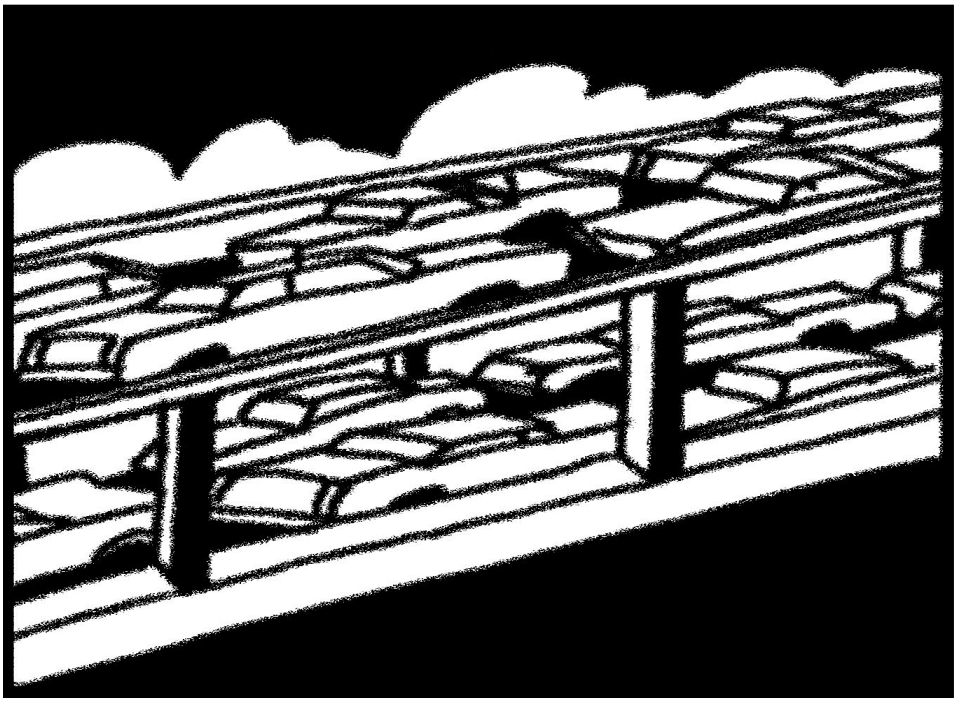
Classifying Software-Based Cache Coherence Solutions

The authors propose a classification for software solutions to cache coherence in shared-memory multiprocessors and show how it can be applied to more completely understand existing approaches and explore possible alternatives.



IGOR TARTALJA AND
VELJKO MILUTINOVIĆ
University of Belgrade

Shared-memory multiprocessor systems are efficient architectural support for applications with a high degree of data sharing. Incorporating a shared cache memory helps close the speed gap between the processors and main memory. However, a shared cache does not address the problem of access contention for both the interconnection network and the memory modules. Private caches can be used to satisfy most memory references locally, but they can cause the memory system to become incoherent (inconsistent). If several processors access the shared data, several copies of shared data will exist in private cache memories. If one processor changes the data, the other copies become stale and another processor may mistakenly use them.

0740-7459/97/$10.00 © 1997 IEEE

WHEN IS A MEMORY SYSTEM INCOHERENT?

Assuming the main memory value of the shared variable is current, incoherence occurs when a value fetched from the cache differs from the value in main memory. According to this definition, Alexander Veidenbaum[1] gives a formal proof about the necessary conditions for incoherence.

When processor P_j (j = 1,2,...) fetches variable X, incoherence arises if

- the value of X is present in the processor's P_j cache, and
- the new value of X is placed into main memory by processor P_k ($k \neq j$, k = 1,2,...), after processor P_j had accessed X last time.

Detecting the necessary conditions of incoherence can be based on the data dependency graph analysis. As Figure A indicates, incoherence will occur if the following is satisfied:

- P_j executes instruction S_1, which writes into or reads from X;
- another processor P_k ($k \neq j$) executes instruction S_2, which writes a new value into X; and
- P_j executes instruction S_3, which reads from X again.

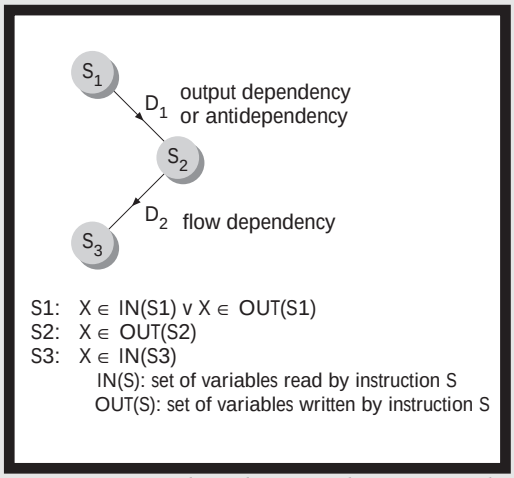


Figure A. *Data dependency graph containing the conditions of incoherence.*

REFERENCE

1. A. Veidenbaum, "A Compiler-Assisted Cache Coherence Solution for Multiprocessors," *Proc. Int'l Conf. Parallel Processing*, IEEE Comp. Soc. Press, Los Alamitos, Calif., 1986, pp. 1029-1036.

This problem is at least two decades old, and many hardware, software, and hybrid solutions have been proposed. Hardware approaches[1] make the maintenance of coherence fully transparent for all levels of software. This makes the programming model very flexible but increases hardware complexity. Software approaches[2] lift the transparency of the problem above the operating system or compiler, so hardware support is less complex. Software approaches generally restrict the programming model more than hardware approaches, but these restrictions are natural and tend to make programs more testable and robust. Finally, software approaches can be more efficient than hardware approaches in some applications.[3]

Despite these important advantages, software solutions have not been widely addressed in the literature. For example, we have not seen a broad, systematic, and flexible classification, nor an exhaustive survey. Existing surveys either focus on hardware solutions; include software solutions, but not in sufficient detail; or concentrate on just a few, usually the authors' own, software solutions. Most existing classifications are not based on an extensive list of criteria clearly distinguishing among existing solutions.

In this article we offer a classification for software solutions that is based on 10 criteria. The first five are binary-choice criteria, serving to roughly label the solution as either dynamic or static, for example. The second five are multiple-choice criteria offering more precise classification, such as granularity of coherence objects: cache line, memory page, data segment, or flexible object. Classes derived from each criterion are used as attribute values of potential or existing approaches.

We also offer a formalization of this classification that looks at solutions as points in an abstract multidimensional criteria space. Designers can use this space to explore the utility of solutions that do not yet exist and identify areas that may warrant closer examination.

We derived our classification from one reported by Sang Lyul Min and Jean Loup Baer.[4] The Min-Baer classification is based on five criteria (some of which we also use) and encompasses a relatively small number of existing solutions. We believe our proposed criteria are broad enough to create clear distinctions among most existing software solutions.

LIMITATIONS OF EXISTING CLASSIFICATIONS

One of the most cited definitions of memory system coherence is this:[5]

> *A memory scheme is coherent if the value returned on a load instruction is always the value given by the latest store instruction with the same address.*

This definition allows that copies of shared data contain stale values temporarily and requires only that the values be updated before they are read. The ability to delay coherence enforcement to the moment it is necessary decreases the frequency of the coherence actions and consequently the system overhead. Most software solutions aim to provide some mechanism for enforcing coherence that incorporates this ability. The boxed text "When Is a Memory System Incoherent?" describes the necessary conditions for incoherence.

Where this coherence mechanism goes—in either hardware or software—is the most common criterion used to classify coherence solutions. This is not an effective basis for classification, however, because many solutions are actually hybrids: Software coherence protocols require some hardware support; hardware solutions can benefit from compiler-level optimization.[6] Here, we view a solution as software-based if the basic coherence protocol does not work without aid from a system software layer.

A more effective view of solutions is in terms of their aspects: dynamic/static, centralized/distributed, communication-based/communication-free, and scalable/nonscalable. In our classification for software solutions, for example, we chose as the first criterion dynamism (dynamic or static). That is, if coherence actions are planned at runtime, the solution is dynamic. All solutions implemented

TABLE 1A
PROPOSED CLASSIFICATION: COARSE-GRAINED CRITERIA

Criterion and Definition	Classes and Examples	Comments
Dynamism (D): Decisions about coherence actions are made at compile time (static) or runtime (dynamic).	**Static** (s): c.mmp page marking, version control, one-bit timestamps. **Dynamic** (d): one-time identifiers, conditional invalidation, adaptive cache management.	Only schemes that require absolutely no compile-time analysis are considered dynamic. Schemes that make final invalidation decisions at runtime, but also perform certain compile-time analysis, are static.
Selectivity (S): Explicit actions (such as invalidation) are applied without any spatial discrimination on the entire cache or selectively on a part of the cache.	**Indiscriminate** (i): cache on/off control. **Selective** (s): fast selective invalidation, programmable cache, conditional invalidation.	Selective schemes differ in the granularity of data objects for which coherence is maintained. The granularity differences are reflected through the granularity criterion (see Table 1B).
Restrictiveness (R): Coherence is maintained preventively (conservative) or only when necessary (restrictive).	**Conservative** (c): cache on/off control, one-time identifiers. **Restrictive** (r): version control, conditional invalidation.	Restricting invalidations reduces cache misses. This criterion could result in a fine division with more than two classes. You could introduce a "level of restrictiveness" measure (the ratio of necessary coherence actions to total coherence actions performed, averaged over a number of benchmark programs). If the scheme is absolutely restrictive, this ratio is 1.
Adaptivity (A): The coherence protocol is fixed or adaptable to the characteristics of the patterns of access to data objects.	**Fixed** (f): one-time identifiers, version control. **Adaptive** (a): adaptive cache management.	So far, the only software adaptive scheme in the open literature is adaptive cache management. Thus, we need not introduce the finer measure "level of adaptivity."
Locality (L): The coherence protocol may reduce misses strictly locally (interepoch reduction) or globally (intraepoch reduction).	**Local** (l): one-time identifiers, cache on/off control. **Global** (g): version control, conditional invalidation.	This criterion extends the restrictiveness criterion and is irrelevant for very conservative schemes.

on the operating system level belong to this group. If the coherence actions are planned at compile time, the solution is static. All compiler solutions belong to this group.

PROPOSED CLASSIFICATION

Tables 1A and 1B list the 10 criteria on which we base our classification scheme. Each criterion has two or more classes. Although the table does not include it, we also have a wild-card class (*) to designate that the criterion is irrelevant for that solution, or that there is insufficient information in the literature to classify the solution. This lets us consistently classify schemes with a 10-element tuple, in which each element represents a criterion plus the appropriate class denoted by X:x (or X:* if the criterion is irrelevant). The tenth element can contain more than one class (denoted by X:x/+y+.../) if the solution uses more than one technique for detecting incoherence. In the table we include examples from the open literature for each class.

Our classification promotes a layered understanding of the techniques being evaluated because it has two levels of characterization: coarse-grained (the binary-choice criteria) and fine-grained (the multiple-choice criteria). However, the classification is flat, not hierarchical. This simplifies the "widening" of the criteria set to accommodate new schemes. Consequently, the criteria set we present is not fully orthogonal, but we believe this disadvantage is not crucial for most practical applications.

CLASSIFYING EXISTING SCHEMES

Our classification is suitable for both understanding existing schemes and exploring new approaches. Here, we apply it to existing software cache coherence schemes. This list of schemes is not intended as a detailed account, but as an overview that gives some idea of how the proposed classification captures a greater depth of characteristics and thus promotes understanding. Using the criteria and classes in Table 1, we associate each scheme with a 10-element tuple to show where it falls in the proposed classification.

Static schemes. Static schemes rely predominantly on program analysis at compile time. Analysis points to potential causes of incoherence, and adds information to the program to avoid fatal incoherence errors during the execution. This additional information can include marking of data or references and insertion of special instructions. The analysis and related actions eliminate the runtime coherence traffic, making the static schemes easy to scale up. On the down side, static solutions potentially degrade performance. Predicting incoherence conditions at compile time is imprecise. Consequently, coherence actions are performed on a worst-case basis, and many actions may be performed that are actually not required for that scenario.

Most static schemes assume that the parallel program can be expressed through a leveled directed task graph, in which nodes denote tasks, and edges denote intertask dependencies. Indeed, such a graph is a natural presentation model for numeric applications, a large class of parallel programs. The pro-

TABLE 1B
PROPOSED CLASSIFICATION: FINE-GRAINED CRITERIA

Criterion and Definition	Classes and Examples	Comments
Granularity (G): The size and structure of the coherence data object	**Line** (l): version control. **Page** (p): c.mmp page marking, one-time identifiers. **Segment** (s): conditional invalidation. **Flexible** (f): RP3 flexible invalidation, adaptive cache management.	This criterion extends the selectivity criterion. It is irrelevant for fully indiscriminate schemes.
Blocking (B): The basic program block as a code unit for coherence protocol	**Critical region** (c): one-time identifiers, conditional invalidation. **Epoch** (e): Ultracomputer program structure analysis, cache on/off control, program analysis and reference marking, version control, timestamps. **Subroutine** (s): programmable cache. **Program** (p): c.mmp page marking.	The epoch class encompasses terms such as "computational unit," "loop," "epoch," and "task level," which are essentially similar coherence blocks.
Positioning (P): Position of instructions to implement the coherence protocol	**Entry/exit of critical region** (e): one-time identifiers, conditional invalidation. **Loop boundary** (b): cache on/off control, version control. **Source/sink of data dependency** (d): programmable cache. **Interrupt procedure** (i): coherence on interrupt request.	This criterion considers only the special coherence instructions (`Invalidate`, `Flush`, `Post`, and so on). It does not include coherence actions such as comparisons at each reference, which some schemes can do with special hardware support.
Updating (U): The main memory is updated either during the write or later	**Write-through** (t): version control. **Write-back** (b): programmable cache, adaptive cache management. **Hybrid** (h): one-time identifiers. **Alternative** (a): conditional invalidation.	Hybrid schemes typically offer both write-through for shared writeable data and write-back for private and shared read-only data. Alternative schemes offer either write-through or write-back.
Checking (C): Technique used to check conditions of incoherence	**Checking data type or reference type** (r): c.mmp page marking, program analysis and reference marking, timestamps. **Program structure analysis** (s): RP3 flexible invalidation, cache on/off control. **Data dependency analysis**: (d): programmable cache. **Bitwise information runtime comparison** (b): fast selective invalidation, lifespan strategy, one-bit timestamps. **Version comparison** (v): version control. **Generation comparison** (g): generational approach. **Monitoring interconnection network traffic** (m): coherence on interrupt request.	Some schemes include multiple techniques for detecting incoherence. Such schemes are classified according to the dominant technique. For completeness, we indicate the lesser techniques with slashes, for example c/+s+d+r/.

gramming model for these applications assumes that the program consists of parallel or serial loops (sequential code between consecutive loops could be viewed as a serial loop with one iteration). If we consider each task graph level as an epoch associated with an iteration or a whole loop, whether or not coherence actions are needed in the epoch depends on the type of loop. The most common place for coherence enforcement is a loop boundary.

In a parallel `DoAll` loop, for example, there is no data dependency between iterations, so they can execute in parallel. Another processor cannot consume the new version of the shared variable produced in the same loop. Thus, no coherence actions are needed, and a temporary incoherence of the memory system is allowed until the end of the loop.

In a parallel `DoAcross` loop, on the other hand, incoherence may arise because parallelization is possible (with additional synchronization) and there is data dependency among iterations.

Finally, because only one processor executes a serial loop (the structure of data dependencies between iterations disables any parallelization), incoherence cannot arise inside the loop, and there is no need for coherence actions.

C.mmp page marking. William Wulf and C. G. Bell[7] of Carnegie Mellon University were among the first to note the problems arising from multiple values of shared writeable variables existing concurrently in the private caches of different processors. Their method keeps the writeable shared data out of the cache at all times, allowing only the read-only shared pages to be cached in the private cache memories. This is particularly useful for pages that contain shared instructions. Page marking uses a bit in the relocation registers to mark pages as cachable. This scheme is extremely conservative, making it relatively easy to implement, but at the cost of considerably lower processing power.[8]

This scheme is classified as (D:s,S:*, R:c,A:f,L:*,G:p,B:p,P:*,U:*,C:r).

Ultracomputer program structure analysis. Allan Gottlieb and coauthors,[9,10] Jan Edler and coauthors,[11] and Kevin McAuliffe[12] describe an approach for the Ultracomputer multiprocessor developed at New York University. The caching of read-write shared data is temporarily permitted during the *safe* execution epochs. Safe epochs are defined as times

when the data is either accessed exclusively by one processor or only read by multiple processors.

The Ultracomputer caches support two instructions for software control.

Our classification promotes a layered understanding of the techniques being evaluated.

The instructions are inserted into the code at compile time. The `Invalidate` (originally `Release`) instruction frees one cache memory entry without copying the data into main memory, thus preventing network traffic. The `Post` (originally `Flush`) instruction copies data from the cache memory entry into the main memory without invalidating the entry. The main memory is updated using the write-back policy[10] (a later paper[11] proposes a combined approach: write-back for private data and write-through for shared data).

The read-write shared variables are cachable only during the safe epochs. At the end of a safe epoch, the variable must be invalidated, and marked as noncachable. For data shared by a parent task and its children, `Post` and `Invalidate` are executed before the child tasks are created. The variable that was private before now becomes shared and is marked noncachable. After the parent task's children are completed, it can continue to work with the same variable, treating it as private, and therefore cachable. Also, `Post` must be executed before a task switch because the execution of the task may migrate to another processor. Both instructions can be performed on the segment or whole cache level.

This scheme is less conservative than the Wulf-Bell scheme. Still, it is not restrictive in the invalidation of shared data. The scheme prevents unnecessary misses only locally within epochs. The coherence instructions are selective with the segment-level granularity, but they are slow because of the need to scan through the cache directory.

This scheme is classified as (D:s,S:s, R:c,A;f,L:l,G:s,B:e,P:b,U:h,C:s/+r/).

RP3 flexible invalidation. W.C. Brantley, K. McAuliffe, and J. Weiss[13] proposed an approach similar to the Ultracomputer scheme. They did the research at the IBM T. J. Watson Research Center on the IBM RP3 multiprocessor prototype.[14]

As with the Ultracomputer cache coherence scheme, the RP3 scheme permits changing the cachability status of shared variables between consecutive epochs. It also introduces a volatility attribute for temporarily cachable data. The attribute enables a specific instruction for invalidating volatile data in many segments or pages. It also supports several other invalidation instructions aimed at different objects: a cache line, a page, a segment, all user space, or all supervisor space. The cachability and volatility attributes of each segment or page are specified in an appropriate segment or page descriptor. These attributes describe segment and page tables as well. The caches are write-through, so the main memory is always up to date.

In general, the cache coherence strategy of the IBM RP3 scheme is still very conservative. However, its selective invalidation is more flexible than that in the Ultracomputer approach because the granularity of the invalidation can be varied. This results in higher precision and consequently a lower miss ratio.

This scheme is classified as (D:s,S:s, R:c,A:f,L:l,G:f,B:e,P:b,U:t,C:s/+r/).

Cache on/off control. Alexander Veidenbaum[15] proposed a relatively simple protocol for cache coherence in the Cedar multiprocessor at the University of Illinois. The protocol assumes the program structure is based on parallel/serial loops, as described at the beginning of this section. The compiler inserts coherence-related instructions only at loop boundaries and at subroutine call points.

The proposed instructions are `Invalidate` (originally `Flush`), which deletes the contents of the entire cache (indiscriminate invalidation); `Cache-on,` which forces all references to go through the cache; and `Cache-off,` which forces all references to bypass the cache. The compiler analyzes program structure and inserts the coherence instructions before, inside, or after the loops, as well as before or after subroutine calls.

In a `DoAll` loop, the cache can be enabled, so the protocol inserts `Cache-on` and `Invalidate` after the `DoAll` statement. In that way, each processor, at the beginning of the iteration assigned to it, enables its own cache and, as a preventive measure, removes the old contents. Generally, in a `DoAcross` loop the cache should be disabled, so the protocol inserts a `Cache-off` after the `DoAcross` statement. Specifically, the cache could be enabled if an `Invalidate` is inserted after the synchronizing wait-on-semaphore operation in the iteration that contains the sink of data dependency. In a `DoSerial` loop the cache can be enabled because only one processor executes the loop. If the cache is disabled before the loop, the compiler must insert `Cache-on` and `Invalidate` before the loop.

After the `DoEnd` statement, the state of cachability is restored to what it was before the loop. If the cache is enabled before a parallel loop, the compiler inserts `Invalidate` after the `DoEnd` statement. It inserts a `Cache-off` before the subroutine call. If the cache is enabled before the call, it inserts `Cache-on` and `Invalidate` after the call.

The basic algorithm includes no analysis of individual references and data dependencies. It disables and/or invalidates the entire cache. For example, the caching of private and read-only shared data is prohibited inside `DoAcross` loops. Consequently, this scheme is very conservative, its restrictiveness of invalidation is very low, and the invalidation is indiscriminate. Nevertheless, this

scheme is very important as a baseline for many subsequent schemes.

This scheme is classified as (D:s, S:i,R:c,A:f,L:l,G:*,B:e,P:b,U:t,C:s).

Program analysis and reference marking. Roland Lee, Pen-Chung Yew, and Duncan Lawrie[16,17] proposed a scheme based on the static analysis of program structure and marking of individual references. They did their research at the University of Illinois and embedded the algorithm into Parafrase, a parallelizing Fortran code restructurer.

The compiler analyzes the program and segments it into a sequence of epochs defined by parallel loops or sequential code segments between consecutive parallel loops. For each reference, it uses data dependence tests to detect the data cachability status. All references to a variable that multiple processors access within an epoch are marked as noncachable if at least one processor writes to the variable. References marked as noncachable bypass the cache at execution time and access the main memory directly. At the end of each epoch, the processor invalidates the entire private cache. The write-back caches are used.

This algorithm efficiently supports *intraepoch* localities, localities of references *within* a single epoch. However, the cache is indiscriminately invalidated at the end of each epoch, which ignores *interepoch* localities, localities *between* epochs.

This scheme is classified as (D:s,S:i, R:c,A:f,L:l,G:*,B:e,P:b,U:b,C:r/+s+d/).

Fast selective invalidation. Hoichi Cheong and Veidenbaum[18] describe a scheme based on efficient invalidation. The invalidation effect depends on the characteristics of the references. The speed of proposed invalidation is close to the speed of indiscriminate invalidation, but its selectivity implies a better hit ratio. This research also originated from the University of Illinois.

The basic scheme considers a cache with one word per line. Each cache word is supplied with a *change bit*. When set, this bit indicates that the word is potentially stale. Arbitrary access to a cache word resets its change bit. The `Invalidate` instruction sets the change bits of all words in the cache in one cycle. Consequently, invalidation is as fast as indiscriminate invalidation. The compiler inserts the `Invalidate` instructions at the loop boundaries.[15] A *clear bit* is also associated with each cache word. A `Clear` instruction sets clear bits of all cache words to provide an empty cache at the beginning of a subroutine.

Read references that can be guaranteed not to fetch a stale data copy are marked as `cache-read` (for example, references to data that are read-only within the analyzed subroutine, references to read-write data coming before the first write to the same data, or references that together with all preceding writes lie within an epoch). Unsafe read references (such as those after the epoch containing some writes to the data, if there is at least one read of the data before the epoch) are marked as `memory-read`. A cache miss is forced if a `memory-read` reference accesses the word with a set change bit. The scheme assumes the use of write-through caches.

This approach has several benefits. First, invalidation has no effect on `cache-read` references; that is, invalidation is selective. Second, when a variable is read more than once within a single iteration of a `DoAll` loop, although all reads could be marked as `memory-read`s, the invalidation will affect the first read only. This effectively supports intraepoch localities. Also, some interepoch localities are supported, for example, when `read` references belong to different epochs and precede the first write. However, the approach still inserts `Invalidate` instructions into the code conservatively.

This scheme is classified as (D:s,S:s, R:c,A:f,L:g,G:l,B:e,P:b,U:t,C:b/+s+r+d/).

Programmable cache. Ron Cytron, Steve Karlovsky, and McAuliffe[19] proposed a solution that is based on a detailed static analysis of a program's data dependencies. The authors were with the IBM T.J. Watson Research Center and the University of Illinois when they published their work.

The special instructions proposed for cache management are `Post` (copies the value of the cache data into the main memory), `Invalidate` (destroys the cache data copy), and `Flush` (performs a combination of the previous two instructions). The approach assumes the use of a write-back cache with one word per line. The authors propose the following algorithm for inserting introduced instructions into the code:

- Insert `Post(X)` after a write into a shared variable *X* if the write is a source of the crossing flow dependency (if the sink of the dependency is a read that can be executed on another processor). Also, if *X* is live after the procedure under analysis, insert the instruction `Post(X)` after the last write to *X* within the same procedure.
- Insert `Invalidate(X)` before the sink of the crossing flow dependency if the source of the analyzed flow dependency acts as a sink in at least one output dependency or antidependency. Also, if a variable *X* has been defined before entry into the procedure, insert `Invalidate(X)` before the first reading of the variable *X* in the procedure.

This method is suitable for both understanding existing schemes and exploring new approaches.

- Insert `Flush` after the source of the flow dependency, substituting `Post` and `Invalidate` if the substitution does not cause an unnecessary miss.

Additionally, the compiler marks each variable in the global address space as

cachable (read-only shared data and private data), temporarily cachable (general read-write shared data), or noncachable (read-write shared data accessed only after invalidation).

This scheme greatly restricts invalidations within a procedure because both the program structure and data dependency analyses are used to maintain coherence. However, unnecessary interprocedural invalidations still occur.

Dynamic software solutions are similar to hardware solutions: they maintain the coherence of private caches entirely at runtime.

This scheme is classified as (D:s,S:s, R:r,A:f,L:g,G:l,B:s,P:d,U:b,C:d/+s+r/).

Version control. Cheong and Veidenbaum[20] describe a scheme that improves on their fast selective invalidation scheme. The version control scheme, also developed at the University of Illinois, is based on the static analysis of the parallel program tasking structure and a dynamic control of the variable version using appropriate hardware support.

The approach centers on the notion that whenever a write is made to a shared variable, a new version of its contents is defined. Each processor locally maintains global information about the current version of each shared variable (scalar or vector) using a *current version number*. When passing over a task-level boundary (for example, after exiting from a `DoAll` loop), processors execute the instructions (inserted by the compiler) for incrementing the current version number for all variables being written (regardless of which task wrote them) at the previous task level.

The authors propose extending each cache line with a *birth version number* field. After a read miss and the loading of the variable into cache, the cache controller assigns the current version number value to the birth version number field. At each cache write, the controller updates the birth version number to the incremented current version number (the number of the next version). At each access to a shared data item in cache, the controller compares the current and birth version numbers. If the birth version number is smaller, the data item in the cache is stale, and a miss is forced.

The version control scheme is based on the local maintenance of global information for a shared data version. The scheme demonstrates a better performance than the previous two schemes[15,18] because it respects the temporal locality of references over the task-level boundary. As a consequence, the hit ratio increases. However, this benefit comes at the cost of relatively complex hardware support.

This scheme is classified as (D:s,S:s, R:r,A:f,L:g,G:l,B:e,P:b,U:a,C:v/+s/).

Timestamps. Min and Baer[21,4] of the University of Washington proposed a scheme similar to the version control scheme, but did so independently of Cheong and Veidenbaum. In the timestamps scheme, each shared data structure is assigned a counter that is incremented at the end of each epoch in which the data structure can be modified. Each cache word is assigned a timestamp, which is set to (*counter* + 1) when the word is modified. At the access to cache memory, a word is valid if the value of the timestamp is equal to or greater than the value of the corresponding counter. The counter corresponds to the version control scheme's current version number; the timestamp, to the birth version number; and the timestamps' epoch, to the version control scheme's task level.

The primary difference between the two approaches (timestamps and version control) is that timestamps has a sophisticated algorithm for reference marking, which better supports the localities between dependent tasks in a `DoAcross` loop.

This scheme is classified as (D:s,S:s, R:r,A:f,L:g,G:l,B:e,P:b,U:t,C:v/+s+r/).

Life-span strategy. Cheong, a coauthor of the fast selective invalidation scheme, went on to improve this scheme. The new scheme,[22] in a basic one-level version, better supports the temporal locality of shared variables over consecutive epochs. The tradeoff is one additional bit per cache coherence unit and a few additional instructions.

Starting from the support for the fast selective invalidation scheme, Cheong proposes adding a *stale bit* to each cache line, and renaming `memory-read` to `memory-read-reset-stale`. Each access to the shared variable (`memory-read-reset-stale` or `write`) resets the stale bit, indicating the validity of the variable within the subsequent epoch. `Invalidate` simply copies the stale bit into the change bit, and then sets all stale bits.

Cheong also proposes an extension for an *n*-level life-span strategy that is based on the use of multiple stale bits. Another extension supports the parallelism of `DoAcross` loops (with additional instructions `memory-read` and `write-set-stale`).

The baseline algorithm of this scheme exploits localities between two consecutive epochs. Consequently, this scheme exploits more localities than the fast selective invalidation strategy, but less than the version control or timestamp schemes. The life-span strategy scheme needs one more bit per cache line, relative to the fast selective invalidation scheme, but unlike the version control or timestamps strategies, it does not need version counters.

This scheme is classified as (D:s,S:s, R:r,A:f,L:g,G:l,B:e,P:b,U:t,C:b/+s+r+d/).

Generational approach. Tzi-cker Chiueh[23] improves on the control of shared-data versions, using a leveled task graph to keep track of task generations instead of separately managing each variable version. He performed this work at the State University of New York at Stony Brook.

The scheme extends each cache line with a *valid generation number* field. To indicate the *current generation number* of tasks, it adds a register to each processor register set. At the end of each task that writes to a shared variable, the compiler inserts an instruction to set the appropriate valid generation number field to the number of the generation, during which a task will be writing into the given variable; thus, the field indicates the last generation number until the appropriate shared variable is considered valid.

When a task is scheduled to a processor, it initializes the current generation number register of that processor to indicate the current generation of tasks. An access to a shared variable is treated as a hit if the appropriate valid generation number field contains a value greater than or equal to the current generation number.

This approach has simpler hardware support than the version control and timestamps schemes. Instead of having a local table for each processor to keep information on the global current version of each shared variable, it requires only one additional register (for the current generation number). Additionally, the generational approach naturally overcomes some inherent problems of the version control scheme, such as `DoAcross` loop handling or the inefficiency of conditional writes.

This scheme is classified as (D:s,S:s, R:r,A:f,L:g,G:l,B:e,P:b,U:t,C:g/+s+r/).

One-bit timestamps. Ervan Darnell and Ken Kennedy[24] of Rice University describe a scheme that is based on timestamps but is implemented using only one additional bit per cache line. The approach exploits the validity of cached shared data immediately after the epoch within which it was locally accessed.

An *epoch bit* is associated with each cache line to flag any arbitrary access to the cache line during the current epoch. All epoch bits are reset at each epoch boundary. The compiler inserts `Invalidate` instructions at the end of each epoch (before resetting the epoch bits). Invalidation can be related to the whole array conservatively or to only the parts that have been written during the current epoch. At runtime, these instructions actually invalidate only the cache lines that have not been accessed during the epoch. Thus, `Invalidate` in effect copies the epoch bit to the valid bit.

This scheme involves considerably simpler hardware support than the timestamps and version control schemes—each cache line grows by only one bit (the epoch bit) and no additional bits are required in instructions or private memory. The scheme also achieves at least the same and frequently a better (as simulation studies show) hit ratio than the timestamps scheme. Although the range of supported localities is limited to consecutive epochs, most shared data references are resolved in the cache.

This scheme is classified as (D:s,S:s, R:r,A:f,L:g,G:l,B:e,P:b,U:*,C:b/+s/).

Dynamic solutions. Dynamic software solutions are similar to hardware solutions in that they maintain the coherence of private caches entirely at runtime. Because they are implemented in the operating system's kernel, they do not contribute to compiler complexity. Incoherence conditions are detected at execution time, so dynamic approaches reduce preventive (unnecessary) actions. On the down side, they are difficult to apply when parallelism is expressed via parallel loops. This makes them ill-suited for numeric applications, which are based primarily on a programming model characterized by parallel loops. On the other hand, dynamic solutions offer a natural solution for concurrent applications based on a programming model characterized by heterogeneous threads communicating via data-sharing mechanisms such as critical regions or monitors.

One-time identifiers. Alan Smith[25] describes a dynamic solution that is entirely embedded in the operating-system-level operations for critical region handling. He conducted this research at the University of California at Berkeley.

Smith proposed that each translation lookaside buffer entry and each line in the processor cache be expanded with the *one-time identifier* field that temporarily uniquely marks a shared data page. When a processor loads a new translation lookaside buffer entry, the scheme places a new and unique value from a special incrementing register into the one-time identifier field. When the processor accesses an address from the shared page for the first time, it loads the line into the cache from the main memory and copies the value of the one-time identifier field from the translation lookaside buffer entry to the one-time identifier field in the cache line.

All subsequent accesses to this variable check the one-time identifier field in the cache for a match with the one-time identifier field in the translation lookaside buffer. If a match occurs, the access is treated as a hit. After exiting from a critical region, the processor executes the instructions that invalidate all translation lookaside buffer entries of the pages that belong to the shared data being protected by the just-exited critical region. At the next access to the shared data, the processor loads the corre-

Dynamic solutions are implemented in the operating system's kernel, so they do not contribute to compiler complexity.

sponding translation lookaside buffer entry again, and obtains a new value for the one-time identifier field.

Smith also proposes using the write-through approach for shared data and the write-back approach for private data.

Our formal classification treats individual schemes as points in an abstract criteria space.

The advantage of the one-time identifier scheme is the resulting complete decentralization, similar to most static schemes (but not most hardware schemes). Coherence is maintained at each processor autonomously without communication with other processors. However, the scheme requires relatively complex hardware support (the one-time identifier extension in the translation lookaside buffer and the cache, comparators for the one-time identifier fields, and so on). Also, when the processor exits from a critical region, there is no longer a way to restrict the execution of `invalidation` instructions, so invalidation is done as a preventive action.

This scheme is classified as (D:d,S:s, R:c,A:f,L:l,G:l,B:r,P:e,U:h,C:v).

Coherence on interrupt request. David Cheriton, Gert Slavenburg, and Patrick Boyle[26] present a true hybrid solution for maintaining the coherence of virtually addressed cache memories. They conducted their work at Stanford University as part of the VMP multiprocessor project. The solution is characterized by hardware-based detection of incoherence conditions and the generation of an interrupt request, followed by the software-based enforcement of coherence.

This scheme is based on the concept of virtual memory: page faults are determined in hardware; the fetching of requested pages from secondary memory is done in software. The proposed cache page size is relatively large (up to 512 bytes), and the proposed hardware for the page transfer is relatively fast (at 40 Mbytes/sec). Consequently, cache misses are relatively rare and can be processed at the operating-system level.

This concept is extended into the domain of cache coherence maintenance. The hardware-based bus monitor detects incoherence conditions and interrupts the local processor. In response, the processor executes the interrupt routine that enforces coherence.

The main memory is viewed as a sequence of cache page frames. A page can be in either a shared or private state. In the *shared* state, the page frame contains the current page value; private cache memories are allowed to contain copies of that page. In the *private* state, only one cache memory contains the page.

Each bus monitor maintains a private table of actions. For each page frame, this table defines what the monitor must do when the address from that page appears on the bus. If the corresponding page is not in the processor's private cache, no action is needed. If the page in the cache is shared, the monitor ignores all `read-shared` requests; it must also abort all `read-private` and `assert-ownership` requests and issue an interrupt to the local processor, causing the page to be invalidated. If the page in cache is private, the monitor must abort the request and issue an interrupt, causing, if the page was dirty, a write-back into main memory; if the request is `read-private` or `assert-ownership`, the processor invalidates the page; if the request is `read-shared`, the page becomes shared. The processor performing the aborted request detects the abortion and requests the page again.

Because the detection of incoherence conditions is completely hardware-based, invalidations are very restricted. However, the cache page must be large enough to make page faults relatively rare. This makes invalidation poorly selective if the shared data is fine-grained.

This scheme is classified as (D:d,S:s, R:r,A:f,L:g,G:p,B:p,P:i,U:b,C:m).

Conditional invalidation. We have developed three dynamic software schemes that test for incoherence conditions when a processor enters a critical region, thereby avoiding unnecessary invalidations.[27] We performed this work at the University of Belgrade in cooperation with NCR Corp.

Coherence is enforced at the entry into/exit from the critical region. Of the three schemes we propose, *version verification* restricts invalidation to the greatest degree, and most effectively uses the existing interregional locality of references.

Upon entry into the critical region, the operating system explicitly compares the real version of the shared segment (information from the shared-segment table) with private information about the most recently used segment version. If the versions do not match, the operating system selectively invalidates the shared segment. If the shared memory is updated using the write-back approach, the operating system updates the shared segment original (in main memory) before exiting from the critical region.

Results of our simulation analysis show that advantage of restrictively applied invalidation grows with number of processors. The advantages were especially prominent if the shared segments were mostly read.

This scheme is classified as (D:d,S:s, R:r,A:f,L:g,G:s,B:r,P:e,U:a,C:v).

Adaptive cache management. John Bennett, John Carter, and Willy Zwaenepoel[28] describe a dynamic adaptive scheme for the Munin system at Rice University. Munin supports several coherence mechanisms and applies them according to the access dynamics of each class of shared objects. Although Munin represents a distributed shared-memory system, we include this work in our list because of its generality regarding the concept of adaptivity.

In such systems, the incoherence

problem exists because the shared address space is physically distributed across several local memories (caches), enabling one memory address to be mapped into several local memories.[29] Munin makes this problem completely transparent for the application; the programmer merely informs the system about the expected access dynamics to shared objects.

The authors' study shows that there are relatively few general read-write objects, that parallel programs behave differently in different phases of execution, and that except in the initialization phase, most accesses are reads. The authors note that the average period between two accesses to synchronization objects is considerably longer than the average access period for other shared objects.

Munin maintains the following object classes using the mechanisms in parentheses after each class: `write-once` (replication), `private` (not managed), `write-many` (delayed update), `result` (delayed update), `synchronization` (distributed locks), `migratory` (migration), `producer-consumer` (eager object movement), `read-mostly` (replication and broadcast updating), and `general read-write` (Berkeley ownership).

In addition to traditional mechanisms for maintaining coherence in distributed shared-memory systems (replication, invalidation, migration, and remote load/store), the authors of the scheme propose using *delayed update*. When a process modifies a shared data item, the system does not immediately send the new value to remote servers to update their copies. Instead, it postpones sending until the next synchronization and then sends all changes in one package.

Simulation studies[28] show that for a given application (such as Quicksort) this method can decrease bus traffic by more than 50 percent relative to the conventional hardware write-update, and by more than 85 percent relative to the write-invalidate mechanism.

This scheme is classified as (D:d,S:s, R:r,A:a,L:g,G:f,B:p,P:*,U:*,C:*).

EXPLORING AND COMPARING SCHEMES

Using a consistent classification approach lets us formalize the classification to consider individual schemes as points in an abstract criteria space. The formalization of our classification lets designers see major differences in existing schemes as well as explore gaps in the criteria space for not-yet-conceived solutions. In the formal view, coordinates within the abstract multidimensional criteria space correspond to the chosen criteria; the values on these coordinates correspond to the classes. The number of criteria determines the number of dimensions of the space. The number of classes per criterion determines the number of discrete values that exist on the corresponding coordinate. If the set of criteria is independent, the space is orthogonal (not the case in our classification).

If the abstract space is described with a sufficient number of criteria and corresponding classes (for example, all 10 criteria for the schemes described here), it is considered *complete;* that is, each solution corresponds to one point. In the preceding survey, we used a complete space.

If the abstract space is described with a subset of criteria or classes, it is considered *reduced*; each point may correspond to more than one existing solution. Figure 1 shows a reduced space for the first three criteria in Table 1 (dynamism, selectivity, and restrictiveness).

Frequently, a reduced space is sufficient to see major differences among selected schemes. If it is not, typically, a significant criterion was not considered, or the multiple solutions in one point are practically the same, such as version control[20] and timestamps.[21] It may also be that solutions are semantically similar, such as timestamps[21] and one-bit timestamps,[24] but different in implementation, yielding different performance and/or complexity. Figure 1 shows only one representative solution per point, although in reality some points correspond to several solutions. For example, the point (s,s,r) contains not only version control,[20] but also the programmable cache scheme,[19] timestamps,[21] the life span strategy,[22] the generational approach,[23] and one-bit timestamps.[24]

With this classification, designers can detect that some solutions are similar (because they belong to points close to each other), even though their authors declare them as different. They can also use the classification in exploratory studies. If a combination of attributes (an *n*-element tuple) corresponds to a nonexistent solution, it is a *free point*. The corresponding solution may make no sense or not be particularly useful in the desired setting, but it may also stimulate research into classes of schemes that offer new levels of performance. For example, we believe it is worth searching for a new solution in the plane of adaptive static schemes. Knowing expected access dynamics to a properly declared shared object, the compiler can predict and embed the appropriate protocol to maintain the coherence of that object.

One can explore major differences in existing schemes as well as gaps in the criteria space for not-yet-conceived solutions.

Also, in exploratory studies, researchers can evaluate if the criteria for a proposed method are sufficiently broad to describe them by associating the new method with a free point in the criterion space; if not, the proposed criteria set may be further enlarged. We believe it may also be possible to roughly estimate a solution's performance solely on the characteristics of the classes that describe it—without analyzing the solution details.

PERFORMANCE PARAMETERS

Various parameters will impact the performance of these schemes, especially parameters related to workload. Several evaluation studies are useful. There is not enough space to cover them in any depth here, but we urge readers to look them up and read further. A perusal will show you how you can use different techniques to evaluate software solutions.

♦ *Min and Baer*.[30] Min and Baer compare a hardware scheme based on a centralized full-map directory[5] with their timestamps scheme.[21] The comparison is based on a simulation driven by real address traces. The authors conclude that the miss ratio is about the same for both schemes, the write traffic is considerably higher with the software scheme, and the network traffic is higher with the hardware scheme.

♦ *Tartalja and Milutinović*.[27] We analyze how the restrictiveness of invalidation conditions affects processing power using a modified Archibald-Baer model of the probabilistically synthesized workload. The modifications consist of introducing new parameters that model the spatial and processor localities and a new operating system model. We propose a selective and conditional self-invalidation class of schemes and compare them on a bus-based multiprocessor simulator. The preliminary results demonstrate the advantages of restrictively applied invalidation, even in systems with only 16 processors.

♦ *Owicki and Agarwal*.[8] The authors use the mean value analysis model to compare the performance of four representative schemes: *Base*, a scheme with no coherence maintenance, which defines the upper limit for performance; *No-cache*, a very conservative scheme that does not allow the caching of shared data;[7] *Software-flush*, a scheme based on static program analysis and insertion of the `Flush` instruction;[18,19] and *Dragon*, one of the best hardware schemes based on snooping and write-broadcast. The results show that software schemes are more sensitive to workload-related parameters than hardware snoopy schemes.

♦ *Adve and colleagues*.[3] The authors compare hardware directory-based schemes and software static schemes. Their analytical method starts from a general program behavior model based on the access dynamics for different classes of shared data. For these classes, they give the contours of the constant ratio of software to hardware efficiency method. From these contours, they identify the parameter domains where the software scheme exhibits an advantage over the hardware scheme, and vice versa. For example, software schemes outperform hardware schemes in handling migratory data; in maintaining of read-write data objects the two classes of schemes are comparable.

These studies give good insight into the real performance differences between software and hardware schemes for different values of technology- and application-related parameters. Especially important is that for some workloads software schemes demonstrate better performance. We expect that in real implementations, this performance advantage is even slightly higher because the complexity of hardware support for software schemes is relatively low. Consequently, VLSI systems that use software schemes potentially have a slightly better internal timing, and operate with a slightly faster system clock.

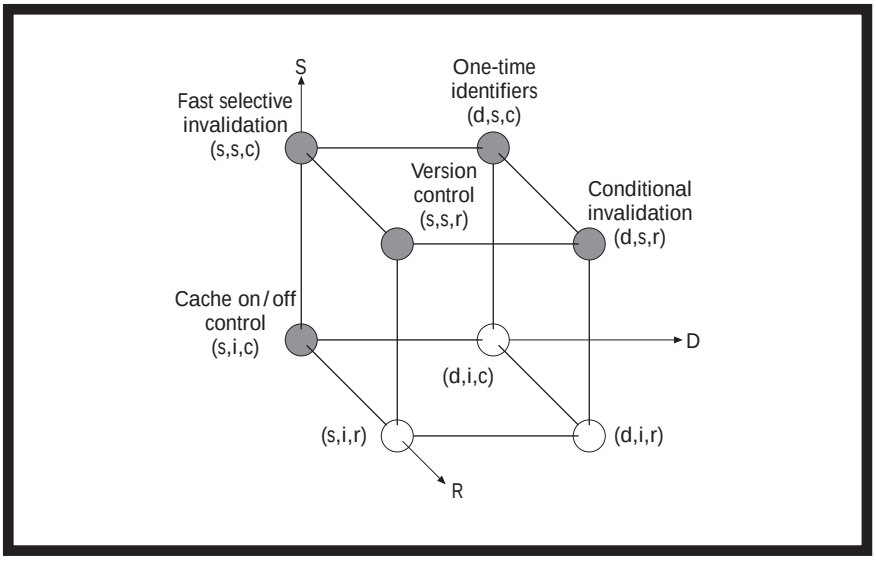


Figure 1. *An example of a reduced abstract criteria space for the first three criteria in Table 1: dynamism: static (s) or dynamic (d); selectivity: indiscriminate (i) or selective (s); and restrictiveness: conservative (c) or restrictive (r).*

The proposed classification makes it easier both to see the characteristics of existing approaches and to anticipate the appearance of new ones. We believe the classification and its formalization will make it easier to see directions for new research. The free points in the criterion space can serve as guides toward new solutions, much as the periodic table aided chemists in discovering new elements.

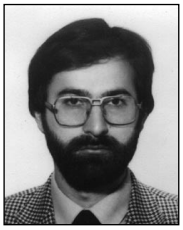
The performance evaluation efforts we have presented include both analytical (mathematical models) and empirical (simulation models) studies. Similar results of several studies point to the relative advantage of software schemes for certain workloads. This conclusion should encourage further research in software-based cache coherence. ◆

REFERENCES

1. M. Tomasević and V. Milutinović, *The Cache Coherence Problem in Shared Memory Multiprocessors: Hardware Solutions*, IEEE Comp. Soc. Press, Los Alamitos, Calif., 1993.
2. I. Tartalja and V. Milutinović, *The Cache Coherence Problem in Shared Memory Multiprocessors: Software Solutions*, IEEE Comp. Soc. Press, Los Alamitos, Calif., 1996.
3. S. Adve et al., "Comparison of Hardware and Software Cache Coherence Schemes," *Proc. 18th Int'l Symp. Computer Architecture*, ACM Press, New York, 1991, pp. 298-308.
4. S. Min and J.-L. Baer, "Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps," *IEEE Trans. Parallel and Distributed Systems*, Jan. 1992, pp. 25-44.
5. L. Censier and P. Feautrier, "A New Solution to Coherence Problem in Multicache Systems," *IEEE Trans. Computers*, Dec. 1978, pp. 1112-1118.
6. J. Skepstedt and P. Stenström, "A Compiler Algorithm That Reduces Read Latency in Ownership-Based Cache Coherence Protocols," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques*, 1995, pp. 69-78. Available from the authors.
7. W. Wulf and C. Bell, "C.mmp—A Multi-Mini Processor," *Proc. Fall Joint Computer Conf.*, IEEE Comp. Soc. Press, Los Alamitos, Calif., 1972, pp. 765-777.
8. S. Owicki and A. Agarwal, "Evaluating the Performance of Software Cache Coherence," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1989, pp. 230-242.
9. A. Gottlieb et al., "The NYU Ultracomputer—Designing a MIMD, Shared-Memory Parallel Machine (extended abstract)," *Proc. Int'l Symp. Computer Architecture*, IEEE Comp. Soc. Press, Los Alamitos, Calif., 1982, pp. 27-42.
10. A. Gottlieb et al., "The NYU Ultracomputer—Designing a MIMD, Shared Memory Parallel Computer," *IEEE Trans. Computers*, Sept. 1983, pp. 175-189.
11. J. Edler et al., "Issues Related to MIMD Shared-Memory Computers: The NYU Ultracomputer Approach," *Proc. Int'l Symp. Computer Architecture*, IEEE Comp. Soc. Press, Los Alamitos, Calif., 1985, pp. 126-135.
12. K. McAuliffe, *Analysis of Cache Memories in Highly Parallel Systems*, Tech. Report 269, Courant Inst. Mathematical Sciences, New York Univ., New York, May 1986.
13. W. Brantley, K. McAuliffe, and J. Weiss, "RP3 Processor-Memory Element," *Proc. Int'l Conf. Parallel Processing*, IEEE Comp. Soc. Press, Los Alamitos, Calif., 1985, pp. 782-789.
14. G. Pfister et al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. Parallel Processing Conf.*, IEEE Comp. Soc. Press, Los Alamitos, Calif., pp. 764-771.
15. A. Veidenbaum, "A Compiler-Assisted Cache Coherence Solution for Multiprocessors," *Proc. Int'l Conf. Parallel Processing*, IEEE Comp. Soc. Press, Los Alamitos, Calif., 1986, pp. 1029-1036.
16. R. Lee, P.-C. Yew, and D. Lawrie, "Multiprocessor Cache Design Considerations," *Proc. Int'l Symp. Computer Architecture*, ACM Press, New York, 1987, pp. 253-262.
17. R. Lee, *The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors*, Tech. Report 670, Center of Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, 1987.
18. H. Cheong and A. Veidenbaum, "A Cache Coherence Scheme with Fast Selective Invalidation," *Proc. 15th Int'l Symp. Computer Architecture*, IEEE Comp. Soc. Press, Los Alamitos, Calif., May 1988, pp. 299-307.
19. R. Cytron, S. Karlovsky, and K. McAuliffe, "Automatic Management of Programmable Caches," *Proc. Int'l Conf. Parallel Processing*, Penn State Press, University Park, Pa., 1988, pp. 229-238.
20. H. Cheong and A. Veidenbaum, "A Version Control Approach to Cache Coherence," *Proc. Int'l Conf. Supercomputing*, ACM Press, New York, 1989, pp. 322-330.
21. S. Min and J.-L. Baer, "A Timestamp-Based Cache Coherence Scheme," *Int'l Conf. Parallel Processing: Vol.1*, Penn State Press, University Park, Pa., 1989, pp. I23-I32.
22. H. Cheong, "Life Span Strategy—A Compiler-Based Approach to Cache Coherence," *Proc. Int'l Conf. Supercomputing*, ACM Press, New York, 1992, pp. 139-148.
23. T.-C. Chiuch, "A Generational Algorithm to Multiprocessor Cache Coherence," *Proc. Int'l Conf. Parallel Processing*, CRC Press, Boca Raton, Fla., 1993, pp. 20-24.
24. E. Darnell and K. Kennedy, "Cache Coherence Using Local Knowledge," *Proc. Int'l Conf. Supercomputing*, ACM Press, New York, 1993, pp. 720-729.
25. A. Smith, "CPU Cache Consistency with Software Support and Using One Time Identifiers," *Proc. Pacific Computer Comm. Symp.*, 1985, pp. 142-150. Available from the authors.
26. D. Cheriton, G. Slavenburg, and P. Boyle, "Software-Controlled Caches in the VMP Multiprocessor," *Proc. 13th Annual Int'l Symp. Computer Architecture*, IEEE Comp. Soc. Press, Los Alamitos, Calif., 1986, pp. 366-374.
27. I. Tartalja and V. Milutinović, "An Approach to Dynamic Software Cache Consistency Maintenance Based on Conditional Invalidation," *Proc. Int'l Conf. System Sciences*, IEEE Comp. Soc. Press, Los Alamitos, Calif., 1992, pp. 457-466.
28. J. Bennett, J. Carter, and W. Zwaenepoel, "Adaptive Software Cache Management for Distributed Shared Memory Architectures," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE Comp. Soc. Press, Los Alamitos, Calif., 1990, pp. 125-134.
29. J. Protić, M. Tomasević, and V. Milutinović, "Distributed Shared Memory: Concepts and Systems," *IEEE Parallel and Distributed Technology*, Summer 1996, pp. 63-79.
30. S. Min and J.-L. Baer, "A Performance Comparison of Directory-Based and Timestamp-Based Cache Coherence Schemes," *Proc. Int'l Conf. Parallel Processing*, Vol. 1, Penn State Press, University Park, Pa., 1990, pp. 305-311.

Igor Tartalja is a teaching/research assistant in computer engineering in the School of Electrical Engineering at the University of Belgrade, where his current research interests include memory consistency models, system software support for shared-memory multiprocessors and distributed systems, and heterogeneous processing. His PhD thesis was on the dynamic software maintenance of cache coherence in shared-memory multiprocessors.

Tartalja received a BSc, an MSc, and a PhD from the University of Belgrade, all in computer engineering.



Veljko Milutinović is on the computer engineering faculty in the School of Electrical Engineering at the University of Belgrade, Serbia, Yugoslavia. He has published more than 50 articles in IEEE periodicals, and has participated in a number of industrial designs. His research interests are in advanced microprocessor and multimicroprocessor architecture (SMP and DSM).

Milutinović received a BSc, an MSc, and a PhD from the University of Belgrade, all in electrical and computer engineering.

Address questions about this article to either author at Dept. of Computer Eng., School of Electrical Eng., University of Belgrade, POB 816, 11000 Belgrade, Yugoslavia; {etartalj, emilutiv}@etf.bg.ac.yu. For a tutorial on the subject see http://ubbg.etf.bg.ac.yu/~emilutiv/.