Dynamic Techniques for Load and Load-Use Scheduling

AMIR ROTH, RONNY RONEN, SENIOR MEMBER, IEEE, AND AVI MENDELSON, MEMBER, IEEE

Invited Paper

Modern microprocessors employ dynamic instruction scheduling to select independent instructions for parallel execution. Good scheduling of loads is crucial, since the long latency of some loads makes them likely to degrade performance. A good scheduler attempts to issue loads as early as possible.

Scheduling loads is not simple. First, safely resolving a load's input dependences can be done only at execution time, after the load address and all previous store addresses are known. Second, varying load latency makes it difficult to prioritize loads and to efficiently schedule load-dependent instructions.

This paper surveys several techniques that optimize load scheduling. ***Memory disambiguation*** *resolves store–load dependences and enables earlier execution of store-independent loads.* ***Memory renaming*** *and* ***memory bypassing*** *short-circuit memory to streamline the passing of values from stores to loads.* ***Critical path scheduling, pre-execution,*** *and* ***address prediction*** *advance long-latency loads by computing load addresses early, or predicting them.* ***Value prediction*** *short-circuits load execution by predicting the loaded data values. Finally,* ***data speculation*** *and* ***hit–miss prediction*** *help the scheduling of load-dependent instructions.*

Keywords**—***Dynamic instruction scheduling, load–store scheduling, memory bypassing, memory dependences, memory disambiguation, memory forwarding, memory renaming, out-of-order processor.*

I. INTRODUCTION

Programs are collections of operations that communicate using named storage spaces. A value flows from a producing operation to a consuming operation by virtue of the producer naming a certain location as its output, the consumer naming the same location as its input, and the consumer executing after the producer. A producer–consumer relationship established in this way is called a *data-dependence* or simply *dependence*. *Data-flow order* is a minimal partial order that preserves data-dependences—correct ordering for all producer–consumer pairs—and, hence, the meaning of the program. By not specifying the order of *independent operations*—operations not linked via direct or indirect data-dependences—data-flow order establishes the upper bound on concurrency in the program and, thus, its performance. The data-flow performance limit cannot be achieved on actual processors, which can execute only a finite number of operations concurrently. *Scheduling* is the process of choosing a restricted partial order—perhaps even a total order—for the operations that minimizes program execution time on a given processor.

Scheduling is performed at several levels. *Control-flow order* is a total order presented to the processor as the result of the combined scheduling effort of the programmer and compiler. The programmer makes high-level decisions while the compiler schedules individual instructions. Control-flow order, also called *program order*, gives the programmer and compiler a sequential interface that produces repeatable executions and aids in reasoning about program correctness. Processors are obliged to externally maintain the appearance of program order execution. However, many modern processors use a technique called *dynamic scheduling* or *out-of-order execution* that allows them to locally rearrange instructions. Fig. 1 shows a sample program consisting of a simple loop. Two variations of the loop are shown; they are used throughout the paper to demonstrate various techniques. The figure presents the loop's C source, a dynamic instruction stream consisting of several iterations in program order, the data-dependences between dynamic instructions, and possible execution schedules.

In this paper, we assume that the programmer and compiler schedule code to the best of their abilities and concern ourselves with scheduling performed by the processor itself, both within the framework of dynamic scheduling and as extensions to it. In making scheduling decisions, the processor can use run-time information that is not available to the programmer and compiler. In addition, when run-time information is not available early enough, the processor can use *spec-*

Manuscript received February 5, 2001; revised June 15, 2001.

A. Roth is with the University of Pennsylvania, Philadelphia, PA 19104 USA (e-mail: amir@cis.upenn.edu).

R. Ronen and A. Mendelson are with Intel Israel 74, Haifa 31015, Israel (e-mail: ronny.ronen@intel.com; avi.mendelson@intel.com).

Publisher Item Identifier S 0018-9219(01)09685-2.

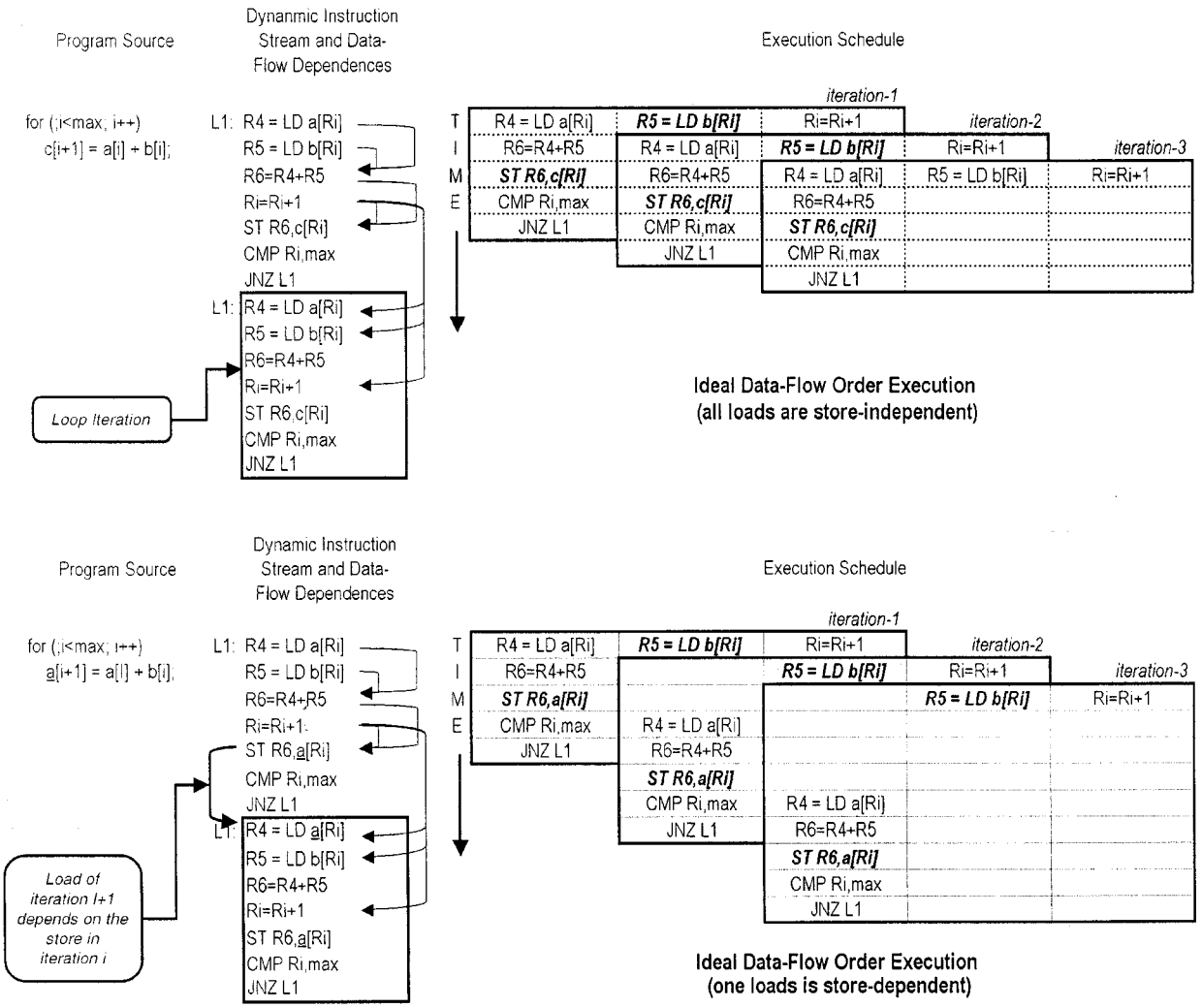

Fig. 1. Ideal data-flow order scheduling.

ulation—proceeding with guessed information, checking the guess when the actual information becomes available, and *recovering* (discarding the work) in case of a *misspeculation* (a wrong guess).

To maintain the appearance that instructions execute in program order, dynamically scheduled processors employ a characteristic *pipeline*. Instructions are pulled into the processor in program order via an *in-order front end*, which *sequences* instructions—fetches instructions from memory, decodes them, and uses program order to extract information about their dependences. Instructions proceed to the *out-of-order core*, where this dependence information enables them to execute in data-flow order—as soon as their inputs dependences are satisfied. Completed instructions are *retired*—their effects are made externally visible—in program order. Performance is measured by instruction retirement throughput. Dynamic scheduling increases retirement throughput by allowing instructions to execute as early as possible and maximizing the number of instructions that execute concurrently. The size of a processor's *instruction window*—the number of instructions that can simultaneously reside within the out-of-order core—determines the degree to which it can reorder instructions and, consequently, how close it can come to the performance bound dictated by global data-flow order.

Like all forms of scheduling, dynamic processor scheduling is most challenging in the presence of *long latency operations* and *incomplete knowledge of dependences and latencies*. Long execution latency increases an instruction's residence in the processor and blocks the flow of operations via a combination of in-order retirement and finite resources. Imperfect information about dependences and latencies leads to suboptimal scheduling in which instructions either wait unnecessarily to execute or execute too early, necessitating some recovery procedure. In practice, the problems of long latency and imperfect information are due primarily to *loads*—instructions that move values from memory to registers. Although memory is typically implemented as a transparent hierarchy that minimizes average access latency, loads still have the longest worst-case—and usually the longest average-case, too—execution latencies of all common classes of operations. As the previous sentence implies, load latencies are also not uniform, a fact that complicates the prompt execution of instructions that consume their values. To make matters worse, the input dependences of loads on *stores*—the complementary

operations of loads, which move data from registers to memory—are not apparent early enough to allow loads to execute as early as possible.

This paper surveys processor techniques that supplement dynamic scheduling by targeting the various problems associated with loads and load-dependent instructions. Our goal is to describe a set of problems, quantify the potential for improvement in each area and give high-level overviews of recent innovations. We do not wish to evaluate the innovations themselves, nor could we uniformly do so even if that were our intent.

The techniques we present fall under four major categories. *Memory disambiguation* techniques [1]–[3] overcome the dependence uncertainties of loads on previous in-flight stores to accurately determine the earliest time at which a given load may safely and correctly execute. *Forwarding and bypassing* techniques [4]–[7] exploit the dependence of loads on recent stores to create a streamlined value forwarding process that often bypasses conventional memory structures altogether. *Load latency reduction and tolerance* techniques [8]–[14] reduce the observed execution latency of loads in various ways. As a complement to memory disambiguation, *load latency speculation* techniques [3] overcome the uncertain latencies of loads to accurately determine the earliest time at which load-dependent operations can execute.

II. Why Loads Are Different

Most instructions create new values or alter the flow of instructions in a program. Loads move unmodified values from memory to registers—i.e., from one form of storage to another. To understand why such conceptually simple operations are so important to performance, we examine the roles loads play in program execution and how their handling differs from that of other instructions.

Operations within a program communicate using named storage spaces. Processors provide two basic name spaces, *registers* and *memory*. Most modern processors allow control and computation operations to access only registers and provide special instructions—*loads* and *stores*—to move values between registers and memory. Even machines that externally allow direct computation on memory elements, internally implement this division. This organization acknowledges that loads and stores must be handled differently from other instructions due to their disproportionately large impact on performance.

The special characteristics of loads and stores derive from the basic differences between registers and memory. The register space is small and register names are hard-coded directly into instructions. Memory is large and typically accessed indirectly, using memory names—commonly called *addresses*—that are computed into registers. There are many inter-related reasons for these differences and as many inter-related consequences.

The small number of registers means that a small, fast implementation of the entire register space is possible. In contrast, a fast implementation of all of memory is impractical to build. Modern computers implement memory as a transparent hierarchy of structures that provides fast, high-bandwidth access to a small dynamic subset of memory with progressively higher-latency, lower-bandwidth access to larger subsets. A *memory hierarchy* exploits spatial and temporal memory access locality to shorten the execution latency of most loads. However, those loads that are not satisfied by the higher, short-latency levels of the hierarchy can have extremely *long execution latencies*, which stall instruction retirement throughput for long periods and significantly degrade performance. The presence of a hierarchy has another subtle effect: it introduces *load latency variation*, which makes scheduling load-dependent instructions difficult.

To sidestep the problems associated with memory access latency, compilers try (very hard) to map as many program locations to registers as possible, a process called *register allocation*. Although register allocation is very efficient, many program locations are invariably mapped to memory. Some program locations are mapped to memory due to the limited number of registers, or due to conventions that define the interfaces between different units of compilation. However, many program locations *must* be mapped to memory. Programming languages provide idioms for *structured data aggregates*—like arrays and dynamic pointer-based structures—that allow the programmer to access multiple locations using the same group of statements. Program locations that correspond to these structures must be mapped to memory not only because the number of elements in these structures is often larger than the number of registers, but because memory's *indirect access mechanism* is required to implement the functionality of a single instruction accessing multiple elements. Memory's indirect addressing enables the use of structured data and the expression of general algorithms, but leads to *memory ambiguity*—the inability to decide *a priori* whether memory references whose addresses are computed into different registers point to the same memory location or not—and complicates the prompt data-flow execution of loads. Executing a load as early as possible requires determining which stores that load depends on and monitoring the completion time of those stores. Due to indirect access, determining whether a load depends on a previous store requires all previous store addresses to be computed.

To summarize, loads present modern dynamic schedulers with several challenges. Long latency loads degrade performance by stalling retirement, varying load latencies complicate the scheduling of load-dependent instructions, and load dependences are difficult to track. The next sections elaborate on these problems and present techniques to alleviate them.

III. Advancing Loads Prior to Independent Stores

The idea behind dynamic scheduling is to execute instructions as early as possible while still enforcing data-dependences. Enforcing a data-dependence between a pair of instructions is only an issue if those instructions are simultaneously in-flight. Only in this case is it possible to erroneously

execute a value consumer instruction before the value producer instruction has finished executing. In order to execute an instruction as early as possible, a dynamic scheduler must know which older in-flight instructions that instruction depends on and precisely track the completion status of those instructions.

Tracking instruction completion status is relatively straightforward. For instructions with only register inputs—all instructions except for loads—precisely determining which older in-flight instructions they depend on is simple as well. Since register names are hard-coded into instructions and instructions are sequenced in-order, register dependences can be computed precisely for every instruction before it enters the out-of-order core. In contrast, the dependence of a load on an older store—the fact that the store writes to a memory address the load later reads—can often not be established that early.

The dependence of a load on a previous store only becomes important at *schedule time*, when the load's register source, its address, is computed and a functional unit is available to process the load. Prior to that point, the load cannot be executed anyway. At schedule time, a *perfect scheduler* would *advance* the load past all older stores it does not depend on. If the load does not depend on any in-flight store, it is issued to the memory system. If it depends on an in-flight store and the store's data is available, the data is *forwarded* to the load. If it depends on an in-flight store and the store's data is not available, the load is made to *wait* until the store's data becomes available.

Absent a perfect scheduler, store–load dependences—and independences—are established by comparing store and load addresses. Differing addresses suffice to establish store–load independence. However, matching addresses are insufficient to establish store–load dependence. Since a load's dependence on a store may be *masked* by an intermediate store with the same address, the addresses of all intermediate stores must also be known to establish store–load dependence. Since addresses are computed and out-of-order execution core does not necessarily compute addresses in program order, the situation in which the processor must schedule a load without knowledge of all older store addresses is quite common. We say that a load is *conflicting* if at schedule time there is an older store with an unknown address. A load is *colliding* if it depends on one of the stores with which it conflicts.

Fig. 2 shows the distribution of load status at schedule time for a machine with a 32-instruction window for various benchmark suites. Generally, about 65% of loads conflict with prior stores, but only about 10% collide. As the size of the instruction window grows, the average number of in-flight stores older than a given load increases and the fractions of conflicting and colliding loads grow with it.

Collectively, the decisions of when to execute conflicting loads affect performance significantly. Stalling a conflicting load while waiting for all store addresses to compute only to discover that the load depends on none of them results in unnecessary delay. This is a *lost opportunity* for performance gain. On the other hand, allowing the load to

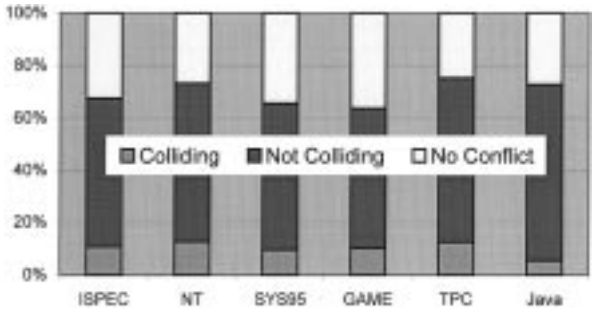

Fig. 2. Load status at schedule time (as percent of all loads).

execute before the addresses of all older stores are known only to discover that a dependence was ignored requires a costly *recovery* procedure. The actual cost of recovery varies greatly depending on its precise implementation. Minimal recovery—often called *selective recovery, selective squashing*, or *selective re-execution*—requires the re-execution of the load and all of its dependent instructions. Selective re-execution is difficult to implement, requiring execution and dependence information to be maintained past execution itself. Intel's Pentium 4 processor [15] is the only production processor that implements it. Selective re-execution can only be used when the dependences themselves are known to be safe. The simpler, safer, but much more costly recovery solution is *squashing*—in which the load and *all* subsequent instructions are aborted, refetched, and re-executed. Squashing requires no execution or dependence information to be maintained after execution and correctly reconstructs misspeculated dependences. It is implemented by all other speculative processors.

Fig. 3—based on numbers from [2]—shows (the bars on the right) the performance potential of a perfect scheduler relative to a processor that allows loads to execute only after all older in-flight stores have executed. Different configurations show somewhat different numbers, but still exhibit the same trends, as presented in [3]. This section presents several techniques that try to approach the performance of perfect memory scheduling by executing loads as early as possible while minimizing recovery cost. Collectively, these techniques are called *load speculation*. Load speculation techniques are driven by *speculative memory disambiguation*, the activity of trying to determine store–load dependences without complete information about their addresses.

A. No Speculation

Conservative scheduling is a simple approach that eliminates misspeculation entirely. In conservative scheduling, loads are executed in-order with respect to older stores. Most actual implementations of conservative scheduling do not require older stores to execute completely prior to executing a given load. Since only addresses are required to determine store–load dependence, load execution may proceed once all previous store addresses are known even if the data of some stores is unavailable. This enhancement requires the ability to detain a load if it is found to depend on a store

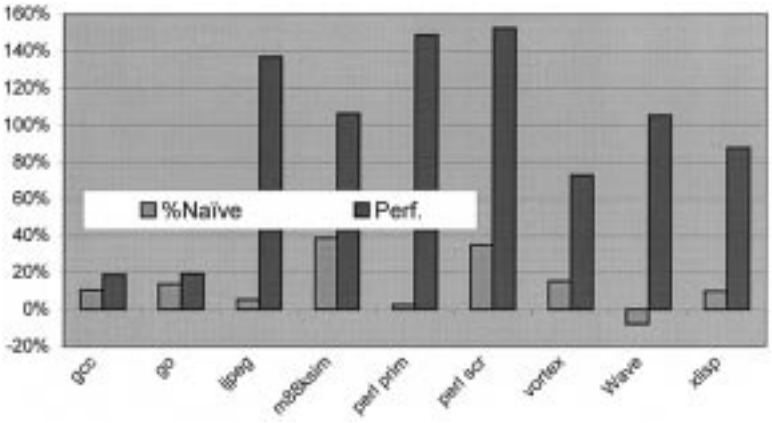

Fig. 3. Speedup of naïve speculation and perfect disambiguation over conservative scheduling.

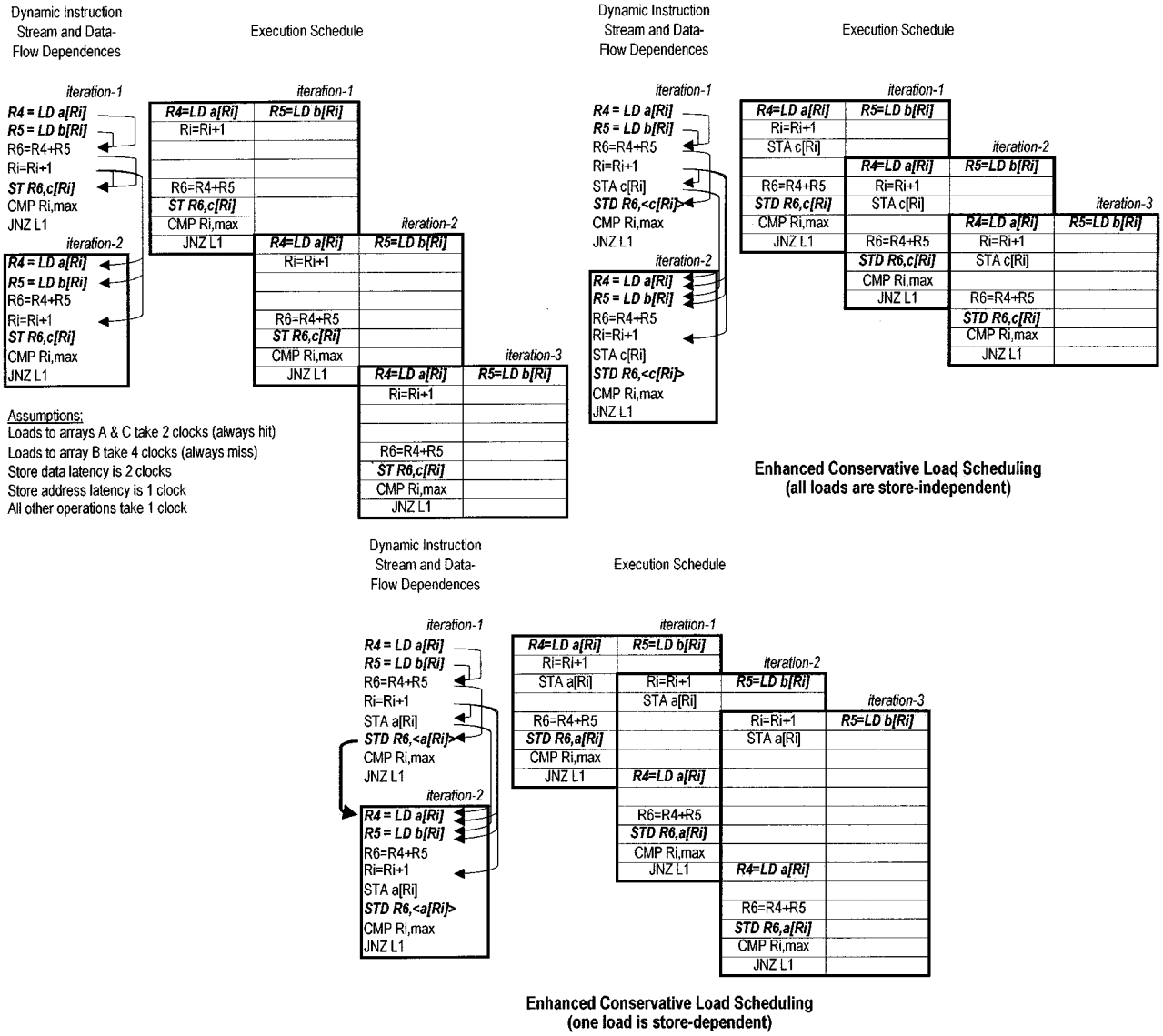

Fig. 4. Conservative load scheduling.

whose data is unavailable. Intel's P6 microarchitecture implements this kind of conservative scheduling by internally decoupling stores into separate *store-address* and *store-data* operations. The differences between basic and enhanced conservative scheduling are shown in Fig. 4. The diagram on the top left-hand side shows a basic conservative scheduler which forces loads to wait for all prior stores to be executed. The diagrams on the top right-hand side and on the bottom

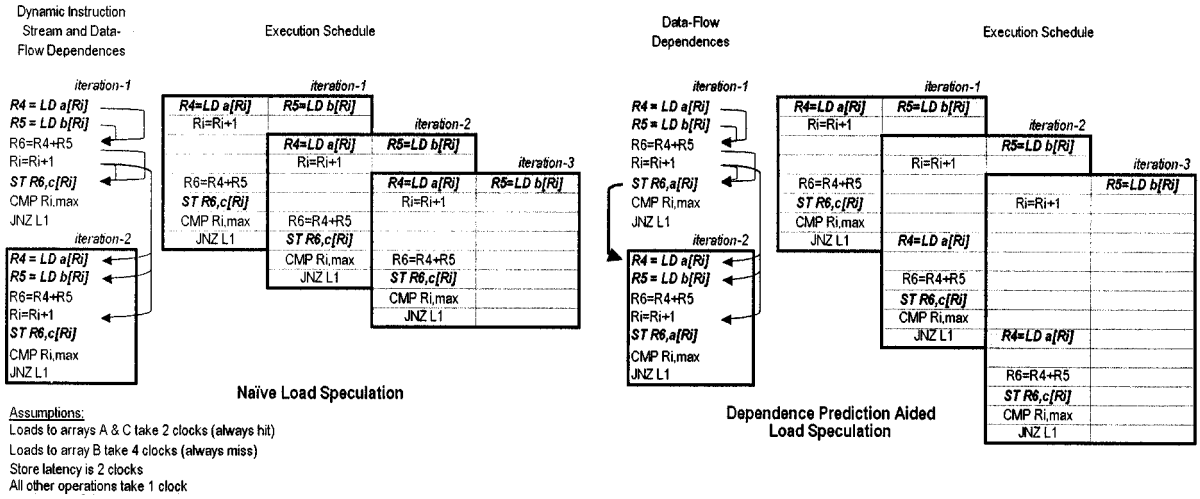

Fig. 5. Naïve load speculation (left) and dependence-prediction aided load-speculation (right).

show an enhanced scheduler that requires loads to wait only for store addresses.

Conservative scheduling of the decoupled store-address variety serves as the baseline performance measure in Fig. 3. Conservative scheduling does not approach the performance of a perfect memory scheduler, sometimes providing only half the performance of the latter.

B. Naïve Load Speculation

Conservative scheduling is simple to implement since it does not necessitate the implementation of a recovery procedure. However, it incurs a high performance cost in terms of lost opportunity for optimization. As the load status distribution in Fig. 2 shows, most conflicting loads do not ultimately collide. Exploiting this observation, *naïve speculation* [16]—also called *blind speculation* or *opportunistic scheduling*—adopts the opposite strategy of conservative scheduling and executes conflicting loads immediately with the assumption that they do not collide. In contrast with conservative scheduling, naïve speculation incurs no cost in terms of lost opportunities for load advancement. However, it does incur the direct performance cost of misspeculation recovery and the static cost of an implementation of a recovery mechanism. The left side of Fig. 5 demonstrates naïve speculation—the potential gain due to aggressive, successful, load advancing is easily seen.

Because most conflicting loads do not collide, naïve speculation largely outperforms conservative scheduling, as shown in Fig. 3. However, its performance often falls short—sometimes by a lot—of the upper performance bound of a perfect scheduler. The reason is that, although the number of colliding loads is relatively small, the penalty of recovery is quite significant. In fact, in programs with a sufficient number of collisions—like *wave*—a naïve approach will trigger recovery too frequently and perform worse than conservative scheduling. As instruction windows increase, the number of collisions increases and the number of recovery procedures triggered may become unacceptably high, drowning the performance gain of advancing loads altogether. These trends suggest a hybrid approach that advances loads that are not likely to collide, and employs conservative scheduling for ones that are.

C. Dependence-Prediction Based Load Speculation

Determining which loads are likely to collide and which ones are not is not as difficult as it first seems. Programs are repetitive and processors are deterministic. Programs repeatedly execute the same code sequences. Asked to schedule the same code sequence many times, a processor will schedule it the same way, or very nearly the same way, every time. Repetition leads to predictability. A load that collides once is likely to collide again; in fact, it is likely to collide with the same store again.

Predicting collisions or store–load dependences directly is preferred over an indirect alternative—resolving or predicting addresses early and using these addresses to speculatively determine dependence. Early address resolution and prediction techniques are useful in certain situations, as we will see in subsequent sections. However, establishing dependences accurately requires correct values for *all* addresses, something these techniques can rarely provide. Furthermore, the dependence—or independence—between multiple instances of a given static store–load pair is typically more stable than the stream of addresses referenced by those instances. The right side of Fig. 5 shows dependence-prediction aided load scheduling. The independent load is advanced, while the dependent one waits for its store to complete. If all loads in the figure were independent, the resulting scheduled sequence would be identical to the one produced by the naïve speculation scheduler, shown on the left.

The differences between store–load dependence predictors lie in the precise relationships they attempt to capture. The simplest predictors predict whether a given load is likely to produce a collision. A load that is predicted to collide is scheduled conservatively—the addresses of all older stores must be known before the load is allowed to execute. Instances of such *binary decision predictors* are the Compaq

Alpha 21264's [17] *wait table* and the *inclusive collision history table (CHT)* [3].

More sophisticated predictors attempt to predict a range of stores past which a load can advance. Their predictions do not come in the form of a binary decision, but rather as the number of immediately older stores with which the load is predicted not to collide—called the *collision distance*. A load whose collision distance is smaller than the number of older in-flight stores is predicted to collide. The load must wait for the addresses of all stores that are at least as far as its collision distance before it can execute. It does not need to wait for the addresses of closer stores. The *exclusive CHT* [3] is this kind of predictor.

One problem with distance predictors is that a load may be arrived at via different control-paths and its distance to the colliding store may vary across paths. To minimize misspeculation, collision-distance predictors typically predict the minimum collision distance along any path—preferring to miss the opportunity to advance a load past one or two stores on one path to triggering a recovery procedure on another. The inclusion of path information may improve collision-distance prediction by allowing different distances to be used for different actual control flows.

Other predictors attempt to predict the identity of the colliding store precisely. By definition, locating a colliding store is more difficult than locating a range of independent stores. While a correct prediction of the colliding store defines the independence range, the opposite is not true. *Synchronization-based dependence predictors* remember not only that a collision occurred, but also which store caused it. Future collision predictions are obtained by first reading the identity of one or more stores that have triggered collisions in the past, then checking for the presence of older unresolved instances of those stores in the window. *Speculation/synchronization* [1] and *store sets* [2], [18] are examples of synchronization-based dependence predictors. Speculation/synchronization remembers the identity of the most recent colliding store for each load and predicts a collision if an older, unresolved instance of that store is found in the window. The store sets technique remembers a set of stores that have caused collisions and synchronizes the load with the youngest unresolved instance of any store in the set.

All of the dependence predictors above predict the identities of colliding loads. The *store barrier cache* [19] implements the complementary approach—it predicts the identities of stores that cause collisions and delays all younger loads until instances of those stores complete.

D. Accounting for Silent Stores

A *silent store* [20]–[22] is a store that writes a memory location with the same value that already existed there. Essentially, silent stores are redundant and can be eliminated or ignored. Recent studies have shown that as many as 25% to 30% of all stores in a program are silent. The fact that so many stores are redundant is surprising, but understandable given the repetitive nature of programs and the compiler's view that storing a value to memory is cheaper than first reading the old value from memory and then storing the new value only if it differs from the old one.

Early studies explored eliminating silent stores to reduce cache and bus traffic. Recent work [23] addressed the effect of silent stores on load scheduling, the central observation being that a load can be advanced prior to an unresolved colliding store if that store is silent. The prediction of whether a given store is likely to be silent can be made independently and combined with conventional dependence prediction to arrive at a speculation decision. However, silence and dependence prediction can be combined quite naturally.

The goal is to treat dependence on a silent store as no dependence at all. Since dependence predictors learn by observing past load misspeculations, a simple way to accomplish this goal is to not trigger misspeculations on collisions with silent stores. Note, this is what we would like to do anyway. There is no point to signaling a misspeculation and triggering a recovery on an ignored collision with a silent store—re-executing the load will not change its outcome. To implement the desired policy, we modify the collision detection mechanism to compare the data values of stores and loads in addition to their addresses and signal a collision only on an address match and a value mismatch.

Surprisingly, advancing a load past a silent store is not a clear win. Although it executes earlier, the load may need to access the memory hierarchy, losing the opportunity to use faster store-to-load forwarding and consuming cache bandwidth that could potentially be used to service another load. In addition, silence prediction introduces the possibility of silence misprediction and may result in additional recovery instances and lost performance.

IV. Streamlining Memory Communication

Memory is a communication medium. Programs use memory to pass values from one operation to another with stores and loads as the communication brokers, establishing the communication paths via address calculation but not creating any new values themselves. *Memory communication* is used when not enough registers are available or when the precise communication pattern itself cannot be unambiguously determined by the compiler. Memory communication is slower than register communication because access to even the highest level of the memory hierarchy takes longer than register file access, but more so because the store and load must each compute an address and because all intermediate stores must also compute an address before communication can be established. The previous section described techniques that tried to identify and advance store-independent loads. This section describes similar techniques that identify *store-dependent loads* and attempt to streamline the memory communication, the passing of the value from producing operations to stores via memory to loads and finally to consuming operations.

The bulk of the delay of memory communication is incurred in establishing the communication path itself. To overcome this delay, we again exploit the fact that store–load communication patterns are repetitive and predictable, this

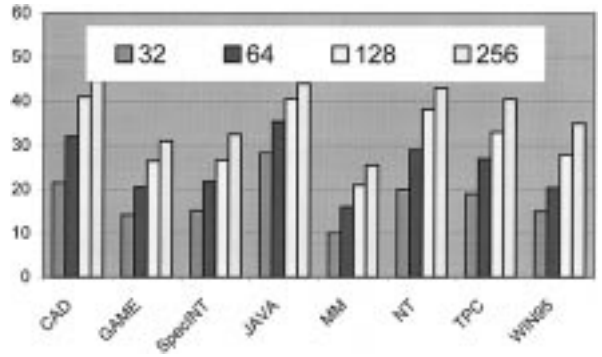

Fig. 6. Number of true memory-dependences as a fraction of all loads for varying communication distance.

time to *predict communication paths* and perform the communication speculatively and via a *streamlined path* that is faster than a round trip to the data cache. *Speculative streamlined memory communication* is analogous to dynamically and speculatively transforming memory communication, which is indirect, ambiguous, and slow, to register communication, which is direct, unambiguous, and fast. This transformation can take place because, ultimately, only the communication itself matters, not the actual way in which it is performed.

Modern microprocessors employ *memory forwarding* to permit memory communication to take place before the participating store has retired and its data has been written to the memory system. Store address/data pairs are buffered on the processor in a structure commonly called the *memory-ordering buffer*. Once communication between a load and a buffered store is established, the store value is forwarded to the load directly without accessing the memory hierarchy. Memory forwarding enhances performance by allowing store-dependent loads and their dependent instructions to execute before the corresponding store retires, and by reducing the demand for cache bandwidth. Beyond these benefits, however, it is also faster than cache access since the memory ordering buffer is typically much smaller than the data cache. Motivated by this speed advantage, some of the techniques presented in this section have natural extensions that operate on retired stores in an attempt to convert as many cache accesses as possible to memory forwarding instances.

The degree to which speculative streamlined memory communication can short-circuit the memory hierarchy—or whether it can convert a cache access to forwarding—depends on the dynamic distance between the store and the load. Certain kinds of speculative short-cuts can only take place when both communicating parties are simultaneously in-flight. Similarly, the conversion of cache access to forwarding requires that the store be recent enough so that its entry in the forwarding structure has not been evicted. Fig. 6, taken from [6], shows the distribution of loads that communicate with stores as a function of the communication distance—in dynamic instructions—for various benchmark suites compiled for the Intel IA32 architecture. Other studies [16], compiling programs for different architectures and measuring distances in stores rather than instructions, show comparable results. Notice, this figure differs from Fig. 2. Here, we do not care about the state of the load at schedule time, only that a communication path exists and that the store and the load are close enough dynamically to allow reasonable sized structures to perform the optimization.

A. Identifying Memory Communication

If streamlined speculative memory communication is the equivalent of converting memory communication to register communication, then *memory communication identification* is the equivalent of the *static register allocation performed by the compiler*. The goal of memory communication identification is to allow the load and store to recognize one another as the communicating parties and to agree on a name they will both use to refer to the communication.

One method for identifying memory communication is *memory communication prediction*, which exploits the stability of program structure to flag store–load pairs that have communicated in the past as likely communicating parties. Memory communication predictors are similar to collision predictors that predict which store caused the collision, both use a load's identity as an index to determine the identity of the communicating store. Note, a collision is nothing else than a misordered communication. However, memory communication predictors have some added complexity. First, many forms of speculative streamlined communication require *some action on the part of the store* as well, potentially before the load arrives. This, in turn, requires a predictor that, at the very least, can use a store's identity to determine whether the store is part of some communication. Second, speculative streamlining can be proactively applied to *all communicating store–load pairs*, not only ones that induce misspeculations. Learning communication patterns by tracking collisions is not sufficient, especially to support techniques that can operate outside the instruction window and optimize communication with retired stores.

Memory communication predictors learn communication patterns by tracking all memory communications that occur within a given distance. A communication predictor typically remembers the addresses and identities of some number of recent stores. The identities of communicating pairs are discovered by comparing the addresses of loads to buffered store addresses and noting the identities of the two parties on a match. Communication predictors are driven by *streams of retired stores and loads*, both to eliminate ordering ambiguities and to avoid learning communication patterns along misspeculated paths.

The representation of communication information resembles that of collision information—with the load's identity as the primary index. Stores can determine their part in a communication via a separate, store identity-based index into the same structure or via associative search on the store-identity data.

Register tracking [7] is another technique that can identify instances of memory communication. As opposed to memory dependence predictors, which remember previous

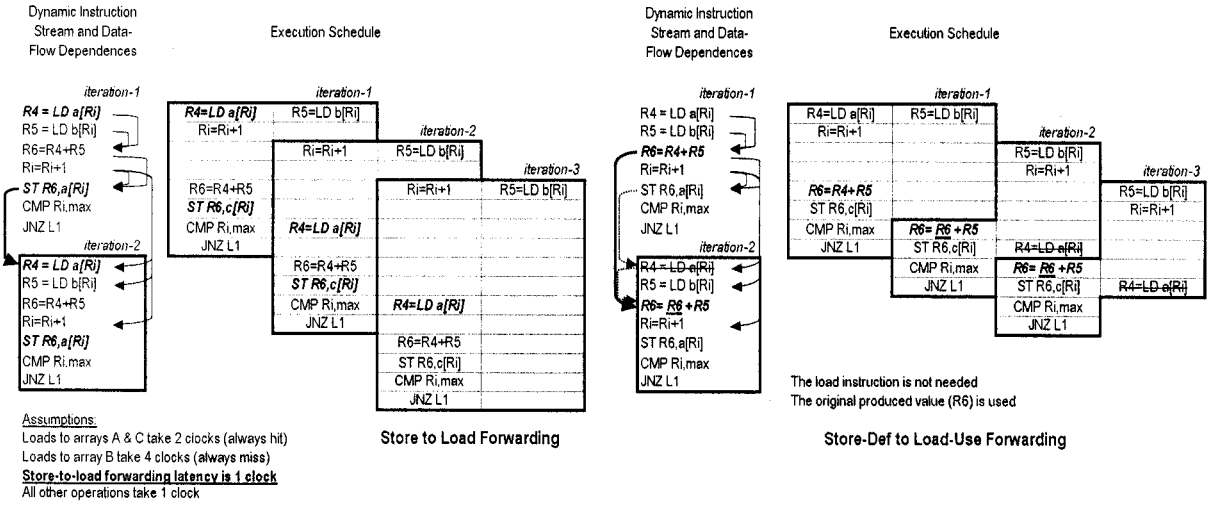

Fig. 7. Store-to-load forwarding (left) and store-def to load-use forwarding (right).

communicating pairs, register tracking establishes store–load dependences by determining the equality of pairs of addresses without knowing the addresses themselves.

Register tracking tracks register values through instructions that add constants to registers. Such updates can be *accumulated* and represented as a single number—an offset relative to a register value. Addresses that are produced from a known value using exclusively a series of such updates are *trackable*. Two addresses trackable from the same register can be tested for equality—and potential store–load communication—by comparing their offsets only. The test can be performed even when the base register value is unknown.

Register tracking is especially useful in identifying stack-related store–load communication that takes place during procedure parameter passing and the saving and restoring of preserved registers. Loads and stores that perform these functions are based off the stack pointer register, a register that is modified almost exclusively using constant additive updates. Register tracking can detect close to 100% of all stack related store–load dependences enabling usage of various forwarding techniques. Not surprisingly, register tracking has a greater impact in stack-based architectures—like Intel's IA32—in which stack manipulation is frequent.

B. Speculative Store to Load Forwarding

The streamlined store-to-load forwarding process is composed of two steps. First, communication prediction is used to assign a shared name to the communicating parties. Second, each *dynamic instance of the communicating pair* is allocated a physical location that will *synchronize* the store and load—i.e., hold the store value until the load arrives or alternatively stall the load until the arrival of the store. Note, there are two namespaces here—the *communication namespace*, which connects static store–load pairs and is not associated with any physical data transfer, and a *synchronization namespace*, which is associated with physical storage and is allocated and deallocated to implement communication between dynamic store–load pairs. This organization enables a small number of physical locations to service a large number of known static communication pairs and allows multiple instances of a single static communication to proceed in parallel.

Stores check the communication predictor when they enter the instruction window. If a store is part of a communicating pair, it is allocated a *synchronization register* from the synchronization namespace; the store will write its data into this register when the data becomes available. The identity of the synchronization register is noted in the appropriate communication predictor entry. Loads also check the communication predictor when they enter the instruction window. If the load is part of a communicating pair to which a synchronization register has been allocated, the load reads that register, potentially waiting for the store to write it. The synchronization register is freed once the load receives the value. Streamlined forwarding naturally supports communication between retired stores and incoming loads—the synchronization register persists until the load has retrieved the value and a pointer to it is maintained in the communication predictor, allowing it to be located after the corresponding store has retired. From the store's point of view, streamlined communication is a supplementary activity. The store still writes its address and data to the memory-ordering buffer, to handle load communication that is not captured by the predictor. The left side of Fig. 7 demonstrates store-to-load forwarding. The reduced communication latency between the store and the dependent load results in a shorter schedule.

Speculative store-to-load forwarding is, obviously, speculative and must be verified. The straightforward, but costly, verification procedure is to actually execute the load and compare the speculated and actual results. A cheaper solution is to verify only the communication path itself, i.e., that the store and load addresses match and that no intermediate store accesses the same address. The latter method eliminates memory hierarchy accesses but is not easily extended to cover forwarding outside of the instruction window, where identifying colliding intermediate stores or memory modifications due to external snoops is complex.

Speculative memory cloaking [4] and *memory renaming* [5] are similar mechanisms that perform speculative store-to-load forwarding. Both support forwarding from retired stores. According to [4], using a simple predictor that can learn communications that do not span more than 256 stores, speculative memory cloaking can successfully forward about 45% of all dynamic loads on integer code and 15% of all dynamic loads on floating-point code, with a communication misspeculation rate of under 4%.

C. Speculative Load to Load Forwarding

The concept of store-to-load forwarding can be extended to cover *load-to-load forwarding*. Load-to-load forwarding is not memory communication *per se*—the participating loads do not truly depend on one another. However, by treating the loads as dependent, the younger load can enjoy the same performance benefits as the load in store-to-load forwarding. Load-to-load forwarding exploits the following scenario: two different static loads accessing the same memory location with no intermediate store to that address. This is the same scenario exploited by store-to-load forwarding with the older load taking the place of the store. There are several cases in which load-load forwarding is useful—for example, repeated access of a read-only global variable.

The mechanics of load-to-load forwarding are quite similar to store-to-load forwarding. Potential load-to-load communication pairs are identified via address comparisons. The older load computes an address and obtains its data via a conventional route, but also deposits the data into a synchronization register allocated for the communication. The younger load locates the synchronization register using the communication predictor and speculatively reads its value without computing an address or waiting for older store addresses to resolve.

Load-to-load forwarding and store-to-load forwarding can be implemented together in the same mechanism. The addition of load-to-load forwarding increases the fraction of forwarded loads from 45% to 65% on integer code and from 15% to 45% on floating-point code [24].

D. Speculative Store-Def to Load-Use Forwarding

Streamlined memory forwarding effectively converts a communicating store–load pair into two register move operations—the store moves its data input into the synchronization register and the load moves the value from the synchronization register to its output and forwards it to dependent operations. Register moves are cheaper and faster than stores and loads, but are themselves redundant operations that do not create new values, simply move values around. The store-to-load communication path can be short-circuited even further by eliminating the register moves and routing the store's data input directly to the load-dependent instructions. The instruction that produces the store's data input is called a *store-def*. The instruction that consumes a load's output is called a *load-use*. This kind of forwarding is called *store-def to load-use forwarding* or *store–load bypassing*, since the store and load are bypassed altogether.

Store-def to load-use forwarding eliminates the use of the synchronization register as an intermediate. Doing so actually shortens the program's *critical path*, converting a sequence of four dependent operations, which in the best case could execute in four consecutive cycles, to a sequence with only two dependent operations, which can execute in as little as two cycles. However, removing the intermediary implies that this kind of forwarding can only be performed if the register holding the store-def value is still valid by the time the load-use enters the instruction window. The right side of Fig. 7 demonstrates store-def to load-use forwarding, and shows that bypassing both the store and the load results in a significant reduction in the length of the data-flow path.

Two proposed techniques implement speculative store-def to load-use forwarding: *speculative memory bypassing* [4] and *unified renaming* [6]. Speculative memory bypassing is a dynamic extension to speculative memory cloaking. Speculative memory bypassing uses register dependence logic to capture the physical register holding the store data for each dynamically cloaked store–load pair. The physical register tag is stored in the communication predictor, and passed directly to uses of the cloaked load as they enter the window.

Unified Renaming is a general mechanism that bypasses *identity operations*. Identity operations are instructions that produce outputs that are equal to previously computed values. Identity operations include, but are not limited to, instructions that simply move values from one place to another, like register moves or communicating store–load pairs. Unified renaming uses register renaming to route the input of an identity operation to the input of its dependent instructions directly.

Unified renaming consists of two components. An *identity detector* detects identity operations. The *modified register-renamer* redirects inputs of identity-dependent operations to the inputs of the identity operation itself, essentially reusing the physical register holding the input of the identity operation to hold its output as well. Unified renaming bypasses any operations captured by the identity detector. For example, a simple, decoder-based identity detector allows it to eliminate register move instructions. To implement store-def to load-use forwarding using unified renaming, the identity detector is augmented with a communication predictor that identifies the participating loads and stores as identity operations. Load-to-load forwarding is implemented in a similar way.

Unified renaming can be performed safely, using a decoder-based detector to identify register moves, or speculatively, predicting communicating load–store pairs. The procedure for detecting unified renaming misspeculations and recovering from them is similar to the one used to recover from store-to-load forwarding misspeculations. As in store-to-load forwarding, bypassing a load–store pair does not imply that the store is relieved from execution, address calculation or writing its data and address to the memory ordering buffer. These tasks must be performed in addition to the bypass to allow conventional, address-based

memory forwarding for loads not captured by the predictor. The presence of store information in the memory ordering buffer also allows eventual address mismatches and the presence of intervening stores to be detected.

V. Reducing and Tolerating Load Execution Latency

Operations with long execution latencies are often on the program's *critical path*—they cause serious performance problems. Long latency operations delay the processing of all subsequent instructions because until they complete, completed younger instructions cannot retire and free processor resources for future instructions.

Loads are the most frequent of long latency operations. The execution of most instructions takes one cycle. Simple arithmetic and logical instructions, stores and control instructions are all single-cycle operations. In the best case—a first-level *cache hit*—load latency is on the order of two to three cycles. Memory hierarchies attack the *average load execution latency* by making a first-level cache hit the common case. *Cache misses* elicit a load's worse- and worst-case behavior, depending on which memory hierarchy component ultimately services the request. A *primary miss* is one that is serviced by the second-level cache. Primary miss latency is around ten cycles on most modern processors. A *secondary miss* is satisfied by memory. Secondary miss latency is already more than one hundred cycles in current processors and is growing (relatively) with each successive processor generation.

Maintaining high performance in the presence of a long latency load requires overlapping the execution of that load with *older* independent instructions. In this section, we discuss four aggressive techniques for reducing and tolerating load latency. The scope of these techniques, and their complexity, depends on the load latency itself. The relatively short latencies of cache hits and primary misses means that enough older instructions to fully, or at least mostly, cover this latency often co-exist with the load within the instruction window. Cache hit and primary miss latency can therefore be hidden by mechanisms that are more intelligent than, but not fundamentally different from, vanilla dynamic scheduling. However, the overlapping capability of dynamic scheduling is limited by the size of the instruction window. Secondary miss latency must be overlapped with many more instructions if it is to be fully tolerated. Techniques that attempt to tolerate secondary miss latency must artificially break the constraints of the instruction window.

A. Critical Path Scheduling

A program's critical path comprises those instructions that most directly limit its performance in a given cycle. As performance is measured by retirement throughput, long latency instructions—which stall retirement until their own completion—often lie along the critical path. Instructions on which long latency instructions depend typically lie along the critical path as well, since an acceleration of their execution directly reduces the time the long latency instruction spends stalled as the oldest instruction in the machine. Criticality is propagated backward through data (and other) dependences—instructions on which critical instructions depend are themselves critical, and so on, transitively. Criticality is not propagated across dependence edges that span more instructions than the size of the instruction window. In such cases, the performance of the younger instruction cannot possibly be affected by that of the older instruction—it would be ready to execute as soon as it entered the window no matter what policies were used to process the older instruction.

From a scheduling standpoint, instructions on the critical path should be scheduled with the highest priority. The instructions that share the instruction window with the critical-path instructions are by definition less performance visible than the critical path itself. Decreasing the scheduling priority and performance of these instructions to benefit the critical path is unlikely to expose their latency and degrade overall performance.

Since loads (especially those that miss the cache) are the most frequent of long latency instructions, they and their computations often lie along the critical path. A scheduling policy that prioritizes critical-path instructions expedites the scheduling of these loads. A critical path scheduling policy effectively maximizes the number of independent instructions that can be naturally overlapped with a primary cache miss by the instruction window. By prioritizing a load and its computation, this policy guarantees that the load is scheduled as early as possible after it enters the instruction window. In doing so, it maximizes the number of older instructions that can be overlapped with the cache miss latency. As the load gradually becomes the oldest instruction in the machine, its execution can continue to be overlapped with an additional window's worth of younger instructions. Fig. 8 demonstrates how simple prioritization of an instruction needed to compute load addresses results in a shorter critical path and a better overall schedule.

Identifying the exact critical path is not simple. However, heuristic approaches [14] have been suggested that can dynamically identify critical instructions and the operations they depend on with high accuracy. Hardware implementations of more rigorous critical-path analyses have also been proposed [25].

B. Address Speculation

When the address computation of a critical load consists of a long chain of dependent instructions, most of the dynamic instruction stream prior to the load becomes critical and critical path scheduling becomes ineffective. One thing that can be done in these situations is to speculatively break the load's input dependence, executing it with an address that is obtained via a mechanism other than execution. Of course, the load and its dependent instructions must be re-executed if the address obtained by eventual execution differs from the speculated one. The left side of Fig. 9 demonstrates how

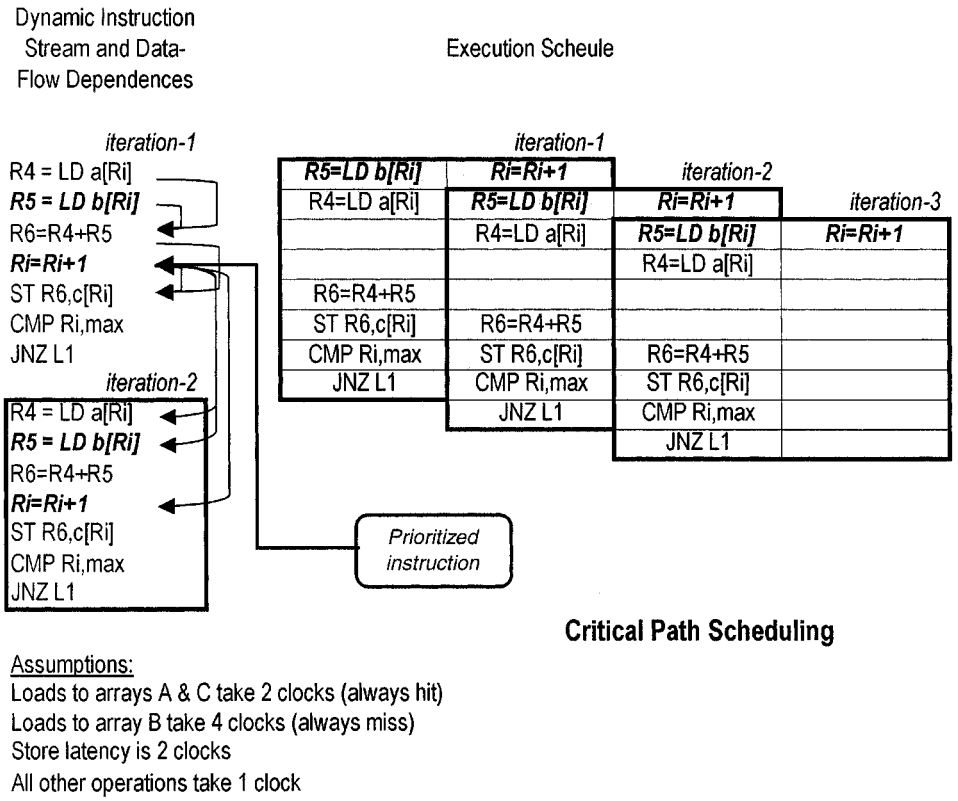

Fig. 8. Critical path scheduling.

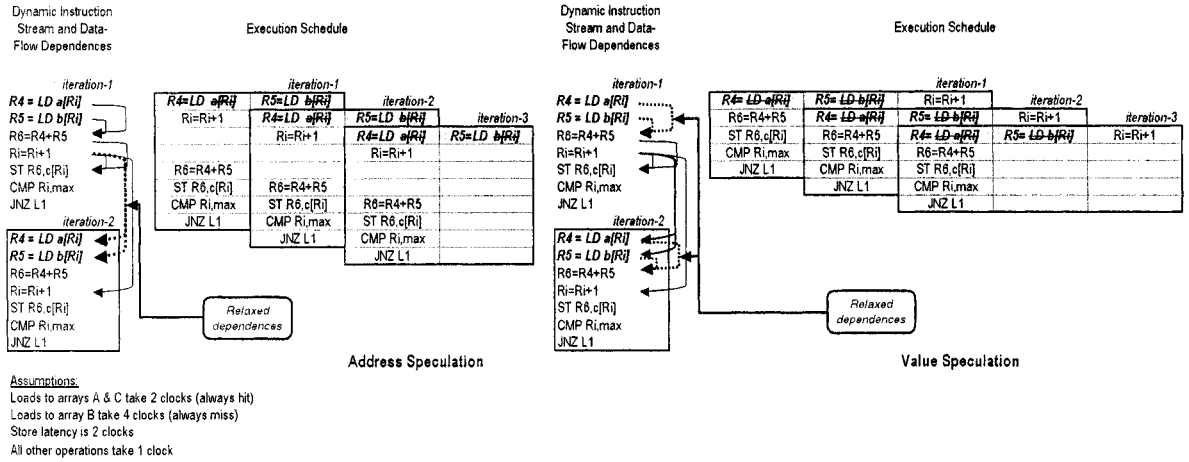

Fig. 9. Address speculation (left) and value speculation (right).

address speculation affects load scheduling by eliminating a load's dependence on its address.

Several possibilities exist for guessing at load addresses. Mechanisms like register tracking (see Section IV-A) naturally track address computations as they propagate through various instructions. These mechanisms often construct internal representations of addresses that compress the effects of chains of instructions into a single operation. Upon the arrival of the input to such a chain, the register tracking mechanism can compute an input address faster and earlier than execution is able to. Register tracking can produce early addresses with high accuracy but only for loads whose chain of address calculations are entirely composed of simple arithmetic operations.

To handle other loads, *address prediction* can be used. Address predictors exploit a program characteristic that execution does not—the repetitious nature of programs. Address predictors store past addresses in tables, which are indexed by some information about execution context such as the load instruction pointer, the program path leading to the load, and previous addresses accessed by that load. They regurgitate those, or somewhat modified, addresses when subsequently queried in similar contexts.

There are several kinds of address predictors [26], [27], which predict different access patterns. *Last value predictors (LVP)* handle the simplest pattern, the same instruction always generating the same address, as in the case of a global variable access. *Arithmetic or stride predictors* handle another simple and common pattern, in which each address differs from the previous by an additive constant. Strided access patterns occur when array elements are accessed sequentially. *Context-based predictors* are the most general and can predict any previously observed address sequence, regardless of the arithmetic relationship between the individual addresses. Context-based predictors simply remember entire address sequences and regurgitate them after seeing a matching sequence prefix. Context-based predictors are especially useful for predicting address sequences in which the addresses are not arithmetically related at all. Such sequences are typically the product of linked-data structure access. Linked data structures are dynamically managed and generally do not adhere to a regular memory layout. More effective and complete address predictors are *hybrid predictors*, which implement all three basic predictors and choose among them on a per load basis. Good hybrid address predictors can predict addresses for over 65% of dynamic loads with prediction errors in the range of 1%.

The scheduling scope of the instruction window is insufficient to tolerate the latencies of secondary misses, which are on the order of one hundred or more cycles. Maintaining a sufficiently high retirement rate in parallel with such a long latency event requires a large number of instructions, much larger than is available in a conventional instruction window. The only way to achieve such a high degree of overlapping is with a mechanism that is completely decoupled from execution. Without the need to bind a value to a register and to wait for the value to arrive to allow the binding to be retired, a decoupled mechanism can initiate cache misses arbitrarily early and theoretically tolerate arbitrary latencies.

Address prediction can also be used to support *hardware prefetching* [28], [29]—a mechanism that attempts to decouple memory latency from execution. Hardware prefetch engines initiate fetching of memory blocks they predict will be accessed in the near future into the cache levels closer to the processor core. Because hardware prefetching does not bind values to registers, the prediction of precise addresses is not necessary, nor must individual predictions be matched with individual loads. The address predictors that drive hardware prefetching can be triggered by either actual data cache misses or by the fetch of likely-to-miss loads and they may predict cache block addresses using either histories of requested blocks or address histories of individual loads.

Address prediction is efficient in that its per-load effort is relatively small, allowing it to handle many loads. Where hardware address prediction works, hardware prefetching is rightfully the tool of choice. However, address prediction fails for some types and access idioms of structured data. Except for sequential access of dense arrays—which is admittedly the most common one—not many data access idioms are arithmetically predictable. Context-based predictors [30] are capable of capturing less regular sequences, but require large tables to handle large data sets—the ones that most benefit from prefetching. *Predictor-directed stream buffers* [31] attempt to reduce this cost while maintaining a reasonable prediction rate by equipping the prefetcher with a combined stride- and context-based predictor.

C. Value Speculation

Where address prediction fails, the complementary approach can be used. Rather than breaking the input dependence of the load by predicting its address, the output dependence of the load is broken by predicting its output value [8], [10]. *Value speculation* is performed as a load enters the instruction window. If a *value prediction* is available, the load-dependent instructions are executed immediately using the predicted value as an input without waiting for the actual load to complete. Of course, the load must be executed to verify the prediction. As with address speculation, in the event of a misspeculation, all dependent instructions are rescheduled and re-executed. Mechanically, value predictors are very similar to address predictors—not a surprise since an address is simply a kind of value. The right side of Fig. 9 shows how value speculation affects load scheduling by making the load consumer independent of the load itself. Good value predictors—which combine last value, arithmetic and context-based components—can deliver correct values for about 75% of the loads in selected applications from the integer SPEC95 benchmark [32].

Value speculation is an alternative to binding address speculation and can be used in conjunction with hardware prefetching. As an alternative, value speculation is generally preferred over address speculation in several respects. Value speculation collapses load latency completely and allows dependent instructions to be scheduled immediately. It also accesses the memory system at most once per load. Address speculation is preferred over value speculation only where addresses are more predictable than their corresponding load values [10], for instance during the sequential access of an array containing unpredictable data, which is quite frequent in floating point and multimedia applications.

D. Pre-execution

Some loads are both likely to incur secondary misses and difficult to address and value predict. We term these *problem loads* because they present problems for most conventional mechanisms found in processors today. Quite simply, address—and ultimately value—generation is impossible for problem loads in any way other than actual execution. Fortunately, this does not mean that early address generation and the effects of prefetching are precluded for these loads. *Pre-execution* is a technique that uses actual execution to generate early addresses for problem loads. Pre-execution generates addresses early by running and keeping ahead of main program execution. It achieves this by identifying the *computations of problem loads*—the instructions that transitively contribute values to the load's input—and *executing only copies of these computations* in parallel with, but decoupled from, the whole program.

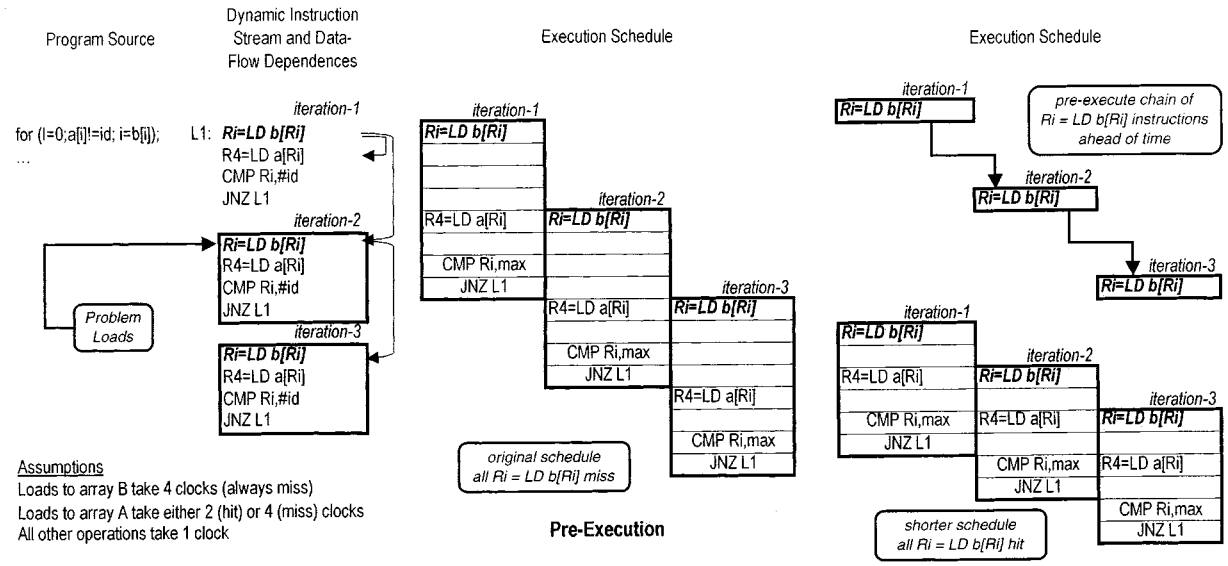

Fig. 10. Pre-execution.

Pre-execution can be viewed as an extension of critical path scheduling. Recall, the limitation of critical path scheduling derived from its restriction to the sequential instruction window. Pre-execution breaks this limitation. Decoupling allows problem load computations to be executed in parallel with main program regions with which they would not normally co-exist in a sequential window. This allows the latencies of problem loads to be overlapped with many main program instructions—a stall in the pre-executing computation does not stall the main program, which keeps retiring and fetching new instructions—and, hence, tolerated more effectively.

Pre-execution is meant as a replacement for neither conventional execution nor prefetching, but as a *supplement* to both. Pre-execution must be implemented in addition to conventional execution because the latter is required to establish program order and produce the intended program output. Because pre-execution is a redundant activity, conventional prefetching is required. Since bandwidth to support the pre-execution of all likely-to-miss loads is simply not available, pre-execution should be used only for problem loads, on which no other known techniques work.

There are several proposals of mechanisms that at the very least achieve the same effect as pre-execution [33], [34]. Two, in particular, pre-execute load computations that are extracted directly from programs: *dependence-based prefetching* [11] and *speculative data-driven multithreading* [13]. The concept behind both is the same. The first step is to identify computations of problem loads that are likely to cause secondary misses and create a processor-internal static representation of these computations. Each static computation is associated with a trigger, a static instruction whose dynamic instances cause copies of the computations to be microarchitecturally forked. The trigger is typically an instruction that supplies the computation with some seed input from the main program. Forked computations execute in parallel with the main program, siphoning bandwidth as needed. Prefetching is achieved naturally as pre-executed loads access memory. Fig. 10 shows how pre-execution can speed up linked list traversal by prefetching future elements ahead of time. The linked list is represented as two arrays, array a holds a value while array b holds the index of the next element. By executing copies of $i = b[i]$ in rapid succession on a separate thread, the latency of $b[i]$ can be decoupled from the main program, producing the condensed schedule on the right.

While conceptually similar, the mechanics of dependence based prefetching and speculative data-driven multithreading are quite different. Dependence-based prefetching targets only loads associated with the traversal of linked data structures. Its implementation uses simple, dedicated finite state machines with private local storage that recursively use loaded values to compute prefetch addresses and contend with the main program for bus bandwidth only. In contrast, data-driven multithreading is capable of pre-executing computations of arbitrary structure and uses general-purpose execution hardware to do so. Data-driven multithreading is implemented as an extension to *simultaneous multithreading (SMT)* [35] with the computations executing as concurrent threads on the additional hardware contexts. Pre-executing threads are introduced into the pipeline at the fetch stage where they are interleaved with the main program on a per cycle basis using an intelligent thread scheduler. The rest of the pipeline is oblivious to their presence. The out-of-order core naturally interleaves their execution with that of the main program while preventing them from modifying architectural program state. Data-driven multithreading exploits the shared physical register file of an SMT processor to allow the main thread to use pre-executed instruction results directly, using a technique called *register integration*.

VI. Scheduling Load-Dependent Instructions

So far, we have discussed methods for completing loads earlier, by allowing them to execute prior to older indepen-

dent stores, ensuring that they don't incur cache misses, or obtaining their output values quickly by forwarding data from older stores or loads. After dealing with the loads themselves, we face the challenge of scheduling the instructions that depend on these loads. Modern schedulers typically schedule instructions several cycles—typically two or three—in advance. In order to schedule dependent instructions in back-to-back cycles, these schedulers must know instruction completion times that far in advance as well. This is not a problem for most instructions, which have fixed latencies. Loads, however, have variable latencies that depend on where they get their value.

To simplify scheduling, most processors are designed so that the two most common load execution scenarios—conventional forwarding from an in-flight store via the memory ordering buffer and a first-level cache hit—have identical latencies. This makes the scheduler oblivious to which of these two cases is taking place. However, for accurate scheduling, the scheduler must know if a load is going to miss sufficiently far in advance so that it does not schedule its dependent operations immediately. This is not a problem if determining hit–miss status takes less time than accessing the data. However, such a configuration is almost never the case, and most schedulers must decide whether to schedule the load dependent operations before the load has accessed the cache.

A. Optimistic Load-Dependent Scheduling

Most processors assume loads always hit in the cache and schedule dependent instructions accordingly. The details of *optimistic scheduling* sound simple, and they are on processors in which cache data access and hit verification are equal latency operations. When a cache miss is signaled, the load dependent instructions have been scheduled, but have not executed. The signal cancels their execution and blocks the scheduling of instructions that depend on them. The canceled instructions are rescheduled immediately after the requested data is fetched, as the miss handler notifies the scheduler of the data's arrival several cycles in advance. On such machines, optimistic scheduling has no direct latency cost. Its only cost is the loss of the execution slot for a canceled instruction—an opportunity cost that may delay the execution of subsequent, load-independent instructions.

Optimistic scheduling is more complex on machines in which the data access latency is shorter than the latency to determine cache hit status. The Intel Pentium 4 processor [15] is one such machine, using *way prediction* to enable very fast data access. On such machines, by the time a cache miss is signaled the dependent instructions (and potentially instructions that depend on them as well) have already executed with incorrect data. The execution of these instructions cannot simply be canceled. It must be discarded and redone. Because cache misses are rather frequent, squashing the load and all subsequent operations would be too expensive. For this reason, the Pentium 4 implements selective re-execution.

B. Hit–Miss Prediction

The optimistic policy is incorrect about 5% of the time—the average rate of first-level cache misses. *Hit–miss prediction* [3], [17], [36] can be used to identify loads that are likely to miss, and delay the scheduling of instructions that depend on them, allowing independent instructions to proceed first. At first glance, hit–miss prediction seems to only lengthen the program critical path, since every predicted cache miss that actually hits results in later than optimal scheduling of dependent instructions. However, if the prediction accuracy is very high—specifically, hits are rarely predicted as misses—hit–miss prediction can improve performance by avoiding clogging the execution units with the wrong instructions.

The challenge of hit–miss prediction lies in devising an accurate predictor. Local history has been shown to accurately predict about 30% to 50% of the cache misses on integer applications and over 80% in floating point applications [3]. The predictor can be biased to minimize the cases in which an actual hit is predicted as a miss. The high prediction rate in floating-point applications is not surprising. Floating-point applications often access arrays in a regular, strided manner in which a specific load misses every fixed number of accesses.

VII. Summary

High performance requires good scheduling which, in turn, requires accurate knowledge of operation dependences and latencies and, ideally, short latencies. The dependence and latency uncertainties and relatively long latencies of loads mean that load scheduling has a major effect on the performance of modern processors. These problems are not likely to go away. In fact, the probability is high that load scheduling will become even more important to performance. Increasing processor frequencies mean that load latencies are becoming relatively longer. Added cache levels make load latencies less predictable. Larger instruction windows increase the potential for active dependences with stores.

In this paper, we present an array of dynamic techniques for optimizing the various aspects of load scheduling. Techniques that identify store–load dependences in the absence of addresses can be used to optimize both store-independent and store-dependent loads, advancing the former and allowing the latter to avoid accessing the memory hierarchy and obtain their data directly from the store via forwarding. Other techniques attack store-independent loads that incur cache misses, attempting to advance these loads even further and to cut their observed latency. Some prioritize the instructions that are needed to compute the load address, while others try to eliminate the load address computation or even the load itself from the program critical path by predicting the load address or the loaded value. Recent techniques attempt to prefetch the needed data from main memory to register by either aggressively running ahead of the program executing only critical load computations, or by asynchronously identifying and prefetching memory blocks that are likely to be used. These techniques are not mutually exclusive. In fact, future processors will likely employ several of these techniques simultaneously.

In this paper, we limit our discussion to microarchitectural techniques. Architectural changes and compiler transformations that can take advantage of them will enable future processors to deliver even more performance. For instance, new architectures like the Intel Itanium Architecture (a.k.a. IA64) expose load speculation and memory disambiguation to software, giving the compiler and programmer the opportunity to aggressively schedule loads, beyond what the processor can do on its own.

Are we at the end of the road? Recent years have seen innovations successfully exceed limits that were previously thought insurmountable. It is our belief that new ideas for further enhancing performance will continue to appear.

REFERENCES

[1] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," in *Proc. 24th Int. Symp. Computer Architecture*, June 1997, pp. 181–193.

[2] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proc. 25th Int. Symp. Computer Architecture*, June 1998, pp. 142–153.

[3] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation techniques for improving load related instruction scheduling," in *Proc. 26th Int. Symp. Computer Architecture*, May 1999, pp. 42–53.

[4] A. Moshovos and G. S. Sohi, "Streamlining inter-operation memory communication via data dependence prediction," in *Proc. 30th Int. Symp. Microarchitecture*, Dec. 1997, pp. 235–245.

[5] G. Tyson and T. Austin, "Improving the accuracy and performance of memory communication through renaming," in *Proc. 30th Int. Symp. Microarchitecture*, Dec. 1997, pp. 218–227.

[6] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A novel renaming scheme to exploit value temporal locality through physical register reuse and unification," in *Proc. 31st Int. Symp. Microarchitecture*, Dec. 1998, pp. 216–225.

[7] M. Bekerman, A. Yoaz, F. Gabbay, S. Jourdan, M. Kalaev, and R. Ronen, "Early load address resolution via register tracking," in *Proc. 27th Int. Symp. Computer Architecture*, June 2000, pp. 306–315.

[8] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," in *Proc. 7th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Oct. 1996, pp. 138–147.

[9] M. H. Lipasti, "Value locality and speculative execution," Ph.D. dissertation, Dept. Elect. Comput. Eng., Carnegie-Mellon Univ., Pittsburgh, PA, May 1997.

[10] F. Gabbay and A. Mendelson, "Using value prediction to increase the power of speculative execution hardware," *ACM Trans. Comput. Syst.*, vol. 16, pp. 234–270, Aug. 1998.

[11] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence-based prefetching for linked data structures," in *Proc. 8th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Oct. 1998, pp. 115–126.

[12] S. Srinivasan and A. Lebeck, "Load latency tolerance in dynamically scheduled processors," in *Proc. 31st Int. Symp. Microarchitecture*, Nov. 1998, pp. 148–159.

[13] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," in *Proc. 7th Int. Conf. High Performance Computer Architecture*, Jan. 2001, pp. 37–48.

[14] E. Tune, D. Liang, D. Tullsen, and B. Calder, "Dynamic prediction of critical path instructions," in *Proc. 7th Int. Symp. High Performance Computer Architecture*, Jan. 2001, pp. 185–195.

[15] M. Upton, "The Intel Pentium® 4 Processor,", http://www.intel.com/pentium4, Oct. 2000.

[16] A. Moshovos and G. S. Sohi, "Memory dependence speculation tradeoffs in centralized, continuous-window superscalar processors," in *Proc. 6th Int. Conf. High-Performance Computer Architecture*, Jan. 2000, pp. 301–312.

[17] R. E. Kessler, E. J. McLellan, and D. A. Webb, "The alpha 21 264 microprocessor architecture," in *Int. Conf. Computer Design*, Dec. 1998, pp. 90–95.

[18] S. Onder and R. Gupta, "Dynamic memory disambiguation in the presence of out-of-order store issuing," in *Proc. 32nd Int. Symp. Microarchitecture*, Nov. 1999, pp. 170–176.

[19] D. Adams, A. Allen, R. F. J. Bergkvist, J. Hesson, and J. LeBlanc, "A 5ns store barrier cache with dynamic prediction of load/store conflicts in superscalar processors," in *Proc. Int. Solid-State Circuits Conf.*, Feb. 1997, pp. 414–415.

[20] G. B. Bell, K. M. Lepak, and M. H. Lipatsi, "Characterization of silent stores," *PACT*, pp. 133–144, 2000.

[21] K. M. Lepak and M. H. Lipatsi, "On value locality of store instructions," in *Proc. 24th Int. Symp. Computer Architecture*, June 2000, pp. 182–191.

[22] C. Molina, A. Gonzalez, and J. Tubella, "Reducing memory traffic via redundant store instructions," in *Proc. Conf. High Performance Computers and Networks*, Apr. 1999, pp. 1246–1249.

[23] A. Yoaz, R. Ronen, R. S. Chappell, and Y. Almog, "Silence is golden?," in *Int. Symp. High Performance Computer Architecture, Work in Progress Session*, Jan. 2001.

[24] A. Moshovos and G. S. Sohi, "Read-after-read memory dependence prediction," in *Proc. 32nd Int. Symp. Microarchitecture*, Nov. 1999, pp. 177–185.

[25] B. Fields, S. Rubin, and R. Bodik, "Focusing processor policies via critical-path prediction," in *Proc. 28th Int. Symp. Computer Architecture*, July 2001, pp. 74–85.

[26] J. Gonzalez and A. Gonzalez, "Speculative execution via address prediction and data prefetching," in *Proc. 11th Int. Conf. Supercomputing*, July 1997, pp. 196–203.

[27] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated load address predictors," in *Proc. 26th Int. Symp. Computer Architecture*, May 1999, pp. 54–63.

[28] T.-F. Chen and J.-L. Baer, "A performance study of software and hardware prefetching techniques," in *Proc. 21st Int. Symp. Computer Architecture*, Apr. 1994, pp. 223–232.

[29] S. Mehrotra and L. Harrison, "Prefetch system applicable to complex memory access schemes," in *Proc. 10th Int. Conf. Supercomputing*, May 1996, pp. 133–139.

[30] G. Kedem and H. Yu. DRAM-page based prediction and prefetching. [Online]. Available: http://kedem.cs.duke.edu/HPMA/Prefetching/index.html

[31] T. Sherwood, S. Sair, and B. Calder, "Predictor-directed stream buffers," in *Proc. 23rd Int. Symp. Microarchitecture*, Dec. 2000, pp. 42–53.

[32] Y. Sazeides and J. E. Smith, "The predictability of data values," in *Proc. 30th Annu. Int. Symp. Microarchitecture*, 1997, pp. 248–258.

[33] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt, "Simultaneous subordinate microthreading (SSMT)," in *Proc. 26th Int. Symp. Computer Architecture*, May 1999, pp. 186–195.

[34] Y. H. Song and M. Dubois, "Assisted execution," Dept. Elect. Eng., Univ. Southern California, Tech. Rep. CENG 98-25, Oct. 1998.

[35] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, D. M. Tullsen, and S. J. Eggers, "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading," *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 322–354, 1997.

[36] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta, "Predictability of load/store instruction latencies," in *Proc. 26th Int. Symp. Microarchitecture*, Dec. 1993, pp. 139–152.

Amir Roth received the B.Sc. degree from Yale University, New Haven, CT, in 1994 and the M.Sc. and Ph.D. degrees in computer science from the University of Wisconsin-Madison in 1997 and 2001, respectively.

He is currently an Assistant Professor at the University of Pennsylvania, Philadelphia. His research interests include computer architecture, microprocessor design, and performance evaluation.

Ronny Ronen (Senior Member, IEEE) received the B.Sc. and M.Sc. degrees in computer science from the Technion, Israel Institute of Technology, Haifa, Israel, in 1978 and 1979, respectively.

He is currently a Principal Engineer with the Microprocessor Research Laboratories, Intel Corporation, Haifa, Israel, focusing on microarchitecture research. He has been with Intel for over 20 years and has led the compiler and performance simulation activities in the Intel Israel Software Department.

Avi Mendelson (Member, IEEE) received the B.Sc. and M.Sc. degrees in computer science from the Technion, Israel Institute of Technology, Haifa, Israel, in 1979 and 1982, respectively, and the Ph.D. degree in electrical and computer engineering from the University of Massachusetts, Amherst, in 1990.

He was with the Electrical Engineering Department at the Technion from 1990 to 1997 and with National Semiconductor from 1997 to 1999. He is currently a Research Staff Member with the Microprocessor Research Laboratories, Intel Corporation, Haifa, Israel. His current research interests include computer architecture, operating systems, and system-level performance analysis.

Dr. Mendelson is a member of the ACM.