A Run-time Optimization Technique for Tolerating Cache Misses

November 16, 2002

Abstract

Stalls caused by data cache misses are a costly problem for high-performance computer systems. Out-of-order processors avoid many of these stalls at the expense of complex hardware and a deeper pipeline. In addition, their continuous rescheduling of instructions (even when it provides no benefit) is a significant source of additional power consumption.

This paper presents *cache-miss stall deferral*, a strategy for allowing in-order EPIC processors to tolerate variable-latency memory operations by continuing issue beyond instructions dependent upon a missed load. The technique uses EPIC control-speculation recovery mechanisms to inexpensively recover from cache-miss stalls which are deferred at run-time. By using an in-line recovery mechanism, no additional code needs to be generated for recovery. Reexecution proceeds only when required, eliminating unneeded stalls on speculative operations.

The strategy presented realizes similar benefit to out-of-order processors in that instructions following a stalled operation can be overlapped with the handling of a cache miss, often enabling overlap of additional cache misses that would not be possible in sequential execution. Our approach maintains the in-order model of execution avoiding the power consumption, complexity and pipeline stages required by out-of-order execution while achieving significant performance improvement over traditional EPIC processors executing aggressively optimized code.

1 Introduction

In order to meet the latency requirements of modern high-frequency microprocessors, first-level data caches have remained relatively small in size. For this reason, the additional latency caused by data cache misses continue to be a major performance bottleneck. Out-of-order processors tolerate this variable latency behavior of memory operations though dynamic scheduling which allows the overlapped execution of instructions that follow a stalled instruction with the handling of the cache miss. However, in in-order processors, an instruction that consumes the destination of a load will stall if that memory operation's result has not yet returned from the memory hierarchy.

As dynamic scheduling allows execution beyond a stalled instruction, some of the inherent benefit from out-of-order execution comes from its ability to perform additional memory instructions which would be blocked by a stalled instruction in an in-order machine. Such loads might also miss in the cache. In this way, considerable performance improvement is achieved simply through

handling multiple cache misses simultaneously that would otherwise be handled sequencially.

This paper presents *cache-miss stall deferal*, a strategy for allowing in-order processors to proceed beyond a stalled instruction waiting on a cache access. By maintaining the in-order model of excuction, the additional complexity and pipeline stages required by out-of-order execution are

This paper presents a new mechanism for use in run-time optimization systems for allowing in-order processors to achieve much of the cache miss toleration that was previously only available in dynamically scheduled machines. While this work builds upon a previously described system, it could be extended to other dynamic optimization systems which exploit hardware support. The next section discusses the relevant previous work. Sections 3 and 4 introduce the *Run-time Optimization Architecture* (ROAR) and the implementation of cache-miss stall deferal within the ROAR framework. Section 5 reports the experimental evaluation of our approach, and Sections 6 and 7 contain future work and conclusions.

2 Related Work

Focus on memory tolerant techniques and less on runtime opti

OOO ====

Instruction Issue Logic in Pipelined Supercomputers

Comparison of different issue models, namely CRAY-1 scalar, Tomasulo's algorithm, thortons scorboard algorithm, and a direct tag search algorithm. Using CRAY as baseline, tomasulo gets 58% thorton gets 28% dts gets 38%

Implementation of precise interrupts in pipelined processors

This is the reorder buffer paper. 25%-3% degradation depending on handling of stores (do they wait for pipeline to drain or issue and wait in memory unit).

Checkpoint Repair for High Performance Out-of-order Execution Machines

Checkpoint repair allows recovery from an erroneous speculative state to a previous, valid state by recording state changes within difference lists. The size of the difference lists limit the amount and distance of reordering that can take place.

out-of-order tolerate cache misses better than in-order Cache miss delay can be hidden by dynamic code scheduler out-of-order is less sensitive to cache miss behavior than in-order

Runtime optimization ==========

Dynamo: A Transparent Dynamic Optimization System

Transparently optimizes a native application as it executes This system does not currently employ a speculation mechanism if precise exceptions are required, avoids optimizations that might violate precise exceptions (limits its aggressiveness).

¡– OOO rePLay: A Hardware Framework for Dynamic Optimization

Frame is similar to a very long trace cache line and is atomic in its execution Speculation failure within a trace forces a recovery of state to a point before execution of the trace began Re-execution resumes from the original code. State recovery throws out all newly generated values including any correctly executed portions.

¡— EPIC (adapt to profile variation but not cache)

Hotspot detection looks for important phases with consistent control paths and adapts to control variation. Does not look for important cache misses nor adapts to cache miss variation within the same hotspot.

Reference EPIC somewhere (maybe in intro?)

EPIC: An Architechure for Instruction Parallel Processors

enable higher levels of instruction-level parallelism without unacceptable hardware complexity

Place bulk of burden of POE on the compiler Predictable runtime environment

complexity. OOO EPIC ========

renaming and dynamic scheduling of predicated code Normally stall renamer if unresolved predicates guard multiple write to same reg hold op in res. station until predicate resolves *With Select u-op: (pX) r5 = , (pY) r5 = , use r5 ===¿ (pX) rW =, (PY) rU = , use rS and a select rS = (rW,rU) , which is a SSA phi-like node However, the reservation station will still fillup waiting for predicate resolution. *With predicate slip: predicated instructions are executed as soon as non-pred sources are available. There results are held in the reorder buffer until the predicate is resolved, in which case bypassed/broadcast occurs. 15% average speedup

Simulataneous Multithreading: Maximizing On-Chip Parallelism

Examination of multiple SMT models compared to a wide superscalar or convential multithreading architectures. Vertical (idle cycles) and Horizontal (unused issue slots) waste

OOO (supposedly the 2001 work) and/or speculative precomputation (using SMT) yeilds 87%, 92%, and, with both,141%

OOO: ability to tolerate memory latency (L1 in particular)

SP: tolerate small number of delinquent loads[4] (L2,L3 in particular) basic trigger: main thread spawns spec thread chain trigger: spec thread spawns spec thread

Both OOO and SP aim to hide memory latency by overlapping execution with cache misses. It appears that OOO gets the bulk of, and most flexible part of the improvement. SP does well and can be combined with OOO, but really only has a chance on memory intensive benchmarks.

3 Speculation Models

This section details the operation of two previously presented techniques for dealing with compiler-controlled speculation. The two models are not exclusive, but they were proposed for different uses. Precise-speculation was proposed as an additional method of control-speculation that would allow instruction reordering that maintained precise exceptions and required a minimal amount of analysis. This technique is particularly attractive for speculation performed by run-time code generation. Sentinal speculation with inline recovery is a related approach that is less restrictive than precise speculation but requires more capable analysis.

The cache-miss deferral technique exploits a combination of both strategies. While the approach is not strictly tied to Precise Speculation, it is proposed in this work for operations that are precise-specuative. The advantages of using this model will be discussed in Section 4. The recovery from a deferred operation uses the inline recovery mechanism presented in [12].

3.1 Precise speculation model

Precise speculation was proposed in [1?] to minimize the restrictions = placed upon the control and data speculation across potentially = excepting instructions, as well as speculative relocation of them and = branches while completely preserving both the ordering and the liveness = requirements to maintain precise exceptions. Both potentially excepting = instructions and branches can change the control flow. Speculative = instructions must not destroy the contents of registers that may be life = if this not sequential path is taken. The mechanism provides a separate = speculative register file where speculative instructions write their = results. At commit time, the architectural registers are updated. = Non-speculative instructions, on the other hand, work directly with the = architectural registers. When a check detects any speculative exception, = the content of the speculative register file is discarded, and a = transition to recovery mode is then forced. Some code is reexecuted and = exceptions are cleared due to completion of pending event or are = reproduced in their proper order. It should be noted than the order of = stores is never changed and that stores are never speculated.

If a taken branch leads to skip a commit, the associated pending = speculative exceptions are no more needed, and can be cleared (?).

As said before and explained in detail in [1?], precise speculation = requires a special register file that contains a speculative half and = non-speculative half as well as a S-valid bit to specify the validity of = the first one. Non-speculative writes always update both halves and = clear S-valid. Speculative instructions write only the speculative half = and set S-valid. All reads from a register are done from the speculative = half. Speculative reads leave the S-valid bit unchanged. Explicit commit = instructions are no-ops composed of the sources ready for committing. = These ones and non-speculative reads clear the S-valid bit and, if it = was set, commits the outstanding speculative value, copying it to the = non-speculative half. The complete register file can be restored to a = non-speculative state by copying the non-speculative half to the = speculative half for each register and setting its S-valid bit.

At runtime, speculative branches except silently, meaning that they flag = an exception but otherwise do nothing. Non-speculative branches except = if either the instruction itself causes an exception or if a preceding = speculative instruction silently excepted. At this event, the register = file is restored to non-speculative state and control is transferred to = =85? a point prior to any of the speculated instructions, executing in = recovery mode (?).

Precise speculation provides greater freedom in code motion by allowing = the merging of basic blocks.

3.2 Exception recovery model

EPIC architecture employs speculation to initiate loads from memory = earlier in the instruction stream. Because moved instructions can = produce exceptions which cannot occur in sequential program order, = mechanism must ensure that exceptions are properly handled. Memory load = is broken into speculative load and commit instructions, which verify = and generate exception if necessary and check for load-store conflicts. Load speculation design tries to avoid unnecessary cache misses as well = as unwanted page faults. A speculation bit (S-bit) is set for any = operation which is control speculated, or data dependent on = data-speculative load. Non-speculative operations report exceptions = immediately. But the architecture must provide mechanisms to detect = potential exceptions on control-speculative operations as they occur and = to record exceptions until check. For this purpose, an exception bit = (E-tag) is added to each register, indicating that exception occurred = during generation of value stored in associated register. E-tag is = forwarded

with its associated register. Architecture must be designed to allow recovery from exceptions on = control speculation. The IMPACT EPIC recovery model, based on an = improved Sentinel model, added an R-bit to each register, which is used = to selectively execute only data-flow successors of excepting = speculative operations during recovery. Every time an operation finishes = without exception, the E-tag in destination register is cleared and = result stored in destination register; otherwise program counter is = stored into destination register and E-tag in destination register is = set. If source operand in register has its E-tag set, program counter = from this source operand is copied into destination and E-tag in = destination register is set. In this way exceptions are propagated until = a non-speculative operation is executed, which make check, or an = explicit commit instruction is reached. If check discovers one of source = operand has E-tag set, then exception is triggered.=20

One of key advantage of EPIC exception recovery model is addition of = selective inline recovery mechanism for fix up of speculative generated = exceptions or data dependence memory conflicts. For this purpose the = R-tag is added to each register, used to identify operations that are = data-flow dependent on the excepting speculated instruction. In EPIC = model semantic, branches and speculative operations (with S-bit) which = are data-flow dependent on values newly generated in recovery are = executed, until check is reached. These instructions must have their = register sources R-bit set.

Recovery can start in case that speculatively generated operand = indicating exception is used non-speculatively and therefore a genuine = exception is indicated, then resolution of exception and re-execution of = dependent instructions are required to program execution continue. = Recovery starts in case that data speculation check signals that = speculative load conflicts with intervening store, re-execution of load = and dependent instructions is required. In both cases the recovery = predicate (pR) is set to indicate recovery mode. The operation at the = recovery point is re-executed non-speculatively, and any exception is = raised immediately. The pR is an implicit predicate to all operations, = and when is set, instructions behave with according recovery semantics. = During recovery, all speculative operations dependent on operations = which trigger exception which recovery is in progress must be executed = until check operation which initiated recovery is reached. These = instructions must have in their register sources R-tag set. When = speculative instructions (including instruction which generates = exception) commit results during recovery

mode, the R-tags on their = destination registers are set. Following non-speculative execution of = instruction which produces exception (including exception handling if = need), recovery continues with re-execution of flow dependent = speculative instructions. During this phase branches and speculative = instructions with at least one R-tag set for source operand are = executed. Because execution of branch instructions is required for = original control path, branches must be executes independent of pR = value. The branch predicate must have same value in recovery as during = execution inside original execution code. Recovery must execute branches = and dependent speculative operations until R-tag on non-predicate = register (predicate register, which cannot initiate recovery, should not = indicate that recovery is completed) source operand reaches = non-speculative use. Only non-predicate register's R-tag can terminate = recovery mode, what is when home block of excepting speculation = instruction has been reached. Now, if no pending exception is indicated = by source operand E-tags then original exception or conflict is fixed, = pR is cleared, and execution continues. During recovery, propagation and = generation of E-tags occurs in same way as during normal execution. If = E-tags are set during a non-speculative use of register with R-tag set = is reached, then additional exception occurred during recovery from = initial exception. In this case, recovery restarts from the new = excepting location. Using this approach in EPIC architecture during recovery can be fetched = a lot of unusable instructions, but this approach prevent code bloating = by selective re-execution of existing code.

4 Cache-miss Stall Deferral

This section presents the architecture semantics that enable cache-miss stall deferral. As discussed is Section 3, Precise Speculation[13] has been used in ROAR to allow instruction reordering within an optimized trace stored in a memory-based code cache. Run-time optimization uses such reordering to adapt the optimization of the traces to the observed execution profile. The cache-miss deferral technique similarly exploits this speculation support to allow the in-order core to adapt execution to the variable latency of memory operations.

As originally presented[14], the selective inline recovery model was proposed for the potential deferral of both faults and cache misses for control-speculative loads. Since page faults are typically infrequent and quite expensive to handle, deferral of faults for speculative loads seems advantageous. However, the trade-offs for deferring cache misses are much less clear. Cache misses are much more common than faults and can be handled without blocking execution of the current thread. Deferring cache misses is advantageous in cases when the load is speculated by the compiler from an instruction block that is not subsequently reached. According to [14] this situation occurs frequently. However, in cases where the load's home block is reached and recovery is initiated, the cost of recovery can be exorbitant. Likely for this reason, Precise Speculation was introduced to support recovering only from spurious faults generated by run-time speculation, and thus was not intended to support deferring cache misses from speculative operations.

4.1 Execution overview

In processors without dynamic scheduling, instructions are issued sequentially. When a dependency is not met because an operation source is not ready, instruction issue stalls; once the dependency is met, instruction issue again proceeds. This behavior prevents the execution of subsequent, independent operations from overlapping with the data cache miss. In Figure 1(a), the two load operations are independent, but because the second load follows the consumer of the first, it is trapped behind consumer whenever it is stalled. It is possible that both loads will miss in the cache causing two sequential stalls. With a different order of the same instructions in Figure 1(b), the two loads are issued before any consumer can cause a stall. When both loads miss in the cache, the handling of the misses are handled simultaneously during the stall. Such an instruction ordering would be

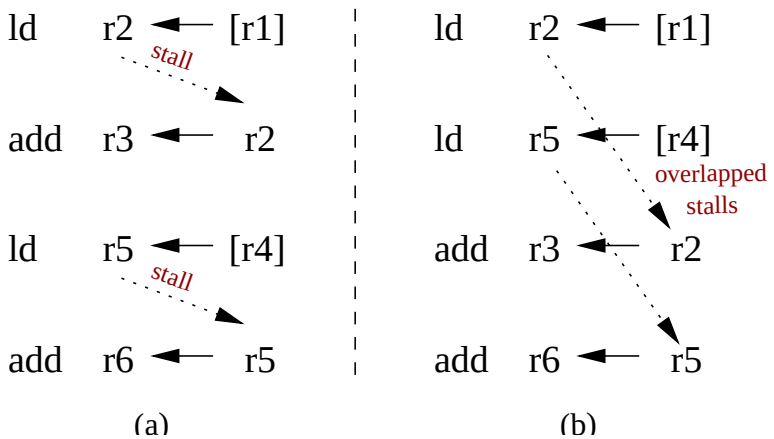

Figure 1: Two alternative instruction schedules; (a) memory access are performed sequentially allowing two possible stalls (b) memory accesses are overlapped allowing only one potential stall location.

found by dynamic scheduling whenever the first load missed in the cache. A static compiler could attempt to schedule instructions in an attempt to allow an in-order processor to overlap as many load misses as possible. However, at compile-time it is not easy to determine which instructions are likely to miss in the cache. In addition, such an ordering might not be the best schedule for the instances when there are not misses (which is often the typical case).

With cache-miss stall deferral, when a speculative operation would normally stall because of a producer's cache miss, its execution is deferred while continuing execution to subsequent instructions. Such instructions might include loads which also miss in the cache as in the example in Figure 1. Operations dependent upon the deferred instruction are skipped, while independent instructions are executed. Eventually, re-execution of all deferred or skipped instructions is initiated at a point at which speculation cannot continue. Using in-line recovery, control is returned to the location of the deferral and only the instructions not originally executed are performed. In some cases, execution from a control-speculative dependent of a speculative load might never reach non-speculative consumers. In these cases, stalls on this dependents are fruitless, and cache-miss stall deferral avoids re-execution of them.

4.2 Cache-miss deferral recovery model

Operation under cache-miss stall deferral is shown in the example in Figure 2(a). If precise-speculative load operation **A** misses in the cache, the miss can be handled as a normal, non-

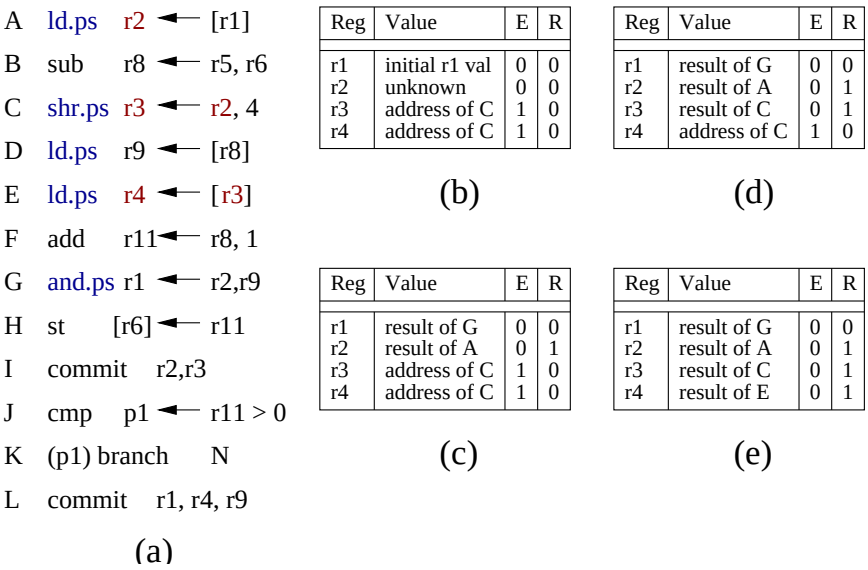

Figure 2: Cache-miss stall deferral operation

blocking miss. However, after such a miss, when operation **C** is reached, a stall would normally occur since `r2` is not ready. However, in a way similar to the operation of the speculation model in [12] explained in Section 3, operation **C** will instead mark its destination, `r3`, with an E-tag denoting that an "exceptional" condition had occurred. This bit continues to share its operation with the original exception fault deferral functionality. Since a real fault has not occured, the system-level exception flag is not set and thus a precise-speculative restore will not need to occur. Additionally, the address of operation **C** is placed in `r3` to enable re-execution of this skipped instruction. Later speculative consumers of `r3` (like operation **E**) will similarly be skipped, and will pass on the address of operation **C** through to their destinations.

Instructions that are independent of operation **A** can execute normally and will not need to be reexecuted. These instructions can include stores like operation **H**. This store must not alias with the loads speculated above it unless the load is also data-speculative (data-speculative operations can also be correctly handled through the normal data-speculation failure recovery from [12]). If later instructions like **G** consume `r2` after load **A** has completed, they will receive the correct value of `r2`. Otherwise such operations would behave exactly like operation **C**, by deferring their execution and propagating their address and the E-tag into their destination register. When operation **A** completes, it will write right its value into `r2` and will use `r2`'s R-tag to indicate if a dependent instruction was deferred and needs to be re-executed during recovery. It is important to

note that this is a slight departure from the previously proposed inline recovery behavior; the need for marking `r2`'s R-tag at this point will be explained in Section 3. By allowing **A** to complete its operation, this load will not need to be reexectued, and subsequent dependant operations that occur after **A** has completed will execute with the correct source values.

In some cases, re-execution of deffered instructions is not necessary. Consider instruction **D** in Figure 2(a). If this load operation misses in the cache, the execution of instruction **G** might be skipped. If branch **K** subsequently leaves the trace shown in Figure 2, this will cause a restore condition where all precise-speculative values are replaced with values from the architectural state. This eliminates and E-tags on registers `r1` and `r9`. Since the result of the speculative chain of instructions, **D**→**G**, is not commited, the **G**'s stall is avoided altogether.

Later, re-execution of deferred instructions might need to occur. A non-speculative dependent of an antecedent, deferred miss will require the recovery from that deferral. Since Precise Speculation allows the speculation of every operation except for stores, instruction issue can never proceed beyond dependent stores. Once re-execution is initiated by a non-speculative consumption of a register with a set E-tag, control will move to the operation whose address is stored in the consumed register. If the instruction causing re-execution has multiple source registers with E-tags set, the source with the earliest address is chosen as the target instruction for beginning the re-execution. As will be explained in Section 4.3, because of the semantics of Precise Speculation, this choice can eliminate cases where multiple re-executions of the same trace are needed.

The register file contents for the example in Figure 2(a) at instruction **F** is shown in Figure 2(b) for the case where load **A** had missed in the first level cache. Instructions **C** and **E** were deferred and thus the destination registers `r3` and `r4` contain the address of instruction **C** and have their E-tag fields set.

In this example, the latency of load **A** was assumed to be short enough that operation **G** did not need to be deferred. When **A** completes, it writes register `r2` and sets `r2`'s R-tag since the instruction, **C**, was deferred. Once instruction **I** is reached, recovery will be required since one of **I**'s sources, `r3`, has a set E-tag. The register file status at this point is shown in Figure 2(c).

With recovery initiated, the program counter is moved to the address of instruction **C** and this instruction will be executed as a non-speculative operation. At this point, since `r2` is available, no stall is needed for instruction **C** and it can compute `r3`'s value, clear its E-tag and set its R-tag.

The register contents at this point are shown in Figure 2(d).

The R-tag bits will serve to indicate which instructions need to be re-executed because of a new source value being available. Since the R-tag of `r3` is set, re-execution of **E** is needed. While it is correct to re-execute any instruction like **G** because one of its sources has an R-tag set, this execution can be avoided since **G's** destination does not have an E-tag set and thus must have received the correct source values during the original execution. Finally, once instruction **I** is reached, re-execution has completed and the pR bit can be cleared and execution can continue as normal. The status at this point is shown in Figure 2(e).

Since the deferred load continues, ... overlap with events like branch mispred.

4.3 Deferral Implementation

If only traditional speculation ...

4.4 Preservation of register lifetimes

Table 1: Simulated EPIC machine model.

Parameter	Setting
Instruction issue	8 units
Integer arithmetic and logic unit	5 units
Floating point arithmetic unit	3 units
Memory unit	3 units
Branch unit	3 units
Branch predictor	10-bit history gshare 3 predictions per cycle
BTB size	1024 entry
RAS size	32 entry
Branch resolution	7 cycles
LD/ST buffer size	8 entry each
L1 data cache	64KB
L1 instruction cache	64KB
Unified L2 cache	512KB
BBB associativity	4-way
Num BBB sets	512 set
Candidate branch threshold	16
Refresh timer interval	8192 branches
Clear timer interval	65536 branches
Hot spot detection counter size	13 bits
Hot spot detection counter inc	2
Hot spot detection counter dec	1
Exec and taken counter size	9 bits

Table 2: Benchmarks and inputs used in experiments.

Benchmark	Inputs	Instructions
099.go	A: SPEC Train	338 M
124.m88ksim	A: SPEC Train	89 M
130.li	A: SPEC Train	122 M
	B: 6 Queens	32 M
	C: Reduced Ref.	362 M
132.ijpeg	A: SPEC Train	1094 M
	B: Custom Faces	57 M
	C: Custom Scenery	320 M
134.perl	A: SPEC Train 1	1512 M
	B: SPEC Train 2	28 M
	C: SPEC Train 3	8 M
164.gzip	A: Reduced Train	1902 M
175.vpr	A: SPEC Test	1012 M
181.mcf	A: SPEC Test	105 M
197.parser	A: UMN_sm_red	178 M
255.vortex	A: UMN_sm_red	63 M
	B: UMN_md_red	315 M
	C: UMN_lg_red	886 M
300.twolf	A: UMN_sm_red	167 M
mpeg2dec	A: Media Train	99 M

5 Experimental Results

6 Future Work

7 Conclusions

References

[1] S. Weiss and J. E. Smith, "Instruction issue logic in pipelined supercomputers," *IEEE Transactions on Computers*, vol. C-33, pp. 1013–1022, November 1984.

[2] W. W. Hwu and Y. Patt, "Checkpoint repair for high performance out-of-order execution machines," *IEEE Transaction on Computers*, vol. C-36, pp. 1496–1514, December 1987.

[3] P. P. Chang, W. Y. Chen, S. A. Mahlke, and W. W. Hwu, "Comparing static and dynamic code scheduling for multiple-instruction-issue processors," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 25–33, November 1991.

[4] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp. 1–12, June 2000.

[5] S. J. Patel and S. S. Lumetta, "rePLay: A hardware framework for dynamic optimization," *IEEE Transactions on Computers*, vol. 50, pp. 590–608, June 2001.

[6] M. C. Merten, A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and W. W. Hwu, "An architectural framework for runtime optimization," *IEEE Transactions on Computers*, vol. 50, pp. 567–589, June 2001.

[7] M. C. Merten, *Run-Time Optimization Architecture*. PhD thesis, University of Illinois, Urbana, IL, 2002.

[8] M. Schlansker and B. R. Rau, "EPIC: An architechure for instruction parallel processors," Tech. Rep. HPL-1999-111, Hewlett-Packard Laboratory, 1501 Page Mill Road, Palo Alto, CA 94304, February 2000.

[9] P. H. Wang, H. Wang, R. M. Kling, K. Ramakrishnan, and J. P. Shen, "Register renaming and scheduling for dynamic execution of predicated code," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pp. 15–25, January 2001.

[10] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 392–403, June 1995.

[11] P. H. Wang, H. Wang, J. D. Collins, E. Grochowski, R. M. Kling, and J. P. Shen, "Memory latency-tolerance approaches for itanium processors: Out-of-order execution vs. speculative precomputation," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pp. 167–176, February 2002.

[12] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, "Integrated predicated and speculative execution in the IMPACT EPIC architecture," in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 227–237, June 1998.

[13] E. Nystrom, R. D. Barnes, M. C. Merten, and W. W. Hwu, "Code reordering and speculation support for dynamic optimization systems," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 163–174, September 2001.

[14] D. I. August, J. W. Sias, J. Puiatti, S. A. Mahlke, D. A. Connors, K. M. Crozier, and W. W. Hwu, "The program decision logic approach to predicated execution," in *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 208–219, May 1999.