and instruction quality.	15
1.4	Thesis overview: from the problems observed to the proposed solutions.	18
2.1	Example of a basic block chaining algorithm. Basic blocks are mapped so that the execution trace is consecutive in memory.	22
2.2	Examples of the procedure splitting algorithm. Infrequently used basic blocks are mapped to a different procedure, and moved away of the execution path.	23
2.3	Routine mapping algorithm by Pettis & Hansen. Two routines which call each other will be mapped close in memory to avoid conflicts among them.	24
2.4	Code mapping algorithm by Torrellas <i>et al.</i> The most heavily executed basic blocks are mapped to a reserved area of the instruction cache, eliminating conflicts in important parts of the code.	25
2.5	Pipelined execution of a branch instruction. The branch is not resolved until the ALU stage, which introduces two delay slots.	27
2.6	Comparison of the branch execution cost for different branch architectures using both software and hardware techniques.	28
2.7	gshare (global history)	30
2.8	PAP (private history)	31
2.9	Dealiased branch predictors.	32
2.10	A fetch mechanism capable of reading one basic block per cycle.	33
2.11	Performance metrics for the fetch engine proposed by Yeh & Patt.	34
2.12	Decoupling the fetch stage: an independent branch prediction mechanism provides fetch blocks to a Fetch Target Queue, and the pipelined instruction cache reads the blocks from memory.	36
2.13	Extension of the superscalar fetch engine with a multiple branch predictor to read multiple consecutive basic blocks per cycle.	38
2.14	Fetch width provided by the branch address cache (IPF), and fetch engine performance measured in instructions per fetch cycle (IPFC).	39
2.15	Processor performance using different sequential fetch policies and using the Collapsing Buffer.	39
2.16	Extension of the wide superscalar fetch engine with a trace cache to allow fetching of non-consecutive basic blocks in a single cycle.	41

2.17	Comparison of the processor performance using different fetch engines, including the Trace Cache.	42
2.18	The trace cache is storing redundant information, because a basic block may be present in more than one trace cache line.	43
2.19	The block based trace cache stores basic blocks in a special purpose block cache, and stores block pointers in the trace table, eliminating the basic block redundancy.	43
2.20	The contributions of this thesis in their historical context.	45
3.1	Major components of the Oracle 7.3.2 server.	50
3.2	Profiling of the baseline application and generation of the optimized code layouts.	52
4.1	Example of the Software Trace Cache basic block chaining algorithm. Basic blocks are mapped so that the execution trace is consecutive in memory.	61
4.2	Trace mapping for a direct mapped instruction cache.	62
4.3	Determining the size of the CFA from the execution frequency and size of the most popular traces.	63
4.4	Instruction cache miss rate for various cache sizes when using different hardware configurations and code layout optimizations.	64
4.5	Instruction cache misses for various cache and line sizes for a commercial database management system running an OLTP workload.	65
4.6	Relative number of misses in the optimized DBMS binary compared to the baseline application for various cache and line sizes.	66
4.7	Code layout optimizations increase the number of sequentially executed instructions.	67
4.8	Layout optimized codes use all the instructions in a cache line before it is replaced.	68
4.9	Instruction cache lines have an increased lifetime in layout optimized codes.	69
4.10	Almost all instructions loaded in the cache are used at least once, and instruction reuse increases in optimized codes.	70
4.11	Code layout optimization increase the fetch width by aligning branches towards not taken.	70
4.12	Code layout optimizations effectively increase the fetch width of baseline and trace cache fetch architectures.	71
4.13	Static branch prediction accuracy for the original and optimized code layouts (self and cross trained).	72
4.14	The use of optimized code layouts reverses branch direction, so that they tend to be usually not taken.	74
4.15	Dynamic prediction accuracy for both the base and the STC optimized code layouts using two-level adaptive prediction schemes.	75
4.16	Percent of dynamic branches which cause interference in the gshare prediction tables for the baseline and optimized code layouts.	75
4.17	Effect of the optimized code layout on dealiased branch predictors.	77
4.18	Percent of dynamic branches which cause interference in the gshare prediction tables optimized code layout and the agree predictor using both code layouts.	78
4.19	Prediction accuracy of the GAg branch predictor compared to that of the gshare predictor using the baseline and the optimized code layouts.	78
4.20	Branch history register value distribution for the baseline code layout (a), and the STC optimized layout (b).	80

4.21	Code layout optimizations impact not only the L1 instruction cache, but the whole memory hierarchy.	81
4.22	Overall processor performance increases beyond the perfect instruction cache using code layout optimizations.	81
4.23	Impact of several code layout optimizations on the overall system performance.	82
5.1	Redundancy of traces between the instruction cache and the trace cache. Traces consisting entirely of consecutive instructions (blue traces) can be fetched from the instruction cache in a single cycle without a trace cache.	86
5.2	Distribution of traces classified by the number of sequence breaks they contain. Numbers shown for both the original code layout and the STC reordered code. Traces with 0 breaks are considered blue traces.	87
5.3	FIPA performance for trace cache sizes from 32 to 2048 entries (2KB to 128KB) using realistic branch prediction. The STS enhanced models are tagged with (+). Setups labeled STC use the optimized code layout.	89
5.4	FIPC performance for trace cache sizes from 32 to 2048 entries (2KB to 128KB) and instruction cache sizes of 32 and 64KB using realistic branch prediction. The STS enhanced models are tagged with (+). Setups labeled STC use the optimized code layout.	91
5.5	FIPA performance for trace cache sizes from 32 to 2048 entries (2KB to 128KB) using perfect branch prediction. The STS enhanced models are tagged with (+). Setups tagged STC use the optimized code layout.	93
5.6	FIPC performance for trace cache sizes from 32 to 2048 entries (2KB to 128KB) and instruction cache sizes of 32 and 64KB using perfect branch prediction. The STS enhanced models are tagged with (+). Setups tagged STC use the optimized code layout.	95
6.1	Prediction accuracy of the base gshare predictor and a gshare predictor using an optimized code layout.	99
6.2	Prediction accuracy of the basic agree predictor, the profile-assisted agree and the compiler optimized gshare predictors	100
6.3	Prediction accuracy of the bimode predictor, the compiler-enhanced bimode, and the compiler optimized gshare predictors.	101
6.4	Prediction accuracy of the gskew predictor, and the gshare and gskew predictors on the compiler optimized code layout.	101
6.5	(a) Combination of any dynamic prediction scheme with a profile-based static predictor. (b) Prediction accuracy of the static-dynamic combination using a compiler-assisted agree for the dynamic component.	103
6.6	(a) The agbias branch prediction scheme. (b) Prediction accuracy of the agbias predictor compared to other compiler-enhanced predictors.	105
6.7	Percent of executed branches which cause conflicts in the dynamic prediction tables. Conflicts classified (top to bottom) in positive, neutral and negative, the aggregate corresponds to the total interference rate. There is a separate column for each predictor.	106

6.8	Prediction accuracy of the agbias predictor, an unbounded predictor with the same history length, and an unbounded predictor with a filtered history. The agree[p] and gshare predictors shown for reference.	107
7.1	Example instruction streams. A stream is a sequential run of instructions.	110
7.2	Average length of a basic block, an instruction trace, and an instruction stream. Streams are longer than traces in optimized codes.	112
7.3	The proposed stream fetch engine. The instruction stream becomes the basic fetch entity.	112
7.4	Path correlated implementation of the next stream predictor.	113
7.5	Fetch target queue update mechanism. A fetch request contains a whole stream and may take several cycles to be fetched.	114
7.6	The instruction misalignment problem. A 3 instruction stream may take more than one cycle to fetch on a 4-wide engine if the instructions cross the cache line boundary.	115
7.7	Cache line reuse buffer mechanism. The line buffer reduces instruction cache activity and exploits longer cache lines.	116
7.8	IPC speedup of the stream fetch engine compared to a BTB fetch engine and IPC slowdown compared to the trace cache.	117
7.9	Activity of the instruction cache and branch predictor for the BTB, stream, and trace cache architectures.	121

List of Tables

3.1	Description of the SPEC'95 integer benchmarks.	48
3.2	TPC-D queries used to obtain the profile information (training) and to obtain performance results (test).	49
3.3	Default simulator setup for the isolated fetch engine simulator.	53
3.4	Default setup for our SimpleScalar simulator.	54
3.5	Default simulator setup for the branch predictor simulator.	55
3.6	Default SimOS simulations setup.	56
3.7	Detailed simulator setup for the complete processor simulator.	57
5.1	Average dynamic trace length. Separate results for blue trace length and red trace length are provided for the original code layout and the STC optimized one. . . .	92
7.1	Fetch performance metrics: actual fetch width and branch prediction accuracy. . .	118
7.2	IPC degradation (%) of the BTB and stream fetch architectures when the latency of the branch predictor increases to 2 cycles.	119
7.3	IPC degradation (%) of the BTB and stream fetch architectures as the number of ports to the branch predictor is limited to just one.	120