An Analysis of Degenerate Sharing and False Coherence[1]

Abstract

False sharing reduces system performance in distributed shared memory systems. A major impediment to solving the problem of false sharing has been that no satisfactory definition for this problem exists. In this paper we provide definitions for several types of degenerate sharing, including false sharing. We also provide an algorithm that computes the cost of unnecessary coherence (false coherence) in a shared memory system. Finally, we provide a counter intuitive example demonstrating that the elimination of degenerate sharing can reduce performance.

Keywords: False sharing. Cache Coherence. Distributed shared memory.

1 Introduction

Past research has attempted to define and quantify a degenerate form of sharing in shared memory systems known as *false sharing*. Since false sharing can have a significant impact on shared memory performance, we studied the existing literature to determine the cause of false sharing and how we could eliminate it from our distributed shared memory system. Unfortunately, the existing research relies mainly on ad hoc methodologies. We could not even find an acceptable definition for false sharing. This led us to develop an architecture independent off-line algorithm that, given a memory reference string with additional semantic information, computes the cost of sharing. Finally, we have discovered that, contrary to intuition, the elimination of degenerate sharing using published techniques may *reduce* performance in certain special cases.

1.1 Background

Many systems use *shared memory* for communication and data exchange. To reduce shared memory access cost, multiprocessor systems often replicate shared data in local caches. Communicating processes generally expect these caches to be *coherent*; that is, when one site reads a shared value, the last value written should be returned. To maintain coherence, the system must implement some mechanism that transmits memory updates between writing and reading sites. These *coherence transmissions* directly impact the performance of shared memory systems. Many shared memory systems rely on underlying memory management hardware to decide when a coherence transmission is necessary. Unfortunately, most memory management units (MMUs) only indicate that a transmission *may* be necessary. If the system uses the MMU to determine when a coherence transmission is necessary, there may be some unnecessary transmissions. Such excess communication will reduce system performance.

There are two component costs of a data coherence operation: a fixed amount of overhead and a per-byte transmission cost. The overhead is substantial, so the system usually processes a block of cached data (i.e., cache line or page) to amortize the overhead over many words of the data. While this amortization suggests that larger blocks are better, the per-byte transmission cost effectively limits the maximum block size. As the block grows larger, it becomes more likely that the system will transmit extra data that the receiving site will never read. Since *any* modification of the block forces a transmission of the entire block, selecting the size of the block requires care. Given expected data access patterns (i.e., locality of reference), there exists an optimal balance between block size and per-byte transmission cost.

As the block size increases, so does the likelihood that different objects will be colocated in the block. Unfortunately, colocating data structures produces undesirable side effects. Suppose two different data structures share a coherence block with one site exclusively accessing one of these objects and a second site exclusively accessing the other. Modifications to the first object forces an unnecessary coherence transmission when the second site accesses its object, *even though the second site will never use the data written by the first site.* This has been called the *false sharing problem.* We refer to this problem as the *false coherence* problem since the *cost* of the unnecessary coherence transmission is the principle concern.

Several recent papers discuss the significance and seriousness of the false coherence problem. References [1], [2], [3], [4], and [5] explore the effects of false sharing in NUMA architectures. References [12], [13], and [15] explore the effects of false sharing in tightly coupled multiprocessor systems employing hardware caches. Finally, references [6], [8], [12], and [15] attempt to quantify false sharing and describe how to eliminate false sharing within a set of programs. The evidence in past work clearly suggests that false sharing is a problem and eliminating false sharing can improve performance. However, currently there is no definition for false sharing that captures intuition, is easy to state, and can be easily computed. Without such a definition, any attempt to "solve" the false sharing problem will rely on ad hoc heuristics. To optimize system performance a thorough understanding of the problem is necessary.

1.2 Overview of Paper

In this paper we furnish three important results. First, we provide definitions for *reference strings*, *true sharing*, *false sharing*, *pseudo sharing*, *overwrite sharing*, *replacement sharing*, and *false coherence*. Second, we supply an off-line algorithm that, given a reference string with some additional semantic information, computes the cost of true and degenerate sharing. Finally, we show that simple attempts to eliminate degenerate sharing may actually reduce performance.

The remainder of this paper is organized as follows. Section two describes past attempts to define and measure false sharing. Section three provides basic definitions. Section four discusses architectural semantics and how this work applies to different shared memory architectures. Section five describes our experi-

1. This research was sponsered, in part, by NSF CCR-9209405.

mental apparatus and how we analyzed resulting reference strings to confirm the validity of our definitions. Section six presents an algorithm for measuring false coherence. Section seven discusses system optimizations and the problems with attempting to improve performance by reducing the effects of degenerate sharing. The final section presents our conclusions.

2 Past Work

Attempts to accurately define false sharing in the past have met with mixed success. Eggers and Jeremiassen wrote one of the earliest papers providing a definition for false sharing; they also provided several suggestions on eliminating false sharing [8]. Bolosky and Scott have published several papers on the topic; they state that "a precise definition and quantification of the problem have proven extremely elusive." Khera, LaRowe, and Ellis provide a relatively simple mathematical equation for false sharing that works for many significant cases, but is imprecise. The following sections describe their work.

2.1 Eggers and Jeremiassen

Eggers and Jeremiassen were among the first to attempt to reduce the effect of false sharing in shared memory programs [8]. They defined false sharing to be any coherence transmission caused by site access to a word that had not been modified since the last time that site possessed a valid copy of the block. A problem with this definition (as they acknowledge) is that it does not consider successive accesses to the same block. For example, suppose site 1 has a valid copy of a block containing the variables *a* and *b*, then site 2 writes to variable *a* and site 1 reads variable *b*. By Eggers and Jeremiassen's definition, this would constitute false sharing, the coherence transmission was unnecessary since site 1 still contains a valid copy of *b*. Suppose, however, that site 1 reads location *a* immediately after reading location *b*. Here the coherence transmission is necessary to communicate the value of *a* from the second site to the first. Unfortunately, their definition still charges the cost of the transmission to false sharing.

Given this problem, Eggers and Jeremiassen defined false sharing as the difference in cost between an application and a "hand-tuned" version of that same application. While they demonstrated some impressive performance improvements, this ad hoc definition is unsatisfactory because it is not possible to automate the computation of the cost of false sharing. Furthermore, it is difficult to determine if performance improvements occur by eliminating false sharing or by other changes made to the program.

2.2 Bolosky and Scott

Bolosky and Scott provide a survey of several possible definitions for false sharing [6]. Although the requirements of these definitions are difficult to meet in practice, Bolosky and Scott's work provides the basis for other definitions that follow. An important contribution they've made is to suggest a set of attributes which false sharing definitions should possess. Their three desirable attributes any definition of "false sharing" should possess are

- It must adequately capture the intuitive notion of false sharing,
- It must be mathematically precise, and
- It must be practically applicable.

To capture the intuitive notion of "false sharing" the authors state that the metric should be monotonically increasing as additional unrelated objects are co-located within a block. Further, false sharing should be zero when the block size is one word[2]. To be mathematically precise, the definition should be capable of being stated as a theorem and proven. In particular, this proof should take the form of an equation or algorithm that describes how to compute the cost of false sharing. It must be practically applicable in the sense that the algorithm described above must be tractable. It would be of little value to attempt to use a program that is NP-complete, or worse.

Bolosky and Scott's paper discuss some problems with defining false sharing. They provide a survey of several definitions and describe the problems (based on the criteria above) with those definitions. These definitions include *One Word Block Definition, Full Duration False Sharing, Interval Definition, Heuristic Interval Selection, The Hand Tuning Method,* and the *Cost Component Method.*

The One Word Block Definition compares the cost of executing a program against the cost of the same program executing on a machine with one word coherence blocks. When using single word coherence blocks the system will, in theory, transfer the fewest number of words. However, the high overhead of network transmissions actually increases the per-byte transmission costs as blocks get smaller. For one-word blocks, this overhead component may be much greater than the time to transmit a single word. A system with an eight word block size might suffer from false sharing and still run faster than a single word block system because there is far less overhead cost associated with each transmitted word. According to Bolosky and Scott, intuition suggests that whenever one reduces false sharing system performance should improve. Therefore, they dismiss this model because it fails to capture the intuitive notion of false sharing. We shall show that, despite intuition, it *is* possible for performance to decrease as one eliminates false sharing. Therefore, their rejection of this model may be unwarranted.

Full Duration False Sharing assumes that variables within a coherence block are either shared or not shared throughout a given memory trace. Unfortunately, it is unlikely that a coherence block is falsely shared throughout execution; rather, true sharing may occur in a block during one phase of execution and false sharing may occur during a different phase. As such, Full Duration False Sharing fails the intuition criterion because it is too restrictive.

2. We shall see that this definition of "intuitive" is unsatisfactory for any reasonable definition for false sharing.

The Interval Definition assumes that each site can write to a partially inconsistent block and the system can, when necessary, merge inconsistent blocks together. By ensuring that each read operation has the most recently written data, it is possible to relax consistency at certain points and reduce the number of unnecessary coherence transmissions. Unfortunately, computing the maximal interval in which consistency is unnecessary is not (known to be) tractable. Therefore, this method is not practical.

The Heuristic Interval Selection relaxes the optimality requirement in the Interval Definition to produce an algorithm that is tractable. Systems such as Munin [7] use this technique to reduce false sharing. Unfortunately, the Heuristic Interval Selection method underestimates the amount of false sharing by a considerable amount since this method provides only a loose lower bound on the amount of false sharing [6].

Another problem with the Interval Definition and the Heuristic Interval Selection techniques is that they require the ability to merge inconsistent blocks. This merge operation also reduces system performance. Munin, for example, saves a copy of an old page, computes the differences between the old and updated page, and then transmits the differences that the receiving site merges into its copy of the page. Although this works well with Munin because of its write update policy, the cost of merging could be prohibitive in other systems (e.g., those systems using a write invalidate protocol).

The Hand Tuning Method is the technique proposed by Eggers and Jeremiassen [8] (see the previous section) whereby they take a program, improve it, and compare the cost of the two. This technique relies upon the skill of the programmer and, as such, is mathematically imprecise.

The Cost Component Method, developed by Bolosky and Scott, attempts to analyze each of the component costs that contribute to false sharing based on the size of a coherence block, how much of the block needs to be transmitted, the overhead of transmitting the block, etc. Unfortunately, while this approach looks promising, it is incomplete; there are some (yet to be discovered) terms missing from their set of equations. Until someone can discover the missing terms, this method is too imprecise.

Bolosky and Scott conclude by stating the difficulties of producing a reasonable measurement for false sharing that possesses the three important attributes. They also point out the difficulty they had separating the cost of false sharing from other sources of performance loss in the system.

2.3 Khera, LaRowe, and Ellis

Khera, LaRowe, and Ellis [12] present a metric for false sharing operating under the following definition:

> *The essence of false sharing is that the contribution made by the individual words of a line toward the observed sharing pattern of the entire line is strictly less than full participation. Alternatively, the sets of processors accessing the individual words are proper subsets of the set of processors accessing the line and the level of false sharing is determined by that difference.*

This system compares the number of sites referencing a specific word in a cache line with the number of sites accessing the entire line, much like the One Word Block Definition[3]. While this ratio does provide an intuitive definition of false sharing, and can provide a metric for measuring the *level* of false sharing, this definition may not accurately reflect the *cost* of false sharing. Therefore, Khera, et. al., emphasize writes over reads since writes generate more coherence traffic. This write weighting improves the correlation between computed and empirical results.

A basic premise of their system is that the relative ordering of memory references should not significantly affect the overall metric. That is, two separate runs of the same program on the same hardware should produce the same measure of false sharing. Further, Khera, LaRowe, and Ellis wanted an *architecture independent* metric, one that would produce the same approximate results for different architectures (e.g., write update vs. write invalidate). As such, their algorithm produces an averaged result rather than an exact result.

One admitted drawback to their approach is that their measurements define windows of observation within traces and lose precise ordering information within each window. Choosing the window size and boundaries affects the result. In many respects, this method is similar to Bolosky and Scott's Heuristic Interval Selection method. Khera, et. al., rely on the user to supply the window size and boundaries for the system (i.e., the heuristics).

Although Khera, LaRowe, and Ellis' currently provide an off-line algorithm, they suggest that they will ultimately produce an on-line algorithm. Such an algorithm, even if imprecise, would be invaluable since it would allow the system to migrate processes to reduce false sharing.

3 Definitions

Existing work attempts to define false sharing as a function of resulting coherence transmissions. We have found such definitions to be ambiguous and often self-contradictory. Furthermore, different definitions produce different results on the same reference string; clearly they are measuring different symptoms of degenerate sharing. For our purposes, the current metrics are too imprecise, too costly to compute, or are too complex. We wanted definitions that matched our analysis and observations.

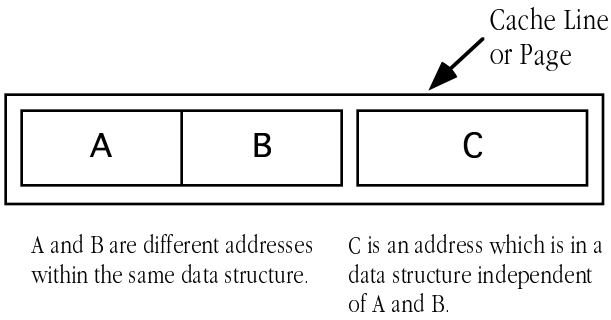
To develop our definitions we generated and analyzed hundreds of random memory reference strings. We discovered several subtly different sharing patterns that produce unnecessary coherence transmissions. Since the cause and, perhaps, the cost were different, we chose to provide separate definitions for each phenomenon rather than trying to provide a complex, all inclu-

3. Unlike the One Word Block Definition, Khera et. al. do not consider the cost of network overhead.

sive, definition. This decomposition let us easily articulate the definitions and simplified the analysis of the problem.

3.1 Simplified Memory Model and Reference Strings

The following simplified memory model provides the basis for our definitions[4]:



Simplified Memory Model

A, B, and C represent sets of one or more addresses within a single coherence block. Addresses in the set C are positionally and structurally independent from A and B, therefore A ∩ C is ∅ (the empty set) and B ∩ C is ∅. A ∩ B is also ∅, but the objects in these sets are members of the same data structure, for example, different rows in a matrix or different fields within a record.

- A *reference string* is a sequence of memory operations $X_1X_2X_3...X_n$ with each X_i taking the form $A_{adrs,site}$, $R_{adrs,site}$, or $W_{adrs,site}$ with "R" denoting a memory read operation, "W" denoting a memory write operation, and "A" denoting either type of memory access. *Adrs* is either lowercase *a*, *b*, or *c* and corresponds to one of the addresses in the simplified memory model above. *Site* is a numeric value denoting the site at which the memory access takes place.

We assume all blocks are invalid at the beginning of a reference string and that some anonymous site has written to each location in the block. This corresponds, for example, to a loader or memory allocator setting up the block before its first use.

Within reference strings we will use variables to denote addresses and sites. Lowercase *v, x, y,* and *z* are variables denoting a single site. Lowercase *i, j, k, m,* and *n* are variables denoting a single address. If some restrictions apply to a variable, then we will enclose the restrictions within square brackets; e.g., "[m ≠ i]" denotes any address *m* which is different from address *i* in the current reference string. Likewise, "[m ≠ i,j]" denotes any address *m* that is different from *i* and *j*. All occurrences of a variable within a reference string denote the same value. For example, consider the reference string:

$R_{i,x}\ R_{[m\neq i],z}\ W_{i,y}\ R_{[m\neq i],z}$

In this example the $R_{i,x}$ and $W_{i,y}$ operations access the same address. Since there is no requirement that $x \neq y$ the sites may be the same or they may be different. The two $R_{[m\neq i],z}$ entries *must* access a location other than *i* (they need not be the same address) and must read data at site *z*.

We will use the symbols "α" and "σ" to denote an arbitrary address or site, respectively. If these symbols appear as subscripts within different terms in the same expression, then each occurrence may represent a different value. For example, in the string "$W_{\alpha,4}\ R_{\alpha,5}$" the write operation could access one address and the read could access another. Similarly, the write and read operations in the reference string "$W_{a,\sigma}\ R_{a,\sigma}$" could occur at different sites.

The symbol "∞" will denote any arbitrary *adrs, site* tuple; for example, "A_∞" denotes a single arbitrary read/write operation at any address and site. If multiple "∞" subscripts appear within a reference string, the accesses need not be at the same site or address. The symbol "$R_\forall$" denotes a single read of the coherence block at all sites. This will force a coherence transmission if the block at a given site is invalid. Note that the address read is not one of *a, b,* or *c*, so this operation will not affect the type of sharing that occurs.

We will make liberal use of the metasymbols "(", ")", "*", and "|" (or/alternation) from automata theory to define sets of reference strings by regular expressions. For example, "$(A_\infty)^*$" denotes a string of zero or more read/write accesses at arbitrary addresses and sites.

Examples:

$$W_{a,1}\ R_{a,1}\ W_{a,1}\ R_{b,2}\ W_{c,3}$$

In the reference string above, site 1 writes to location *a*, reads from address *a*, and then writes to location *a*. Then site 2 reads from location *b*. Finally, site 3 writes to address *c*.

$$W_{a,2}\ (R_\infty\ |\ W_{[\alpha\neq a],\sigma})^*\ R_{a,[x\neq 2]}$$

This expression generates all reference strings that write to location *a* at site2, then have zero or more memory reads at any address or writes to *b* or *c*, followed by a read of location *a* at some site other than site 2.

The use of *a, b,* and *c* within our notation should not be underemphasized. These names provide certain semantic information that is not present in a typical memory trace (i.e., the physical relationship of these variables). The definitions that follow depend upon this additional semantic information. Fortunately, compilers for many languages can easily generate this information.

3.2 True Sharing

True sharing occurs when one site reads a value written by another site. This implies that a *necessary* coherence transmission takes place that copies the data between sites.

- *True sharing* occurs when a site reads an address it has never before accessed or when one site writes to an

4. Our definitions do not rely on this simplified model, but they are easier to explain in the context of this model.

address and another site reads that value before writing over it. Formally, true sharing occurs when a reference string takes the form:

[S1] $(A_\infty)^* W_{i,x} (R_{\alpha,[\sigma \neq v]} | W_{[\alpha \neq i],[\sigma \neq v]})^* (A_{[\alpha \neq i],v} | R_{\alpha,[\sigma \neq v]})^* R_{i,[v \neq x]} (A_\infty)^*$

A decomposition of this expression will make it easier to understand.

$(A_\infty)^*$:

This term, at the beginning and end of the reference string, simply allows true sharing to occur anywhere within a reference string. This term appears without further comment in the regular expressions for all forms of sharing.

$W_{i,x}$:

True sharing always begins with a write to some location *i*. Remember, our model assumes all coherence blocks are invalid when a program first loads into memory. Therefore, the loader or shared memory allocator could be the process responsible for this write.

$(R_{\alpha,[\sigma \neq v]} | W_{[\alpha \neq i],[\sigma \neq v]})^*$:

After site *x* writes to location *i*, any site except *v* can read variables in the coherence block. They may also write any location except *i*. Such accesses (especially writes) may cause the system to transfer data from site *x* to another site *y* that finally transmits the data to *v* when *v* accesses *i*. This will not, however, affect the cost of true sharing. A single coherence transmission to site *v* is necessary whatever the source of that transmission. Of course, no other site may write to location *i* after site *x*. Were this to occur, the true sharing would be between that other site and *v*, not *x* and *v*.

$(A_{[\alpha \neq i],v} | R_{\alpha,[\sigma \neq v]})^*$

Once site *v* accesses a location in the coherence block, a coherence transmission will take place. Since this transmission, ultimately, provides the value of *i* to site *v*, true sharing is responsible for this transmission. The term above allows site *v* to access any location other than *i* with other sites reading any values in the coherence block. No other site may write to a variable in the coherence block since that would invalidate the data at site *v* and force a new transmission. This could change the type of sharing that occurs. Note how this term handles the problem present in Eggers and Jeremiassen's definition.

$R_{i,[v \neq x]}$

Ultimately, site *v* must read the data at location *i* written at site *x*. More importantly, there can be only one coherence transmission to site *v* between the time site *x* wrote to *i* and site *v* reads *i*. The subexpressions between the $W_{i,x}$ and $R_{i,[v \neq x]}$ terms ensure this.

Intuitively, two sites truly share a variable when one site passes information to the other via that variable. Clearly, a coherence transmission is necessary when true sharing occurs in order to copy the value written from the writing site to the reading site[5]. The cost of this transmission is necessary and acceptable.

3.3 False Sharing

We have chosen to restrict our definition of false sharing. Existing definitions generally define false sharing as any memory access that results in an unnecessary coherence transmission. We have chosen to limit our definition of false sharing to a specific memory access pattern and provide additional definitions for other patterns that also produce unnecessary coherence traffic. By our definition, false sharing occurs when two sites access structurally unrelated objects within a coherence block.

- *False sharing* occurs when one site contains a valid copy of a coherence block, a second site writes to an object in that block, then the first site accesses a different, structurally independent object within that same coherence block, and there are no other writes between the two operations. Formally, false sharing occurs whenever a reference string takes one of the forms:

[F1] $(A_\infty)^* A_{k,x} (R_{\alpha,\sigma} | W_{[\alpha \neq c],[\sigma \neq x]})^* W_{a,[z \neq x]} (R_{\alpha,[\sigma \neq x]})^* A_{c,x} (A_\infty)^*$

[F2] $(A_\infty)^* A_{k,x} (R_{\alpha,\sigma} | W_{[\alpha \neq c],[\sigma \neq x]})^* W_{b,[z \neq x]} (R_{\alpha,[\sigma \neq x]})^* A_{c,x} (A_\infty)^*$

[F3] $(A_\infty)^* A_{k,x} (R_{\alpha,\sigma} | W_{[\alpha \neq c],[\sigma \neq x]})^* W_{c,[z \neq x]} (R_{\alpha,[\sigma \neq x]})^* A_{a,x} (A_\infty)^*$

[F4] $(A_\infty)^* A_{k,x} (R_{\alpha,\sigma} | W_{[\alpha \neq c],[\sigma \neq x]})^* W_{c,[z \neq x]} (R_{\alpha,[\sigma \neq x]})^* A_{b,x} (A_\infty)^*$

These four expressions are essentially identical; they handle all the possible permutations of *a*, *b*, and *c* that can produce false sharing. A single explanation for [F1], therefore, will suffice.

$A_{k,x}$:

The $A_{k,x}$ term ensures that site *x* has a valid copy of the block. If site *x* did not have a valid copy of the coherence block prior to this access, the $A_{k,x}$ term will force a coherence transmission and site *x* will now have a valid copy of the block.

$(R_{\alpha,\sigma} | W_{[\alpha \neq c],[\sigma \neq x]})^*$:

This expression allows all read operations and it allows sites other than *x* to write anywhere except *c*. Writes to *c* may not occur since such a write would produce true sharing rather than false sharing.

$W_{a,[z \neq x]}$:

The $W_{a,[z \neq x]}$ term is the *last* write to the coherence block by a site other than *x*. This invalidates the block at site *x*.

$(R_{\alpha,[\sigma \neq x]})^*$:

Once some site has written to location *a*, invalidating the block at site *x*, other sites may read data in the coherence block.

5. Technically, a coherence transmission is *not* necessary if the write operation does not change any values in the coherence block (i.e., it writes the same values already present at the target addresses). We will not consider this case here.

$A_{c, x}$:

When site *x* accesses location *c* a coherence transmission occurs because the block at site *x* is invalid. Since $A_{c, x}$ accesses *c*, not *a*, the transmission from site *z* is unnecessary.

Note that there is some ambiguity between the definitions for true sharing and false sharing. That is, the intersection of the languages described by the true and false sharing regular expressions is not empty. We shall explore this ambiguity later and discuss how to eliminate it.

One important difference between our definition and most other definitions is that the sites must access structurally independent objects within the coherence block. If two objects are structurally independent one can move either object around in memory without affecting the position of the other. Conversely, if two objects are not structurally independent, moving one object requires that they both move as a single block. For example, you can place two integer variables, I and J, at any two arbitrary locations in memory. However, A[1][4] and A[5][8], elements of the same array, are not structurally independent. Moving either one implies that you must move the entire array which includes the other object. Likewise, two fields of a record (structure) are structurally dependent since moving one field around in memory implies that you are moving the entire structure.

3.4 Pseudo Sharing

We define a new term, *pseudo sharing*, to describe "false sharing" of structurally related objects. For example, two sites accessing different rows of the same matrix or different fields within a record (or structure) suffer from pseudo sharing.

The significant distinction between pseudo and false sharing concerns the techniques one uses to eliminate them. False sharing is generally easier to eliminate than pseudo sharing by simply rearranging objects in memory. To eliminate pseudo sharing often requires algorithmic changes, process migration, or the use of indirection [8] [12].

- *Pseudo sharing* occurs when one site has a valid copy of the coherence block, another site writes to a location within the block, and then the first site accesses a structurally related location. These accesses force a coherence transmission from the second site to the first even though the first site's variable was not affected by the write at the second site. Formally, pseudo sharing occurs whenever a reference string takes one of the following forms:

[P1] $(A_\infty)^* A_{i, x} (R_{\alpha,[\sigma\neq x]} \mid W_{[\alpha\neq b],[\sigma\neq x]})^* W_{a, [z \neq x]} (R_{\alpha, [\sigma \neq x]})^* A_{b, x} (A_\infty)^*$

[P2] $(A_\infty)^* A_{i, x} (R_{\alpha,[\sigma\neq x]} \mid W_{[\alpha\neq b],[\sigma\neq x]})^* W_{b, [z \neq x]} (R_{\alpha, [\sigma \neq x]})^* A_{a, x} (A_\infty)^*$

The description of these expressions is identical to that for false sharing except sites *x* and *z* access structurally related variables (a and b).

3.5 Overwrite Sharing

Overwrite sharing occurs when one site writes to a variable and that value is never read before another site overwrites it.

- *Overwrite sharing* occurs when two sites write to an address without some site reading that address between the two writes. Formally, overwrite sharing occurs whenever a reference string has the following form:

[V1] $(A_\infty)^* W_{i,x} (R_{i, x} \mid A_{[\alpha \neq i], \sigma})^* W_{i, [z \neq x]} (A_\infty)^*$

$W_{i,x}$:

This is the initial write whose value will be overwritten.

$(R_{i, x} \mid A_{[\alpha \neq i], \sigma})^*$:

As long as no site other than *x* accesses location *i*, no sharing occurs. This expression matches all strings that do not access location *i* except at site *x*.

$W_{i, [z \neq x]}$:

This is the write operation which overwrites the value written earlier. Since no site (except, possibly, *x*) has read location *i* since the initial write, a coherence transmission from site *x* to site *z* may be unnecessary.

Overwrite sharing is significant because two processors accessing the same location can produce unnecessary coherence traffic. Some previous definitions for false sharing assume optimal coherence traffic in a one word coherence block system. Clearly this may not be the case. Although we believe that overwrite sharing is not very common, any definition whose optimal solution includes unnecessary coherence transmissions will not be satisfactory.

3.6 Replacement Sharing

Replacement sharing occurs if a site has an invalid copy of a coherence block and it writes to a location without first reading the data at that location. This will force a coherence transmission (since the block was invalid) but the transmission will be unnecessary since the write operation does not use the data provided by the coherence transmission. Typically, replacement sharing occurs when one site reads a variable and then another site writes to that same variable.

- Replacement occurs when one site writes to an address in an invalid coherence block after another site has read that address. Formally, replacement occurs whenever a reference string takes the following form:

[R1] $(A_\infty)^* W_{i, [y \neq x]} (R_{\alpha, [\sigma \neq x]} \mid W_{[\alpha\neq i],[\sigma\neq x]})^* R_{i, [\sigma \neq x]} (R_{\alpha, [\sigma \neq x]} \mid W_{[\alpha\neq i],[\sigma\neq x]})^* W_{i,x} (A_\infty)^*$

$W_{i, [y \neq x]}$:

This expression ensures that the coherence block at site *x* is invalid. In particular, another site has written a value to some location i. Remember, the coherence block is invalid at the beginning of the reference string, this write could be by the loader or shared memory allocator.

$(R_{\alpha,\ [\sigma \neq x]} \mid W_{[\alpha \neq i],[\sigma \neq x]})^*\ R_{i,\ [\sigma \neq x]}\ (R_{\alpha,\ [\sigma \neq x]} \mid W_{[\alpha \neq i],[\sigma \neq x]})^*$:

This term guarantees that some site other than *x* reads location *i*. Note that there are no writes to location *i* within any string this expression matches.

$W_{i,x}$:

This term overwrites the value written by $W_{i,\ [y \neq x]}$ earlier. Since there was at least one read operation between the two writes, overwrite sharing does not occur. However, since site *x* has not read location *i* prior to this write, the coherence transmission to site *x* may be unnecessary.

3.7 False Coherence

The extra cost associated with degenerate sharing is of concern to system designers. One problem with existing definitions has been a confusion between the cause of false sharing and the cost of false sharing. Our analysis indicates that although degenerate sharing often increases the amount of unnecessary coherence traffic, often there is no cost to such sharing and sometimes the presence of degenerate sharing can *reduce* coherence traffic. This is one reason past attempts to provide a uniform definition for false sharing have been unsatisfactory.

- *False coherence* is an unnecessary coherence transmission from one site to another. The coherence transmission is unnecessary because the receiving site already has the data it needs or coherence is unnecessary, e.g., when overwrite sharing occurs. The coherence operation occurs because the transmitting site modified some address in the coherence block, even though the receiving site does not use the modified value.

An algorithm that computes the amount of false coherence resulting from an arbitrary reference pattern is complex. We cannot simply union the regular expressions for false, pseudo, overwrite, and replacement sharing to obtain a single regular expression that generates all reference strings involving false coherence. That language is ambiguous because a single sequence in the reference string may exhibit several types of sharing simultaneously. For example, if one site writes to location *a* and a second site reads locations *c* then *a*, false sharing occurs as defined by regular expression [F1]. It takes exactly one coherence transmission to move the data from the first site to the second. Since site 2 accesses both locations, we could attribute the cost to true or false sharing. The transmission is necessary (to communicate the change to *a*), so this is true sharing. There are many examples of ambiguity which occur in reference strings. Section Five describes how we handle this ambiguity and charge the cost of a coherence transmission to an appropriate form of sharing.

3.8 Summary

The following table summarizes, in order of precedence, the definitions of the forms of sharing appearing in the previous sections and describes their important features:

Table 1: Summary of the Different Types of Sharing

Form	Salient Feature
True Sharing	Write at one site followed by read at another site.
Overwrite Sharing	Write at one site followed by a write to the same address at another site. No read operation between the two writes.
Pseudo Sharing	Write at one site that invalidates the block at a second site, followed by an access of a structurally related object at that second site.
False Sharing	Write at one site that invalidates the block at a second site, followed by an access of a structurally unrelated object at that second site.
Replacement Sharing	Write at one site followed by a write at a second site.

4 Architectural Semantics

Definitions for degenerate sharing and false coherence should be relatively independent of the underlying machine architecture. Ideally we would like to use a false coherence metric to compare architectures. If the metric is sensitive to architectural differences, it will not serve well in this task. There are two key architectural issues that may have an effect on the measurement of a system: the coherence protocol and the coherence semantics.

The coherence protocol determines how the system propagates writes. When a system uses the *write invalidate* protocol, any site writing data to a shared block invalidates that block on the other sites. Upon accessing the block, these other sites *pull* the new data from the invalidating site. In a *write update* protocol, sites that write to the block immediately transmit the update to other sites sharing the block.

Coherence semantics may also impact the metric. Does a reference string have the same cost if it is issued on a system that supports strict, sequential, weak, processor, release, or another form of coherence? (See [10] for a survey of memory coherence models.) Consider the reference string:

$W_{a,3}\ R_{a,1}\ W_{a,1}\ R_{a,2}$

If the $R_{a,2}$ operation reads the data written by $W_{a,3}$ rather than $W_{a,1}$ the results would be different from those systems that maintain sequential coherence.

To simplify our definition, we have chosen to base it on a write invalidate sequential coherence architecture. To show that the definition is architecture independent, we will show how to convert a reference string on one architecture to an equivalent reference string on a write invalidate sequential coherence machine.

Theorem 1: Write Invalidate Transformation. Any reference string produced on a machine using the write update protocol can be transformed to an equivalent reference string on a write invalidate machine.

We offer the following informal proof by construction:

The write update (WU) protocol transmits a copy of the coherence block to all interested sites at the end of the modification. The write invalidate (WI) protocol only transmits an invalidation request after modification and waits for the other sites to request a copy of the page. A write on a WU machine is equivalent to a write on a WI machine followed by a read at all other interested sites. Therefore, we can transform a string of the form:

$$\ldots W_{x,y} \ldots$$

on a write update machine to a string that produces the same result on a WI machine:

$$\ldots W_{x,y} R_{\forall} \ldots$$

The $R_{\forall}$ operand forces all sites to read the cache line. This forces a coherence transmission to each site on a WI architecture, forcing the same result as a WU machine.

•

Theorem 2: Coherence Semantics Transformation. Any reference string produced on an architecture with less than sequential coherence can be rewritten as a reference string that, on a machine with sequential coherence, would produce the same result.

We provide the following informal proof by construction:

In the sequential coherence model, any read of an address that occurs after a write to that same address returns the value last written. In a reference string of the form $W_{a,3} \ldots W_{a,1} R_{a,2}$ the read operation reads the data written at site 1, not site 3 (the read is *globally performed*. See [10]). As long as the coherence model in use obeys this rule, no transformation of the reference string is necessary. If it is possible to read stale data, that is, if $R_{a,2}$ above reads the data written by the $W_{a,3}$ operation, then we can transform the reference string to a comparable one by moving the read operation to just before the last write following the write whose value the read operation fetches. In the example above, assuming there were no other writes to address *a* between $W_{a,3}$ and $W_{a,1}$, you could transform the string to $W_{a,3} \ldots R_{a,2} W_{a,1}$ that would produce the same result on a machine employing the sequential coherence model.

•

An application of theorem 1 and theorem 2 to any reference string can transform that string to one that produces the same result on a write-invalidate, sequential coherence architecture. Therefore, we will assume, without loss of generality, the use of a machine supporting write invalidate and sequential coherence.

5 Reference String Analysis

To develop an algorithm which would compute the cost of false coherence for a given reference string we began by analyzing the data in the reference string. Through this analysis we determine when coherence transmissions occur and whether they were necessary.

5.1 Experimental Setup

We developed our definitions for degenerate and true sharing by studying sample reference strings and exploring the patterns present in those strings. At first we considered instrumenting some existing applications to capture shared memory references. However, we were concerned that such applications would generate a very limited set of reference strings that might influence our analysis. So we wrote a short Icon program[6] to generate a set of random reference strings for our analysis. The hundreds of random strings we worked with included far more patterns than one would expect from a few standard applications.

The first version of the Icon program simply generated random reference strings. After manually analyzing these strings, we began modifying the program to perform the analysis automatically. This stepwise refinement ultimately led to a program which not only generated the reference strings, but computed the coherence costs as well. For interested individuals, this Icon program is available via anonymous ftp.

5.2 Analysis

Since the number of possible reference strings is infinite, we could not enumerate and analyze them. To overcome this, we organized the reference strings into *equivalence classes*. Two reference strings belong to the same equivalence class if they exhibit the same sharing patterns and have the same cost. The set of equivalence classes is small and easy to enumerate. Upon isolating these equivalence classes, we could compute the costs attributable to coherence and assign the costs to various types of sharing.

A reference string provides a globally sequential description of memory activity. To compute coherence costs we detected the points at which a coherence transmission occurs within a reference string. Since coherence transmissions are only necessary when one site writes and another site accesses a block, the first

6. We chose the Icon programming language because it was applicable to our work (pattern matching) and the resulting code ports with no changes to Unix, PC/MS-DOS, Macintosh, and many other machines. Icon is freely available from the University of Arizona.

step is to insert boundaries into the reference string across which a coherence transmission occurs. These boundaries separate the reference string into *groups*. A group is a sequence of memory accesses at the same site. Consider the following reference string:

$W_{a,1}\ R_{a,2}\ W_{b,2}\ R_{a,1}\ R_{c,3}\ W_{a,3}\ W_{a,1}\ R_{b,2}\ R_{c,2}\ R_{a,2}\ R_{a,2}\ R_{a,1}\ W_{a,1}$

Grouping the memory references by site produces the following:

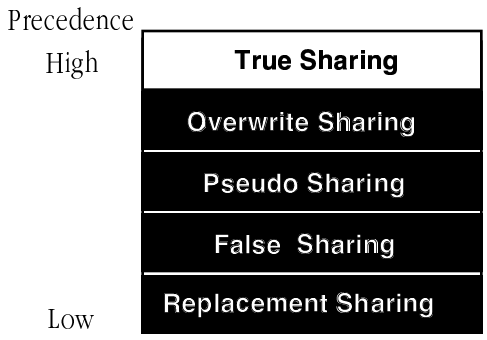
$(W_{a,1})(R_{a,2}\ W_{b,2})(R_{a,1})(R_{c,3}\ W_{a,3})(R_{a,1})(R_{b,2}\ R_{c,2}\ R_{a,2}\ R_{a,2})(R_{a,1}\ W_{a,1})$

There will be no more coherence transmissions than there are groups since there are no transmissions within a group and there is, at most, one transmission into any single group[7]. There may, of course, be fewer transmissions than there are groups. For example, the "$(R_{a,1}\ W_{a,1})$" term at the end of the string above does not require a coherence transmission since site 1 already contains a valid copy of the block.

Note that it is possible for several types of sharing to occur simultaneously between two groups. For example, consider the following reference string:

$$(W_{a,1})(R_{c,2}\ R_{b,2}\ R_{a,2})$$

False sharing occurs because site 1 writes to *a* and site 2 reads *c*. Similarly, pseudo sharing occurs because site 2 also reads location *b*. Finally, true sharing occurs because site 2 also reads location *a*. Although several forms of sharing occur simultaneously, there is only one coherence transmission between the two sites. Therefore, we cannot attribute the cost of one transmission to each form of sharing. Although we could evenly split the cost between each form, we have chosen to charge the cost to only one. By assigning a precedence to each form of sharing we resolve the ambiguity by attributing the cost to the form with the highest precedence. Our precedences are the following:



True sharing has the highest precedence. If true sharing occurs between two sites a coherence transmission will be necessary. In the previous example, true sharing coexists with false and pseudo sharing. However, eliminating the pseudo and false sharing would not improve performance. A single coherence transmission remains necessary to support true sharing. In effect, false sharing and pseudo sharing that coexist with true sharing require no additional coherence traffic. Note how this precedence solution handles the problem encountered by Eggers and Jeremiassen [8].

In the absence of true sharing, degenerate forms of sharing generally produce false coherence. Since the total cost of false coherence is of primary concern, attributing such costs to the various forms of degenerate sharing may seem like extra work. However, knowing the source of the false coherence may help you eliminate it. Therefore, this classification is useful.

Ambiguity also exists between the various forms of degenerate sharing. In the following reference string, overwrite, pseudo, and false sharing all coexist:

$$(W_{a,1})(R_{c,2}\ R_{b,2}\ W_{a,2})$$

Again we use precedence to attribute the cost of the coherence transmission to a single form of sharing. In the example above overwrite sharing occurs because it has the highest precedence.

Although the precedence ordering is somewhat arbitrary, this ordering was chosen based on our experiences analyzing reference strings. This precedence ordering produced a good mix of sharing types within the random reference strings generated by our Icon program. The algorithm, which follows in the next section, is easily modified to change the precedence levels or even list all concurrent forms of sharing.

6 A False Coherence Algorithm

Producing a tool which measures false coherence is an important step toward optimizing programs suffering from false coherence. The following is an algorithm that computes the cost in "coherence transmissions" for an input reference string.

[1] Assume that just before the reference string, some site, not appearing in the reference string, writes to all locations in the block. This invalidates the data at all sites and provides an initial write to each location.

[2] Break the reference string up into groups (by site) and apply the following steps to each group:

[3] For each group, set *Current Group* to the next available group and repeat steps 3..9.
Current Site ← *Current Group*'s site.

[4] If the current site contains a valid copy of the data, then
The cost is zero.
If a write occurs in the group, invalidate all other sites.
Go to step 3.
Comment: If we get beyond step four, the current block must be invalid.

[5] Check for true sharing.
If within *Current Group*, or a following group at the same site with no writes between the two groups, there exists an $R_{i,CurrentSite}$ with no preceding $W_{i,CurrentSite}$, *and*

7. Assuming perfect transmissions. We will not consider the cost of retransmission due to error in this paper.

within some previous group there exists a $W_{i,[x\neq CurrentSite]}$, *and*
there is no $A_{\alpha,\ CurrentSite}$ between the two groups above, then
True sharing occurs.
If there are any writes in the current group, invalidate all other sites
go to step 3.
Comment: If we get beyond step five, we must have some form of degenerate sharing.

[6] Check for overwrite sharing.
If within *CurrentGroup* there exists a $W_{i,\ CurrentSite}$ with no preceding $R_{i,CurrentSite}$ *and*
the last access to location *i* in a previous group was of the form $W_{i,\sigma}$ then,
Overwrite sharing occurs.
Invalidate all other sites.
goto step 3.

[7] Check for pseudo sharing.
If within *CurrentGroup* there exists an $A_{i,\ CurrentSite}$, *and*
between the last write to location *i* there exists a $W_{j,[x\neq CurrentSite]}$ such that i=*a* and j=*b* or i=*b* and j=*a*, then
Pseudo sharing occurs.
If there were any writes in the currrent group, invalidate all other sites.
goto step 3.

[8] Check for false sharing.
If within *CurrentGroup* there exists an $A_{i,\ CurrentSite}$, *and*
between the last write to location *i* there exists a $W_{j,[x\neq CurrentSite]}$ such that i=*a* or *b* and j=*c*, or i=*c* and j=*a* or *b*, then
False sharing occurs.
If there are any writes in this group, invalidate all other sites
goto step 3.
Comment: If you get past point eight, then the current group contains only write operations.

[9] If none of the above hold, the result must be replacement, so
Replacement sharing occurs.
Invalidate all other sites
goto step 3.

This algorithm operates in O(n) time, where *n* is the number of memory accesses in the reference string. Therefore, this algorithm obeys Bolosky and Scott's "practically applicable" criterion.

As noted earlier, this algorithm depends upon the additional semantic information associated with variables *a*, *b*, and *c*. However, the definition of true sharing does not require this semantic information. Therefore, one can still compute the cost of true and degenerate sharing without this semantic information. This is easily achieved by replacing steps six through nine above with the following step:

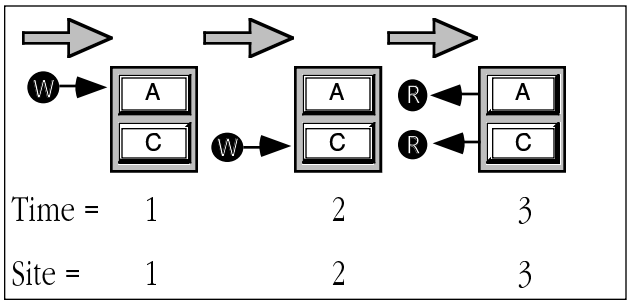
[6] If this is not true sharing, it must be some form of degenerate sharing.
False coherence occurs.
If there are any writes in the current group, invalidate all other sites.
goto step 3.

With this modification, the algorithm given corresponds to traditional definitions for false sharing.

7 Reducing False Coherence

Since false coherence generates unnecessary network traffic, intuition suggests that eliminating the source of false coherence will eliminate unnecessary network traffic and, therefore, improve system performance. Eggers and Jeremiassen [8] have suggested several ways to eliminate false sharing for exactly this reason. Unfortunately, shared memory systems do not always behave intuitively. In certain cases, eliminating one form of degenerate sharing may *increase* the coherence traffic.

Consider the following diagram:



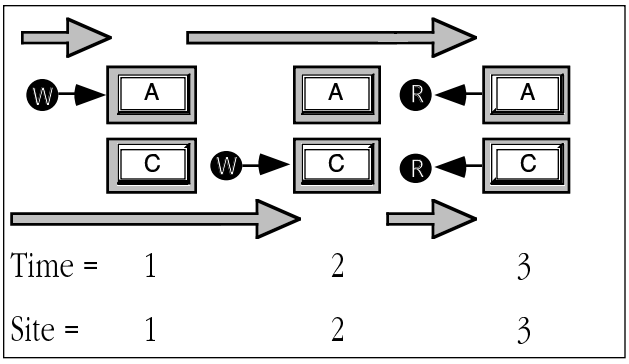
At time T=1 site 1 writes to some variable *a*. If we assume that the block at site 1 was invalid before the write, then a false coherence transmission will occur since a write operation always produces some form of degenerate sharing. This write operation will also invalidate the coherence block at all other sites.

At time T=2, site 2 writes to some variable *c*. Once again, a false coherence transmission takes place because of either false sharing (site 1 wrote to a structurally unrelated variable) or overwrite sharing (if some other site wrote to *c* before T=1). Once again, this write operation invalidates the coherence block at all other sites.

A time T=3, site 3 reads variables *a* and *c*. This requires a coherence transmission from site 2. Since site 3 is reading the variables written by site 1 and site 2, true sharing occurs and the coherence transmission is necessary.

There are a total of three coherence transmissions in this example (denoted by the gray arrows). The first and second are false coherence transmissions, the third results from true sharing. Suppose we attempt to reduce the unnecessary coherence traffic by eliminating the possible false sharing which occurs at site 2.

We can easily do this by moving *c* into a different coherence block. This produces the following memory activity:



As before, the write to location *a* results in degenerate sharing and, therefore, a false coherence transmission. The write to location *c* at site 2 still produces a false coherence operation. Note that moving *c* to a separate coherence block has *not* reduced the number of false coherence transmissions. True, false sharing may not exist; however, it has been replaced by overwrite or replacement sharing. Finally, site 3 reads the *a* and *c* variables. Since they are in separate blocks the system uses two coherence transmissions (*c* from site 2, *a* from site 1) to update these variables. The total is four transmissions: two due to false coherence and two resulting from true sharing. This fails Bolosky and Scott's intuition constraint. By colocating objects we reduce the coherence traffic. This is a problem with Bolosky and Scott's definition of "intuitive," not with the definition of false coherence. We, however, are unwilling to dismiss this definition because real reference strings produce non-intuitive results. When discussing the Heuristic Interval Selection Bolosky and Scott allude to this problem when they discuss the possibility that a maximal interval could have a negative value [6]. If we consider the extra coherence transmission the "cost of merging" modified values, this problem fits in well with Bolosky and Scott's experiences.

There is another problem confronting those who wish to eliminate unnecessary coherence traffic: different executions of the same parallel program generally produce different reference strings that may affect the number of false coherence transmissions. Immediately after a synchronization point one could expect different executions to produce similar reference strings. Shortly after the synchronization point, however, the two reference strings will quickly get out of phase. This means that any static (e.g., off-line) analysis may not provide sufficient information to properly optimize system performance.

8 Conclusion

This paper provides precise and intuitive definitions for *true sharing*, *false sharing*, *pseudo sharing*, *overwrite sharing*, *replacement sharing*, and *false coherence*. We also provide an O(n) algorithm that computes the cost of coherence transmissions and attributes these costs to true or degenerate sharing. We have also noted that optimizing system performance requires careful consideration; simply "eliminating the false sharing" may not provide an optimal solution.

Past attempts to quantify the effects of false coherence have relied on ad hoc methods. Our work provides a solid foundation upon which future research may rely. There are still some major problems to solve. As noted in the previous section, an off-line algorithm such as ours may not provide consistent results for different executions of the same program with the same input data. Furthermore, an off-line algorithm which computes the cost of false sharing can only suggest modifications to a program. An on-line algorithm would allow a program or operating system to modify its behavior, during execution, to reduce unnecessary coherence. We believe that our analysis of this problem is the first step in the development of such an on-line solution.

References:

[1] W. J. Bolosky. *Software Coherence in Multiprocessor Memory Systems*. Ph.D. Thesis, TR 456, Computer Science Department, University of Rochester, May 1993.

[2] W. J. Bolosky, R. P. Fitzgerald, M. L. Scott. *Simple But Effective Techniques for NUMA Memory Management*. Proceedings of the Twelfth ACM Symposium on Operating System Principles, pages 19-31, Dec 1989 In ACM SIGOPS Operating Systems Review 23:5.

[3] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. *NUMA Polices and Their Relation to Memory Architecture*. Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 212-221, Apr 1991. Also, ACM SIGARCH Computer Architecture News 19:2, ACM SIGOPS Operating Systems Review 25 (special issue), and ACM SIGPLAN Notices 26:4.

[4] W. J. Bolosky and M. L. Scott. *A Trace-based Comparison of Shared Memory Multiprocessor Architectures.* TR 432, Computer Science Department, University of Rochester, July 1992.

[5] W. J. Bolosky and M. L. Scott. *Evaluation of Multiprocessor Memory Systems Using Off-line Optimal Behavior.* Journal of Parallel and Distributed Computing, 15(4):382-398, Aug 1992.

[6] W. J. Bolosky and M. L. Scott. *False Sharing and its Effect on Shared Memory Performance.* Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV), Pages 57-72, Sept. 1993.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152-164.

[8] S. J. Eggers and T. E. Jeremiassen. *Eliminating False Sharing*. Proceedings of the 1991 International Conference on Parallel Processing, I:377-381, Aug 1991.

[9] S. J. Eggers and R. H. Katz. *A Characterization of Sharing in Parallel Programs and its Applicability to Coherency Protocol Evaluation.* Proceedings of the 15th

Annual International Symposium on Computer Architecture, pages 373-383, May 1988.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.* Proceedings of the 17th International Symposium on Computer Architecture. 1990, pages 15-26.

[11] A. Gupta and W.D. Weber. *Cache Invalidation Patterns in Shared Memory Multiprocessors.* IEEE Transactions on Computers, 41(7):794-810, Jul 1992.

[12] V. Khera, R. P. LaRowe, and C. S. Ellis. *An Architecture-Independent Analysis of False Sharing.* TR 93-006, Computer Science Department, Duke University, October 1993

[13] R. P. LaRowe and C. S. Ellis. *Experimental Comparison of Memory Management Polices for NUMA Multiprocessors*. ACM Transactions on Computer Systems, 9(4):319-363, Nov 1991.

[14] W. Nitzberg and V. Lo. *Distributed Shared Memory: A Survey of Issues and Algorithms.* IEEE Computer, pages 52-60, Aug 1991.

[15] J. Torrellas, M. S. Lam, J. L. Hennessy. *Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates.* Proceedings of the 1990 International Conference on Parallel Processing, Volume II, pages 266-270, Aug 1990.