CS–1994–37

Factors Affecting False Sharing on Page-Granularity Cache-Coherent Shared-Memory Multiprocessors

Vivek Khera

Department of Computer Science

Duke University

Durham, North Carolina 27708–0129

December 1, 1994

Factors Affecting False Sharing on Page-Granularity Cache-Coherent Shared-Memory Multiprocessors

by

Vivek Khera

Department of Computer Science
Duke University

Approved: December 1, 1994

Carla Schlatter Ellis, Supervisor
Kishor S. Trivedi
Henry S. Greenside
Gopalan Nadathur
Joseph W. Kitchen

This is a reformatted version of the dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University.

Abstract

Efficiently supporting a shared memory paradigm in a large-scale multiprocessor generally involves some form of data caching. One of the drawbacks of caching shared data is the cost of keeping the multiple copies coherent. One source of unnecessary coherency overhead is caused by a problem known as *false sharing*. Unfortunately, the lack of a precise, universally accepted, definition of false sharing hinders research to detect and eliminate the problem.

We articulate our intuitive notion of false sharing and address the problems encountered in previous attempts at defining false sharing. We motivate the importance of a concrete measure by demonstrating that false sharing related coherence overhead comprises a significant portion of the coherence costs in real applications, especially when page-granularity coherence is required. An architecture-independent measure of the false sharing exhibited in a reference trace for cache lines of a specified size is proposed and evaluated experimentally.

The proposed measure attempts to summarize the false sharing impact by approximating some factors and discarding others. The evaluation of this formulation reveals that such summary statistics lose too much information to be of practical use in predicting performance. We use this work to motivate experiments to determine the relative importance of the various workload and architectural factors that affect coherence data traffic. The conclusion from these experiments is that the precise memory reference interleaving order is the most significant factor affecting false sharing coherence data traffic.

Our methodology is to use an execution-driven simulation of specific architectures and applications to generate memory reference traces. The traces are then analyzed off-line.

Acknowledgements

There are many people who have made my pursuit of this degree possible. Some helped academically and others socially.

First, I'd like to thank my parents for giving me the opportunities while I was growing up to explore my interests. Without their support and encouragement I would never even have attempted this.

Among the many friends I've made during my years in Durham are some of the finest people I know — they have helped make life enjoyable. Dave Kotz and Rick LaRowe as my academic "brothers" have provided many ideas and much guidance in my work, as well as being good friends. Thomas Alexander, Chris Connelly, and Apratim Purakayastha have been invaluable in helping me work through some of the rough spots while conducting this research.

Friday afternoon happy hours with Rick and the other AHH-ers, Dov Bulka, Dave Reed, Owen Astrachan, Jonathan Polito, Varsha Mainkar, and Jitendra Apte provided life with some of its most interesting moments. I thank Deganit Armon, Eric Anderson, and Marge Dietz for the long philosophical conversations about life, the universe, and everything. I also thank Eric for making me camp out for basketball season tickets, especially since I got to see all those awesome home games! Finally, for throwing interesting twists and turns into my life, I want to thank Lars Nyland, Stacy Doyle, and Katya Prince. The Ultimate Frisbee gang at Forest Hills Park have also helped me enjoy my last year in Durham much more.

A very special thank you goes to my second "mom", Carla Schlatter Ellis, for supporting me through the ups and downs of the entire process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Efficiently supporting a shared memory paradigm in a large-scale multiprocessor generally involves some form of data caching. Data caching is an important technique to help reduce the effective memory reference latency and various forms of memory reference contention such as bus traffic and memory module contention. One of the drawbacks of caching shared data is the cost of keeping the multiple copies coherent.

One source of unnecessary coherency overhead is caused by a problem known as *false sharing* in large granularity cache units (e.g., multi-word cache lines or pages). False sharing is a result of co-location of unrelated data in the same line: The data may be used by different processors such that the line is shared among them but the individual data elements contained in the line are not each referenced by all these processors.

Informal definitions, such as this, attempt to capture intuitive notions of "bad" packaging of data relative to multiprocessor access patterns. Descriptions have varied considerably. At one extreme, informal false sharing definitions have included only *de facto* private data items residing together within a cache line (sharable, but actually accessed by different single processors). On the other hand, any access behavior falling short of a uniform, pure sharing pattern can be called false sharing. Unfortunately, the lack of a precise, universally accepted, definition of false sharing hinders research to detect and eliminate the problem.

In the next section, we describe previous attempts to characterize or solve the false sharing problem. We motivate the importance of false sharing by demonstrating the impact on coherency costs for real applications. In Section 2.1, we review some of the subtle issues that have made defining false sharing difficult and we articulate our intuition of false sharing. Section 2.3 formally defines our false sharing measures. In Chapter 3 we present our experimental methodology. In Chapter 4, we describe the sample programs we use in Chapter 5 and Chapter 6 to experimentally evaluate how well the measures capture our intuition and predict false sharing performance impacts. Finally, in Chapter 7 we determine the importance of the various workload and architectural factors on the impact of false sharing.

1.1 Evidence of the importance of false sharing

A first-order approximation to the cost of false sharing in the performance of a system is the amount of additional coherence traffic that it causes. Here we present results of a simple set of simulation experiments designed to measure the amount of coherence traffic caused by false sharing under typical cache coherence policies. These data serve to motivate our investigation of false sharing behavior by illustrating the seriousness of the false sharing problem and identifying some of the factors involved in finding a good definition.

The simulator used in our experiments is based on Tango [8] and is driven by the SPLASH

benchmarks [24]. We compare the performance of both update-based and invalidate-based coherence protocols to an optimal coherence mechanism. For these experiments we have written a machine description that is a simplified, parameterized NUMA[1] architecture in which the physical memories of each processor node act only as caches for the global virtual address space, sometimes referred to as COMA (Cache Only Memory Architecture). Each referenced virtual address is resident in the local memory cache of at least one processor node. To eliminate cache replacement effects, our simulator assumes infinite caches.[2] We simulate both 64-byte and 8192-byte cache line sizes. The 64-byte line size is reasonable in modern hardware-based caching systems, and the 8192-byte size is representative of systems based upon page-granularity caching (e.g., Galactica Net [27]).

Our primary metric is the number of data bytes transferred among the processors[3] in order to ensure cache coherence. For the invalidation-based coherence protocol, coherence traffic occurs when a processor misses on a line that would have been in the cache if it were not invalidated by a write to that line by some other processor. Our measurements do not include the traffic required to transmit invalidate messages when a shared line is modified. Coherence traffic for the update-based protocol consists of update messages sent to processors with copies of the line being modified. We count each destination independently, so that a single four-byte write to a shared location may result in the transfer of $4(n-1)$ data bytes if there are n processors sharing the updated line. Only data bytes are counted; all message overhead is ignored for both coherence protocols.

The optimal coherence policy that we simulate is based upon the following simple observation: When a value is written to a shared location, that value need only be propagated to processors that will actually read that value. It is obviously necessary to transfer a value written by one processor to any other that expects to read that value. Furthermore, it is easy to see that the data bytes transferred by this policy are sufficient, since any value that is never read by a processor cannot affect the outcome of that processor's computation. Since this policy generates both the sufficient and necessary coherence data traffic, it is optimal. A bit of reflection reveals that our optimal coherence policy is essentially an overhead-free invalidation-based coherence policy with single-word line size.

In Figure 1.1, we graph the results of our experiments. The data clearly show that the number of data bytes transferred to maintain coherence is significantly fewer for the optimal policy than for the four more realistic alternatives (note the logarithmic y-axis) in nearly all cases. Since the additional coherence traffic incurred by the realistic alternatives can be attributed to false sharing effects, these data clearly demonstrate the potential significance of false sharing, especially for page-based granularity caching systems.

1.2 Background

False sharing has been recognized as a serious problem in several recent studies. Some of the earliest uses of the term "false sharing" appear in discussion of memory management for NUMA architectures where page granularity migration and replication are employed to take advantage of the faster local memory access times [4, 2, 16]. False sharing has been blamed as a cause of increased coherency overhead in multiprocessor hardware caches with increasing line sizes in workload characterization studies of shared memory reference patterns [12, 11].

Techniques for ameliorating the false sharing problem have also been proposed in [7, 9, 1]. The solution provided by Munin [1, 6] addresses only the most conservative form of false sharing. Other proposals [7, 9] deal with the granularity of coherency which addresses one contributing

[1] Non-Uniform Memory Access time.

[2] This is a common simplification in cache studies.

[3] Note that we use "processors" and "caches" interchangeably when writing about the transfer of coherence data in a system.

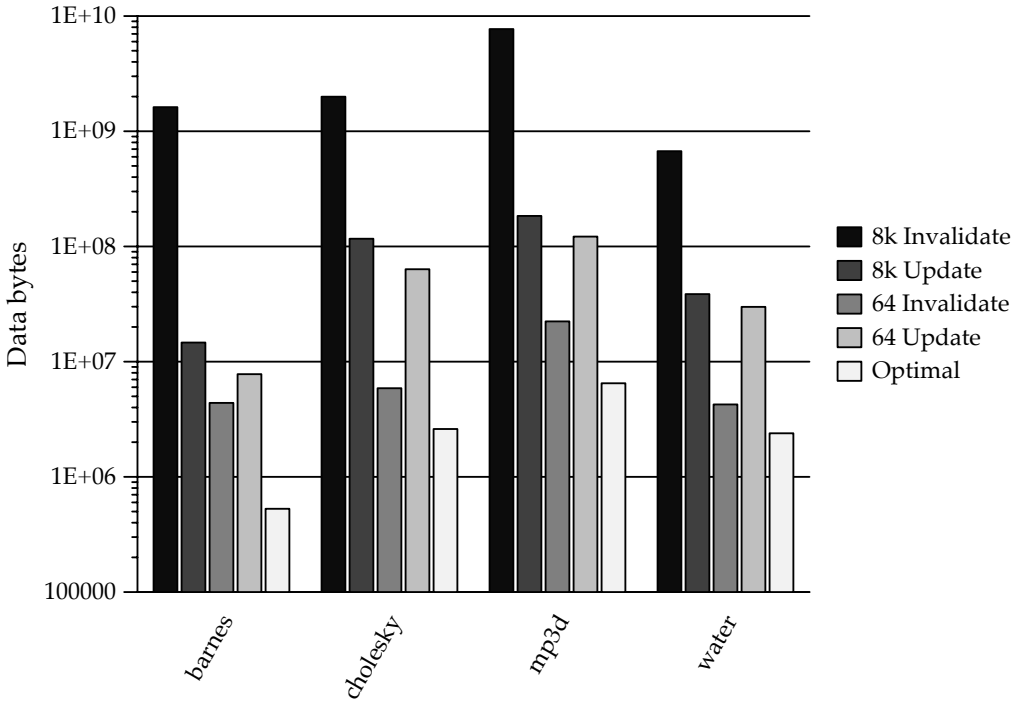

Figure 1.1: *Coherence data bytes transferred (real applications)*

factor without necessitating (or offering) a thorough understanding or definition.

Two recent papers explicitly deal with quantifying false sharing effects and assessing proposed optimizations aimed at eliminating them [26, 10]. An important contribution of these papers is that each offers a concrete measure of false sharing *impact*. In each case, this is the number of false sharing cache misses. The measures subtly differ in how false sharing misses are identified and distinguished from misses caused by true sharing. The implementations result in a different way of counting the misses to be blamed on false sharing and a different categorization when applied to the same reference pattern. In [26], the method involves running two trace-driven simulations, one with single-word cache blocks and one with multi-word blocks. In an infinite single-word cache (i.e., no replacement), all misses (other than cold misses) result from coherency operations caused by true sharing. Cache misses that are present in the multi-word simulation that have no corresponding miss in the single-word simulation are attributed to false sharing. Some true sharing misses in the single-word simulation may also be eliminated in the multi-word simulation by successful prefetching. This technique is not amenable to real-time detection by an operating system. The mechanism for classifying misses as false sharing misses in [10] is based on tracking the current state (shared *vs.* non-shared) of referenced words, as determined by recent invalidation history, in a multi-word granularity trace-driven simulation. Both versions of this measure have the drawback that they depend on a particular coherency scheme (i.e., invalidation protocols) for the definition of false sharing. A definition that is less architecture dependent is highly desirable.

The real focus of these papers is on eliminating false sharing effects. Torrellas *et al.*, in [26], discuss methods of reducing false sharing related cache misses. Their primary methods are to rearrange data such that variables that exhibit false sharing are placed on different cache blocks and to put variables protected by a lock on the same block as the lock. By applying

these optimizations to several applications, they were able to reduce the number of cache misses. Eggers and Jeremiassen [10] also propose techniques to eliminate false sharing in caches. Their transformations involve allocating data objects together that have similar sharing properties and the use of indirection.

Another effort aimed at defining and measuring false sharing is described in [3]. Bolosky and Scott consider a number of alternative formulations using the criteria that the definition should capture intuition, be mathematically precise, and be practical to apply. None of the alternatives is found to be satisfactory on all counts. However, this negative conclusion is in a sense inevitable since the intuitive notion the authors are trying to capture is not clearly articulated and the level of precision sought is unrealistic. Neither of the measures of false sharing used by Torrellas *et al.* [26] or Eggers and Jeremiassen[4] [10] were included in this study.

[4]Although other aspects of this work were cited.

Chapter 2

Proposed definition

Detecting false sharing automatically requires some sort of metric to be defined. The "I'll know it when I see it"[1] detection method obviously cannot be automated. Our goals are (1) to detect when such a problem exists, (2) to identify the specific root causes of the problem (e.g., the data structures that don't belong together in a page based on having radically different access patterns or the processor(s) responsible for generating a different access pattern), and (3) to propose solutions that address the inherent causes (e.g., repackaging data, migrating processes). We address the first two of these issues in this thesis.

2.1 Difficulty of defining false sharing

The characteristics we desire in a definition (to be translated into a concrete measure) of false sharing are that it should (1) capture intuition, (2) be as architecturally independent as possible and yet at least loosely predict performance impacts for various architectures, and (3) have practical application in solving the false sharing problem.

The primary difficulty with providing a precise, formal definition of false sharing that captures intuition has been that the intuition has not proved easy to articulate. The only easily described intuitive notion (*de facto* private data items co-located in a shared line) seems clearly too narrow, although it is an interesting special case that yields to creative solutions (e.g., Munin's merge capability [6]) and is universally accepted as an example of false sharing. This case can be illustrated by the following trace of references for a line consisting of four words accessed by four processors. We denote the referenced word by an upper case letter, A–D, and the processor making the reference by a numeric subscript, 0–3 (e.g., A_0 represents processor 0's reference to word A):

$$A_0,\ B_1,\ C_2,\ D_3,\ B_1,\ D_3,\ C_2,\ A_0$$

Now consider the following trace which we claim also exhibits false sharing:

$$A_0,\ A_1,\ C_2,\ C_3,\ B_0,\ B_1,\ D_3,\ D_2$$

In this case, all words are shared, but only by different subsets of all the processors referencing the line as a whole (processors 0 and 1 reference words A and B while processors 2 and 3 reference C and D).

We need to be able to articulate the intuition that identifies such examples as forms of false sharing in order to derive and interpret more formal definitions or measures. This appears to be the missing step in all previous work. Consequently, we offer the following statement:

[1] To paraphrase Supreme Court Justice Potter Stewart (though not to equate obscenity with false sharing).

> The essence of false sharing is that the contribution made by the sharing patterns on 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 the difference.

This statement covers various scenarios that we believe are examples of false sharing. It also translates into quantitative measures (in Section 2.3) that correspond to our intuitive feel for "more" or "less" false sharing in such scenarios.

Our second goal is architectural independence. As demonstrated in the previous section, using the difference between coherence costs of different architectures as the defining metric of false sharing is problematic is several ways. In particular, invalidation-based protocols and update-based protocols may yield very different results. One feature of such characterizations is a sensitivity to the precise ordering of references in a trace. The relative ordering of references from different processors is obviously a factor in quantifying false sharing impact, as illustrated by the difference between the following two reference strings under an invalidation-based protocol:

$$A_0,\ A_1,\ A_0,\ A_1,\ A_0,\ A_1$$

and

$$A_0,\ A_0,\ A_0,\ A_1,\ A_1,\ A_1.$$

The second reference string above is considered to be *sequentialized*: All references from a single processor occur prior to those of another processor. However, it is not clear that attending to minute reorderings between two traces that could be produced by the same program in an asynchronous parallel environment is desirable. We prefer a definition tied more to the program and line size and less to a particular execution of that program. Our measures define a window of observation in a trace and lose precise ordering information within that window. While choosing the window size is acknowledged to be an issue, losing some ordering information may be viewed as acceptable in that a particular trace is not attributed more precision than it is likely to deserve in a parallel system. It can also be argued that such summary information may be more practical for dynamic resource monitoring and management.

2.2 Definitions for factors

The following workload and architectural factors have been identified to affect the false sharing performance of a shared memory parallel program. All definitions for values that change during program execution implicitly cover only the interval of execution under study. The interval may be the entire execution of an application or may be some subset thereof. Intervals may be expressed either in units of time (such as microseconds) or in number of references (such as 5000 shared memory references).

Number of processors: The number of processors participating in the computation. One processor/memory pair constitutes a single "node".

Page size: The size, in bytes, of the virtual memory pages provided by the system under study. For systems with a fixed page size, this is an architectural parameter.

Remote memory reference latency: The number of cycles required to reference a word on another node. On some architectures, this is not constant. It is not defined for systems which only do block copy operations.

Page copy time: The number of cycles required to make a copy of a memory page on a node from another node. This may be shorter than the time needed to copy each word, since it can be achieved with a block-copy operation.

Coherency protocol: The two policies we consider for keeping data coherent across nodes are *invalidate* and *update*. Invalidate coherency means that when one processor writes to a shared page, all other copies on other nodes are marked as invalid. When a node containing a page marked invalid needs to reference that page, it must fetch a new copy from a node with a valid copy. In the update coherence policy, when a page is modified on a node, all other nodes with a copy of the page get a message containing the new value for the modified word. The specific implementation details of these policies are not relevant here, since we are only concerned with the volume of data traffic not the time it takes to perform the coherency operations.

Update issue time: The number of cycles required to send an updated value of a word from one node to another.

Update expire policy: In the update coherency policy, all nodes with a copy of a given page need to be updated when any of them modifies a word in the page. Without some mechanism for removing inactive copies, unnecessary updates result and space is wasted. The expiration can be based on either time or reference counts. We choose the latter: It is defined as a number indicating how many updates for a given page a node will accept before it discards the page from its memory if there have been no intervening local references to that page.

Processor set size: For any given word, the number of processors that read or write that word. For any given page, the number of processors that read or write any word in that page.

Number of reads and writes: For any given word, the number of reads (or writes) made by all processors to that word. For any given page, the number of reads (or writes) made by all processors to any word in that page.

Memory reference interleaving order: For a given word, the relative order in which each processor references that word in a global ordering.

If processor A references a word at absolute time t_A, and processor B references the same word at absolute time t_B where $t_A \leq t_B$, then the reference from processor A is before that from B. We denote this by the ordered list (A, B). The memory reference interleaving order is the ordered list resulting from the transitive closure of this relation for all references to the word from all processors.

We can extend this to the page or whole address space by replacing "word" in the paragraph above with "page" or "address space" respectively.

There are a few particular patterns of reference interleavings in which we are interested, primarily dealing with *locality of reference*. We base our definitions on the classical uniprocessor definition of locality from [13, Chap. 8].

Temporal run: If a word is referenced, it will tend to be referenced again soon by the same processor, without intervening references by another processor. The length of the run is the number of consecutive references by the processor.

With this definition, we attempt to capture the notion of a processor making exclusive use of a word for a period of time (locality in time). Thus, the memory reference interleaving order for a word would contain sequences of the form $(\ldots, B, A, A, \cdots, A, A, C, \ldots)$, i.e., containing a long sequence of references from only one processor. The following definitions deal with locality in space (memory locations).

Cycles: The word is actively shared. The memory reference interleaving contains repeated references from a particular processor with references from other processors intermixed. The length of a cycle is the number of intervening references. For example, the trace excerpt $(\ldots, A, x, x, A, x, x, A, x, x, A, \ldots)$ contains a cycle from processor A of length two (x indicates any processor other than A).

Thrashing: Extreme manifestation of cycles. The word is shared between a subset of the processors, and the reference interleaving order contains interwoven cycles from these processors, resulting in cycles from each of these processors with an average length one less than the number of processors in the subset. For example, if the three processors A, B, and C are thrashing a word or page, the trace would contain segments of the form $(\ldots, B, A, C, B, C, A, \cdots, B, A, C, \ldots)$, where the average length of the cycles is two. During the period of thrashing, no other processor references that word.

Unstructured: Memory reference orderings that do not have one of the above structures.

Working set: Traditionally, it is the set of pages in a process's virtual address space to which memory references have been made over some period of time [18]. We extend this definition to be the set of pages to which memory references have been made from any processor, and restrict the references counted to the shared data pages. Suggested periods of time are the last 10,000 instructions or 10ms [19].

Phase change: The reference patterns dramatically change between "phases" of execution, such as between initialization and computation phases of a numerical computation. Formally, it is when the rate of change in the working set for a given processor is higher than the local average rate of change. This may be exhibited as a sudden burst in the size of the working set.

Sharing participation: The degree to which the set of processors referencing a particular word in a page is different from the set of processors referencing any word in that page. This is quantitatively defined by the $\mathcal{F}$ metric in Equation 2.4. The $\mathcal{G}$ metric from Equation 2.6 quantifies the participation for the entire page.

The definitions presented for working set and phase change are provided only to be able to define analysis interval boundaries. The significance of each of these factors in false sharing impact is the subject of Chapter 7. Since our cost metric is based on data bytes transferred over the interconnection network, we keep the timing-based factors fixed throughout our experiments.

2.3 Measures for false sharing

The coherence traffic cost measure of the previous chapter quantifies the *impact* of false sharing, but only indirectly suggests the degree of false sharing caused by the particular composition and sharing patterns applied to the page or line. In this section, we define more direct measures for false sharing that capture the intuition expressed in Section 2.1. These are based on shared-memory reference traces but are independent of any coherence mechanism.

Initially, each referenced word is assigned a false sharing value. These values are then combined to calculate a measure for each line. This measure is useful in helping to isolate where false sharing is taking place.

We define our false sharing measure based on a basic unit of memory reference, which we will call a word (w). The size of a word is not defined here, and in fact may not be of fixed size. Typically, references come in two sizes of a single word (4 bytes) and a double word (8 bytes). We use the term *word* to refer to all atomic memory references, regardless of their actual size. Words that are not referenced are treated as if they do not exist, since unreferenced words have no direct effect on memory coherence costs.

The unit of memory coherence is a line (l) which is a set of words:

$$\begin{aligned} l_j &= \{w_i | \text{word } i \text{ is part of line } j\} \\ & l_j \cap l_k = \emptyset\ ,\ j \neq k \end{aligned} \tag{2.1}$$

In our discussion, we also refer to lines as pages, since page-granularity coherence motivated this work.

The number of references made to word w_i is denoted $w_{i,r}$ and the number of write-references (modifications) is denoted $w_{i,m}$. Similarly, we denote the total number of references to line l_j as $l_{j,r}$ and the number of write-references to the line as $l_{j,m}$. All of these counts are taken over the interval of time that is under study (this may be the entire duration of execution.)

The *processor set* of a word is the set of processors that reference the word over the time interval of interest:

$$W_i = \{\text{processors referencing } w_i\} \tag{2.2}$$

The processor set of a line is the union of the processor sets of the words in the line:

$$\begin{aligned} L_j &= \bigcup_{w_i \in l_j} W_i \\ &= \{\text{processors referencing } l_j\} \end{aligned} \tag{2.3}$$

Using these definitions, we can derive an expression for the false sharing that can be attributed to a particular w_i in l_j:

$$\mathcal{F}(i,j) = 1 - \frac{|W_i|}{|L_j|} \tag{2.4}$$

The key idea being that the greater the difference between the word's processor set size and the line's processor set size, the greater the degree of false sharing associated with that word. If a word has not been referenced, we do not use it in any further calculations.

For most cache coherence schemes, the primary cause of coherence overhead is due to write references. With an invalidation-based protocol, writes cause the invalidations that in turn can cause false sharing misses, and in an update-based protocol, it is the write references that cause false sharing updates. Thus, it is often useful to weight $\mathcal{F}(i,j)$ by the fraction of references to w_i that are writes:

$$\mathcal{F}'(i,j) = \mathcal{F}(i,j) \times \frac{w_{i,m}}{w_{i,r}} \tag{2.5}$$

so that words used in a mostly read-only fashion will have $\mathcal{F}'(i,j)$ close to zero, while words with high write-to-reference ratios will have $\mathcal{F}'(i,j)$ values closer to $\mathcal{F}(i,j)$.

The definition for false sharing associated with a given line is:

$$\mathcal{G}(j) = \sum_{w_i \in l_j} \mathcal{F}(i,j) \times \frac{w_{i,r}}{l_{j,r}} \tag{2.6}$$

which is just the weighted average of the false sharing measures of the individual words in the line. To see the importance of the weighting, consider a line with n words, one of which has a very high $\mathcal{F}(i,j)$ value and the rest with no false sharing (i.e., $\mathcal{F}(i,j) = 0$). Clearly, if there are very few references made to the one falsely-shared word, we would not consider the line to be heavily falsely-shared. On the other hand, if the majority of references to the line were to that one word, we would definitely consider the line to be falsely-shared. The weighting makes it possible to distinguish these situations from one another.

As in the per-word case, we also define a write-weighted version of this measure:

$$\begin{aligned} \mathcal{G}'(j) &= \sum_{w_i \in l_j} \mathcal{F}'(i,j) \times \frac{w_{i,r}}{l_{j,r}} \\ &= \sum_{w_i \in l_j} \mathcal{F}(i,j) \times \frac{w_{i,m}}{l_{j,r}} \end{aligned} \tag{2.7}$$

The weighting adjusts for the different contributions of each word to the overall false sharing associated with the line.

2.4 Summary

In this chapter we reviewed the intrinsic and previously encountered difficulties in previous attempts to define false sharing. We then presented our intuitive notion of false sharing and the characteristics that a good definition for false sharing must have. In preparation for defining our measure of false sharing, we defined the workload and architectural parameters that affect false sharing. Finally, we developed in detail the measures we propose as candidates to be evaluated for their usefulness in defining and predicting false sharing.

Chapter 3

Experimental Methodology

To evaluate the proposed measures, we need a testbed in which we can easily modify architectural parameters and collect memory reference traces. We need traces in order to accurately measure the actual false sharing, true sharing, and prefetching that occurs. Using a multiprocessor execution-driven simulator allows us these capabilities.

The simulation environment we use is Tango [8] with our own machine model description. Tango allows us to run each program multiple times with the same input and generate trace files which are based on the specific architectural parameters under study. Trace driven simulation would not suffice, as evidenced by the radical difference in execution times and number of memory references in the Barnes-Hut program from the SPLASH benchmark suite (see Section 6.1.1). In this program, a change in the type of memory coherence from update to invalidate caused the run time of the program to increase by more than three times. The potential inaccuracy of trace-driven simulations is one of the reasons for which Tango was originally written. Holliday and Ellis [14] discuss the accuracy of such studies.

3.1 The simulator

Tango simulations are built such that a single executable contains both the simulation environment and the application program. When the program is run, the simulator reads the necessary parameters from a configuration file, allocates the system resources it will need, and finally creates the parallel environment. It then transfers control to the main routine of the application.

Programs are written using the Argonne PARMACS macros [5]. At compile time, the appropriate hooks into the Tango libraries are inserted into the application object code. As much of the code as can be is directly executed by the host CPU. For code that makes shared-memory references, the Tango simulator code calls a routine in our NUMA simulator to determine how long the reference will take. At this time, the status of the referenced page is updated.

3.1.1 The machine we are modeling

The machine we model consists of multiple processor-memory pair nodes connected by an arbitrary network (Figure 3.1). The network is used to implement the coherent shared memory abstraction for running MIMD (multiple instruction, multiple data) programs. A MIMD program runs on multiple processors, with each processor progressing independently of the others, except at explicit synchronization points. Remote memory references are more expensive than local ones (NUMA) and each node's memory serves as a cache for the global virtual address space — there is no centralized global memory. Every reference must go to physical memory since there are no instruction or data caches. Each node is assumed to have enough physical

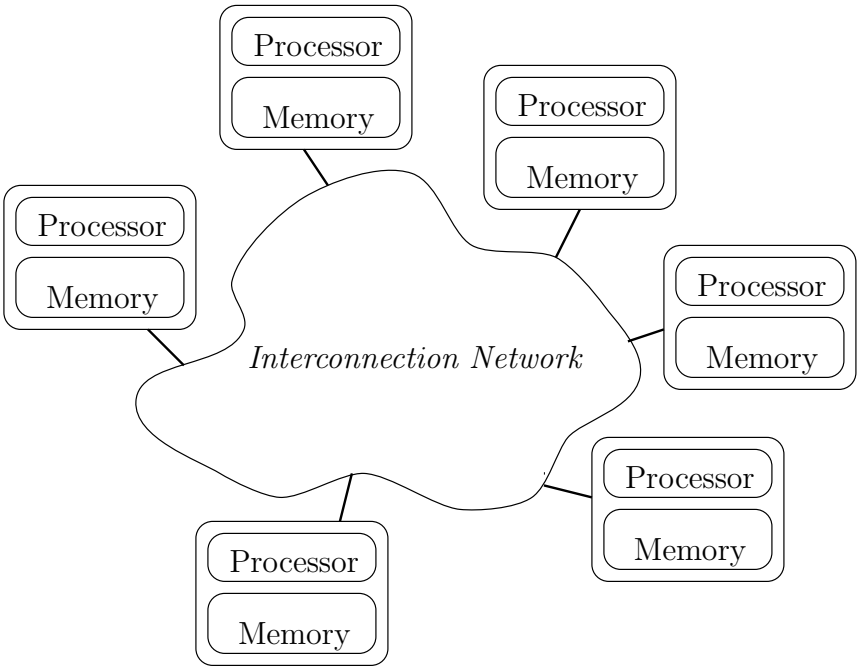

Figure 3.1: *Idealized NUMA shared memory architecture*

memory to hold all the virtual memory pages it will need so we don't have to worry about page-replacement policies.[1] For these studies, all remote references are handled by replicating the referenced page onto the local memory.

3.1.2 Architecture parameters

There are six options that modify the architecture of the machine being simulated. These parameters allow for a very large number of qualitatively different architectures to be evaluated. The options controlling the parameters are read from a configuration file by the simulator at start-up, along with the other standard Tango parameters that set the time needed to execute each instruction and reference local memory.

The simulator can run either an invalidate-based or an update-based memory coherency policy. Invalidate coherency means that when one processor writes to a shared page, all other copies on other nodes are marked as invalid. When a node containing a page marked invalid needs to reference that page, it must fetch a new copy from a node with a valid copy. In the update coherence policy, when a page is modified on a node, all other nodes with a copy of the page get a message containing the new value. This keeps all copies of the pages in the system identical. The coherency policy is selected by the CacheType parameter.

In either update or invalidate coherency, we use a sequential consistency model. Sequential consistency implies that any changes to global memory are visible to all processors simultaneously and before *any* future references are made. This affects our studies in that every coherency operation causes traffic. In other consistency models, the coherency operations can be delayed and batched together to reduce overheads when it is known that a particular word will not be used by the target node during the delay. Other memory consistency models are discussed in

[1] A common simplification in cache studies.

[20].

For the update memory coherency policy, another parameter limits the number of "useless" updates sent to a node. If a page on a node has not been used locally for a while, it is expired (marked as invalid) so future modifications to that page on other nodes will not send additional updates to the node. This reduces network traffic for pages which are no longer used by a given node. The threshold used to determine when to expire a page in this situation is determined by the following formula [17]:

$$C \times T_u = T_p \tag{3.1}$$

The value C is the count of the number of updates sent to the node for the page in question, T_u is the time required to send an update for one word, and T_p is the time needed to copy an entire page from another node. Since the latter two are fixed by the architecture, we can easily compute C for our simulations. When a node has received C updates for a page without referencing it locally, it will expire the page. The UpdThresh parameter determines the C value. If this value is set to zero, the pages are never expired in this manner.

The size of the virtual memory pages can be set to any power of two from one word up to 8192 bytes (PageSize). The number of processors being simulated must also be set (NumNodes). Only one thread runs per processor node.

As part of its operation, the memory system will need to copy an entire page from one node to another. On a real machine, this can be done with a block copy if the hardware supports it, or with a word-by-word copy loop implemented in software. Two parameters in the simulator allow for maximum flexibility in deciding how block copies are done. The first parameter adjusts the size of a block transfer (BlockSize). This can be anywhere from a single word up to PageSize, but must divide PageSize evenly. The other parameter sets the time needed to transfer a single block across the network (BlockCost). Setting these parameters to a single word sized block and a transfer cost equal to that of a remote memory reference would simulate a machine with no hardware block transfer capability.

In our experiments, BlockCost is set to 200.0 and BlockSize is set to 64 bytes, except in the cases where the PageSize is four: BlockCost is set to 20.0 and BlockSize set to 4 bytes. CacheType, PageSize, and NumNodes are varied for the different experiments. For the 64-byte page size experiments, UpdThresh is computed as 10, and for the 8192-byte page size experiments, it is computed as 1280.

3.1.3 Detailed operation of the simulator

Every reference to shared memory results in Tango calling a routine in our NUMA simulator to compute the latency[2] of the reference. In this procedure we track the status of each word and page in the shared memory space for each node. Every page from the shared address space is in one of three states on each node:

unaccessed The page has never been accessed by this processor.

valid There is a copy of this page in the local memory.

invalid The copy of the page on this node has been invalidated.

The initial access to a page by a processor brings the page from unaccessed state to valid state and is counted as a cold miss. Data transfer costs associated with a *cold miss* are not used in our analysis. A reference to a page which is marked invalid for the processor is counted as an *invalidate miss* and brings the page to the valid state. If the reference is a write, then copies of the page on all other processors are marked as invalid for the invalidate protocol (by sending a message to each processor with a copy), or an update message containing the new value is sent to each processor holding a copy of the page for the update protocol.

[2] time to complete

Messages are not really sent—just a count of how many and of which type is kept. There is no overhead cost associated with these messages since we are interested primarily in the amount of data traffic over the network. However, since we know the exact count of messages delivered, we can add in this overhead at a later time during analysis.

The memory system simulated always does a page replication for non-local memory references. If a replication is necessary, the time taken to satisfy the read or write is equivalent to the time it takes to transfer $\mathsf{PageSize} \div \mathsf{BlockSize}$ blocks at BlockCost time for each. We do not model any contention for the transport network. If a reference is local because the page is in the node's local memory, the time taken is the same as any other local memory reference. In the update coherency case, once a page is replicated on a node it is always sent updates regardless of whether it will actually use the page again.

The output from the simulator is a trace of all references to the global memory space. The code and private data segment references are ignored since they are assumed to be in a private area of each node's memory and cannot affect the cost of data communication. The trace format consists of one machine-independent trace file per simulated processor, where each trace entry consists of the following fields:

- address space (shared or private)
- processor node number
- type (read or write)
- length (number of bytes)
- time
- address

During analysis, the traces from each processor are merged and sorted by reference time. This provides us with a trace of the global memory reference interleaving order.

3.2 Calculating true costs

The traces collected when running the simulations are used to calculate the actual false sharing and true sharing costs as well as the amount of benefit due to prefetching on a per-page basis. These calculations are done off-line since there is a considerable amount of state information necessary, and in some cases we need to know what the future references are.

3.2.1 Invalidate Coherency

The technique used is similar to the one used in [25], but we calculate the bytes transferred due to false sharing rather than just counting the false sharing misses. Two concurrent trace driven simulations are run: one is identical to the simulation that generated the trace (multi-word line size), and the other tracks the location and status of each shared word (single-word line size). On each shared memory reference, both simulations are updated, and the result (hit or miss) from each is compared.

In Table 3.1 are listed the four possibilities of the comparison. If the single-word simulation resulted in a hit and the multi-word simulation indicated a miss for a reference, then we would conclude that the miss was induced by false sharing, and thus add to the false sharing cost the amount of data we needed to transfer across the network (the size of a page). The other cases contribute nothing to the false sharing cost calculation. The simulator is capable of distinguishing a cold and true sharing miss by setting an additional flag.

single-word	multi-word	indication
HIT	HIT	normal — no data transfer
MISS	HIT	prefetch benefit
HIT	MISS	false sharing induced miss
MISS	MISS	cold or true sharing miss

Table 3.1: *Comparison chart for invalidate coherency*

3.2.2 Update Coherency

Costs for update coherency are calculated in a similar fashion. However, every write operation has potential for causing data traffic. On a write reference, if an update is sent to a node where the word was not or will not be used, the data transfer is charged as false sharing, otherwise it is true sharing. To determine if the word will be used, we need to look ahead in the trace. The page faults induced by pages that have been expired based on the setting of UpdThresh are charged as false sharing. These page faults must be charged as false sharing because they are caused by unused words — if the words were actively used, the page would not have been expired.

Chapter 4

Workload description

So far we have defined many terms and proposed summary measures to predict false sharing. We now describe the programs we use to evaluate the effectiveness of the proposed measures.

4.1 Synthetic programs

To evaluate the false sharing measures presented, a set of synthetic workload programs with simple and clear characteristics was developed. We chose a synthetic workload to do the initial studies because it is possible to create programs with well defined sharing patterns.

For each program, there are four variations (selected as run-time options):

normal No extra options are given to the program. Each word is referenced once, and there is no synchronization among the processors.

synchronized The program is run with all processors synchronizing on a barrier prior to processing each page.

multiple reference Each memory reference is executed a fixed number of times. In these experiments, this number is ten.

combined Both synchronized and multiple reference.

Each program was run on sixteen processors under the simulator to collect shared-memory reference traces. Only references to shared memory are captured, and only the references to the data pages are considered in their analysis.

In the next two sections, we describe in detail the programs we created for these experiments.

4.1.1 Varying only sharing participation

The program synth-FS generates a set of pages which have varying amounts of sharing participation ranging from full in the last page, to none in the first page. All other factors remain the same from page to page.

To understand how this program works, we present an example. For $N = 4$ processors, we have five pages. The first page has each processor reference exactly one word (all but the first four words remain unreferenced). The second page would have two processors reference one word each, and two processors reference the third word. The whole pattern of the number of processors referencing each word is shown in Table 4.1. Note that each row sums up to $N = 4$, and all the possible ways of making this sum are listed. For $N = 16$, there are 231 pages. In the multiple reference case, each word is referenced multiple times immediately, as each page is completely processed prior to a processor going to the next page.

page	procs/word			
1	1	1	1	1
2	1	1	2	
3	1	3		
4	2	2		
5	4			

Table 4.1: *Reference pattern for synth-FS with four processors*

```
static void
worker(void)
{
    int i, j, n;

    for (i = 0; i < numPages; ++i) {
        int high_proc = 0;
        if (synchronized)
            BARRIER(glob→BA, numProcs); /* synchronize at each page */
        for (j = 0; j < numProcs; ++j) {
            high_proc += sums16[i][j];
            if (my_id < high_proc) {
                for (n = 0; n < nrefs; ++n)
                    /* reference each word nrefs times from each processor */
                    glob→page[i][j] *= numProcs; /* read/modify/write */
                break;          /* go to next page. */
            }
        } /* for j */
    } /* for i */
} /* worker() */
```

Figure 4.1: *Memory reference code for synth-FS*

The main code executing on each processor is listed in Figure 4.1. The array sums16 contains the list of ways of summing to 16 (like the example above for 4 processors), and is pre-computed for efficiency. The reference to glob→page[i] is a pointer to the start of data page i.

4.1.2 Varying sharing participation and processor set size

The program synth-FS+PSS allocates one page per processor. Each page is referenced by a different number of processors in the most conservatively false-shared manner. Each word is referenced by exactly one processor: page 0 is referenced only by processor 0, page 1 is referenced by processors 0-1, each using every other word, page 2 is referenced by processors 0-2, each using every third word, etc. For the multiple reference per word case, each processor loops multiple times over all the words on a page it is to reference so each word is referenced once before the first word in the page is referenced again.

4.2 Real applications

The applications we use to represent real workload are taken from the SPLASH [24] benchmark suite of shared-memory parallel applications. These programs were written and collected specifically to aid computer designers in developing and evaluating new architectures.

These programs are by no means representative of all parallel programs, but they do provide a variety of complete, actual programs with which to evaluate our proposed measures. The four programs we evaluated perform scientific computations. All exhibit more read-sharing activity than write-sharing. Because we are running these programs under a parallel simulator, we are limited in the problem size we can solve with each. The data sets input into the programs are not trivial, but they are not as large as a typical input might be on a real parallel computer.

In the next four sections, we summarize each program briefly, based on the description in [23]. For complete details of the programs, see that paper.

4.2.1 Barnes-Hut

The Barnes-Hut program simulates the evolution of a system of bodies under the influence of gravitational forces. Each body is a point mass and exerts forces on all other bodies in the system. In each time step of the simulation, the net force on each body is computed and its position and other attributes are updated. Since all bodies affect each other, computing the $O(n^2)$ pairwise forces would be impractical for large problems. The Barnes-Hut algorithm uses a hierarchical tree based method to reduce the complexity to $O(n \log n)$. The tree is deeper for regions where the body density is higher, with each node deeper in the tree representing a successively smaller region.

For each body, the tree is traversed once to compute the forces acting on it. If the body is sufficiently far away from the center of mass for the given subtree, the region is approximated by a single particle in the center of mass of that region. If it is not far enough away, the subtree is traversed. The majority of the time is spent in computing the inter-particle forces in each time step.

Data locality is provided by exploiting physical locality in the problem domain. By making the partitions spatially contiguous and equally sized in all directions, the interprocessor communication is minimized, and data reuse maximized. Barriers are used to synchronize at phases of the program where overlap must not occur.

4.2.2 Cholesky

This program performs a parallel Cholesky factorization of a sparse positive definite matrix; i.e., given a positive definite matrix A, compute a lower triangular matrix L such that $A = LL^T$.

Cholesky factorization involves three steps in general: ordering, symbolic factorization, and numerical factorization. The first is not done in this program, and the second is done by one processor. The only step performed in parallel is the last one, which is the most time consuming.

The tasks are scheduled from a task queue, and any free processor can pull off any task which needs to be completed. The only synchronization in the program occurs when accessing the task queue, which is controlled with a lock.

4.2.3 MP3D

MP3D solves a problem in rarefied fluid flow where the flow involves extremely low density. Under such conditions, traditional models such as the Navier-Stokes equations, are unusable because of assumptions that the medium is continuous. With the low density, the discrete particle nature of the medium becomes significant. MP3D uses a Monte Carlo method which simulates the trajectories of a collection of representative molecules, subject to collisions with boundaries of the physical domain, objects under study, and other molecules. Once a steady state is reached, statistical analysis of the trajectory data is used to produce an estimated flow field.

Each molecule is statically scheduled to one processor, which always computes its movement. The partitioning is not related to the particles location in space, so communication is significant. Barriers are used to synchronize the processors between the phases of the computations.

4.2.4 Water

Water is an N-body molecular dynamics application which evaluates forces and potentials in a system of liquid water molecules. Each time step involves setting up and solving the Newtonian equations of motion for water molecules in a cubical box with periodic boundary conditions. The method used for solving these equations is Gear's sixth-order predictor-corrector method. The total potential computed is the sum of intra-molecular and intermolecular potentials. A spherical cutoff radius of half the box length is used to reduce the total number of pairwise interactions computed. The box length is large enough to hold all of the molecules, and is computed by the program. The program uses static scheduling of processors to data since the communications patterns are predictable. Barriers are used to synchronize the processes before and after computing intermolecular interactions.

Chapter 5

Analysis of synthetic workload program execution

5.1 Introduction

We present here the characterization in terms of the measures defined in Section 2.3 of the traces collected from running the synthetic programs described in Section 4.1. Since we know the characteristics of these programs, we can identify the origins of the characteristics we discover. Six architectures are compared for each program. We select the page size from either 8192 or 64 bytes, and set the memory coherency policy to either update, update with expires, or invalidate. The goal here is to predict false sharing impact on an execution of an application, defined as the number of data bytes that are transmitted across the interconnecting network among processors that can be attributed to false sharing (as per the method in Section 3.2). Specifically, we want to know (1) Is a summary measure any good as a predictor of false sharing impact? (2) Do our proposed measures, $\mathcal{G}$ and $\mathcal{G}'$ from Section 2.3, predict false sharing impact for individual pages?

The plots in the discussion below compare the false sharing impact with a proposed predictor for the entire run of a program. The x-axis is the value of the proposed predictor measure and the y-axis is the number of data bytes transferred due to false sharing. Each point denotes a single data page in the application. For interpreting these plots, we consider a "good" plot to have the following properties:

- An increase in x-coordinate implies an increase in y-coordinate as a general trend (monotonically).

- For the point (x, y), given the horizontal interval $(x-\delta, x+\delta)$, for small δ, the range of the impact is within $(y-\varepsilon, y+\varepsilon)$, for small ε, over the horizontal interval. The term "small" is defined relative to the page size.

Plots that exhibit these properties are said to have a good correlation between the two functions compared and indicate that the proposed predictor is useful for its intended purpose.

We organize the analysis by the run-time variants of each synthetic program. We do not treat $\mathcal{G}'$ here as a separate case because the programs we are analyzing are all doing a read/modify/write loop. The number of writes is the same as the number of reads, thus these plots are nearly identical to those comparing $\mathcal{G}$ in all cases.

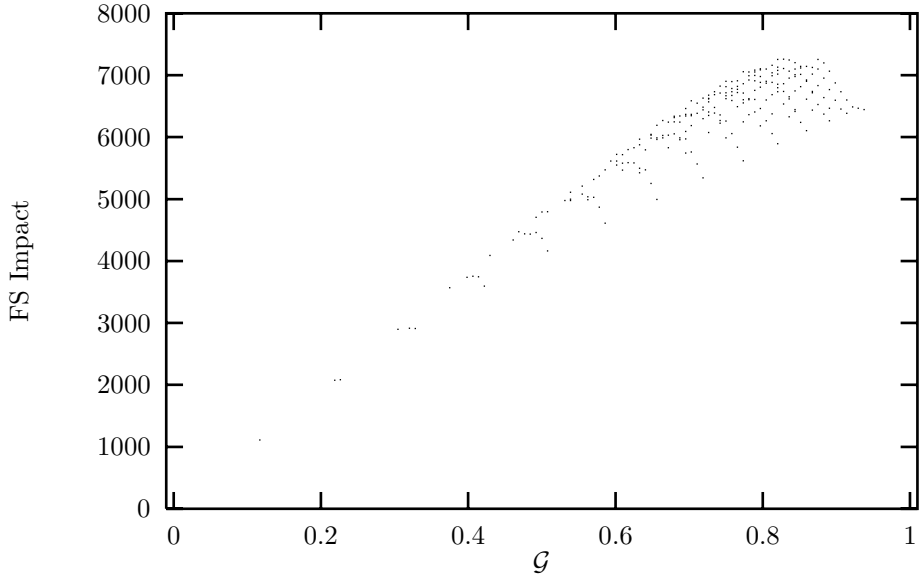

Figure 5.1: *synth-FS-s-n10, 64-byte page, update coherency*

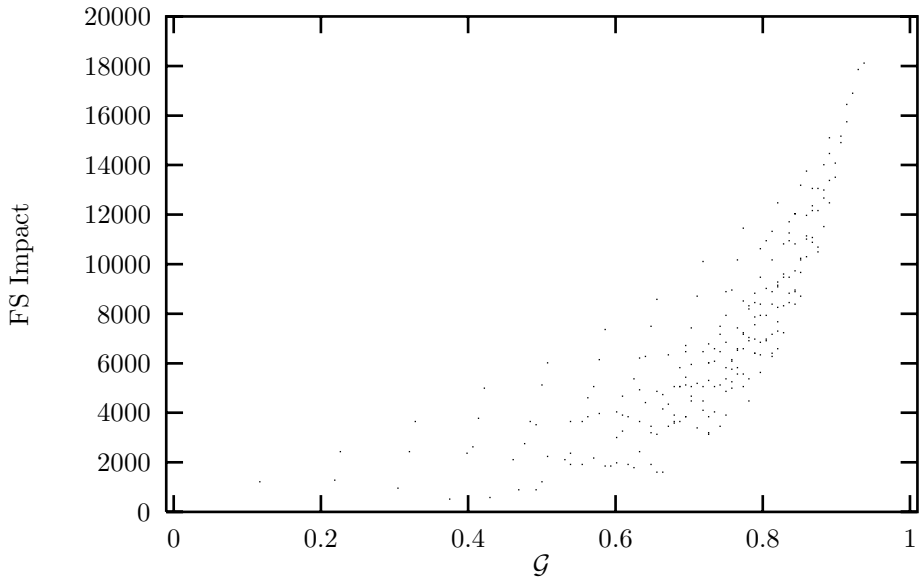

Figure 5.2: *synth-FS-s-n10, 64-byte page, invalidate coherency*

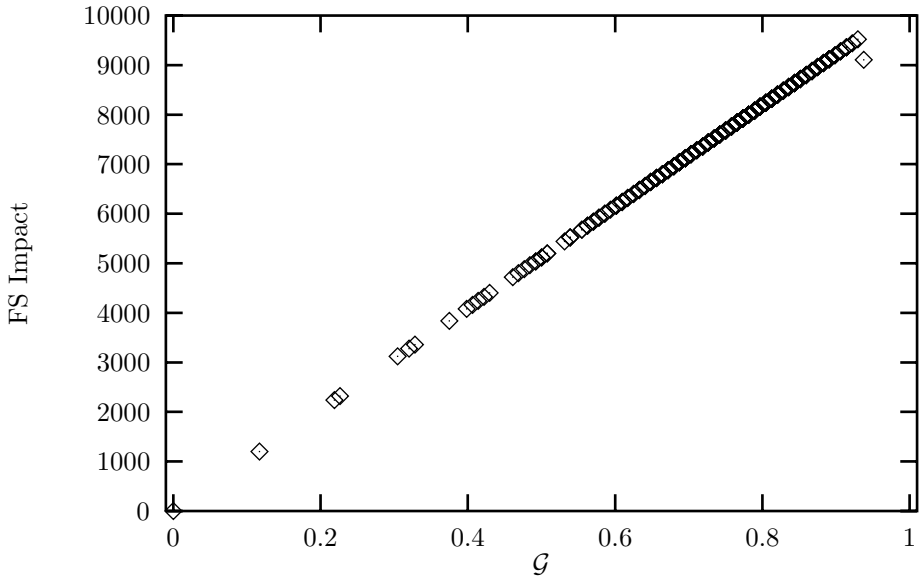

Figure 5.3: *synth-FS-s-n10, 8k-byte page, update coherency*

5.2 Synchronized and multiple references

Figure 5.1 contains the plot for the synth-FS program run with per-page synchronization (-s) and ten references per word (-n10) on a 64 byte page size with update coherency architecture. We can clearly see a good correlation between the $\mathcal{G}$ measure and the false sharing impact. The scattering is due to the loss of ordering information in the $\mathcal{G}$ calculation. Some of the references are being *sequentialized*[1] at the outset of each step while the processors are taking cold misses and the processor set to which updates are sent is still being established; thus there is not always as much traffic as the value of $\mathcal{G}$ indicates one could expect to see. When we switch to an invalidate coherency policy, as shown in Figure 5.2, the shape of the curve changes, and the accuracy of the prediction is lessened. There is too much variation for a fixed $\mathcal{G}$ to consider this to be a good correlation. However, the maximum impact for a given $\mathcal{G}$ does define a clear upper bound to the impact.

In Figure 5.3 we see the plot of synth-FS for an architecture with update coherency and 8192-byte (8k-byte) pages. The comparison is perfectly linear in this case. Compared with the 64-byte update coherency case, the larger granularity pages make it more likely that the processor set will be complete when each processor emerges from its initial cold miss of each iteration due to the increase in time which it takes to satisfy the cold miss and the small data size of this program. Ordering of reference is not as big an effect in the update case as in the invalidate case because of the implementation of the sharing mechanism: once a page is considered shared at a node, it will always get updates for that page. In an architecture that expires pages that are no longer needed, the correlation is not quite so perfect. In Figure 5.4, we plot the results of the update architecture with expiring pages. A page is expired on a node if it has received five updates without an intervening local reference. Notice that the magnitude of the impact is much lower, and that the relationship between impact and $\mathcal{G}$ is not as ideal. The one page clipped is at $\mathcal{G} = 0.0$, with an impact of 401408 bytes, which is exactly 49×8192. This page receives many updates before being used again, and is expired too often in this case. Figure 5.5

[1] Recall the definition of this term from Section 2.1

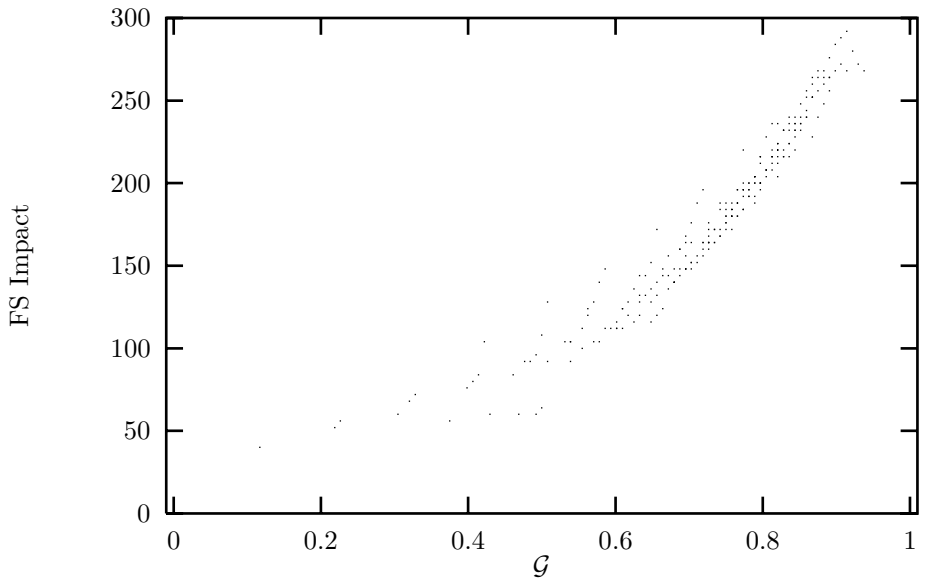

Figure 5.4: *synth-FS-s-n10, 8k-byte page, expiring update coherency (clipped)*

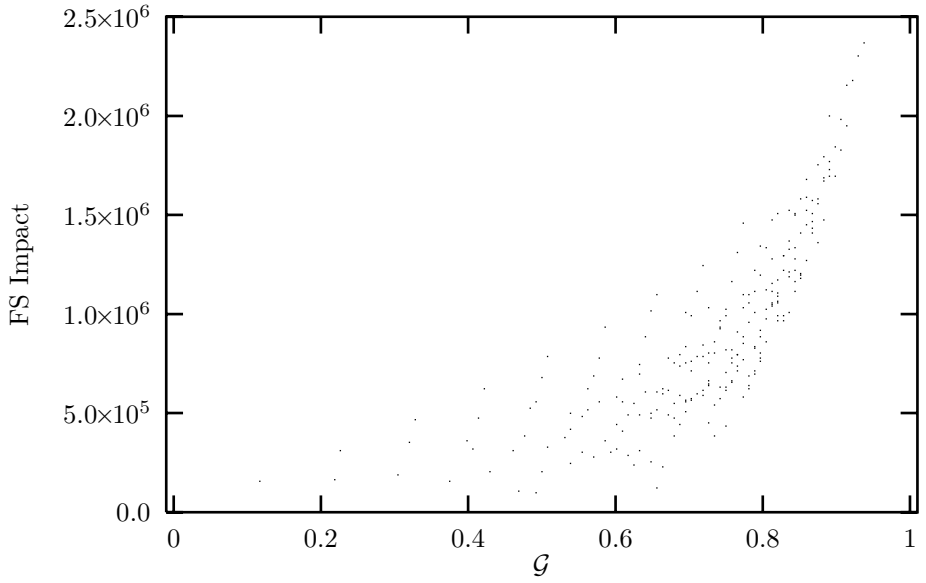

Figure 5.5: *synth-FS-s-n10, 8k-byte page, invalidate coherency*

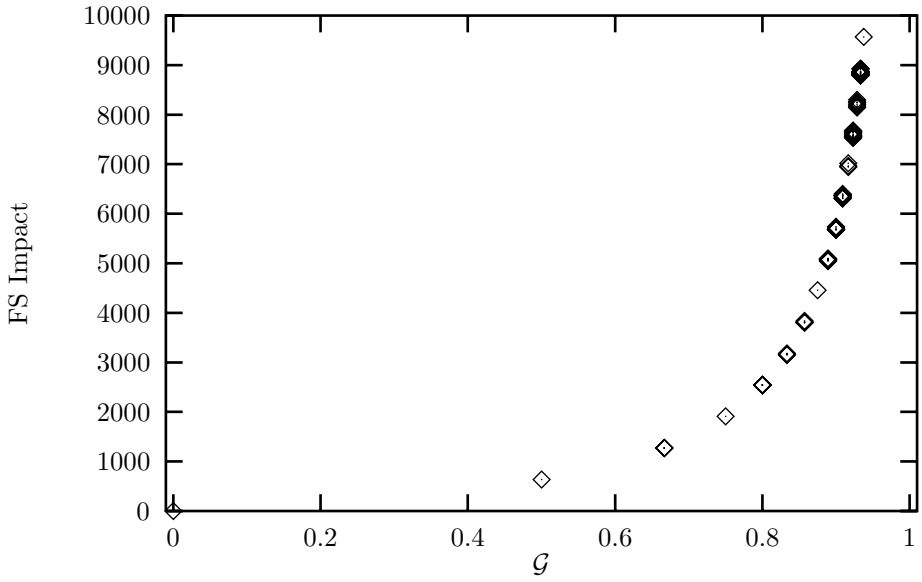

Figure 5.6: *synth-FS+PSS-s-n10, 64-byte page, update coherency*

contains the plot for invalidate coherency on an 8k-byte page. It is not substantially different from the 64-byte page case of the same program, except in the magnitude of the cost incurred.

The next four figures present the same set of plots for the synth-FS+PSS program. We again see in Figure 5.6 a very good predictive capability of $\mathcal{G}$ for the 64-byte page size update-coherency. The first page (with the lowest $\mathcal{G}$) is only used by one processor, the second page is used by two processors, and so on, with no true sharing of the words on a page. The more processors that use a page the more data is transferred, so it follows that the higher the value of $\mathcal{G}$, the higher the false sharing impact.

In the invalidate case, Figure 5.7, we see the trend is not monotonic, but in general, impact increases as $\mathcal{G}$ does. The "blip" at $\mathcal{G} = 0.75$ is caused by the alignment of pages within the data array. Each row of the data array consists of 2048 4-byte words, aligned over exactly 128 pages. The row that is referenced by three processors contains the pages with $\mathcal{G} = 0.75$. Since each processor references every third word in the row, and the row consists of 128 contiguous 64-byte pages, each page is referenced by each processor a slightly different number of times. The change per page induced by three processors is what causes the additional false sharing traffic observed. For the other rows in which a similar phenomenon occurs (6, 9, 12, and 15 processors per row), the additional traffic is not as significant.

When we increase the size of the page to 8k bytes, we see no difference from the 64-byte page size in the update case shown in Figure 5.8. This is because the cost of sending updates is unchanged by the page size, and the program allocates data aligned to 8k byte boundaries. All references by the program are identical, with the only difference being the size of pages copied when we use an invalidate protocol. As we can see in Figure 5.9, the costs due to false sharing in the 8k-byte invalidate case are orders of magnitude higher than the other cases. The "flattening" of the plot at high $\mathcal{G}$ values is because each page is referenced the same number of times, and thus has a maximum amount of data traffic associated with it. These pages incurred that maximum amount of impact.

The characteristics of the other run-time variations of these programs will be compared to these results.

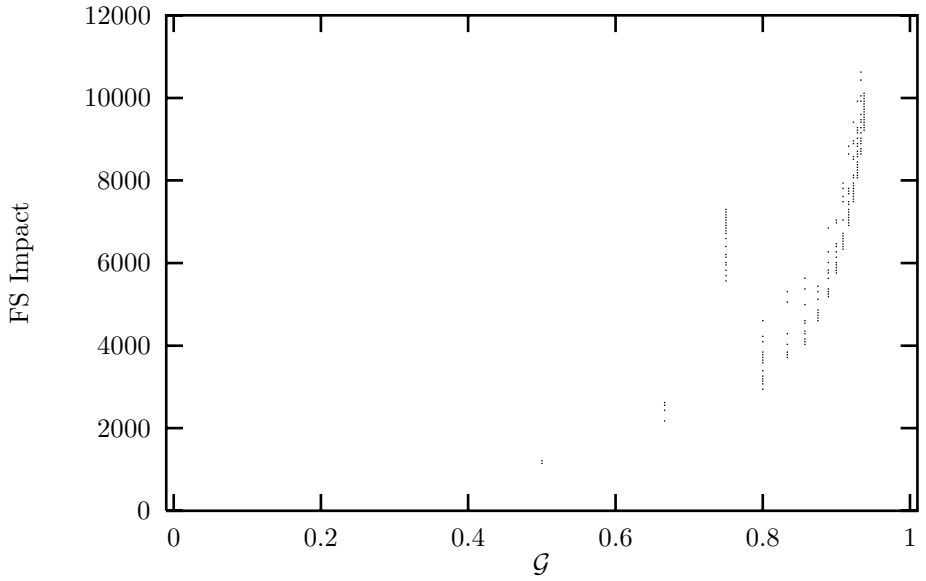

Figure 5.7: *synth-FS+PSS-s-n10, 64-byte page, invalidate coherency*

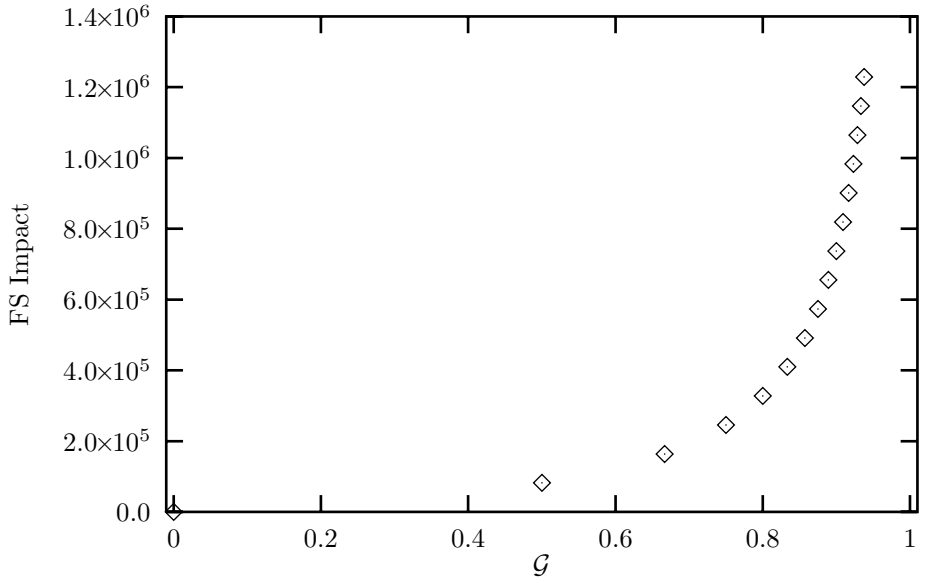

Figure 5.8: *synth-FS+PSS-s-n10, 8k-byte page, update coherency*

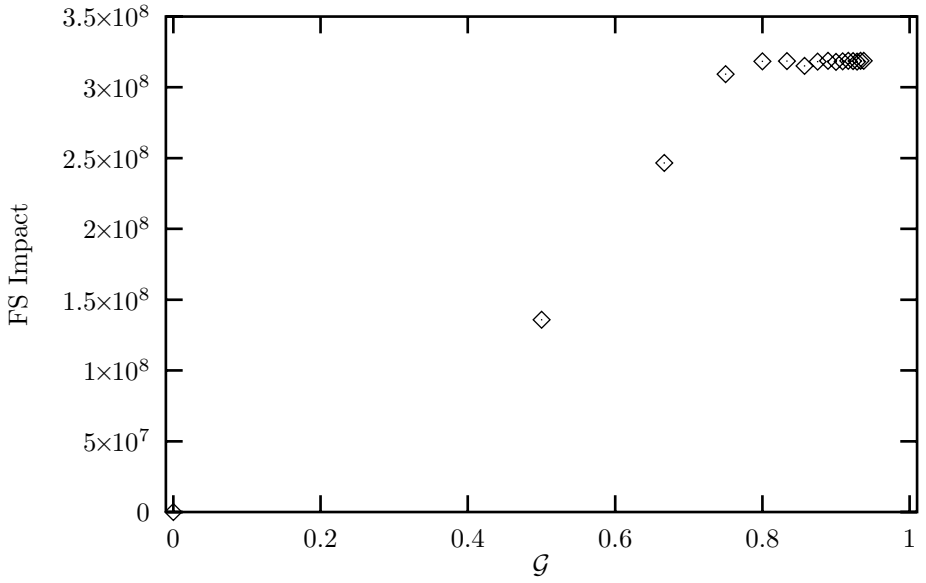

Figure 5.9: *synth-FS+PSS-s-n10, 8k-byte page, invalidate coherency*

5.3 Multiple references without synchronization

When the synth-FS program is run without any synchronization, many of the references get sequentialized. In the invalidate cases, interpreting $\mathcal{G}$ as a predictor of impact would predict a high amount of traffic caused by false sharing when in fact there is very little. This is due to the access order of the pages — once a node is done with a page, it will not use it again. Figure 5.10 illustrates this effect. Our predictive ability is non-existent. Even when $\mathcal{G}$ is high, the false sharing data traffic (impact) is not, except for five pages at the highest values of $\mathcal{G}$.

With the 8k-byte page size, shown in Figure 5.11, there is almost no traffic caused by false sharing for the majority of pages. Only a very few pages are invalidated even once, and a few pages are invalidated many times. The few pages with high cost associated with them are a side effect of the simulation. These pages are accessed simultaneously by multiple processors, so the invalidated pages are re-fetched. The highest $\mathcal{G}$ measures are associated with the last few pages accessed, and this is when the simulator manages to context switch rapidly enough, after most of the processors are terminated.

The $\mathcal{G}$ measure appears to be biased in a pessimistic direction and gives a worst-case prediction. Certain interleavings, like the sequentialization in this program, can result in better performance than predicted. This interleaving effect may suggest limiting the size of the window of observation when calculating the $\mathcal{G}$ measure for such programs. This type of analysis is done for the SPLASH programs in Chapter 6.

The plots for the update cases (64 byte page in Figure 5.12 and 8k-byte page in Figure 5.13) are identical, as expected, due to the design of the program. The data are aligned on 8k boundaries, and the references all occur in the first 64 bytes of a page. Both show perfect correlation of $\mathcal{G}$ and false sharing impact.

Looking at the synth-FS+PSS program for the same sets of parameters, we find that for the update coherency architectures (64-byte page in Figure 5.14 and 8k-byte page in Figure 5.15) the correlation is quite good. In the 64 byte page invalidate architecture, there is no data communication due to false sharing (Figure 5.16). Similarly, in the 8k-byte page invalidate case

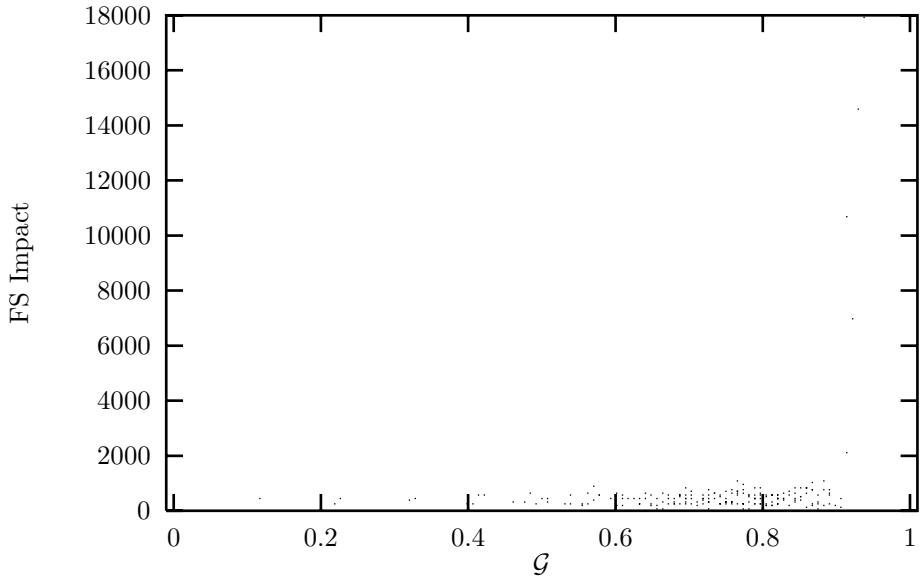

Figure 5.10: *synth-FS-n10, 64-byte page, invalidate coherency*

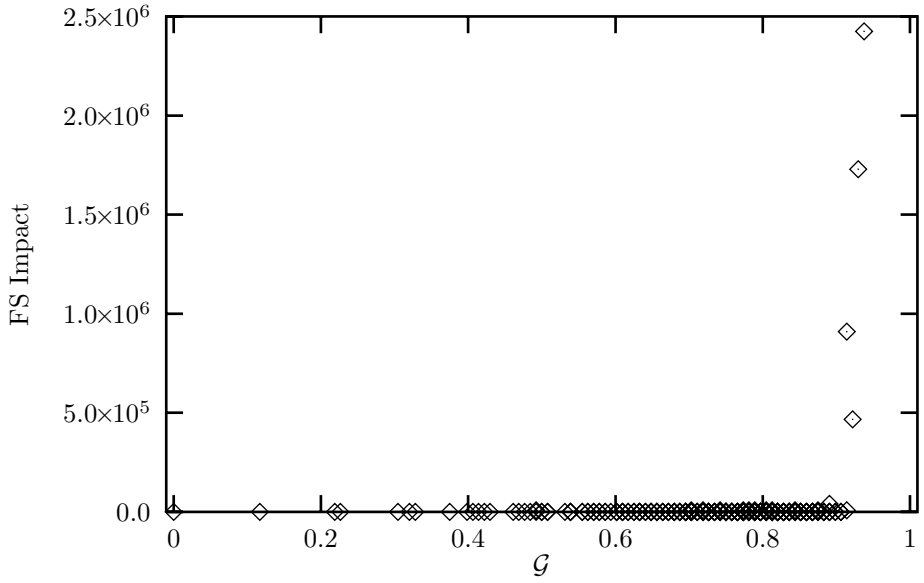

Figure 5.11: *synth-FS-n10, 8k-byte page, invalidate coherency*

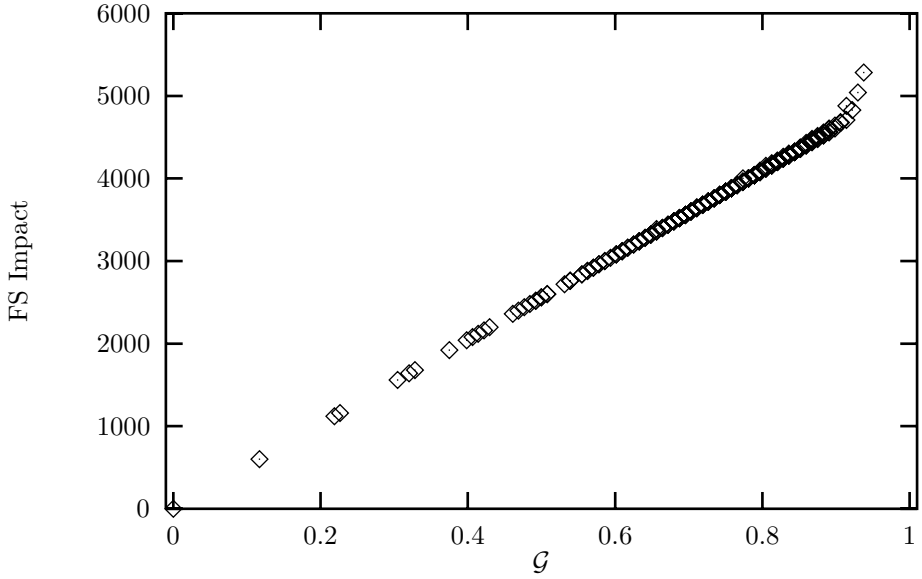

Figure 5.12: *synth-FS-n10, 64-byte page, update coherency*

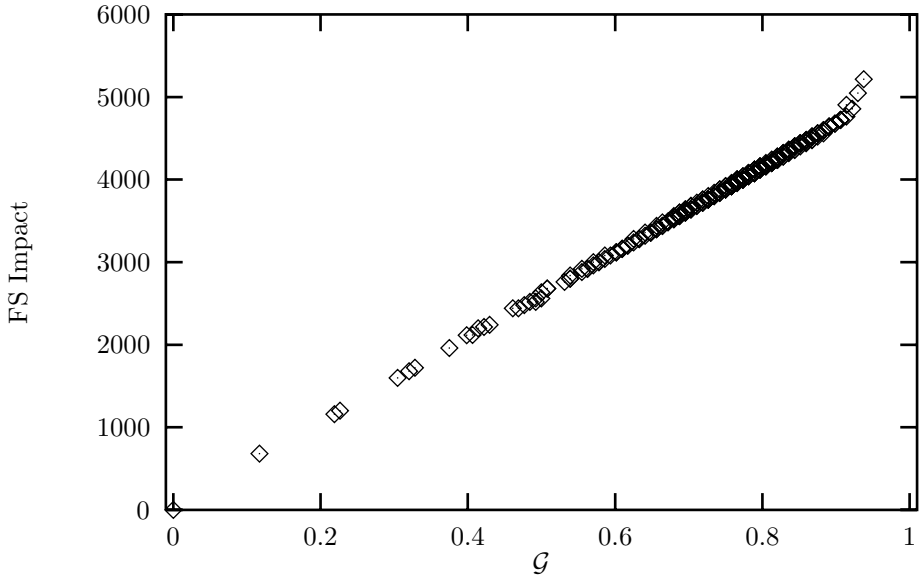

Figure 5.13: *synth-FS-n10, 8k-byte page, update coherency*

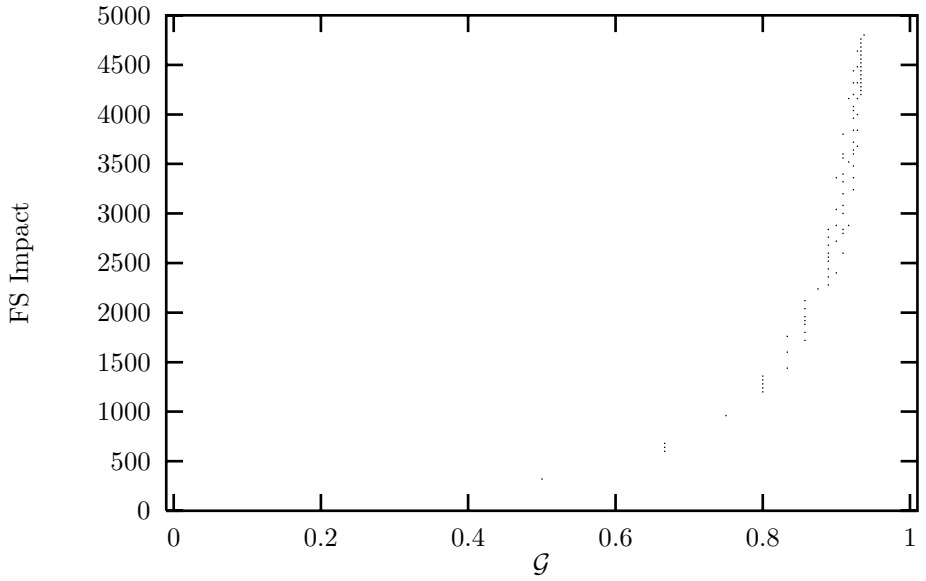

Figure 5.14: *synth-FS+PSS-n10, 64-byte page, update coherency*

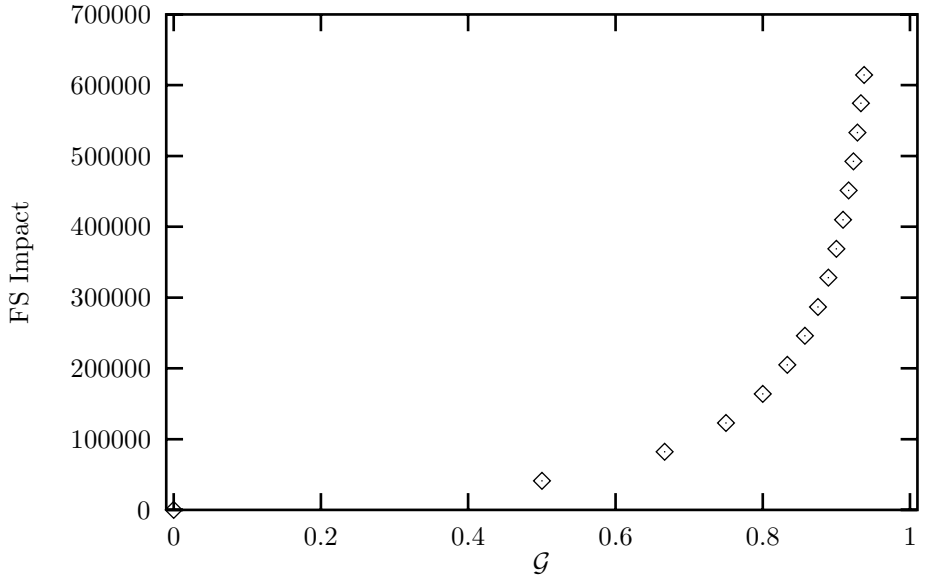

Figure 5.15: *synth-FS+PSS-n10, 8k-byte page, update coherency*

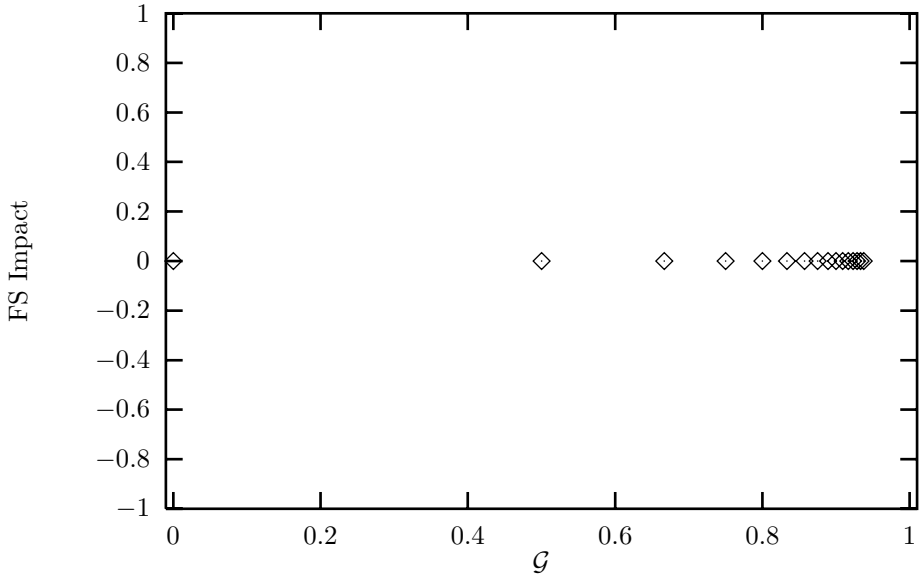

Figure 5.16: *synth-FS+PSS-n10, 64-byte page, invalidate coherency*

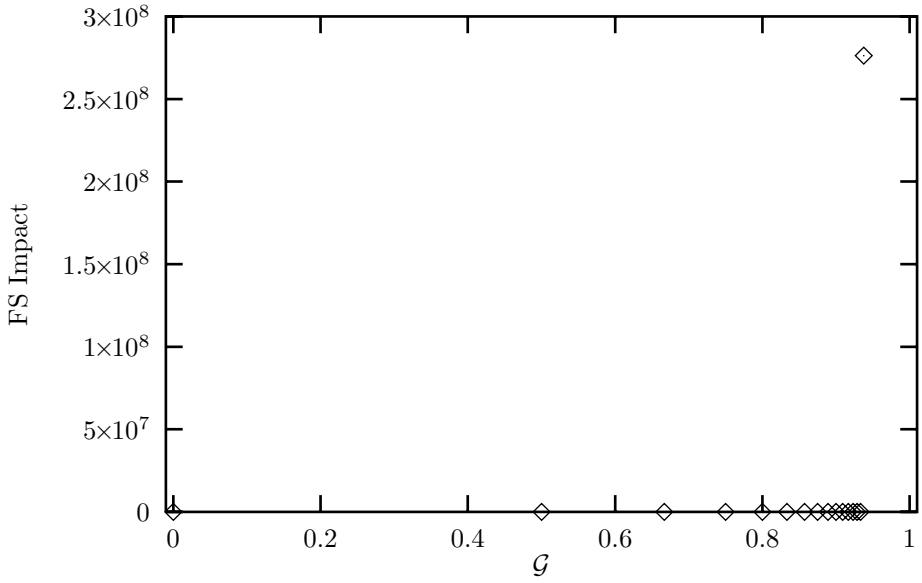

Figure 5.17: *synth-FS+PSS-n10, 8k-byte page, invalidate coherency*

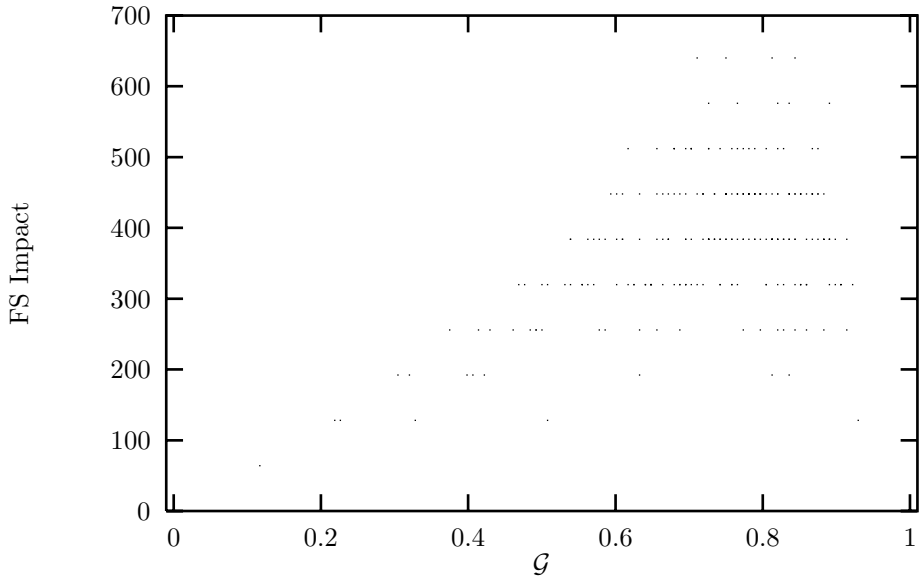

Figure 5.18: *synth-FS-s, 64-byte page, invalidate coherency*

shown in Figure 5.17, there is no false sharing impact except for one page, which incurs all of the costs. This is due to a simulation artifact as identified at the beginning of this section. There is no false sharing impact in these two simulations because of the way the simulator executes the code. Since different processors start out referencing different pages, by the time another processor references a given page no other processor is concurrently referencing it. Invalidation messages are sent, but the pages are never re-fetched (except for the one page in the invalidate case).

5.4 Synchronized

Maintaining the synchronization but reducing the number of references has an effect on the invalidate architectures, but little effect on the update ones. We present in Figure 5.18 the 64-byte page invalidate coherency case of synth-FS. The plot emphasizes one characteristic of all of the invalidate case values: The false sharing transfer bytes is always a multiple of the page size. We have a wider variation in $\mathcal{G}$ for a given amount of false sharing impact because of the relatively few number of references. The predictive value of $\mathcal{G}$ is generally good, but there is too much variation for $\mathcal{G} > 0.6$, highlighting the lack of consistency in the trend. In Figure 5.19 (8k-page, invalidate coherency) we do not have a good predictor primarily because of a wide variation in false sharing cost for the high $\mathcal{G}$ values. Most high-cost pages have high $\mathcal{G}$, yet many of the low-cost pages also have high $\mathcal{G}$ which would mean many false-positive predictions. In both of these latter two cases, the maximum impact for $\mathcal{G}$ is monotonically increasing with $\mathcal{G}$, which is good.

As we've seen before, the update architecture cases of synth-FS+PSS remain similar to the other run-time variations of it. There remains a high predictive capability for $\mathcal{G}$. However, in the invalidate architectures, problems develop. In Figure 5.20, the 64 byte page invalidate coherency case is shown. There is absolutely no correlation between $\mathcal{G}$ and false sharing impact. Figure 5.21 contains a plot of the 8k byte page, invalidate coherency architecture case for this program. We see some correlation here. In both of these last two cases, the reason for the observed effects is

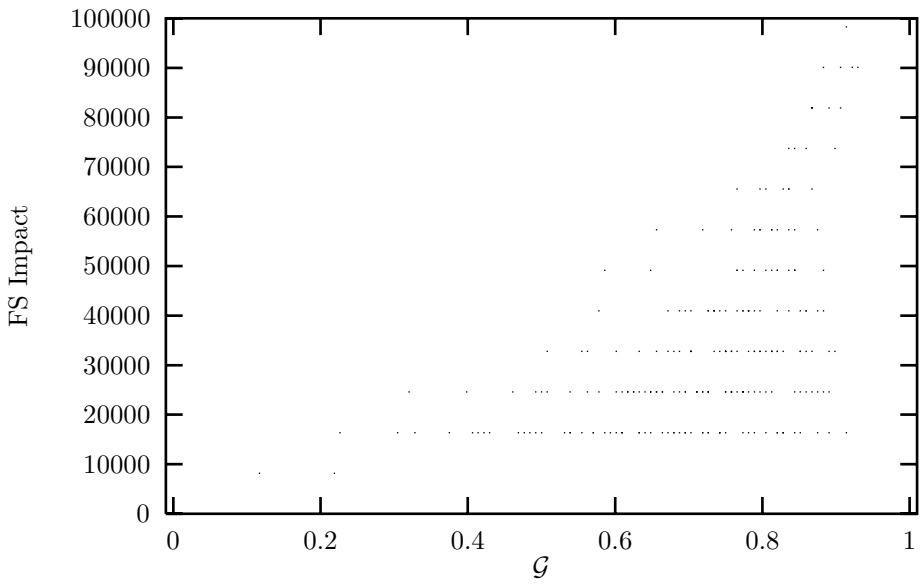

Figure 5.19: *synth-FS-s, 8k-byte page, invalidate coherency*

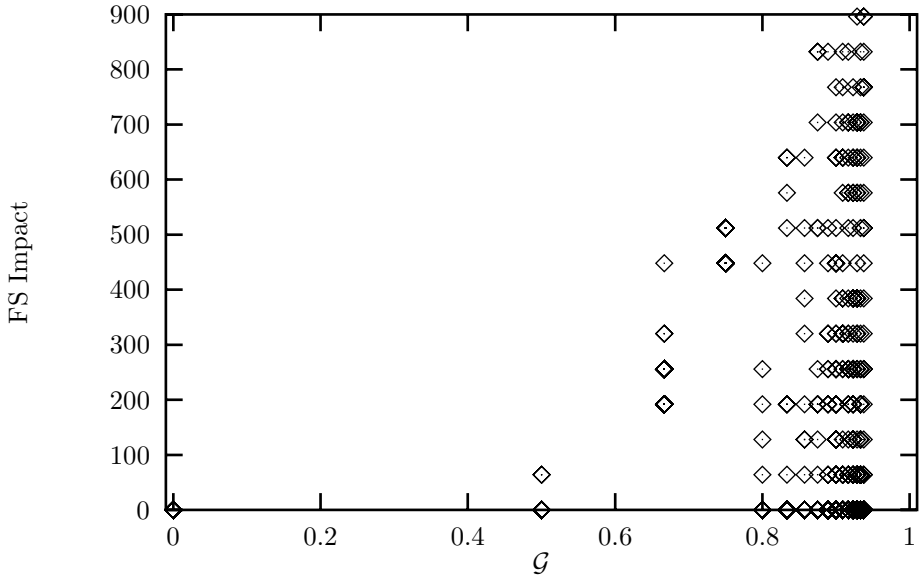

Figure 5.20: *synth-FS+PSS-s, 64-byte page, invalidate coherency*

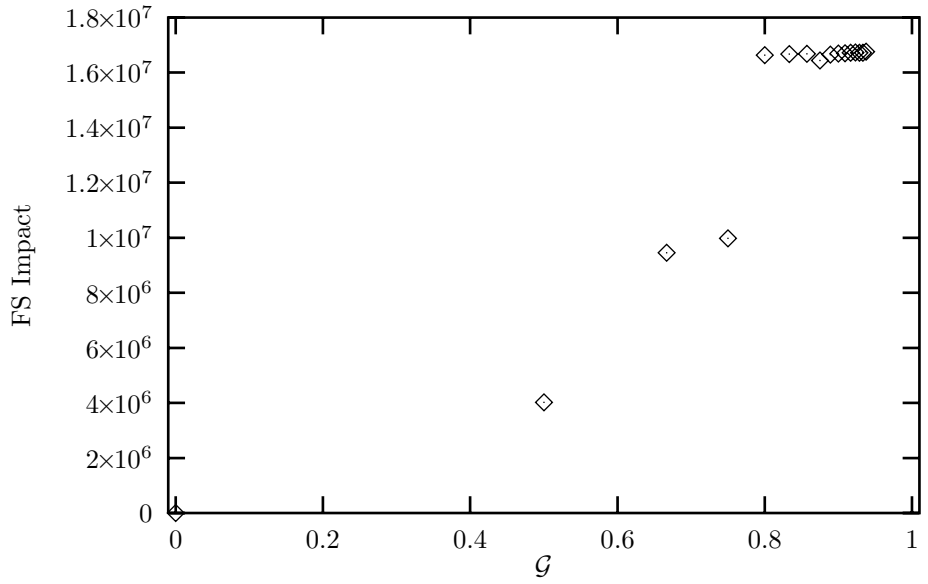

Figure 5.21: *synth-FS+PSS-s, 8k-byte page, invalidate coherency*

due to the very low number of references made to each page by each processor. The pages are not actively shared in that the pages are referenced sequentially by different processors.

5.5 Normal execution

When we remove synchronization from the above program executions of synth-FS, we have an application that makes very few references to shared memory, and these references tend to become sequentialized. The sequential nature of the references is a big hindrance to accurately predicting false sharing costs based only on the amount of references. This is clearly demonstrated by the plot in Figure 5.22. The 8k and 64 byte page invalidate coherency architecture cases look virtually identical other than the scale of the vertical axis. In both of these cases, a page has at most one page fault after being invalidated. This is partially due to the coarse granularity of the simulation — the majority of the work from one processor can be done before any context switch to simulate another processor starts. This results in very little data traffic due to invalidated pages. It should be noted, however, that for the update cases in this program, we again have good correlation, and the plots do not differ significantly from those of the other variations of this program.

The analysis for the synth-FS+PSS program is identical to that of the multiple reference case in Section 5.3. There is no major difference because of the order in which references are made. Each processor references one page, then moves on to the next page. While it is referencing that page, chances are that no other processor will reference it during that same time. This is partly due to the granularity of the simulator.

5.6 Discussion

There were a few situations where $\mathcal{G}$ was very accurate in predicting the false sharing impact. When a given page was accessed actively during the same period of time by many processors, the

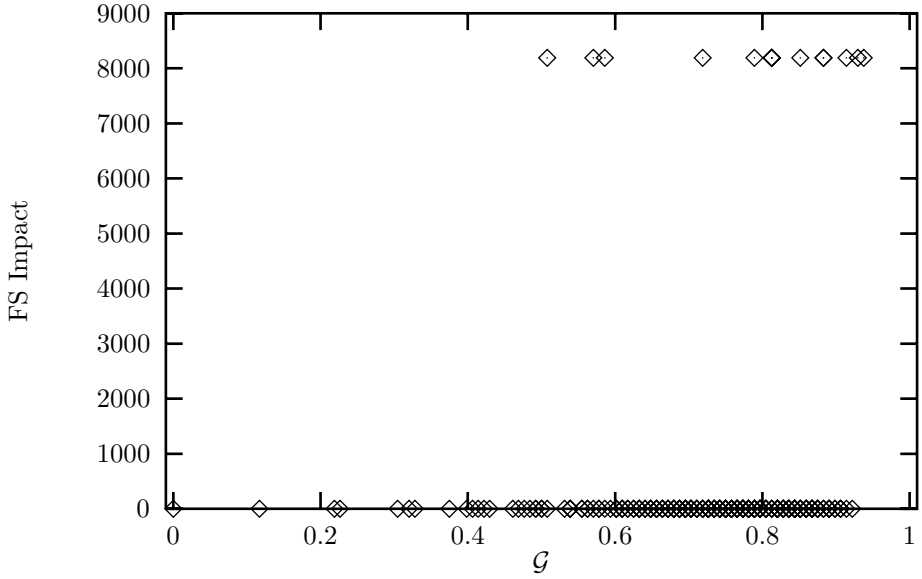

Figure 5.22: *synth-FS, 8k-byte page, invalidate coherency*

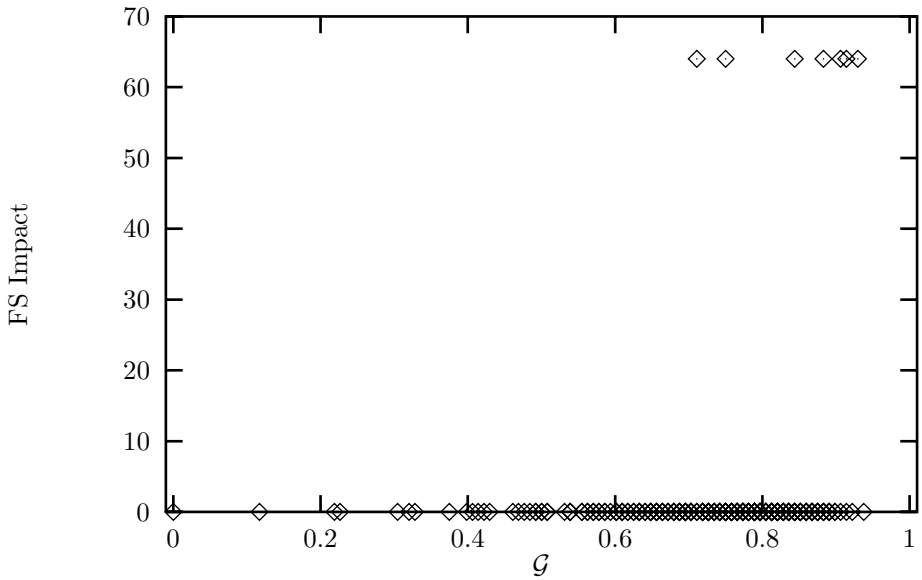

Figure 5.23: *synth-FS, 64-byte page, invalidate coherency*

accuracy of $\mathcal{G}$ increased. This was demonstrated by the multiple-reference, synchronized versions of the programs. When a page was accessed by processors at different time localities the accuracy of $\mathcal{G}$ as a predictor was diminished, as demonstrated by the non-synchronized versions of the programs.

In the following chapter, we analyze some real parallel programs in a similar fashion. We defer final interpretation of the results of this chapter until the end of that chapter.

Chapter 6

Analysis of SPLASH program execution

We now evaluate some real programs using the techniques developed in Chapter 5 for the synthetic programs. These programs are from the SPLASH benchmark suite as described in Section 4.2. As before, we evaluate the six variations of the architecture: 64-byte and 8k-byte page sizes with either invalidate, update, or update with expire coherency. Where we do not specifically address the expiring update coherency cases, there is no difference in the interpretations derived from them and those derived from the standard update coherency cases.

In the previous chapter evaluating the synthetic programs, we saw only partial success with using $\mathcal{G}$ as a predictor. It is becoming clear that $\mathcal{G}$ loses some information needed to predict the false sharing impact of a program. In evaluating the SPLASH programs, we try additional prediction measures which take into account more factors to answer the following questions: (1) Does $\mathcal{G}'$ provide a better predictor for false sharing impact? (2) Does scaling $\mathcal{G}$ by the number of writes to each page improve the predictor? (3) Does computing $\mathcal{G}$ over shorter intervals as defined by phase changes provide a good prediction? First we evaluate the programs using the $\mathcal{G}$ metric. We will then proceed by answering the above questions in order, one per section. The results we present are largely negative, and we investigate the causes of the failures.

6.1 Evaluation of $\mathcal{G}$ as predictor

6.1.1 Barnes-Hut

In Table 6.1 we present summary information about the simulation runs for the Barnes-Hut program. The run time is in nanoseconds of simulated execution time. The number of read and write references made by the executing program are important in interpreting the results below. The column labeled Data Bytes indicates the number of data bytes transferred across the interconnection network to keep the pages coherent (this includes both true sharing and false sharing traffic). There is more data traffic in the 8192-byte update-expire than in the non-expiring update version. For this particular application and input data, the update-expire threshold was too low, resulting in pages being expired too soon. There were nearly as many expire-induced page faults as there were expired pages. With the 64-byte page size, there is a significant reduction in data transfer cost. Correspondingly, the number of expire-induced page faults in this case is less than 70% of the number of expired pages.

It is interesting to note the large change in the runtime and number of reads for the 8k-byte invalidate case as compared with the others. This large discrepancy demonstrates the qualitative difference of the simulations when architectural parameters are changed. It is precisely for this

Memory Architecture	Run time	Reads	Writes	Data Bytes
Optimal	136747780ns	5099480	183128	528300
64-byte, Invalidate	138297560ns	5107078	183141	4384896
64-byte, Update	137370440ns	5098001	183128	7788550
64-byte, Update-expire	137684720ns	5108251	183130	4041284
8k-byte, Invalidate	515639340ns	8550031	183108	1618911232
8k-byte, Update	137901000ns	5110158	183128	14652222
8k-byte, Update-expire	138434560ns	5166504	183128	14842210

Table 6.1: *Summary of simulation runs for Barnes-Hut*

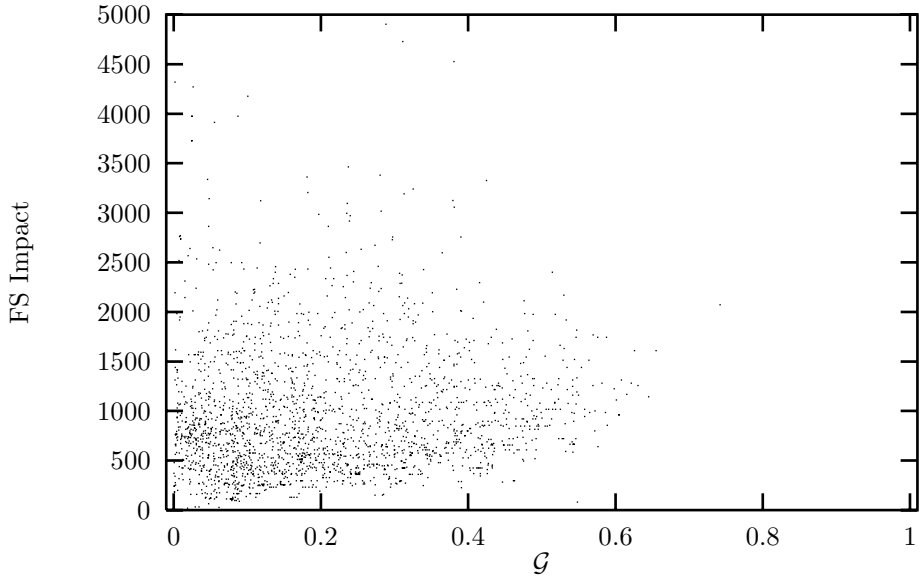

Figure 6.1: *Barnes-Hut, 64-byte page, update coherency ($\mathcal{G}$)*

reason that we do not use trace-driven simulations in our experiments. The execution-driven experiments capture the difference in memory reference timing and orderings that the trace-driven execution would not, making our results are more representative of reality. Holliday and Ellis elaborate on this further in [14].

Running the analysis on the Barnes-Hut application, we see that for the 64-byte page architectures we achieve no predictive ability. In Figure 6.1 (update coherency), the dispersion of pages with high and low false sharing impact for the same value of $\mathcal{G}$ most likely indicates that there is significant information loss which we cannot tolerate if we are to properly predict the impact. Similarly, Figure 6.2 shows us that for the invalidate coherency, we suffer similar loss. This plot in fact shows the inverse of what we had hoped for — the higher impacts are for pages with the lower $\mathcal{G}$ values. Note that this plot is zoomed into the range where the detail can be seen; the actual range of the impact on the vertical axis goes up to 700000 bytes transferred, but only a few pages incurred that high cost. The high impact for pages with low $\mathcal{G}$ is the result of sharing data on pages across separate phases of the program run. Different sections of each page are shared by different subsets of processors at different times. We will see later how limiting the computation of $\mathcal{G}$ to single phases of the execution improves the accuracy of the prediction

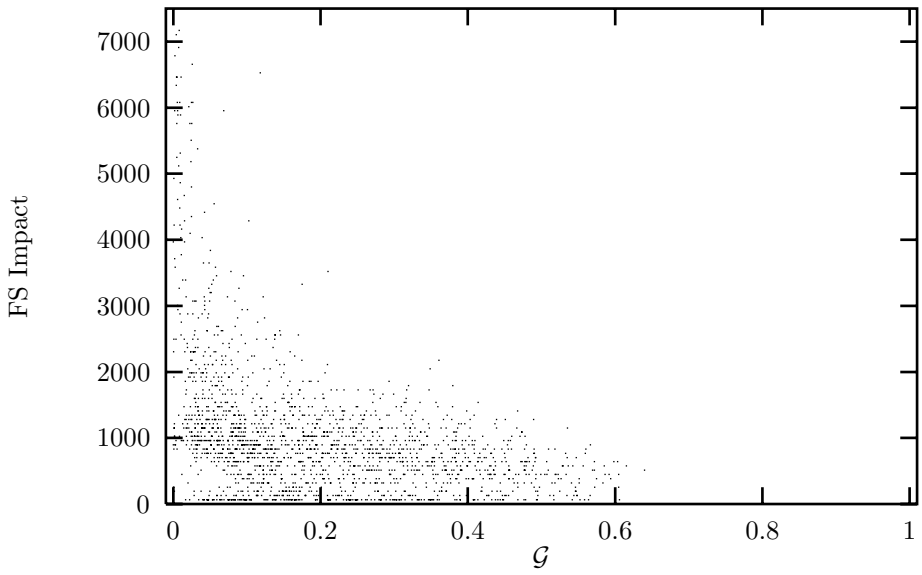

Figure 6.2: *Barnes-Hut, 64-byte page, invalidate coherency ($\mathcal{G}$) (clipped)*

in this type of situation.

When we increase the page size to 8k bytes, we see in both the update (Figure 6.3) and invalidate (Figure 6.4) coherency models absolutely no correlation between $\mathcal{G}$ and the false sharing impact. We conjecture that this is due to the loss in fine-grain information about the ordering of the memory references.

The same program run with the expiring update coherency model shows a much wider dispersion of impact for values of $\mathcal{G}$ close to each other. The plot in Figure 6.5 shows this for the 64-byte page size. This plot is clipped to the same vertical scale as Figure 6.1 for comparison. The few points that are clipped from this image are all near $\mathcal{G} = 0.0$ and have a maximum value of 132992 — the pages are shared at different times during the execution and repeatedly get expired then later paged back in. Most of these pages contain lock data structures.

6.1.2 Cholesky

The next application we evaluate is Cholesky. The simulation run information is summarized in Table 6.2. Like the Barnes-Hut program, the simulation of the 8k-byte page with invalidate coherency runs much longer, and makes many more references to shared memory. The ratio of reads to writes is lower than in Barnes-Hut (approximately $6\frac{1}{2}$ times more reads than writes) except for the one case where it is over 20 times. The expiring update coherency provides a significant reduction in data transfer over the regular update coherency. For both page sizes, the number of expire-induced page faults is less than 45% of the number of pages expired.

Examining the plot of $\mathcal{G}$ compared to false sharing impact per page in Figure 6.6 we see that for 64-byte page size with update coherency that most pages have a relatively low $\mathcal{G}$ value. Further, the impact of the set of pages with the same $\mathcal{G}$ value varies widely from near zero to several thousand bytes; there is no correlation between the two. Expiring pages with update coherency results in reduced false sharing impact for the majority of the pages (Figure 6.7). The majority of the higher impact pages have low $\mathcal{G}$, and there is no increasing trend. This is caused by these pages being expired often and paged back in because of a cyclical access pattern to

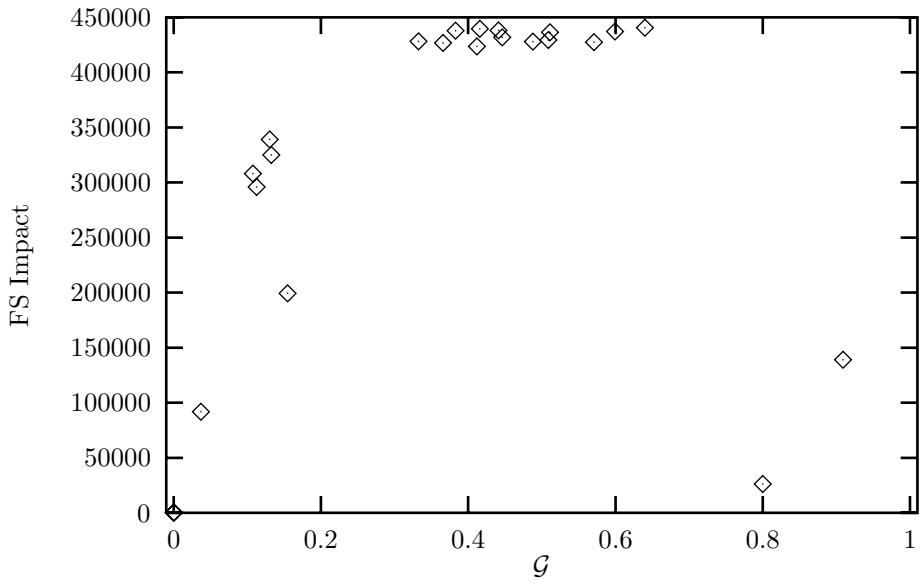

Figure 6.3: *Barnes-Hut, 8k-byte page, update coherency ($\mathcal{G}$)*

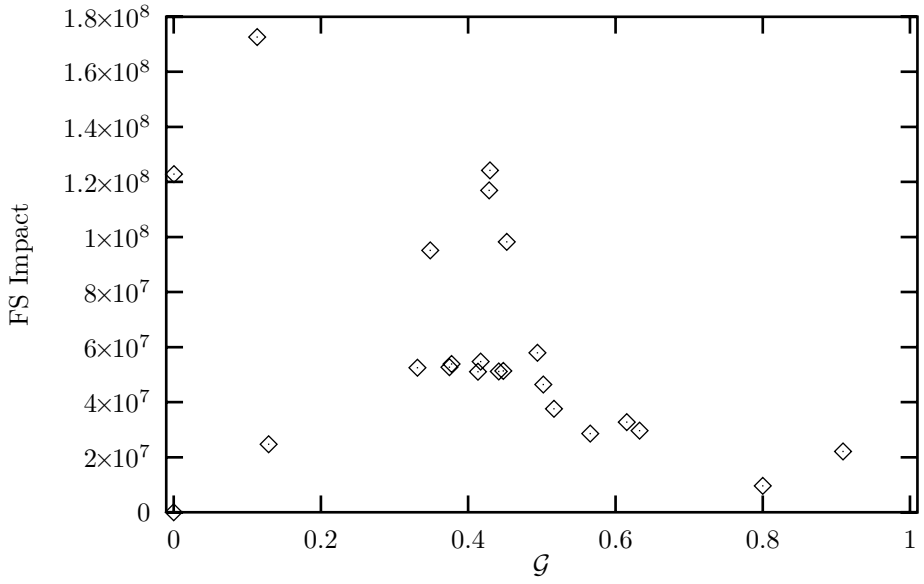

Figure 6.4: *Barnes-Hut, 8k-byte page, invalidate coherency ($\mathcal{G}$)*

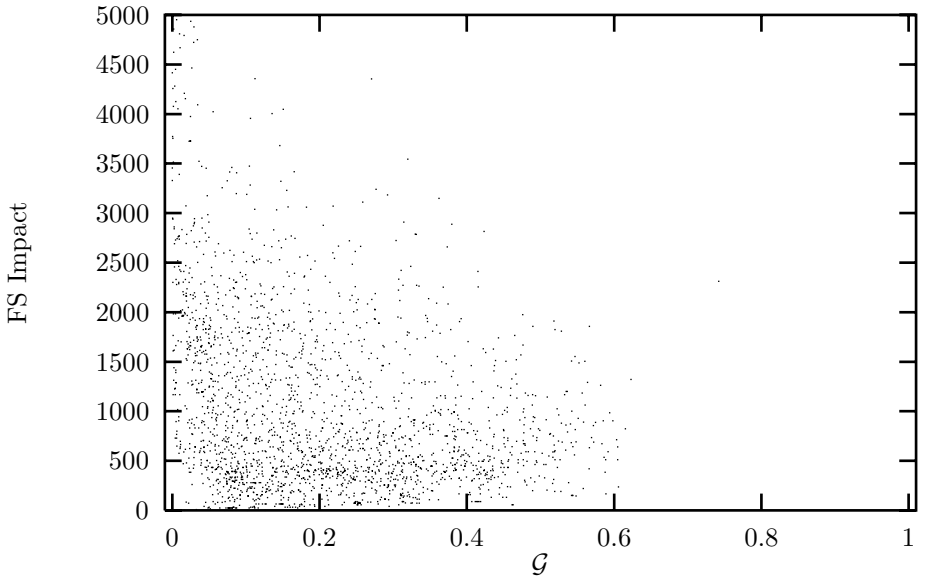

Figure 6.5: *Barnes-Hut, 64-byte page, expiring update coherency ($\mathcal{G}$) (clipped)*

Memory Architecture	Run time	Reads	Writes	Data Bytes
Optimal	316798140ns	12009536	1820214	2602588
64-byte, Invalidate	325613580ns	12103461	1820193	5880192
64-byte, Update	322095520ns	12050717	1820169	63434052
64-byte, Update-expire	322095520ns	12050717	1820169	14065668
8k-byte, Invalidate	904599740ns	38175037	1820103	1995030528
8k-byte, Update	324589740ns	12113467	1820190	116686416
8k-byte, Update-expire	324589740ns	12113467	1820190	47402188

Table 6.2: *Summary of simulation runs for Cholesky*

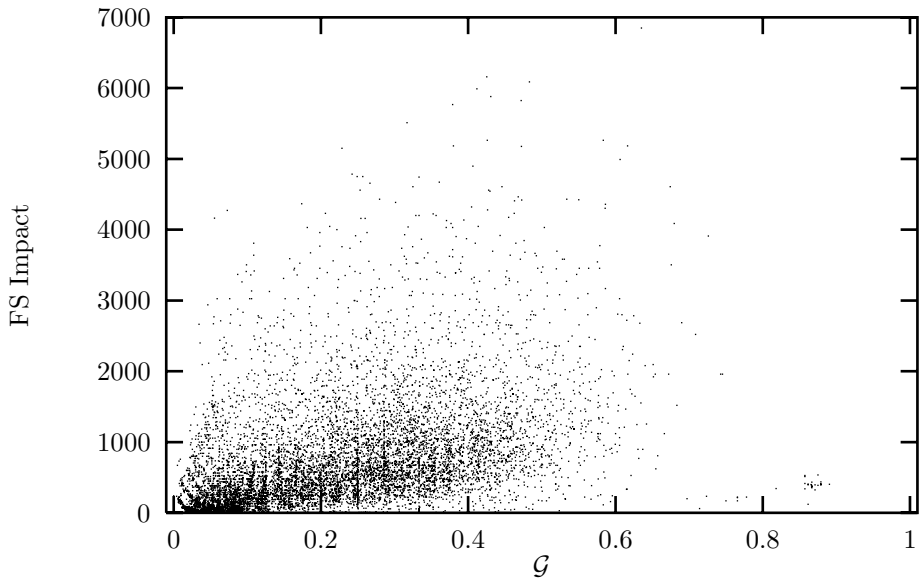

Figure 6.6: *Cholesky, 64-byte page, update coherency ($\mathcal{G}$)*

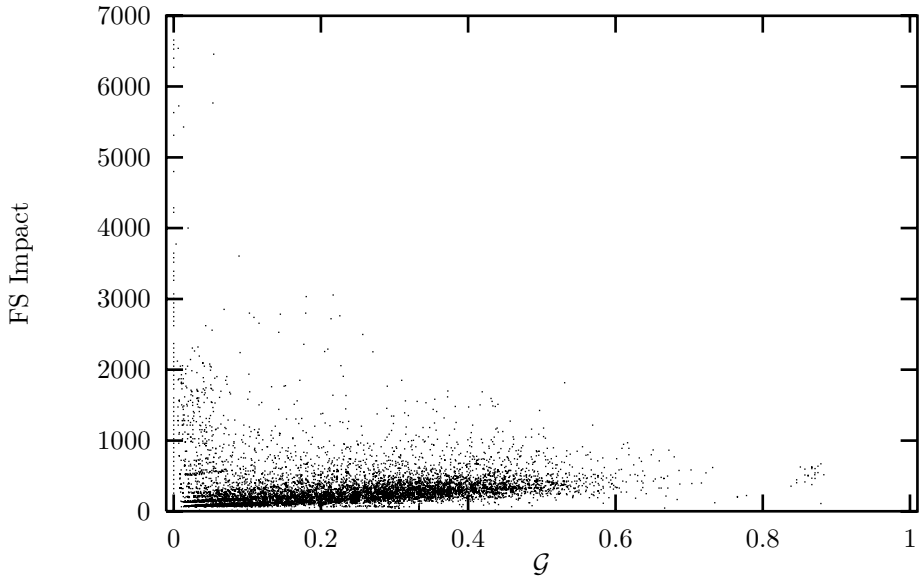

Figure 6.7: *Cholesky, 64-byte page, expiring update coherency ($\mathcal{G}$) (clipped)*

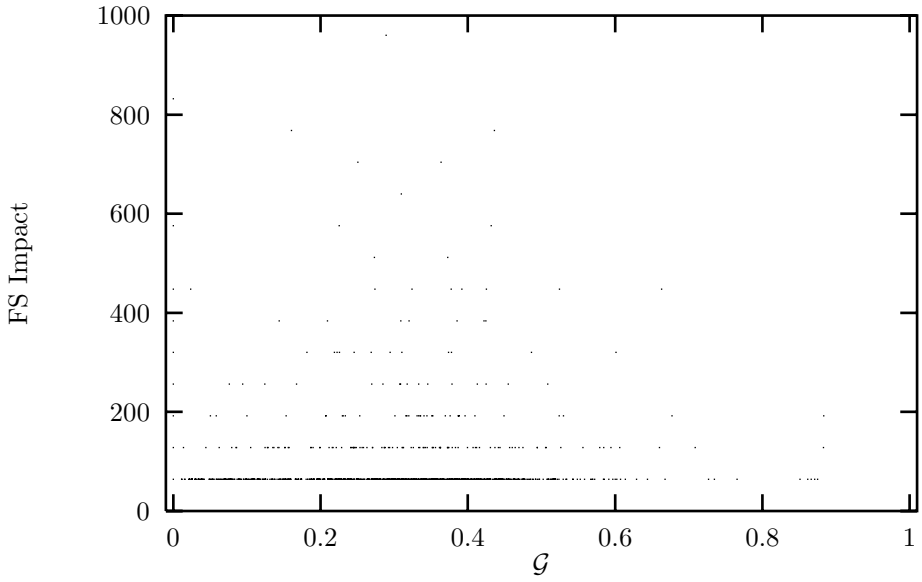

Figure 6.8: *Cholesky, 64-byte page, invalidate coherency ($\mathcal{G}$) (clipped)*

the different data on them. For the invalidate case, we examine a clipped region of the plot in Figure 6.8. The full vertical axis range goes up to 400000 bytes; all of the pages that are outside of the presented plot are near $\mathcal{G} = 0.0$. Again, there is no real correspondence between increasing $\mathcal{G}$ and increasing impact.

With the 8k-byte page sized architectures, we again see little correlation. For the update case, Figure 6.9, the majority of pages fall in the range $0.4 < \mathcal{G} < 0.8$, and the impact of those pages varies widely. For the invalidate protocol, Figure 6.10, we present a clipped view again. In this plot, any pages which were clipped out had $\mathcal{G} < 0.4$. Once again, the value of $\mathcal{G}$ does not help significantly in pinpointing pages that contribute the most to false sharing impact. There are too many pages with high $\mathcal{G}$ with very low impact, which would mean many false-positive predictions.

6.1.3 MP3D

The simulation information for Mp3d is summarized in Table 6.3. Like the previous two programs, the 8k-byte page with invalidate coherency architecture case runs much longer; unlike the other programs, it does not make many more references to shared memory. The ratio of reads to writes is nearly unitary for all architectures.

As in the Barnes-Hut application, the 8192-byte page expiring update coherency caused more data traffic than the non-expiring version, but for the 64-byte page size, the data traffic is reduced. In both cases, the number of expire-induced page faults is at least 96% of the number of expired pages. The reduction in the 64-byte case results from the long duration between expiring and re-loading the pages.

Shown in Figure 6.11 is the comparison of $\mathcal{G}$ to the false sharing impact (64-byte page, update coherency). For low values of $\mathcal{G}$ there is a wide range in impact, but for higher values, the impact remains quite low. The high impact for pages with low $\mathcal{G}$ is caused by sharing of the pages during different phases in the execution, as we also observed in the Barnes-Hut application. A similar situation occurs in the invalidate coherency architecture shown in Figure 6.12. This plot has four

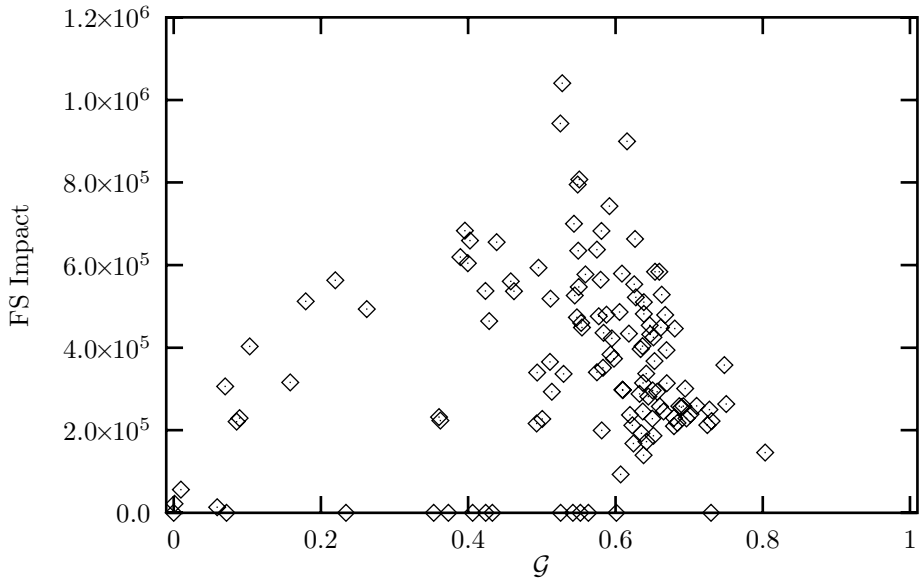

Figure 6.9: *Cholesky, 8k-byte page, update coherency ($\mathcal{G}$)*

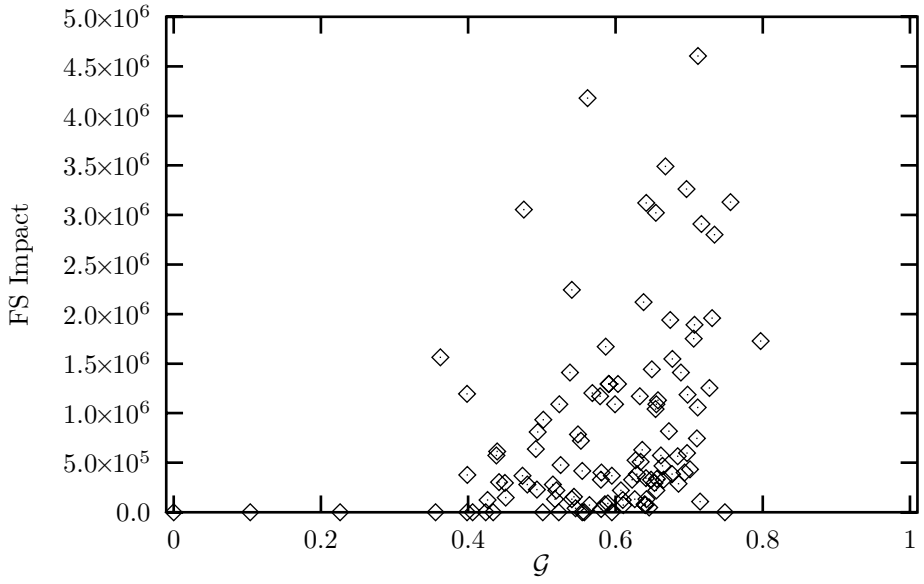

Figure 6.10: *Cholesky, 8k-byte page, invalidate coherency ($\mathcal{G}$) (clipped)*

Memory Architecture	Run time	Reads	Writes	Data Bytes
Optimal	52453561ns	3271963	3141549	6492356
64-byte, Invalidate	58269521ns	3272740	3142056	22362240
64-byte, Update	53383721ns	3270817	3140968	121814272
64-byte, Update-expire	53383721ns	3270817	3140968	30810496
8k-byte, Invalidate	2070405921ns	3273222	3142414	7710507008
8k-byte, Update	55295141ns	3274282	3142852	184052348
8k-byte, Update-expire	55295141ns	3274282	3142852	187392012

Table 6.3: *Summary of simulation runs for Mp3d*

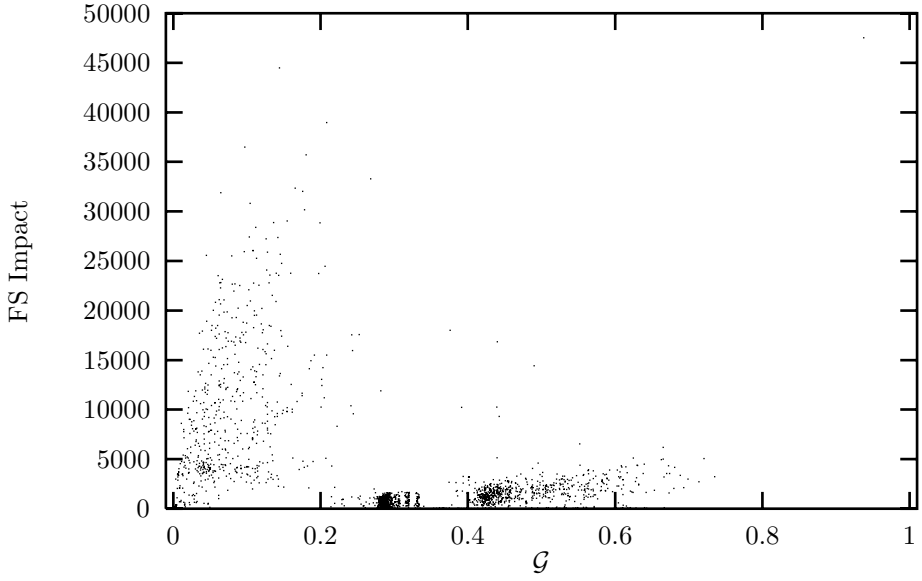

Figure 6.11: *Mp3d, 64-byte page, update coherency ($\mathcal{G}$)*

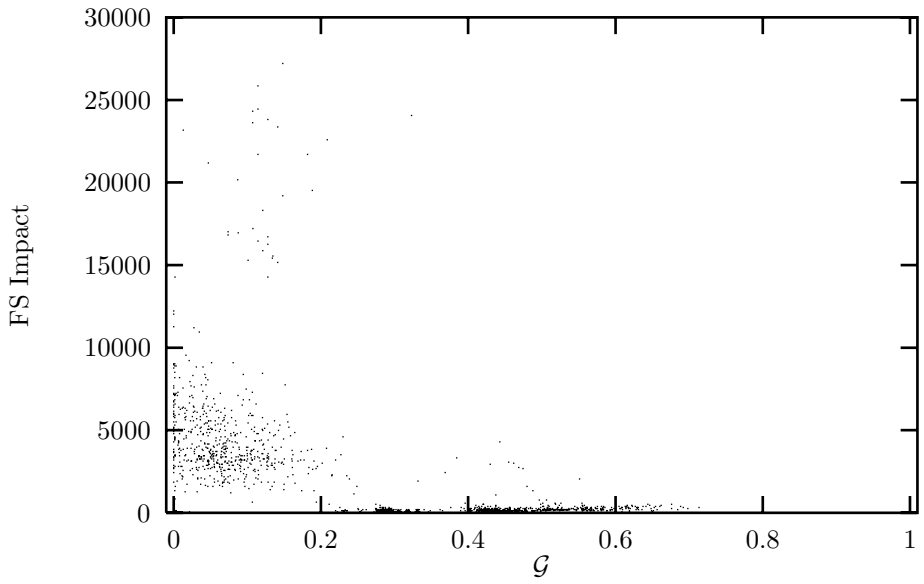

Figure 6.12: *Mp3d, 64-byte page, invalidate coherency ($\mathcal{G}$) (clipped)*

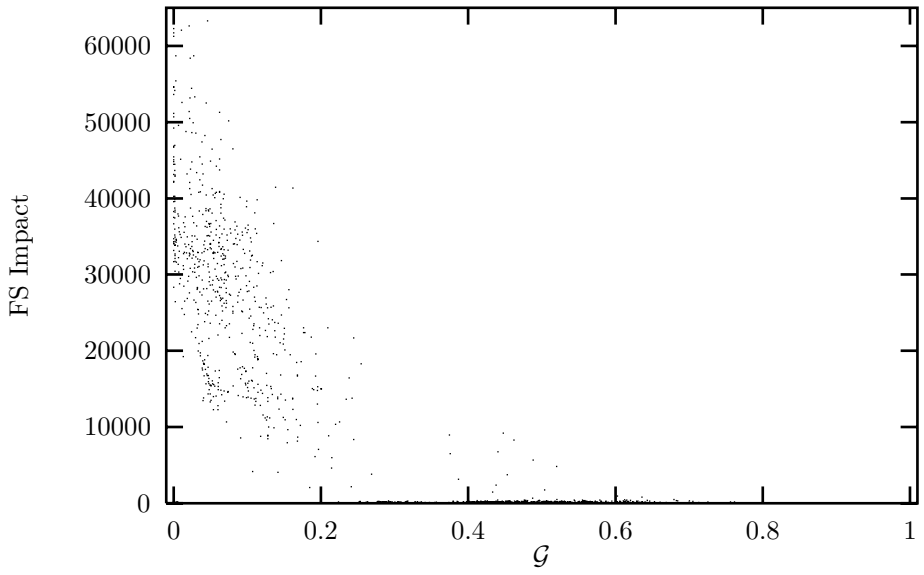

Figure 6.13: *Mp3d, 64-byte page, expiring update coherency ($\mathcal{G}$) (clipped)*

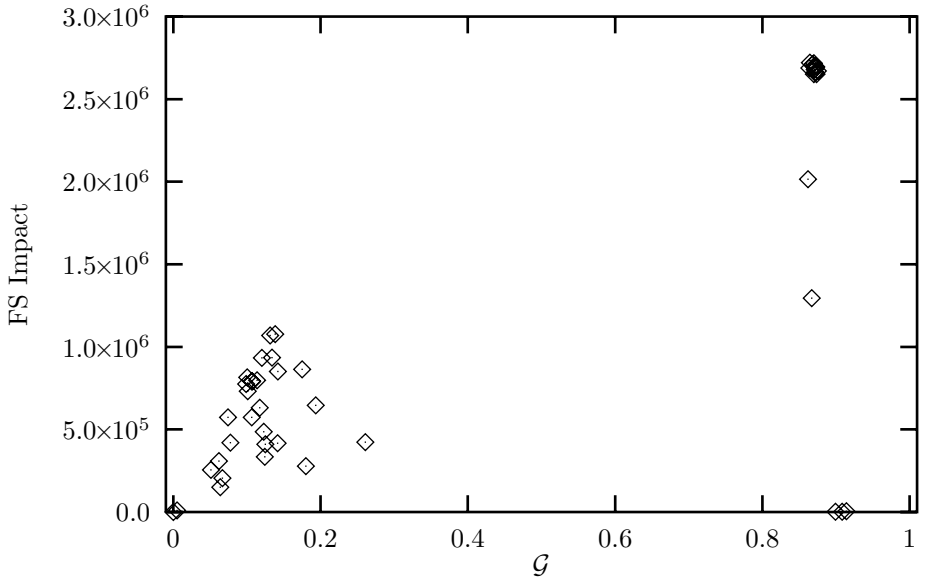

Figure 6.14: *Mp3d, 8k-byte page, update coherency ($\mathcal{G}$)*

points clipped in order to show the detail of the majority of the pages. The clipped points range in impact value from 30000 to 60000 and span the entire range of $\mathcal{G}$. In the expiring update coherency protocol case, there is a wide variation in impact for $\mathcal{G} < 0.2$, and little impact for $\mathcal{G} > 0.3$. This is shown in Figure 6.13, which has four points clipped to show the detail. The overall trend in this case is the opposite of what we expect from a predictor.

The value of $\mathcal{G}$ for the 8k-byte, update coherency architecture (Figure 6.14) provides a reasonably good prediction of false sharing impact. Only a few pages with high $\mathcal{G}$ exhibit no false sharing impact, and all of the pages with high impact have high $\mathcal{G}$ values. We have few false-positive predictions. The invalidate coherency case lacks this predictive power, as shown in Figure 6.15. There are many pages with high $\mathcal{G}$ but low impact, which means the prediction results in many false-positive indications of high false sharing impact. The expiring update case is similar to the update case, but shows a uniform increase in false sharing impact per page.

6.1.4 Water

The simulation information for water is summarized in Table 6.4. Like the previous programs, the 8k-byte page size with invalidate coherency architecture simulation runs longer, but in this case only by about fifteen percent. The number of reads and writes is identical for each execution, and the ratio of reads to writes is approximately nine times more reads than writes.

Expiring pages in the update coherency for 64-byte page size greatly reduces the amount of data transferred among the nodes, but for the 8k-byte page size, it results in a small penalty (equivalent to fewer than eight page faults). In both cases, the number of pages that were expired is just slightly more than the number of pages which were used after being expired.

In Figure 6.16 is a plot of the impact of each of the 2191 pages in the Water program for the update coherency, 64-byte page size architecture simulation. The pages exhibit only 24 different ($\mathcal{G}$,$Impact$) tuples, as shown. The range for the impact is very wide for $\mathcal{G}$ values close to each other. The highest impact measured in this application is at $\mathcal{G} = 0.019886$. The expiring update case is plotted in Figure 6.17. The scale of impact is much higher, due to the high number of

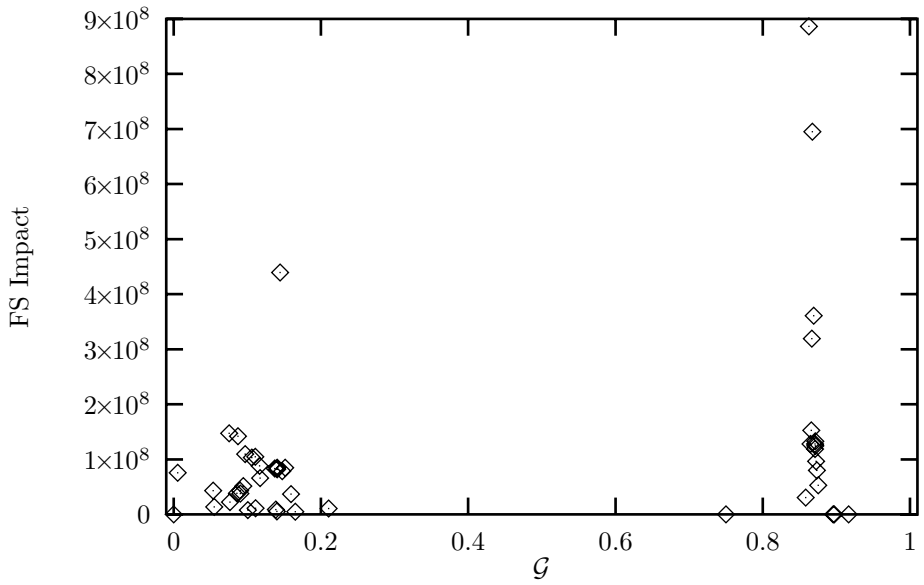

Figure 6.15: *Mp3d, 8k-byte page, invalidate coherency ($\mathcal{G}$)*

Memory Architecture	Run time	Reads	Writes	Data Bytes
Optimal	998252601ns	7607660	819431	2389048
64-byte, Invalidate	999481201ns	7607660	819431	4251840
64-byte, Update	998553201ns	7607660	819431	29890416
64-byte, Update-expire	998553201ns	7607660	819431	8828792
8k-byte, Invalidate	1149306701ns	7607660	819431	671473664
8k-byte, Update	998906341ns	7607660	819431	38581528
8k-byte, Update-expire	998906341ns	7607660	819431	38646632

Table 6.4: *Summary of simulation runs for Water*

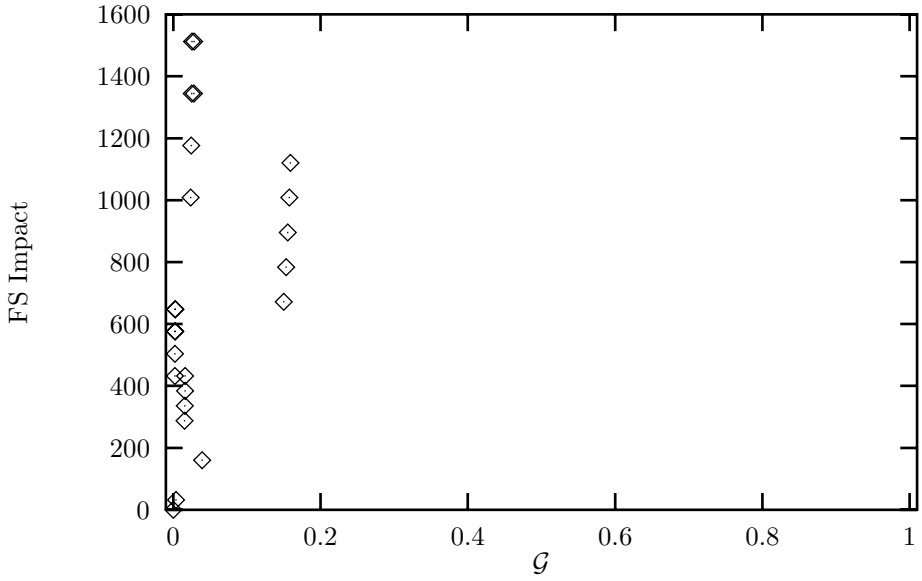

Figure 6.16: *Water, 64-byte page, update coherency ($\mathcal{G}$)*

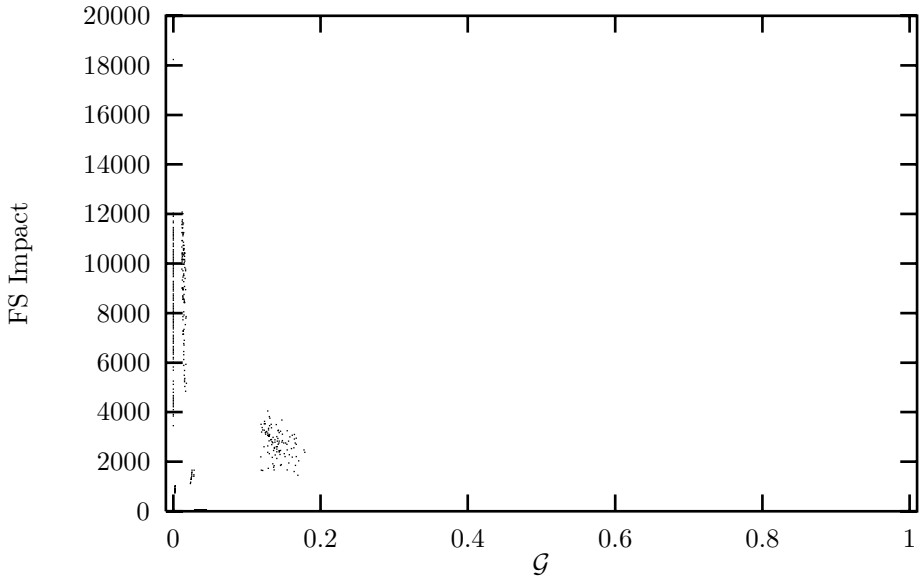

Figure 6.17: *Water, 64-byte page, expiring update coherency ($\mathcal{G}$)*

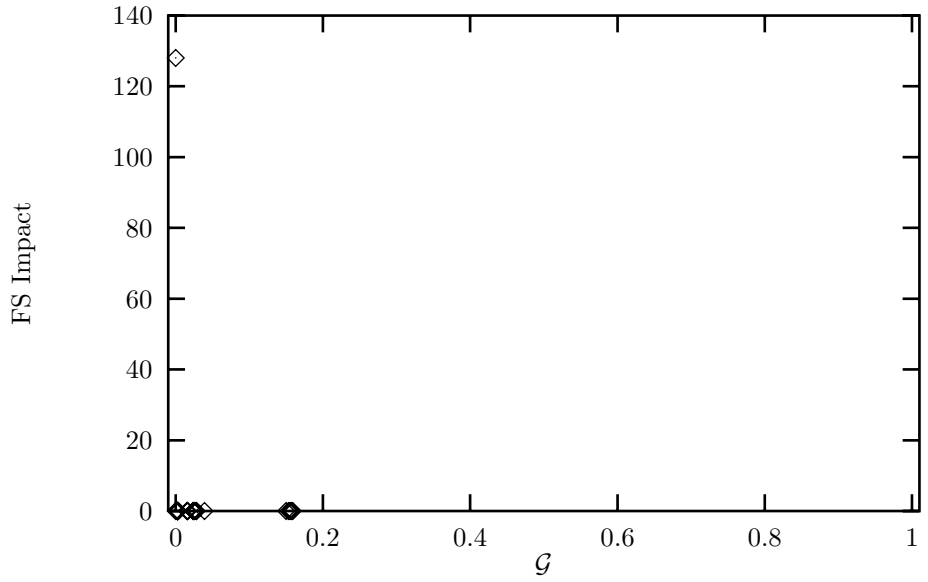

Figure 6.18: *Water, 64-byte page, invalidate coherency ($\mathcal{G}$)*

expire-induced page faults. Even though we attribute more false sharing cost to each page, the overall cost of all data traffic is reduced. For the invalidate case, shown in Figure 6.18, all but one page exhibits no false sharing impact, and that page has $\mathcal{G} = 0.0$. The impact of this page is equal to only two page faults. This program has very little sharing since the data structures are accessed by the same processor repeatedly. When there is sharing, it is mostly read-sharing, and thus causes very little coherency traffic, especially when using the invalidate and update with expire coherency policies. It is clear that $\mathcal{G}$ does not predict impact at all in these cases.

In the 8k-byte update coherency architecture, there is a clear increasing trend of the impact as $\mathcal{G}$ increases, except for the one page with the highest $\mathcal{G}$ (which has zero impact). This is shown in Figure 6.19. $\mathcal{G}$ seems to be a useful predictive tool in this case. The expiring update case looks similar, but has a slightly higher impact scale. In the invalidate case, all but three pages have very high impact and all have values of $\mathcal{G}$ close to each other. As seen in Figure 6.20, the page with the highest $\mathcal{G}$ value exhibits no false sharing impact.

6.1.5 Discussion

The data from this section show that the proposed predictor, $\mathcal{G}$, only works for a couple of particular cases. The data access patterns for these cases are where the shared data is written simultaneously by multiple processors with interspersed reads. This happens in the Mp3d and Water programs. The effect is greatest in the 8k-byte page size with update coherency architectures. The Mp3d program has low processor locality and a high degree of write-sharing. The same data structures are written by the same processors at each time step in the computation, but are referenced by all processors in each time step. In the Water program, the updates to the data structures are all clustered by phases, so they happen pretty much concurrently. Even in these cases, the range of $\mathcal{G}$ is very small, and the correlation somewhat disappointing.

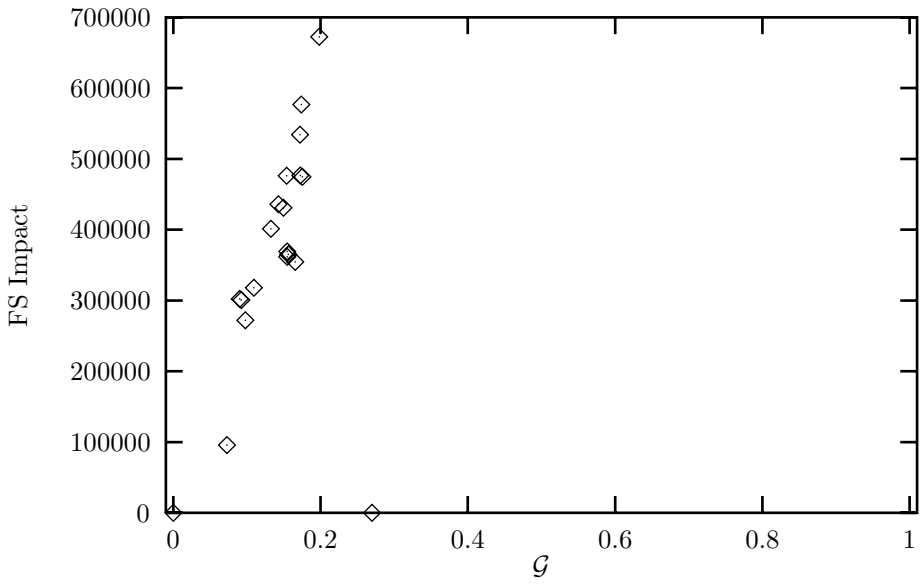

Figure 6.19: *Water, 8k-byte page, update coherency ($\mathcal{G}$)*

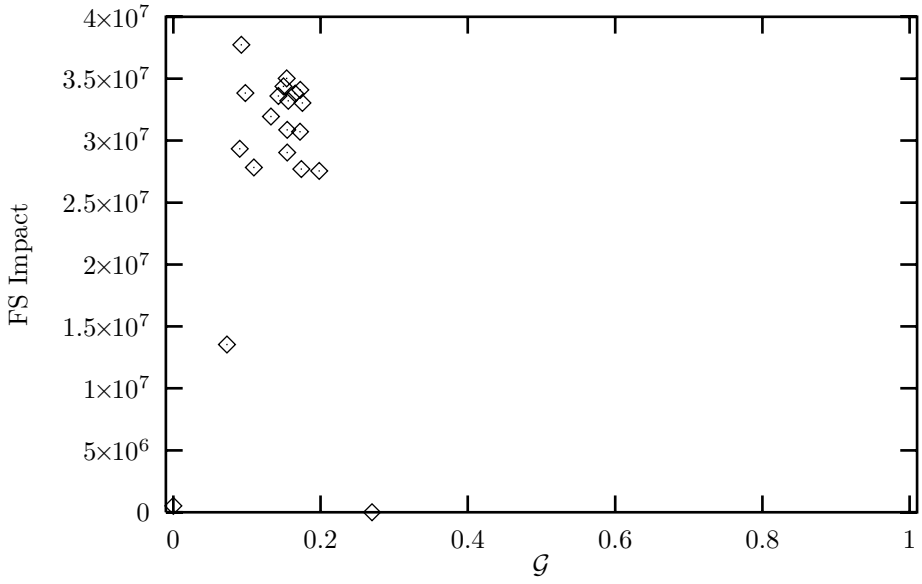

Figure 6.20: *Water, 8k-byte page, invalidate coherency ($\mathcal{G}$)*

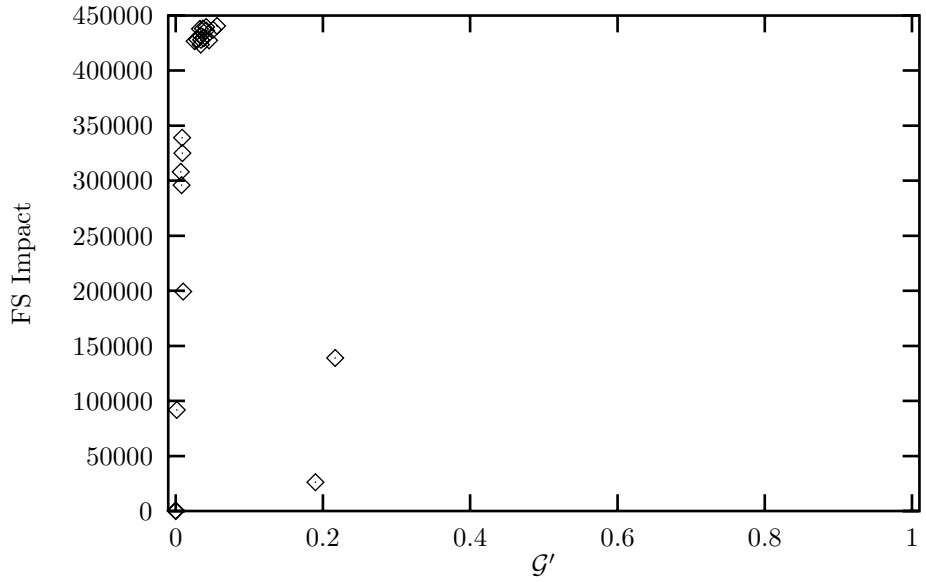

Figure 6.21: *Barnes-Hut, 8k-byte page, update coherency ($\mathcal{G}'$)*

6.2 Evaluation of $\mathcal{G}'$ as predictor

6.2.1 Barnes-Hut

Since Barnes-Hut is a real program, it will tend to make more read references than write references. For this application, the number of reads is roughly 28 times that of the number of writes. Because of this difference, we would expect the $\mathcal{G}'$ measure (from Equation 2.7) to be more accurate. However, as we see in Figure 6.21, this is not the case. Figure 6.22 also displays this for invalidate coherency. The only difference between the comparison of $\mathcal{G}$ and $\mathcal{G}'$ in these cases is that the range of the metric compared is smaller for $\mathcal{G}'$ than for $\mathcal{G}$. This can be attributed to the large difference in total number of references compared to the number of writes to each page.

6.2.2 Cholesky

In comparing $\mathcal{G}'$ to the false sharing impact for this program we do not get any better correlations than for $\mathcal{G}$. In Figure 6.23 is the 8k-byte update coherency case. It is similar to the $\mathcal{G}$ comparison in Figure 6.9, but the range of $\mathcal{G}'$ is reduced. Similarly for the invalidate case in Figure 6.24, we see that $\mathcal{G}'$ contains virtually no information about the actual impact on false sharing of those pages. For the 64-byte page size architectures, we observe similar compression of the range in the proposed predictor measure, $\mathcal{G}'$.

6.2.3 MP3D

The number of writes is nearly equal to the number of reads in this program. Based on the results of the previous programs, we would expect that the $\mathcal{G}'$ measure would be about half of $\mathcal{G}$ for each page. The plots in Figure 6.25 and Figure 6.26 confirm this. Comparing $\mathcal{G}'$ does not lead us to conclude that we can predict impact based on its value. Indeed, using it as a predictor would result in a very high number of false-negative indications of high impact.

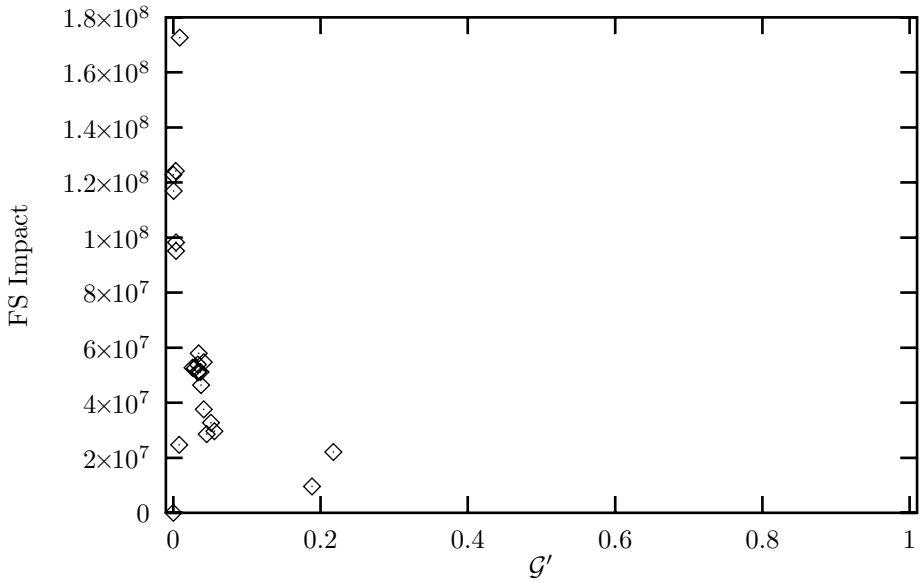

Figure 6.22: *Barnes-Hut, 8k-byte page, invalidate coherency ($\mathcal{G}'$)*

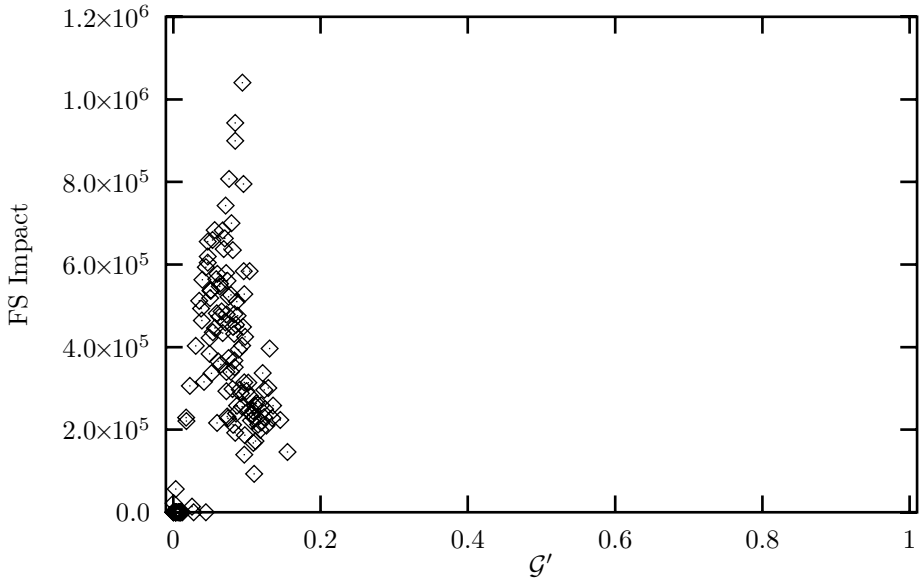

Figure 6.23: *Cholesky, 8k-byte page, update coherency ($\mathcal{G}'$)*

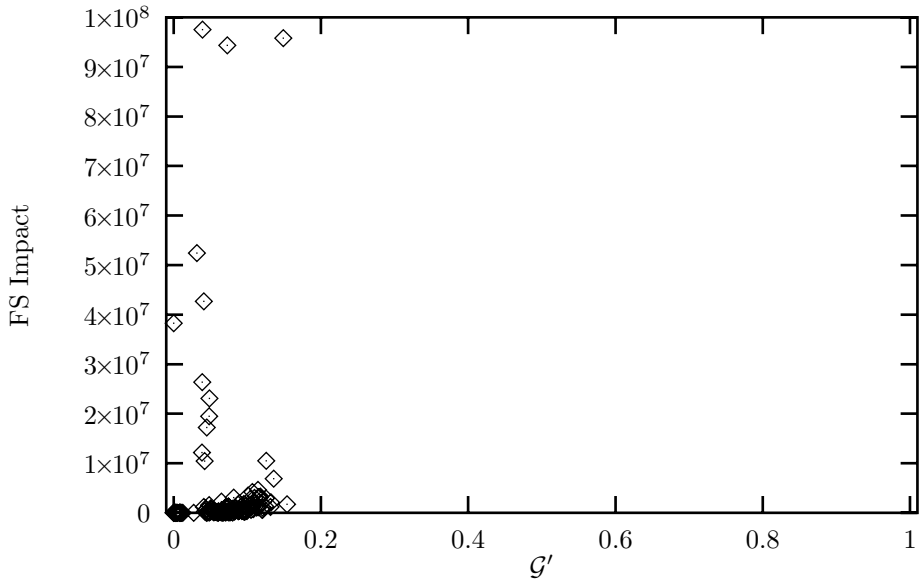

Figure 6.24: *Cholesky, 8k-byte page, invalidate coherency ($\mathcal{G}'$) (clipped)*

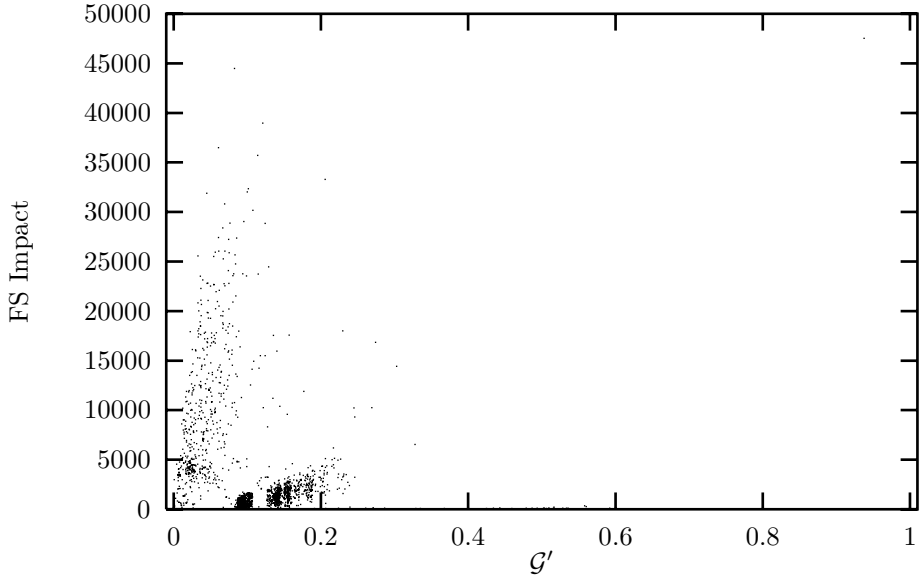

Figure 6.25: *Mp3d, 64-byte page, update coherency ($\mathcal{G}'$)*

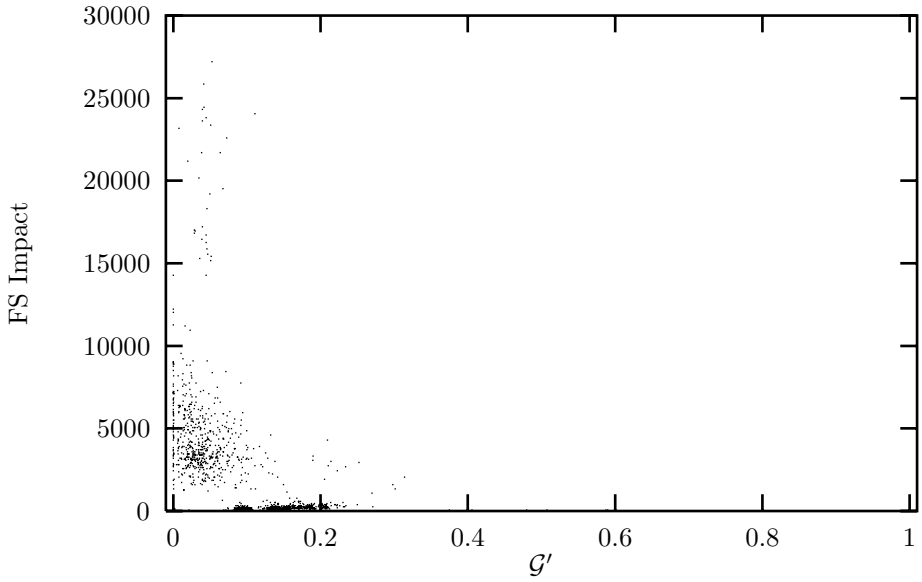

Figure 6.26: *Mp3d, 64-byte page, invalidate coherency ($\mathcal{G}'$) (clipped)*

6.2.4 Water

The comparison of $\mathcal{G}'$ to false sharing impact reveals very little new information over that of $\mathcal{G}$. Figure 6.27 contains the plot for the update coherency 64-byte page size architecture. All that we observe is that the range of $\mathcal{G}'$ is smaller than that of $\mathcal{G}$ and the points representing the pages are condensed in the horizontal axis. There is no improvement of predictive power. The plots for the other cases do not provide any additional insight, so are not presented here.

6.2.5 Discussion

Evaluating the programs in terms of $\mathcal{G}'$ does little to improve the prediction over $\mathcal{G}$. We had expected a more accurate prediction because this measure is biased toward write references. It turns out that the number of writes to shared data structures is low in these programs. While writes to memory directly induce coherency traffic, read references are important in building up the current set of active pages. The importance of writes designed into $\mathcal{G}'$ appears to be misguided in these cases.

6.3 Evaluation of $\mathcal{G} \times Mods$ as predictor

It is clear that using $\mathcal{G}$ or $\mathcal{G}'$ alone is ineffective at predicting the false sharing impact in a real program. In an attempt at equalizing the importance of the predicting metric across pages, we scale $\mathcal{G}$ by the number of writes to each page. This takes into account the relative importance of a page in its direct contribution to false sharing impact. The intuition here is that coherency operations are triggered by write references. We hope to re-organize the pages in the horizontal direction such that a trend might appear. This is different than using $\mathcal{G}'$ as the predictor; in this case we are scaling by the absolute number of writes, rather than the ratio of writes to reads.

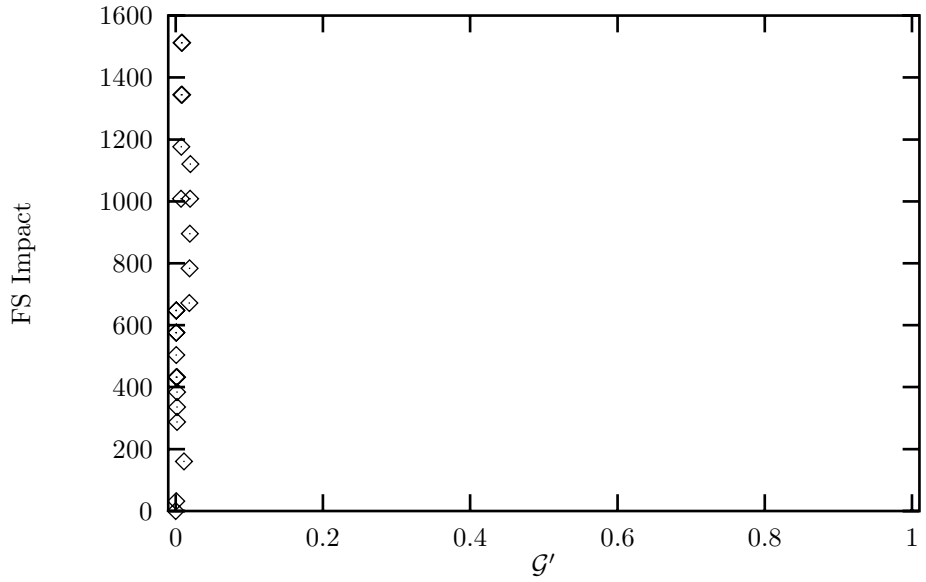

Figure 6.27: *Water, 64-byte page, update coherency ($\mathcal{G}'$)*

6.3.1 Barnes-Hut

In Figure 6.28 we see this plot for the 64-byte page invalidate coherency architecture. This plot is clipped to show the detail of the plot; the full range goes up to 700000 bytes. There is no correlation for this proposed metric. Figure 6.29 presents a similar plot for update coherency. Higher values of the $\mathcal{G} \times Mods$ metric do not imply higher false sharing impact. Therefore this metric is of little use to the programmer to isolate the causes of false sharing related performance problems.

We present the plots of $\mathcal{G} \times Mods$ for 8k-byte pages in Figure 6.30 and Figure 6.31 for completeness. The update-expire case (Figure 6.32) also has little correlation. Only the update case shows the slightest hint of correlation of the predictor metric with the impact.

6.3.2 Choleksy

In the 64-byte page update coherency case, Figure 6.33, we see a trend of increasing cost with increasing $\mathcal{G} \times Mods$, but closer inspection shows that there is too much variation for the lower values of the predictor for it to be accurately used. The highest impact is measured for lower values of the proposed predictor, a false-negative prediction. In the invalidate case (Figure 6.34), we find no useful relationship between the predictor and the impact.

In a pleasing departure from the failures of many of the preceding tests to show any predictive capability, the 8k-byte update case of this program shows a remarkable correlation between this proposed predictor and the false sharing impact, presented in Figure 6.35. Only one or two pages fail to conform to the linear relationship. However, for the case where the pages are expired, the reduction in false sharing impact also totally destroys the linear relationship. This is shown in Figure 6.36. The data structures are referenced by multiple processors until a certain condition is met, after which the data structure is used by only one processor. In the normal update case, updates are continually sent to all processors that ever referenced the page holding the data structure, but in the invalidate and update with expire coherency mechanisms, these extra data coherency operations are eliminated. The $\mathcal{G}$ measure expects constant data traffic, thus

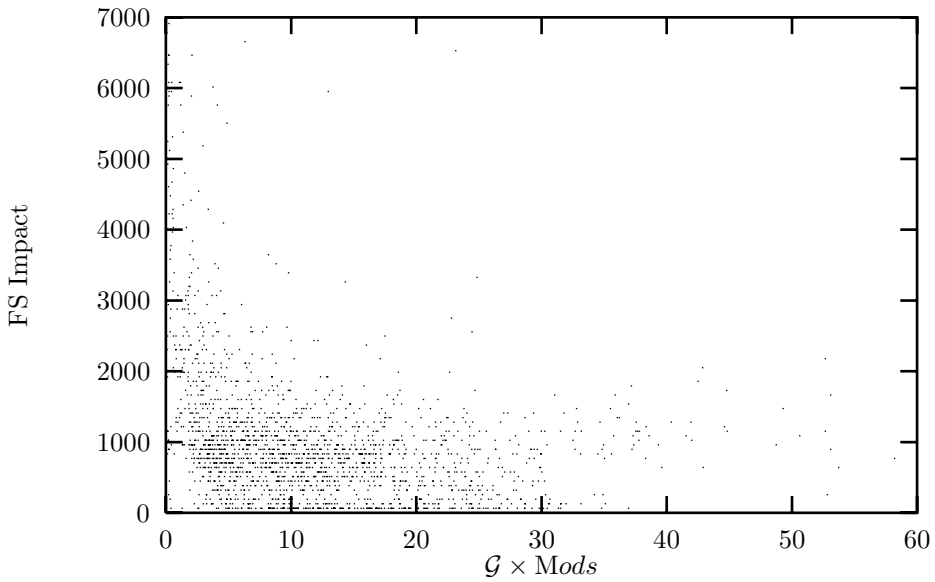

Figure 6.28: *Barnes-Hut, 64-byte page, invalidate coherency ($\mathcal{G} \times Mods$)*

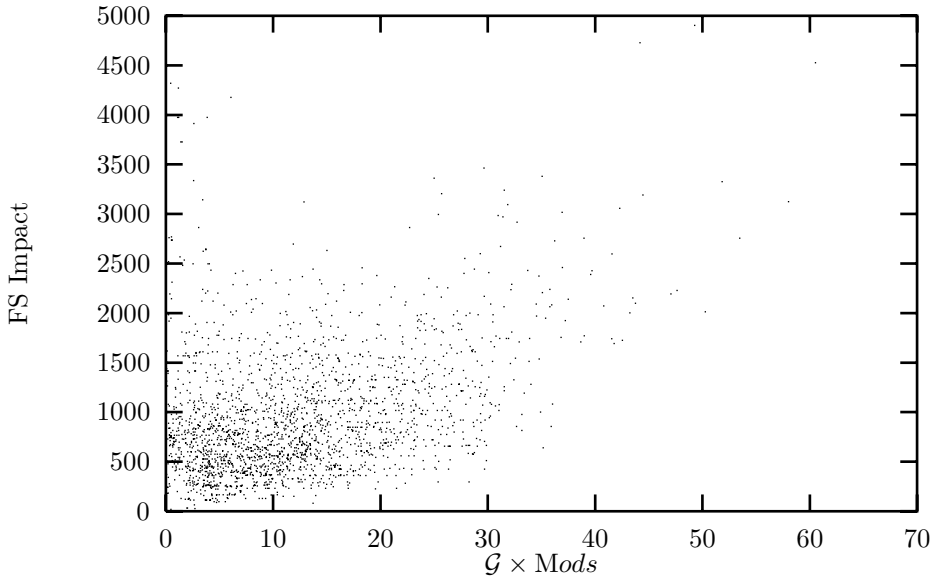

Figure 6.29: *Barnes-Hut, 64-byte page, update coherency ($\mathcal{G} \times Mods$)*

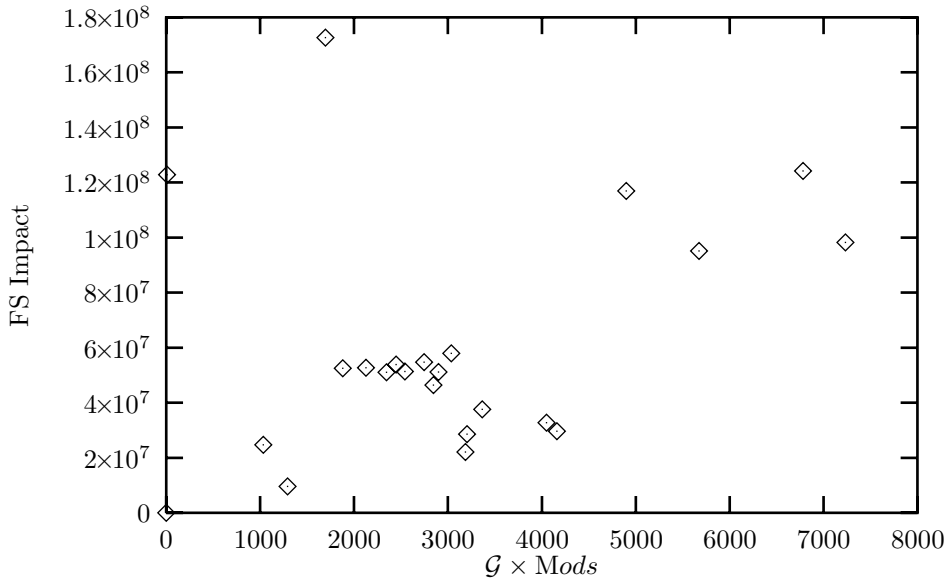

Figure 6.30: *Barnes-Hut, 8k-byte page, invalidate coherency ($\mathcal{G} \times Mods$)*

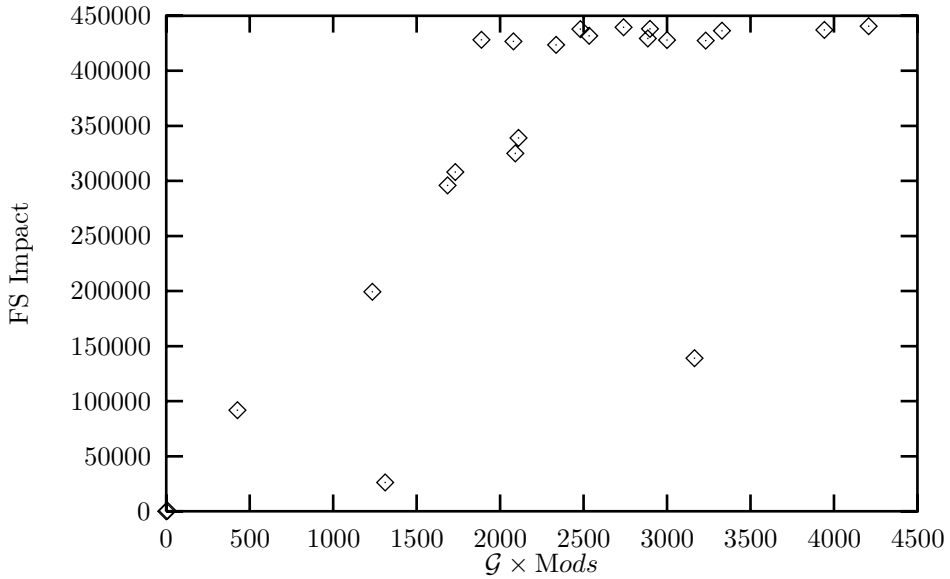

Figure 6.31: *Barnes-Hut, 8k-byte page, update coherency ($\mathcal{G} \times Mods$)*

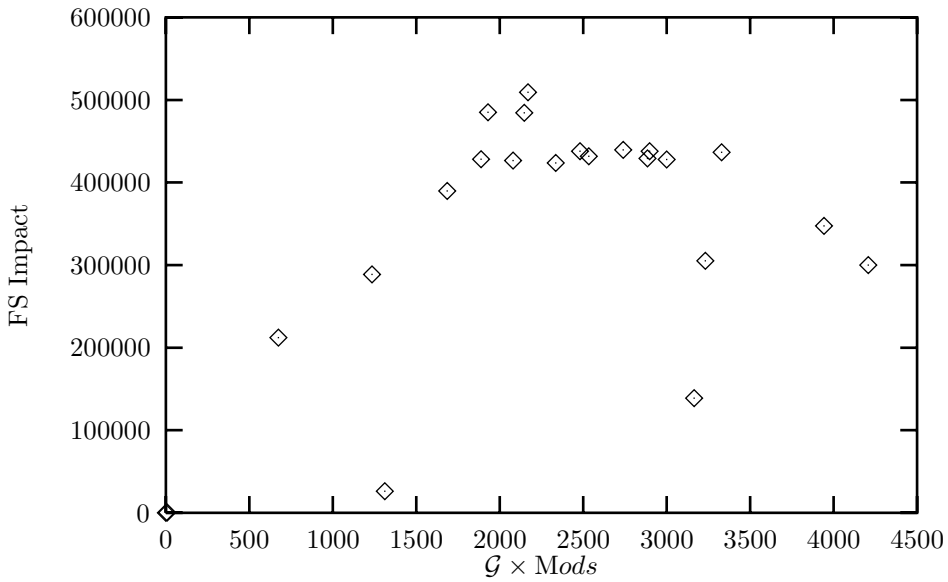

Figure 6.32: *Barnes-Hut, 8k-byte page, expiring update coherency ($\mathcal{G} \times Mods$)*

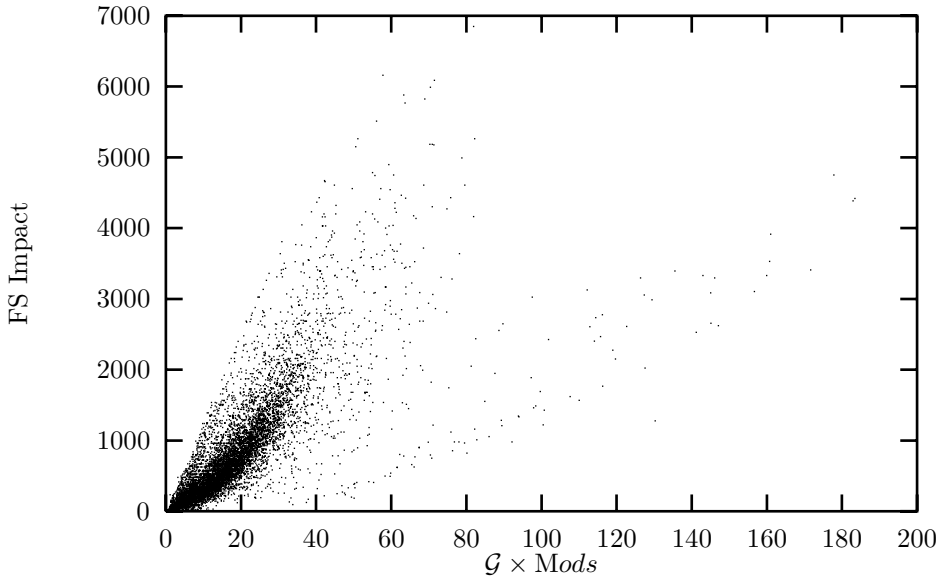

Figure 6.33: *Cholesky, 64-byte page, update coherency ($\mathcal{G} \times Mods$)*

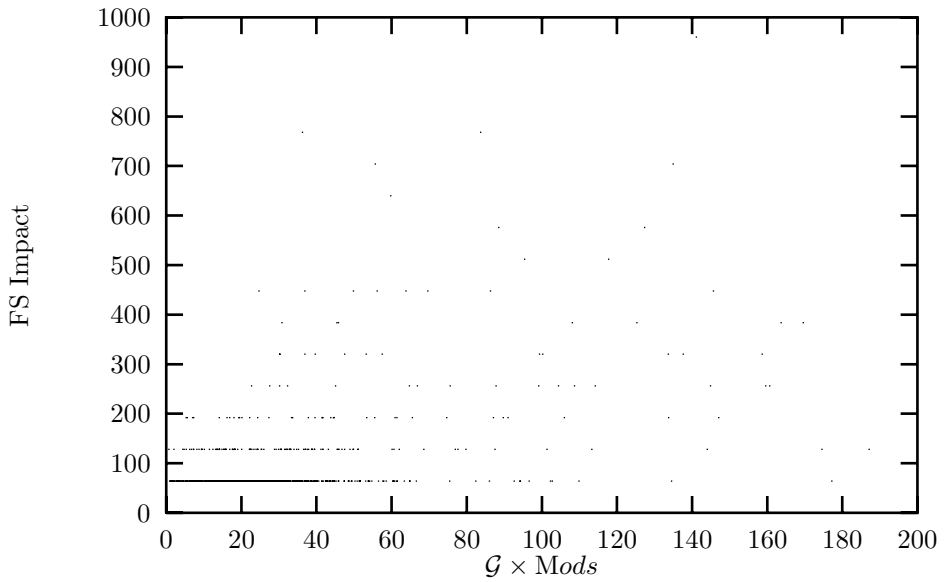

Figure 6.34: *Cholesky, 64-byte page, invalidate coherency ($\mathcal{G} \times Mods$)*

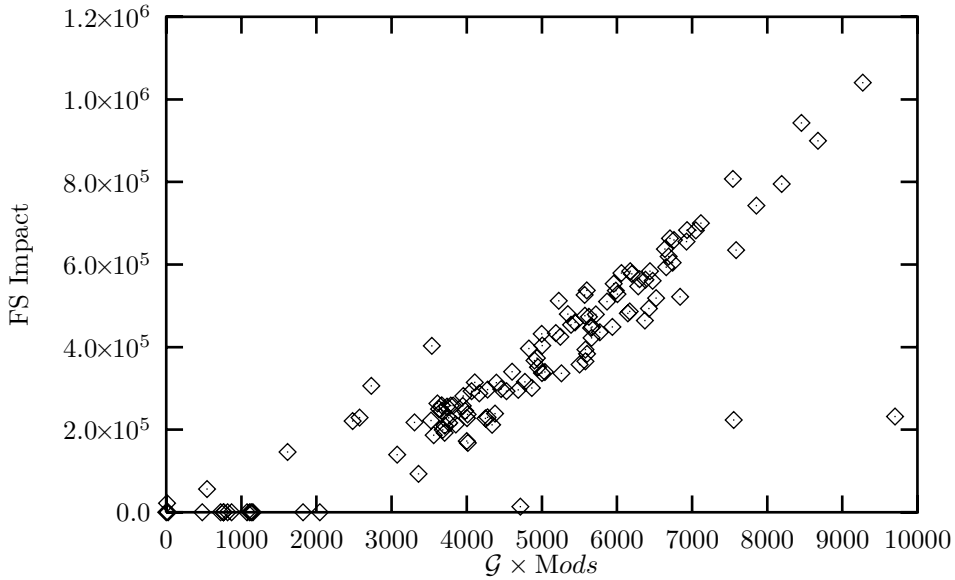

Figure 6.35: *Cholesky, 8k-byte page, update coherency ($\mathcal{G} \times Mods$)*

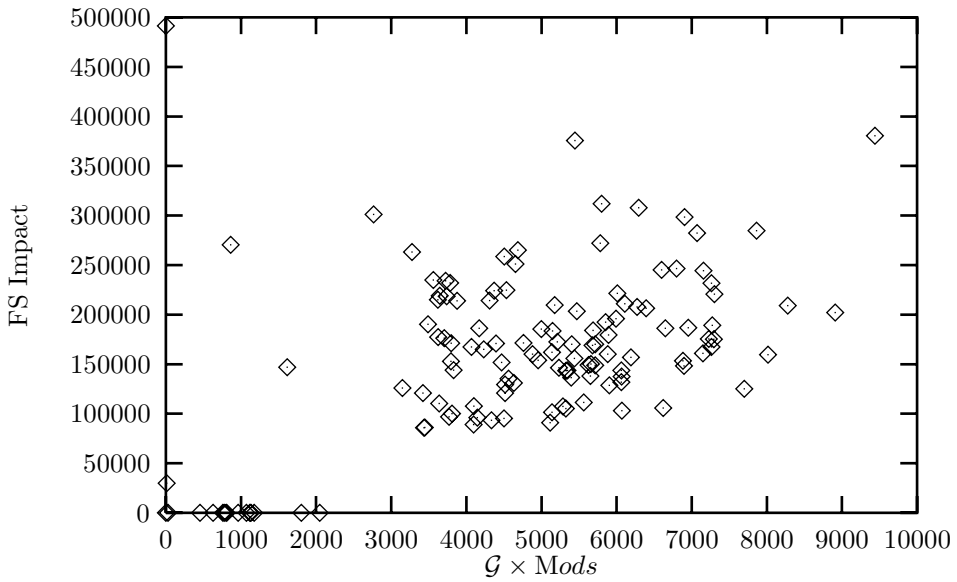

Figure 6.36: *Cholesky, 8k-byte page, expiring update coherency ($\mathcal{G} \times Mods$)*

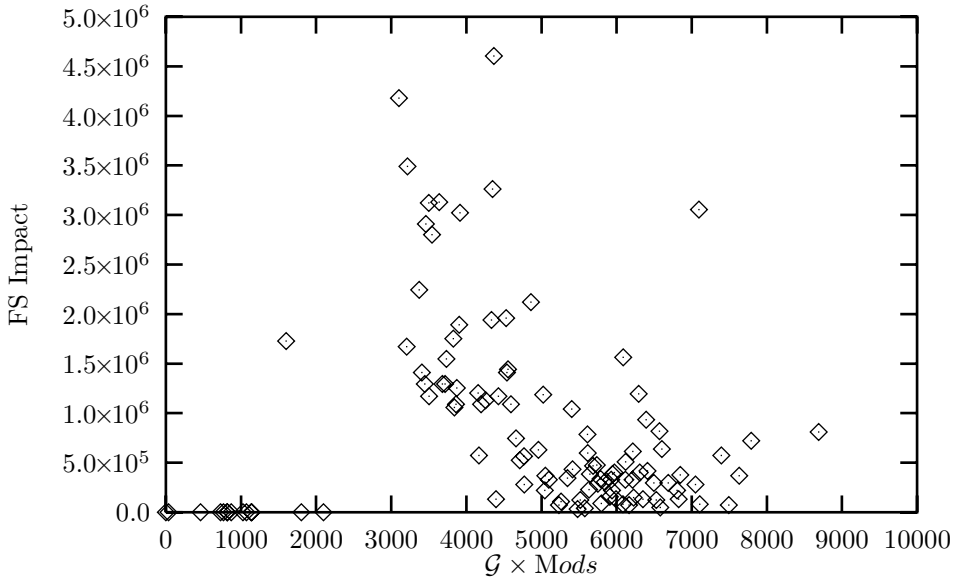

Figure 6.37: *Cholesky, 8k-byte page, invalidate coherency ($\mathcal{G} \times Mods$)*

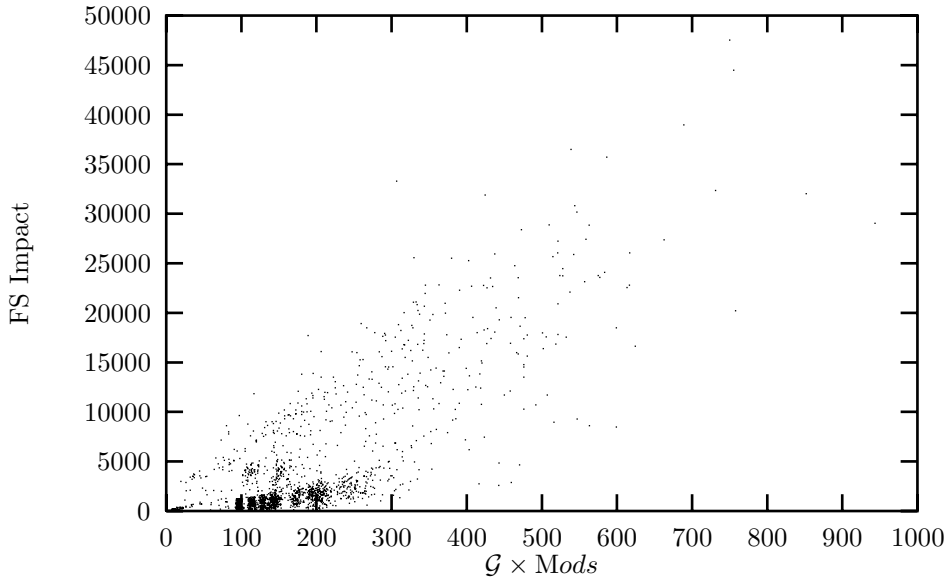

Figure 6.38: *Mp3d, 64-byte page, update coherency ($\mathcal{G} \times Mods$)*

these extra updates result in traffic that more closely match the assumptions. For the invalidate coherency protocol, there is also no correlation as seen in Figure 6.37. The full range of this clipped plot is up to 4×10^8 bytes, with only ten pages (points) being outside the shown range, mostly in the central horizontal region.

6.3.3 MP3D

By scaling the $\mathcal{G}$ measure by the number of writes to each page, we see a nice trend of higher impact for a higher measure in the 64-byte update architecture (Figure 6.38). There is still some variation for each predicted value, but this must be expected given the summary nature of the $\mathcal{G}$ metric. An imaginary line connecting the points along the top edge of the "cloud" of points serves to show the upper bound of impact that the measure predicts. With expiring pages in the update case, Figure 6.39, the trend changes from increasing to decreasing as the metric increases. The change is similar to the change in the $\mathcal{G}$ *vs. Impact* plots. For the invalidate coherency case, we see no useful correlation between the measure and the impact, as shown in Figure 6.40. The highest measured impact values occur at the lower values of the proposed predictor. At the same time, the lowest measured impact occurs in the same region.

For an architecture with 8k-byte page size and update coherency, we again observe a very nice correlation between the metric and the impact. Figure 6.41 shows a nearly perfectly linear relationship between the two. The linearity is still visible in the expiring update case (Figure 6.42) but is not as perfect. This is because many unnecessary updates have been eliminated for some pages. As before with the invalidate coherency case (Figure 6.43) we completely fail to be able to predict.

The reason for these changes in predictive ability with coherency policy is similar to that for the Cholesky program — the locality changes. The update coherency policy continues to send updates where they are no longer needed, but the other policies do not. Since $\mathcal{G}$ expects this traffic, it predicts the impact more accurately.

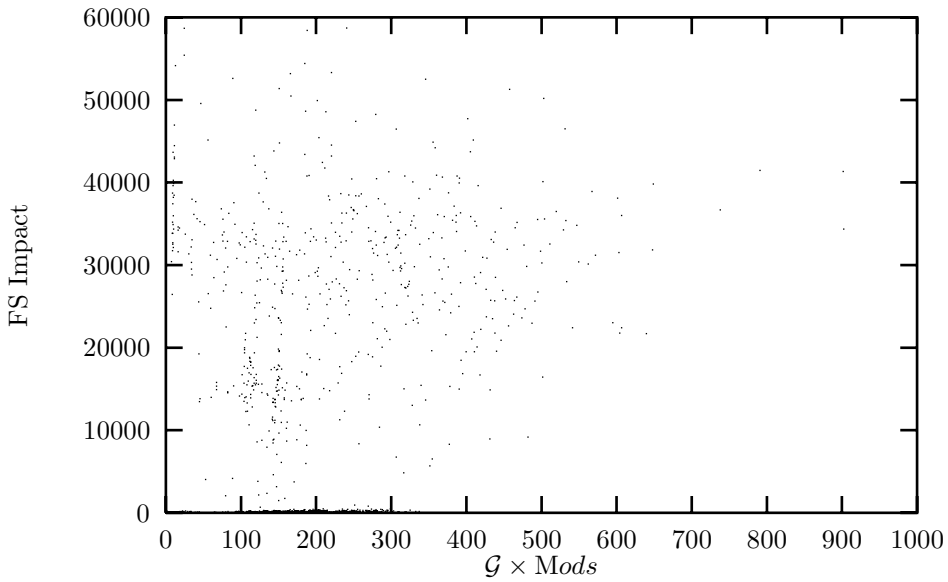

Figure 6.39: *Mp3d, 64-byte page, expiring update coherency ($\mathcal{G} \times Mods$) (clipped)*

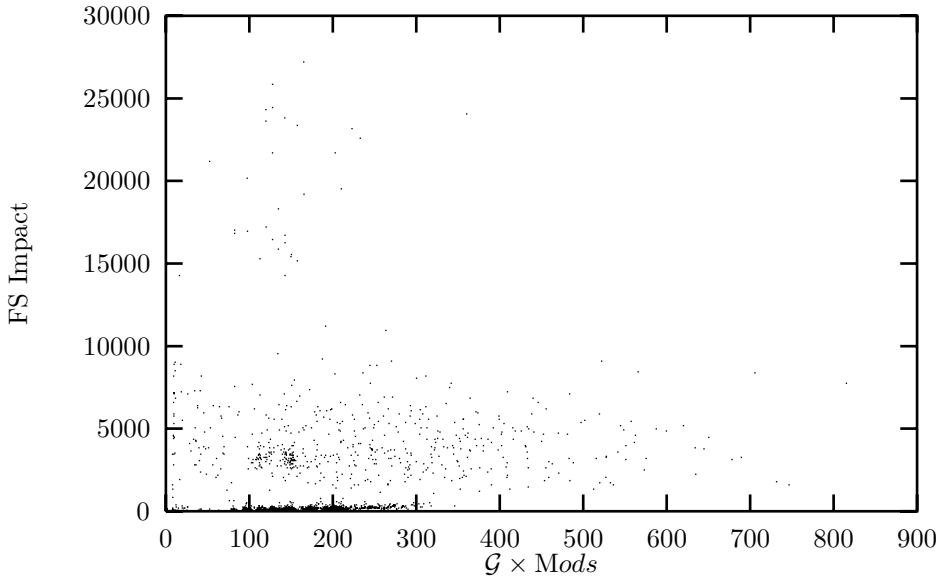

Figure 6.40: *Mp3d, 64-byte page, invalidate coherency ($\mathcal{G} \times Mods$) (clipped)*

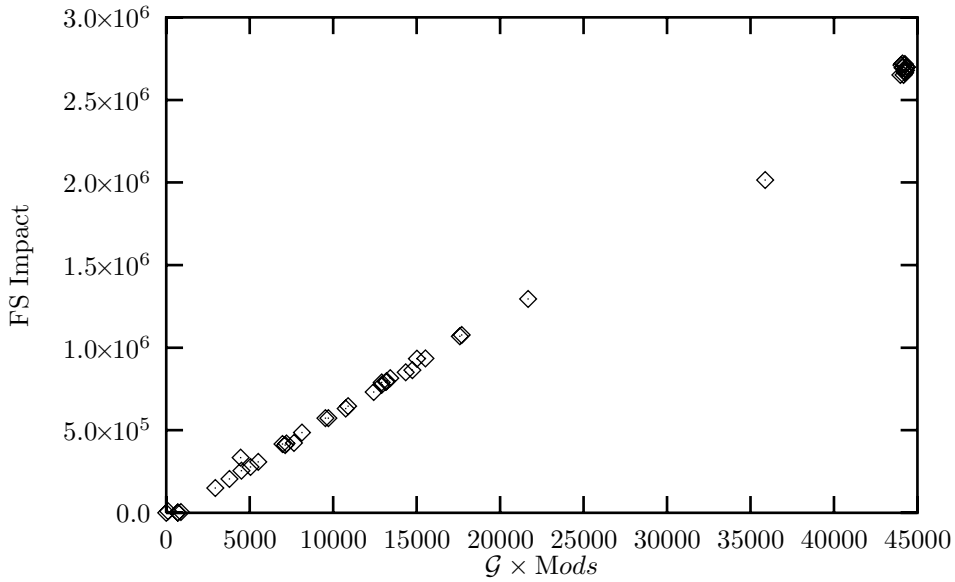

Figure 6.41: *Mp3d, 8k-byte page, update coherency ($\mathcal{G} \times Mods$)*

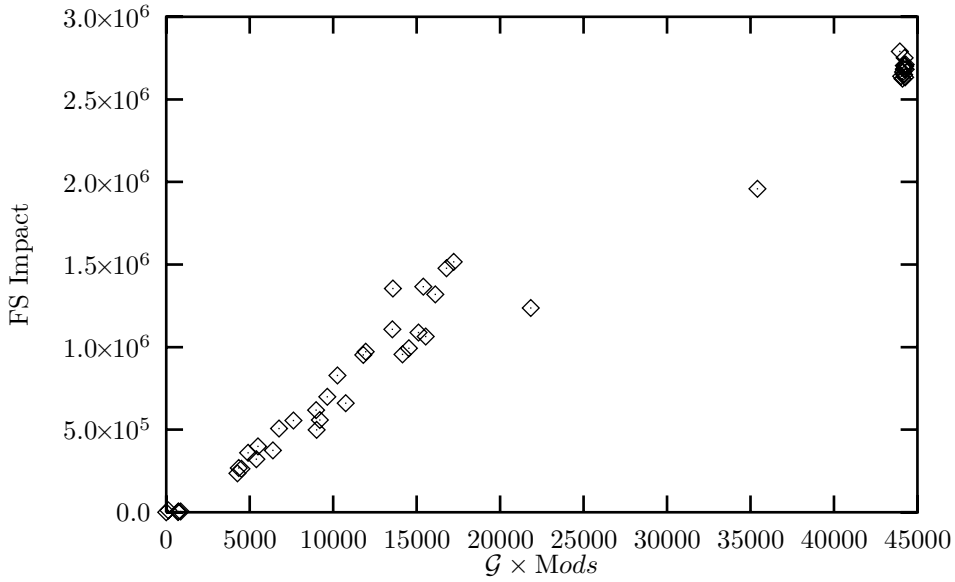

Figure 6.42: *Mp3d, 8k-byte page, expiring update coherency ($\mathcal{G} \times Mods$)*

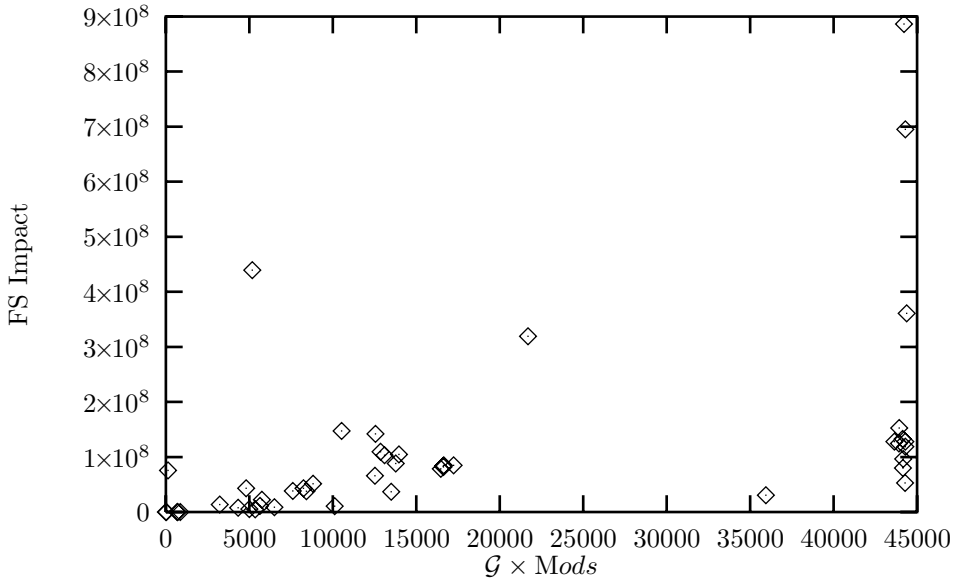

Figure 6.43: *Mp3d, 8k-byte page, invalidate coherency ($\mathcal{G} \times Mods$)*

6.3.4 Water

Even with the scaling of the number of modifications (writes) to each page, we are not able to predict impact based on the measure. The 64-byte page size update coherency case shown in Figure 6.44 has three regions of $\mathcal{G} \times Mods$. Each region has false sharing impact that varies widely from low to high. The expiring update case has a similar distribution with a higher impact scale. There is little correlation between the metric and the impact. The invalidate coherency case (Figure 6.45) has all but one page with zero impact, so does not improve over the original unscaled measure in Figure 6.18.

Both the update and invalidate coherency models for the 8k-byte page size architectures provide reasonably good prediction between the $\mathcal{G} \times Mods$ measure and the false sharing impact. In the update case of Figure 6.46, the trend is nearly linear. The expiring update case (not shown) has a similar plot, but with a lower impact scale. For the invalidate case, Figure 6.47, the high measure values all have high impact, and low measures indicate low impact. However, there is little distinction in impact among the higher predictor measure values. The accuracy of the update case is due to the static nature of the pages being referenced. The same processors always reference the same data structures, so relatively few pages get expired.

6.3.5 Discussion

The evaluations of the programs in terms of this metric were no more successful than those of the previous section ($\mathcal{G}'$). The only successful predictions were in the same cases and for the same reasons as in that evaluation. This lends more support to our conjecture that the importance of writes is not as significant as we originally thought when developing the measures.

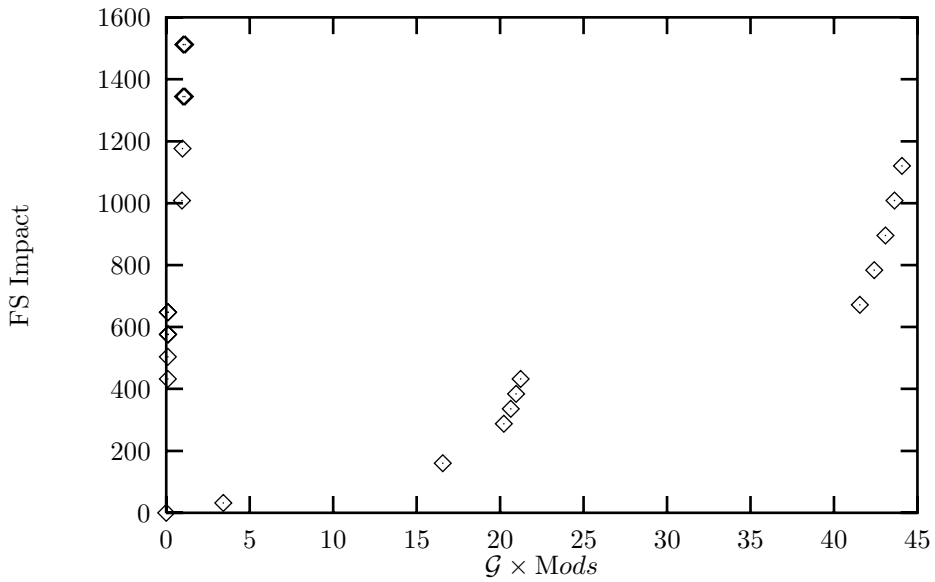

Figure 6.44: *Water, 64-byte page, update coherency ($\mathcal{G} \times Mods$)*

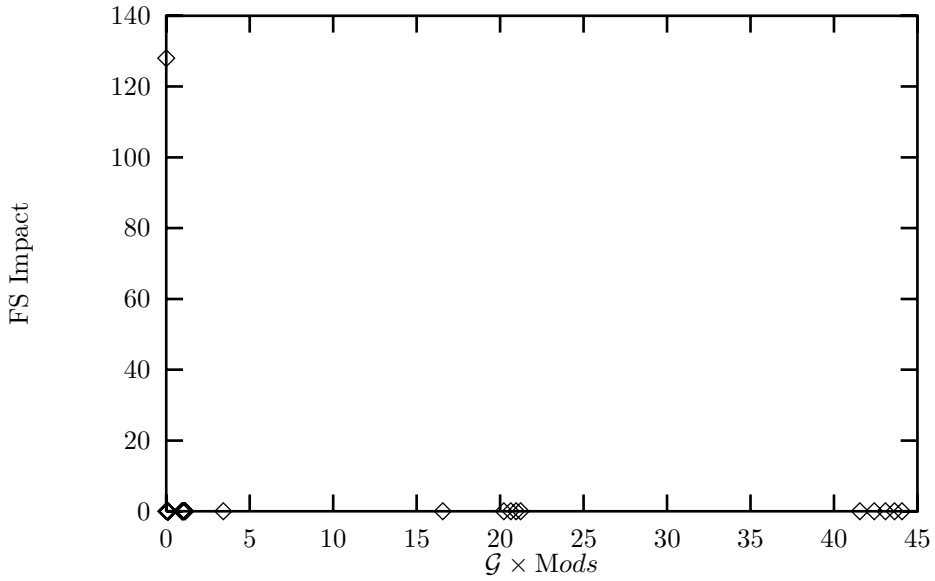

Figure 6.45: *Water, 64-byte page, invalidate coherency ($\mathcal{G} \times Mods$)*

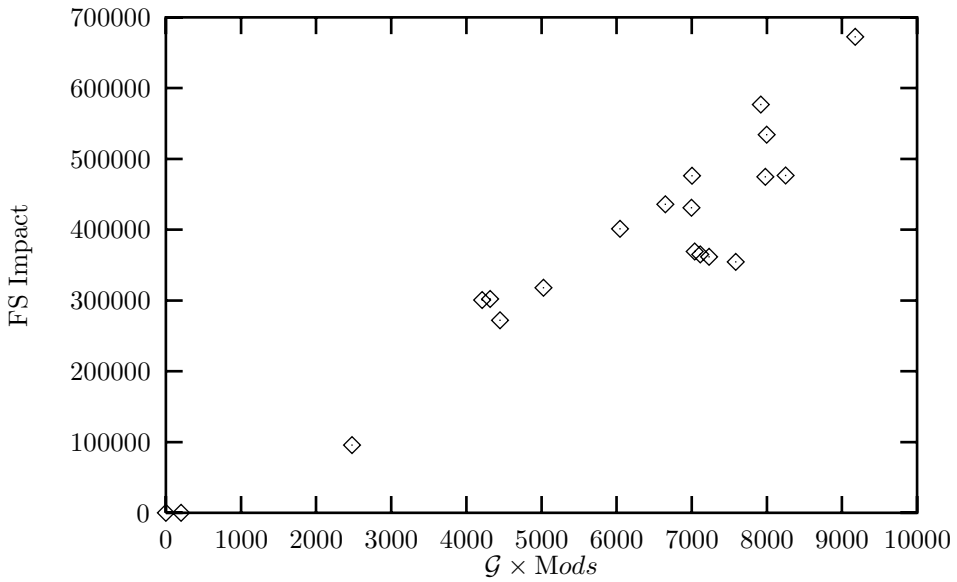

Figure 6.46: *Water, 8k-byte page, update coherency ($\mathcal{G} \times Mods$)*

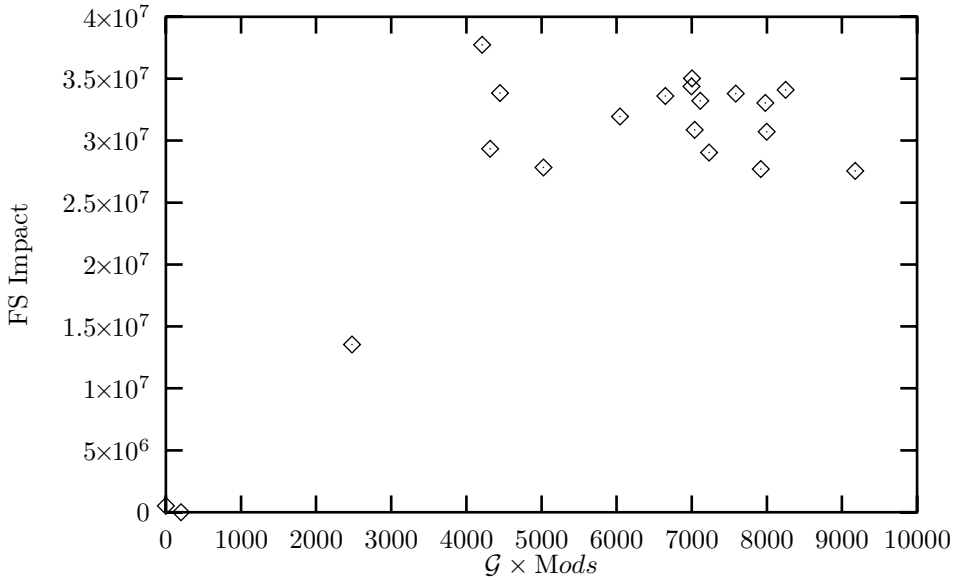

Figure 6.47: *Water, 8k-byte page, invalidate coherency ($\mathcal{G} \times Mods$)*

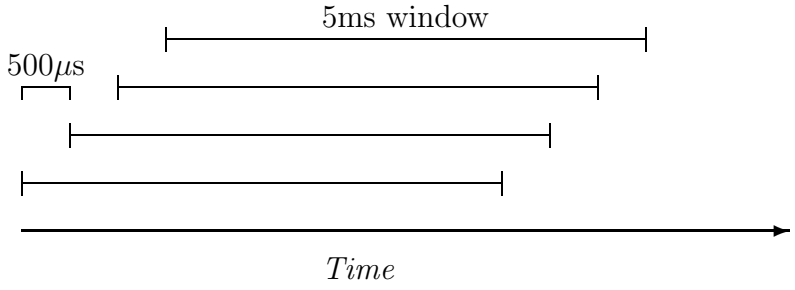

Figure 6.48: *Overlapping windows for working set calculations*

6.4 Limiting evaluation to phases

The observations of the synthetic programs indicate that the synchronized versions are more predictable than the others. The synchronization points indicated times during the execution that denoted a change in the area of shared memory which the program was accessing. The plot of $\mathcal{G}$ *vs. Impact* in Figure 6.2 for the Barnes-Hut program also showed high impact for low $\mathcal{G}$, which we attributed to sharing of pages in separate phases. Within phases, the pages had more false sharing than they appeared to have across phases. If we can determine the synchronization points or phases of execution in these programs, we can expect to be able to do better prediction. To this end, we use the size of the working set and its change to detect phase change boundaries.

The working set is computed over a sliding window of 5 milliseconds, and is reported every 500 microseconds ($\frac{1}{10}$ of the window size). That is, the working set is first reported after 5ms of runtime of the program, and again every 500μs thereafter for the previous 5ms of runtime. Figure 6.48 shows how the first four windows overlap as time progresses. The working set is reported at the end of each of the intervals.

6.4.1 Barnes-Hut

Determining phase changes

The working set size history of Barnes-Hut is presented visually in Figure 6.49 for the 64-byte update architecture. The dashed line indicates the size of the working set across time, and the impulses indicate the relative change in the size of the working set from the previously reported value. The same plot for invalidate coherency is in Figure 6.50.

From these plots, we see that there are definite phase changes as evidenced by the change in working set. After the initial burst at the left edge, which represents all the pages referenced since the beginning of the program, we see a large drop in the working set size. This is due to the way the size is reported, and shows that the start of the program uses many pages and then settles down to a relatively stable working set of approximately 1800 pages. Near time 5×10^7ns, we see a burst of new pages added to the working set, and 5ms later a corresponding drop. The duration between the rise and drop is exactly the length of our window. Thus, we can call this a phase change as the set of pages in the working set has changed — the program has moved on to another set of pages. We consider the phase change to take place at the point where the size of the working set increases sharply (5ms before it declines sharply). After an interval of about 2×10^7ns, the cycle repeats, so we say that this program has a phase of that length.

Figure 6.51 contains a plot of the working set size from the 8k-byte page, invalidate coherency run. We see similar patterns to the ones for the 64-byte page case, but with a lower number of pages. The reduced page count is expected because the data size remains the same but the size

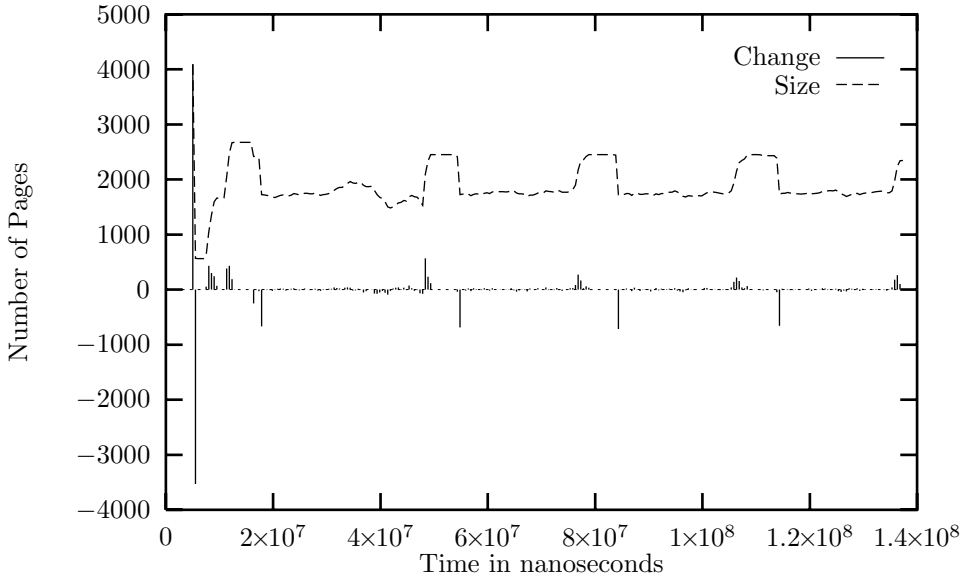

Figure 6.49: *Barnes-Hut working set size, 5ms window (U64)*

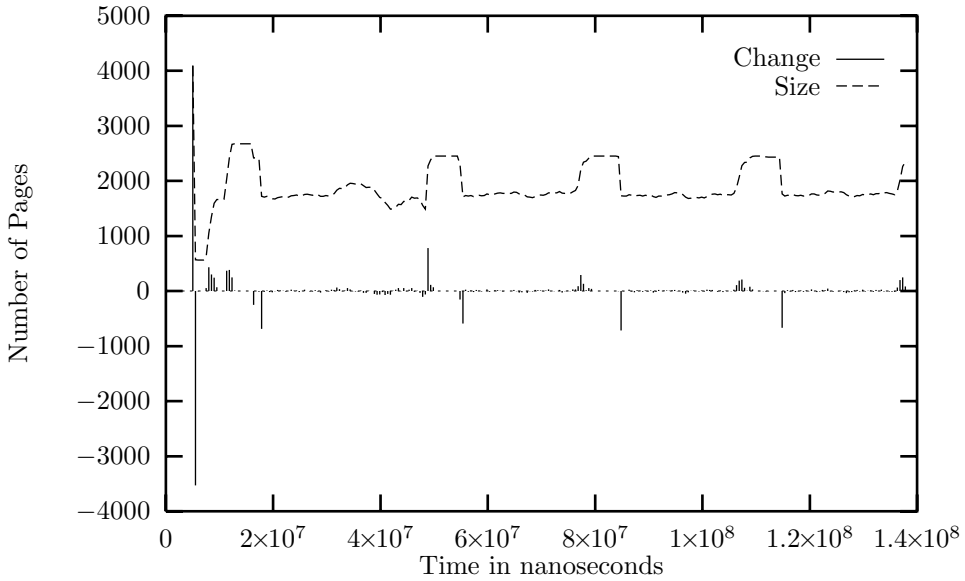

Figure 6.50: *Barnes-Hut working set size, 5ms window (I64)*

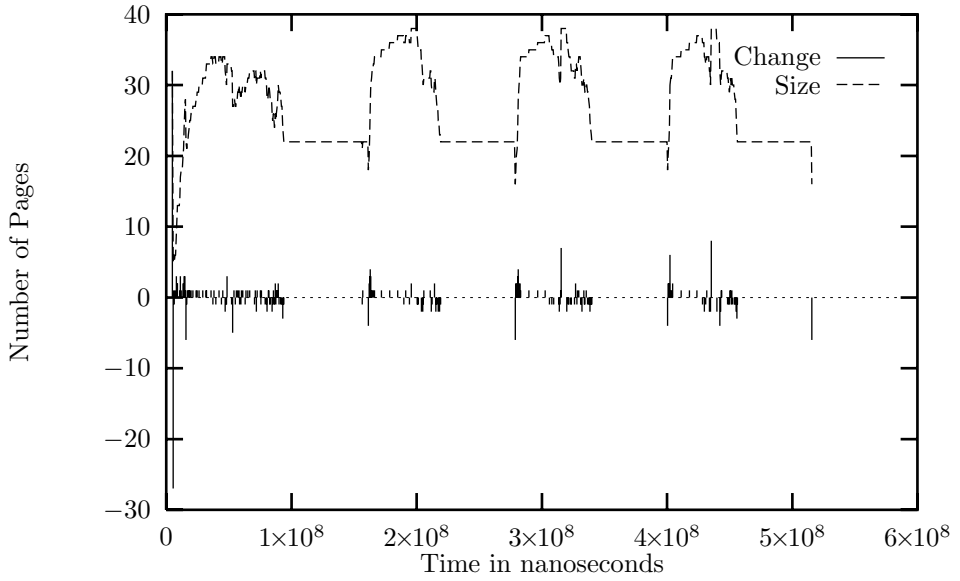

Figure 6.51: *Barnes-Hut working set size, 5ms window (I8k)*

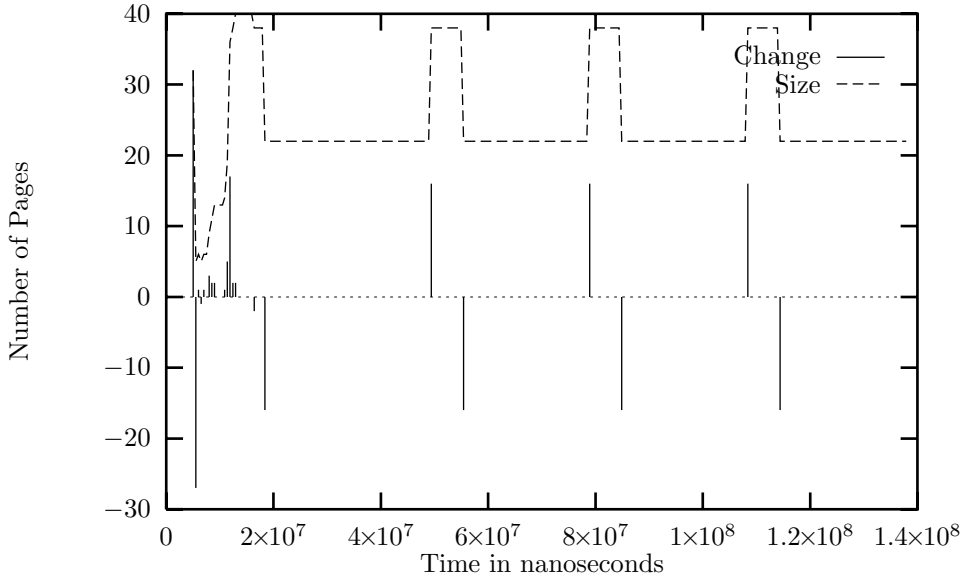

Figure 6.52: *Barnes-Hut working set size, 5ms window (U8k)*

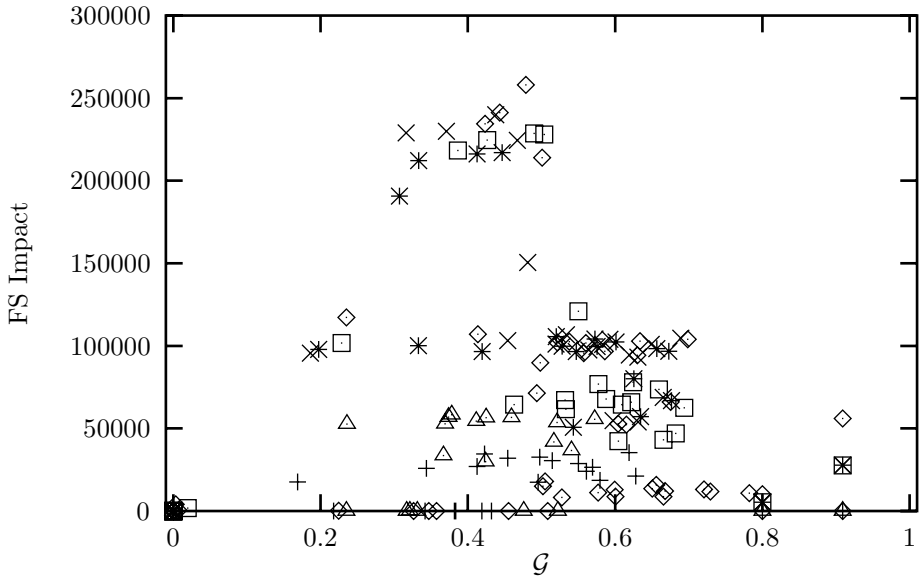

Figure 6.53: *Barnes-Hut, impact of $\mathcal{G}$, 20ms window (U8k)*

of the pages is increased. The invalidate coherency execution ran much longer and generated many more memory references than the other executions. There are many more pages which are active during this execution. The low level of 22 pages is a common feature of both this and the update coherency architecture, but the duration of the peaks is much longer, indicating more activity for periods longer than the time window. We can consider this to be evidence of some thrashing, and conclude that it is qualitatively different than the other runs of the same program. For 8k-byte pages with update coherency, Figure 6.52, we see the same sharp changes in working set size as we did with a 64-byte page size.

Changing the window size from 5ms to 10ms or 20ms results in similar looking plots, and confirms our conclusion of where the phase changes occur. The expiring update coherency cases are not substantially different in interpretation. These plots are not presented here.

Analysis of shorter time windows

Based on the evidence of phases in this application and previous experimental results, one would expect that by limiting the computation of $\mathcal{G}$ to the duration of a phase, the predictive ability would be improved. To test this hypothesis, we analyze the traces for the Barnes-Hut program in windowed segments. We chose a window size of 20ms, which corresponds to the observed phase length of the program. Unlike the windows used for the working set evaluation, these windows do not overlap.

In Figure 6.53 we present a plot comparing $\mathcal{G}$ to the false sharing impact for each of the six 20ms windows in the program's execution under update coherency. Notice that the range of the impact (the vertical axis) is lower than that of the same plot when computed over the entire run of the program (Figure 6.3). This is because the impact of a page is computed over a shorter interval of time, and there are fewer total references made. The different symbols in the plot indicate the different intervals from which each set of pages came. Each page appears as a point for each interval in which it was referenced. There seems to be very little correlation between $\mathcal{G}$ and the impact, as in the original comparison. Analysis of the expiring update coherency

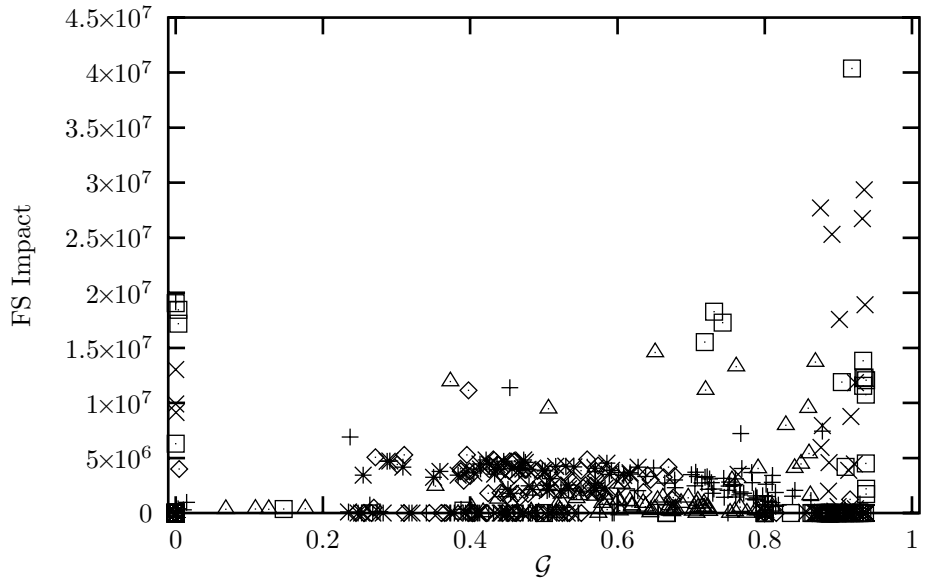

Figure 6.54: *Barnes-Hut, impact of $\mathcal{G}$, 20ms window (I8k)*

architectures shows no improvement in predictive ability, so is not presented here in detail.

The same plot for invalidate coherency is presented in Figure 6.54. There are many more points in this plot because the run time was much longer, thus we had 26 intervals. For clarity, a detail of the plot with the vertical axis range limited is shown in Figure 6.55. These plots show a slight increase in impact as $\mathcal{G}$ gets higher, but there are many pages with very low $\mathcal{G}$ having significant false sharing impact.

For the 64-byte page size cases, the plots are qualitatively the same as those for the entire program execution (Figure 6.1 and Figure 6.2) so are not presented here. Increasing or decreasing the window size in these analyses resulted in similar interpretations.

6.4.2 Cholesky

Determining phase changes

We plot the working set size and its change over the entire run of the program, computing the size over a 5ms interval, reporting it every 500μs. The Cholesky program doesn't show nice distinct phases like the Barnes-Hut program did. In the 64-byte page size cases, the change in working set size for both invalidate (Figure 6.56) and update (Figure 6.57) coherency follows the same progression over time.

For the 8k-byte page cases, the update coherency (Figure 6.58) looks pretty much identical to the 64-byte page cases. The invalidate case shown in Figure 6.59 has a much longer time scale, but the initial portion corresponds quite closely with the other cases. Because the trace runs much longer, the remainder of the working set plot is unique to this run.

Shorter time window analysis

Based on the working set size analysis above, we run the analysis comparing $\mathcal{G}$ to false sharing impact on non-overlapping time windows of lengths 50ms and 100ms. These durations are fixed-size intervals which divide the executions into major phases.

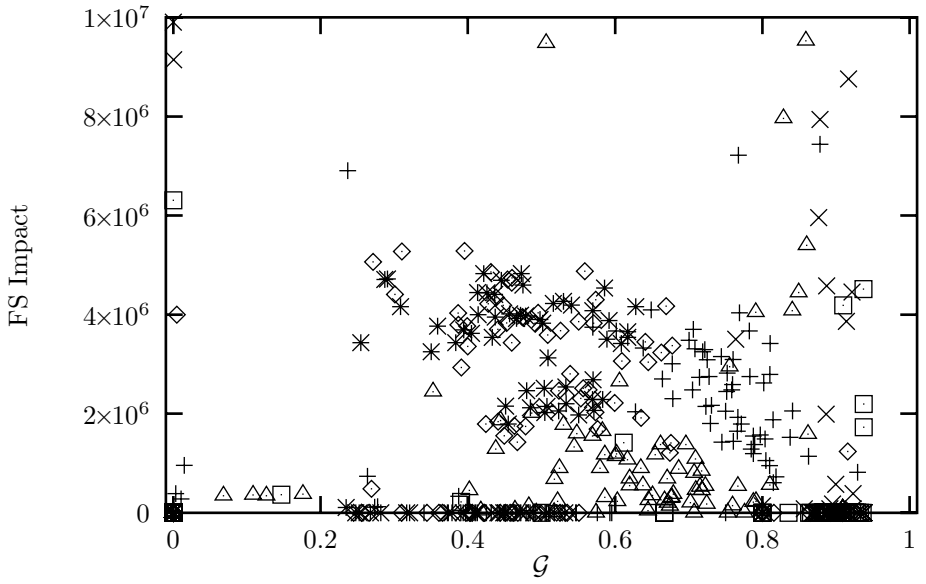

Figure 6.55: *Barnes-Hut, impact of $\mathcal{G}$, 20ms window (I8k) (clipped)*

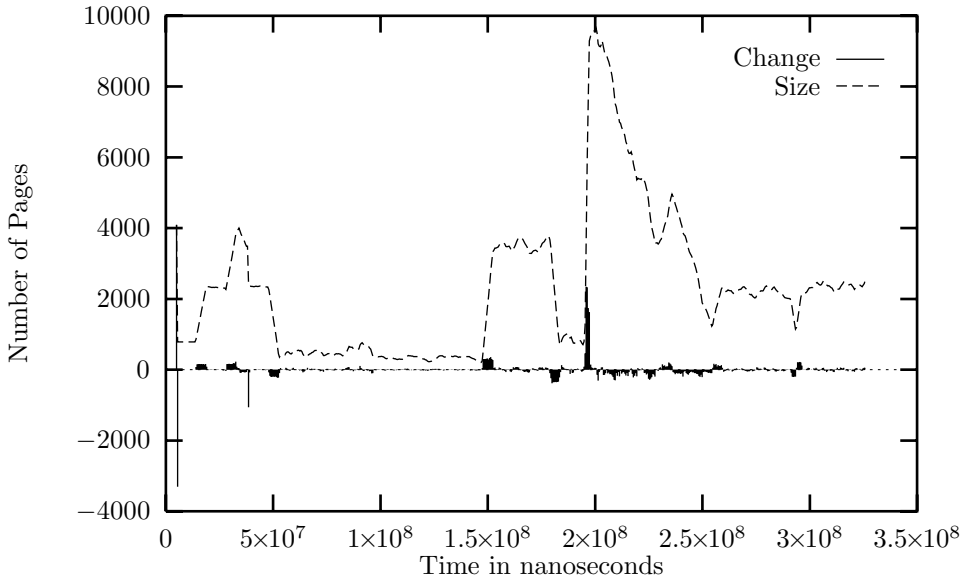

Figure 6.56: *Cholesky working set size, 5ms window (I64)*

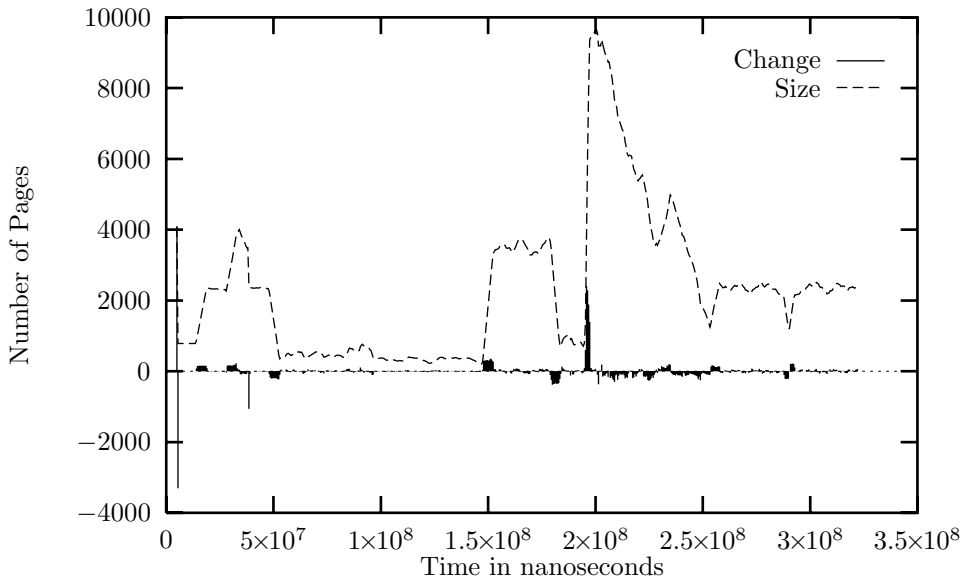

Figure 6.57: *Cholesky working set size, 5ms window (U64)*

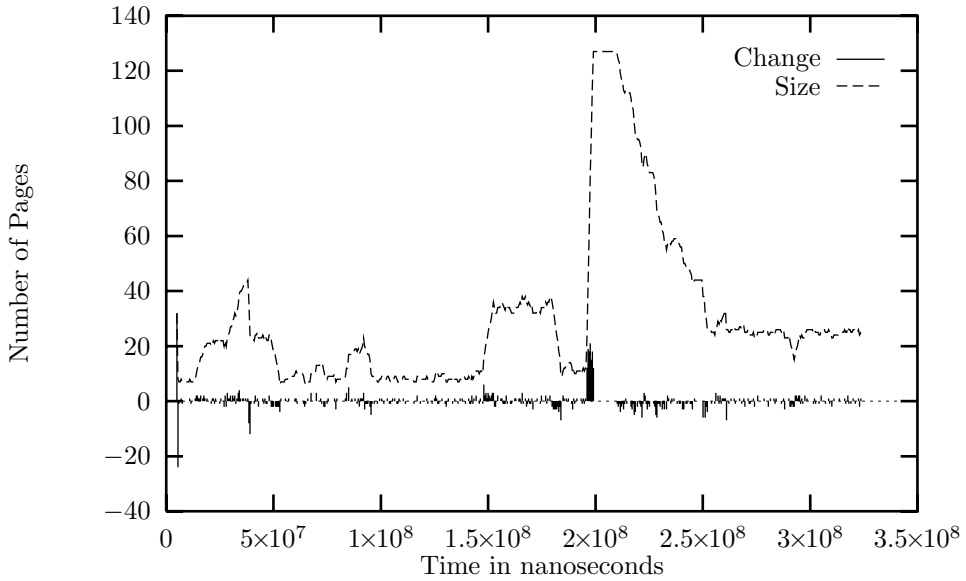

Figure 6.58: *Cholesky working set size, 5ms window (U8k)*

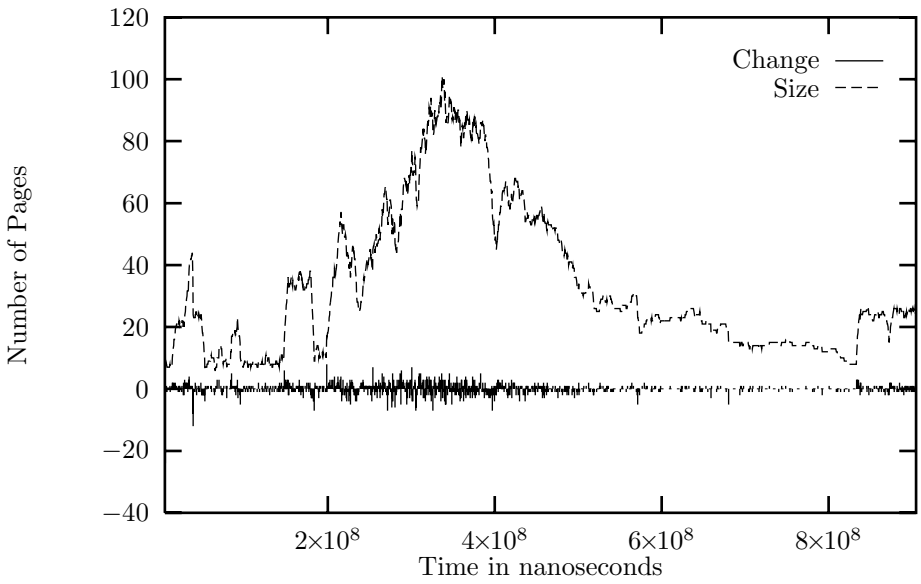

Figure 6.59: *Cholesky working set size, 5ms window (I8k)*

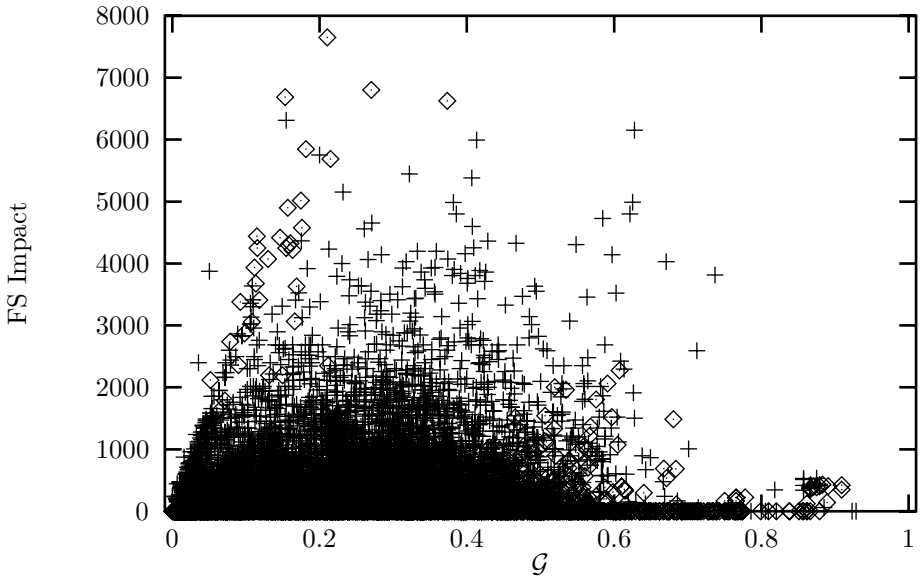

Figure 6.60: *Cholesky, impact of $\mathcal{G}$, 100ms window (U64)*

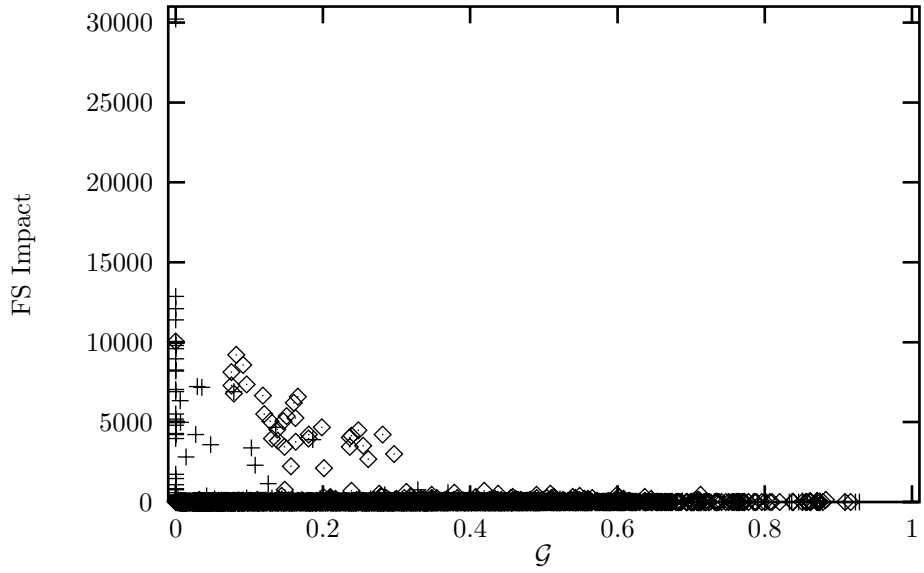

Figure 6.61: *Cholesky, impact of $\mathcal{G}$, 100ms window (I64) (clipped)*

In Figure 6.60 we present the analysis of $\mathcal{G}$ compared with false sharing impact for each page in 100ms intervals for the 64-byte page size update coherency policy. One fact not demonstrated by the plot is that for the first and fourth (last) intervals, all pages had $\mathcal{G} = 0.0$ and a corresponding false sharing impact of 0 bytes, i.e., no false sharing. The initial and final intervals have no false sharing because all the work is done by one processor — there is no data sharing whatsoever. The two middle time windows are plotted in the figure. In the area where $\mathcal{G} < 0.5$, the variation in impact is very high. For the invalidate case shown in Figure 6.61, we present a clipped view of the plot. The only point omitted from this figure is for $\mathcal{G} = 0.000007$ and with the false sharing impact of 370048 bytes. We see no useful correlation in this plot. The difference between this plot and the entire execution analysis plot (Figure 6.8) is that much of the false sharing traffic is eliminated because the pages are shared across phases, and we have limited our analysis to single phases. Analysis of the expiring update coherency architectures shows no improvement in predictive ability, so is not presented here.

The interpretation of the 8k-byte cases are virtually identical to the 64-byte cases. The update architecture is shown in Figure 6.62 and the invalidate in Figure 6.63.

Reducing the size of the window in which we compute the measure to 50ms does not seem to make any difference for the cases where we have a 64-byte page size. For the 8k-byte page size cases, however there is a slight difference compared to the 100ms windows. Figure 6.64 contains a plot of the update policy case with the 50ms window. It differs just slightly from the 100ms window by being a little larger in the impact scale, and having more pages with zero bytes of impact. For the invalidate case shown in Figure 6.65, we see that the pages with the highest impact transfer an order of magnitude less data across processors. We see this as a more accurate assessment of false sharing impact for the following reason: If the window size is too small, there would be an increase in false sharing impact because data that is truly shared in the program might not be used by all the processors during the interval, thus appearing to be falsely shared. However, in this case we still observe many pages with zero impact at high $\mathcal{G}$ values.

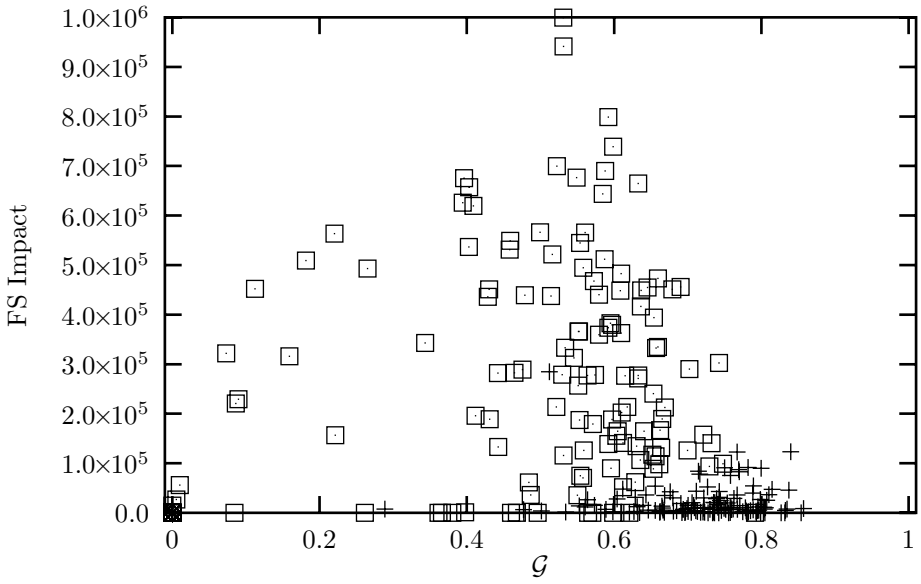

Figure 6.62: *Cholesky, impact of $\mathcal{G}$, 100ms window (U8k)*

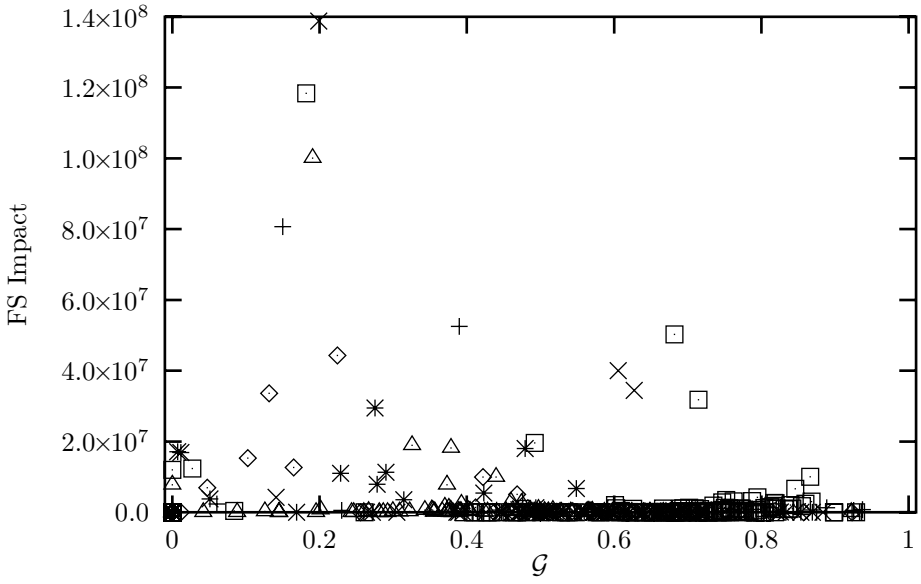

Figure 6.63: *Cholesky, impact of $\mathcal{G}$, 100ms window (I8k)*

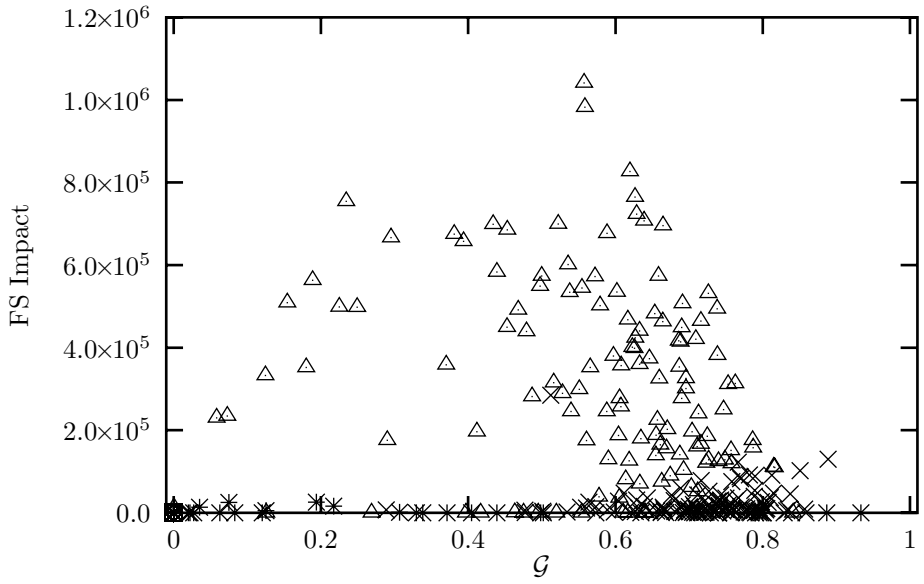

Figure 6.64: *Cholesky, impact of $\mathcal{G}$, 50ms window (U8k)*

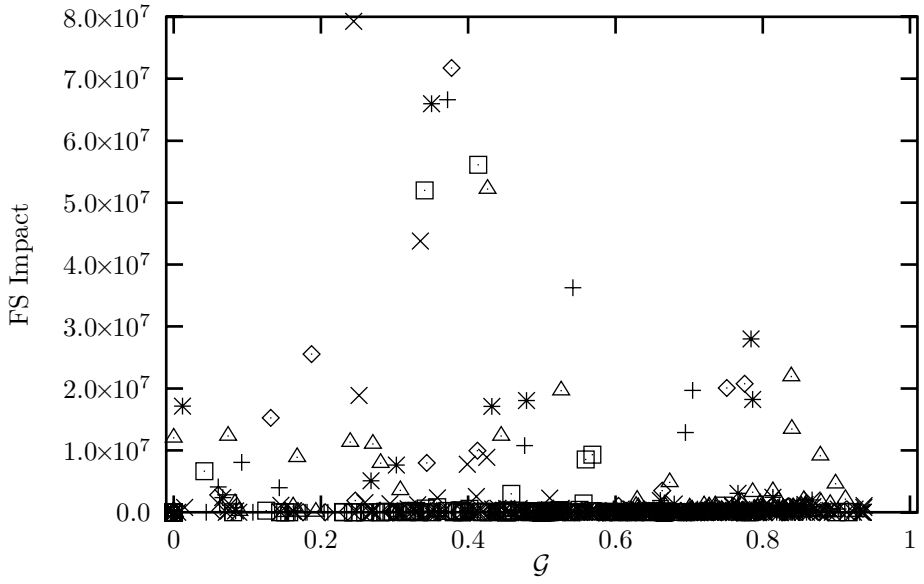

Figure 6.65: *Cholesky, impact of $\mathcal{G}$, 50ms window (I8k)*

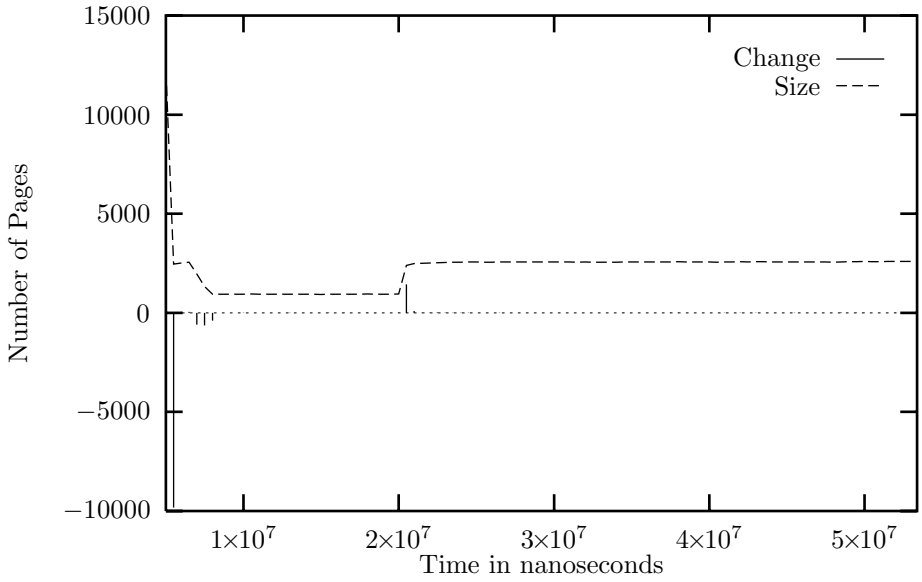

Figure 6.66: *Mp3d working set size, 5ms window (U64)*

6.4.3 MP3D

Determining phase changes

The working set size is computed for 5ms intervals and reported every 500μs as before. For the architectures with 64-byte pages, we see three distinct phases. Figure 6.66 shows the update coherency case, and in Figure 6.67 is the invalidate case. In both, the phases are marked at the points where there is a sharp change in the working set size, aside from the initial spike caused by the startup of the program.

For the 8k-byte page size architectures, the update case shown in Figure 6.68 exhibits a similar pattern to the 64-byte cases, but has a less pronounced initial phase change. Also, during the middle phase, the working set size changes slightly more than it did for the 64-byte cases. The invalidate coherency case, however is substantially different. Shown in Figure 6.69, we see a large change in the working set size approximately every 40ms. This program runs substantially longer than the others, too, and the 40ms interval is nearly two-thirds of the runtime of each of the other simulations.

Shorter time window analysis

Based on the above analysis of working set size changes, we compute the $\mathcal{G}$ metric over smaller time windows appropriate for each architecture. For the 8k-byte invalidate case, we use a 40ms window, as that is the average duration of the "bumps" in the plot of Figure 6.69. For the other cases, we select a 10ms window which divides the execution at the two large changes in the working set size.

Computing $\mathcal{G}$ over 10ms intervals during the program execution results in many pages having very low false sharing impact. In addition, many pages with high impact have relatively low $\mathcal{G}$ values. The situation is similar for both the 64-byte page update, Figure 6.70, and invalidate, Figure 6.71. Compared with the entire run analysis, many pages with higher $\mathcal{G}$ have increased impact. Again, we can attribute this to the sharing of these pages across phases. In the entire run

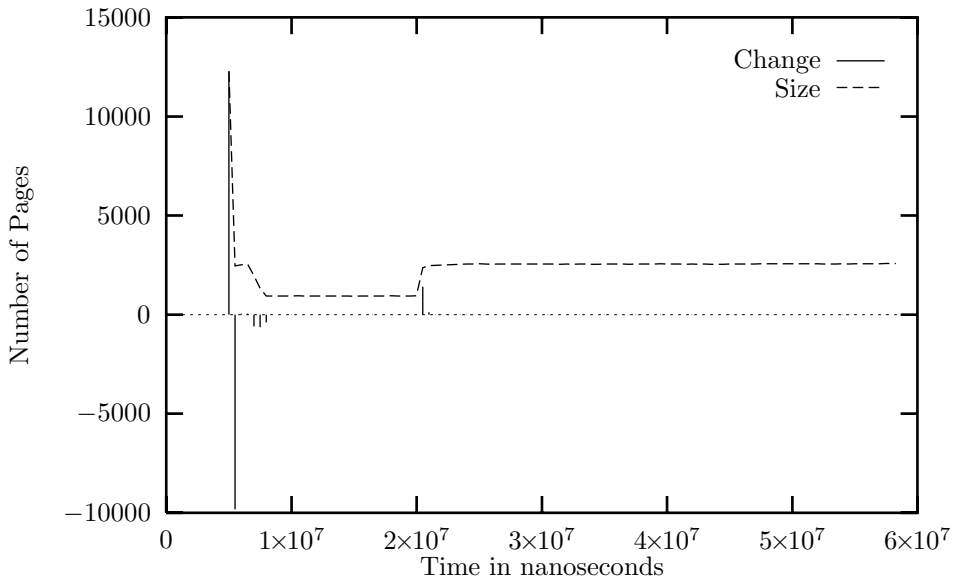

Figure 6.67: *Mp3d working set size, 5ms window (I64)*

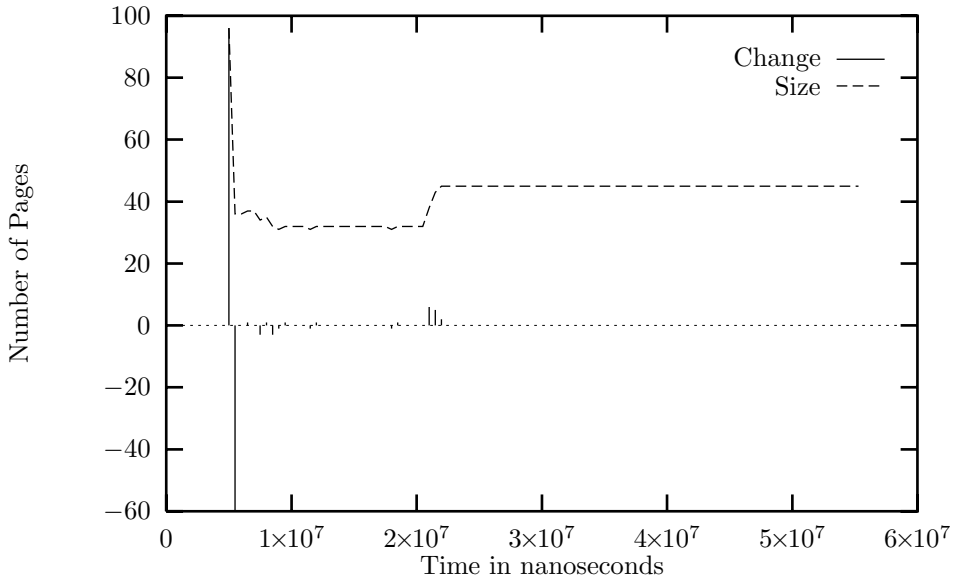

Figure 6.68: *Mp3d working set size, 5ms window (U8k)*

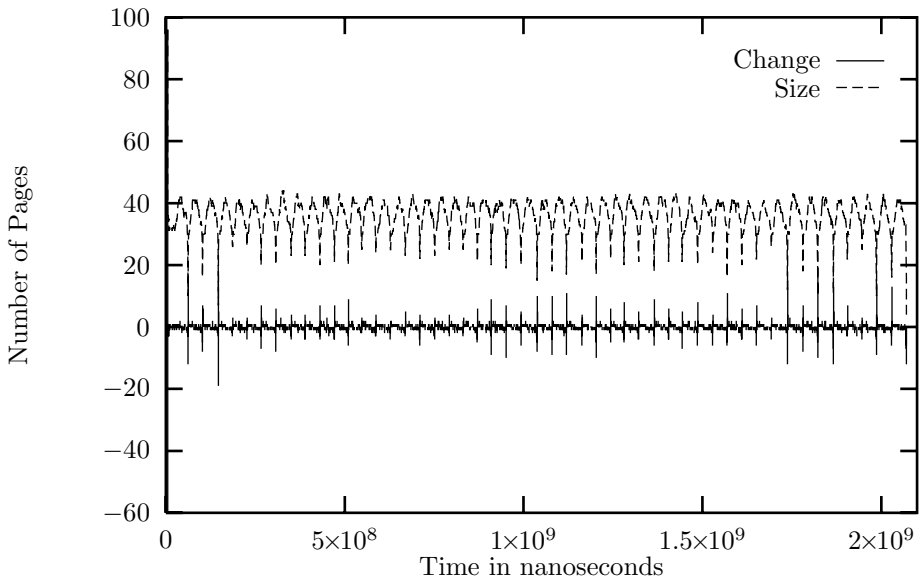

Figure 6.69: *Mp3d working set size, 5ms window (I8k)*

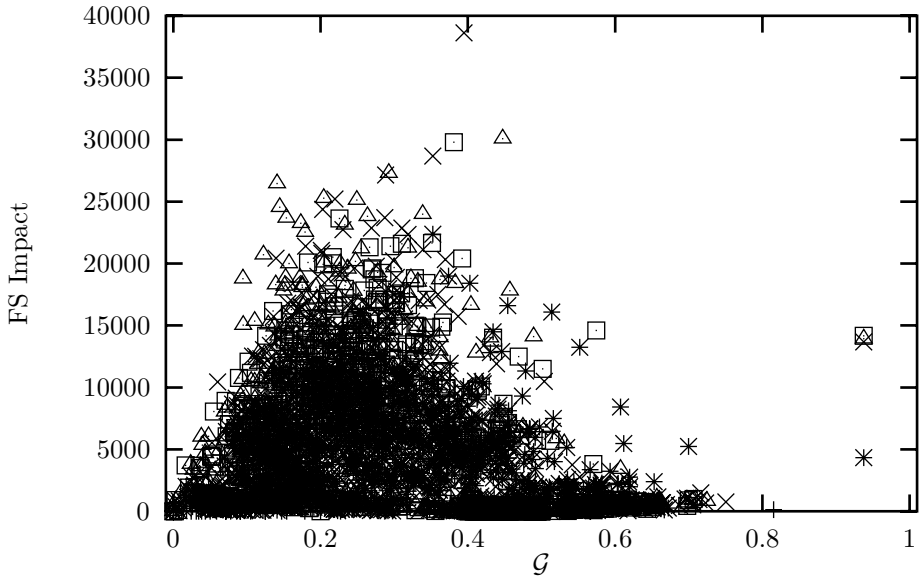

Figure 6.70: *Mp3d, impact of $\mathcal{G}$, 10ms window (U64)*

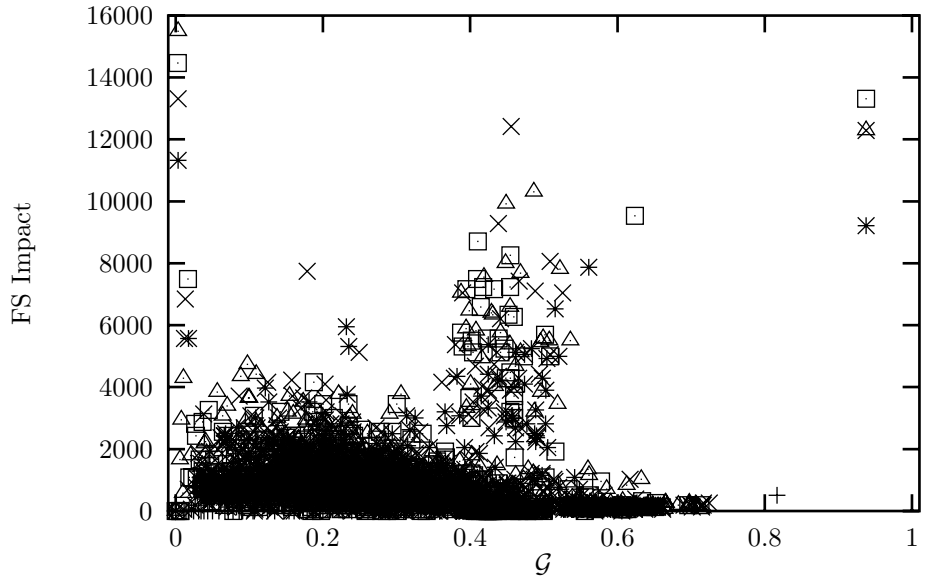

Figure 6.71: *Mp3d, impact of $\mathcal{G}$, 10ms window (I64)*

analysis, the pages are known to eventually be fully shared, so the $\mathcal{G}$ measure is low. However, since the page is not fully shared at the time of the coherency operation, we must charge it to false sharing.

The 10ms interval analysis of the 8k-byte update architecture trace shows similar results to the 64-byte cases, as shown in Figure 6.72. There are a large number of pages with middle-range $\mathcal{G}$ values with a wide range in impact. For the invalidate coherency version, we used a 40ms window for reasons stated above. The plot in Figure 6.73 shows us that for most values of $\mathcal{G}$ we cannot tell if there is very little impact or considerable impact. In particular, pages with a value of $\mathcal{G} \approx 0.9$ exhibit anywhere from no false sharing impact to the highest impact for any page in this application. Using $\mathcal{G}$ to isolate pages with high impact does not work in this case.

For completeness, the 8k-byte invalidate was also analyzed using a 10ms interval. The plot of the pages and their impact is presented in Figure 6.74. There is little improvement in the predictive capability, yet we have many more pages with higher $\mathcal{G}$ values that retain the same impact values, confirming our choice of a longer interval for this architecture.

6.4.4 Water

Determining phase changes

As before, we compute the working set size every 500μs over the last 5ms of run time. For the 64-byte page size architectures (update coherency in Figure 6.75, invalidate coherency in Figure 6.76), there is a nearly constant number of active pages during the majority of execution time. Other than the initial startup and the final data output, there are only slight variations in the working set size at regular intervals. The very slight change occurs at a repeating interval of 150ms, which we consider to be the length of each phase in this program. At the end of the program execution, there are three spikes in the working set size. The width of these spikes is 5ms, an artifact of the resolution at which we compute the working set size.

The working set size on the 8k-byte page size architectures remains nearly constant at ten pages. Every 50ms there is a change, and we use this value as the length of a phase. At the

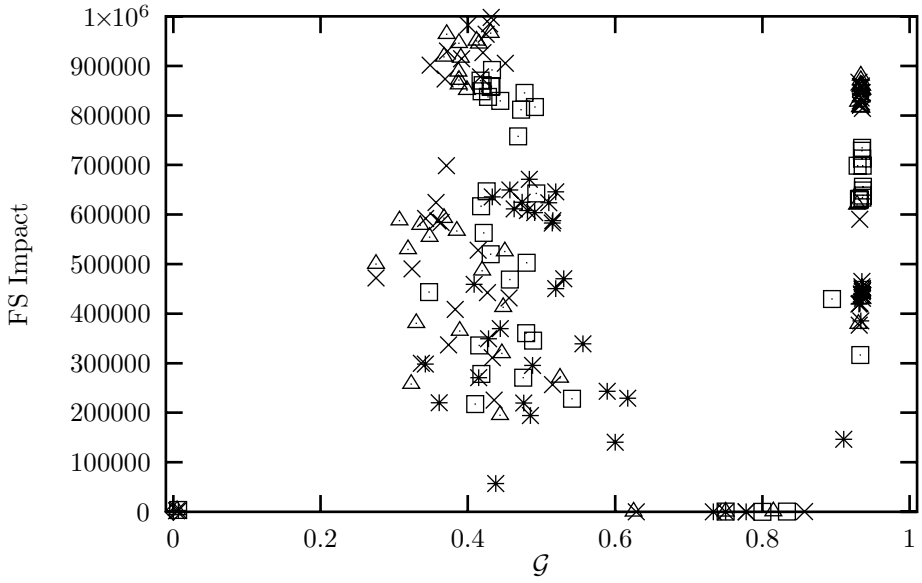

Figure 6.72: *Mp3d, impact of $\mathcal{G}$, 10ms window (U8k)*

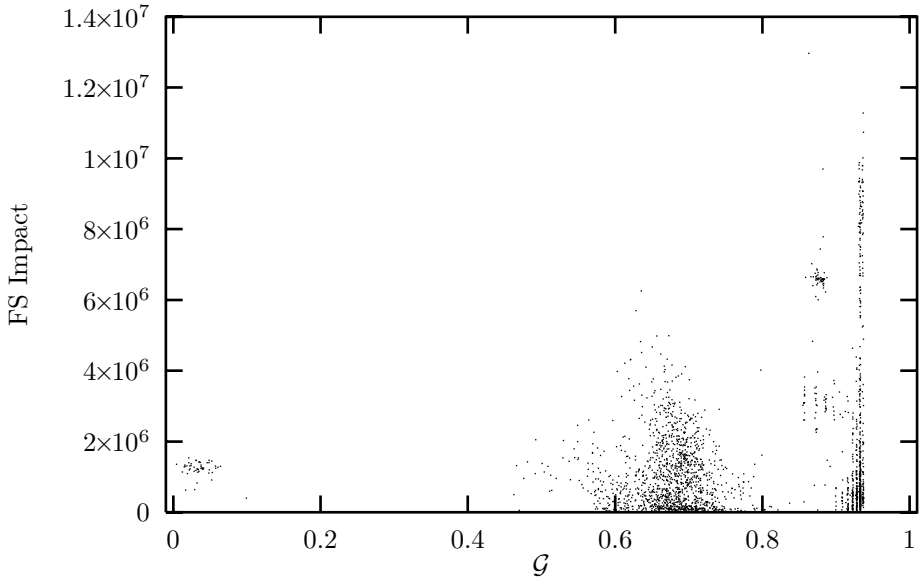

Figure 6.73: *Mp3d, impact of $\mathcal{G}$, 40ms window (I8k)*

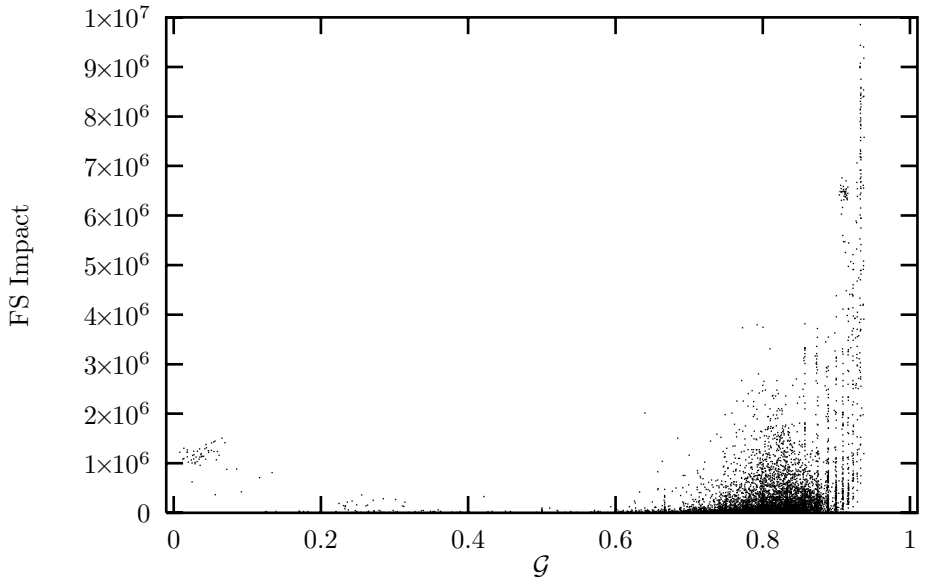

Figure 6.74: *Mp3d, impact of $\mathcal{G}$, 10ms window (I8k)*

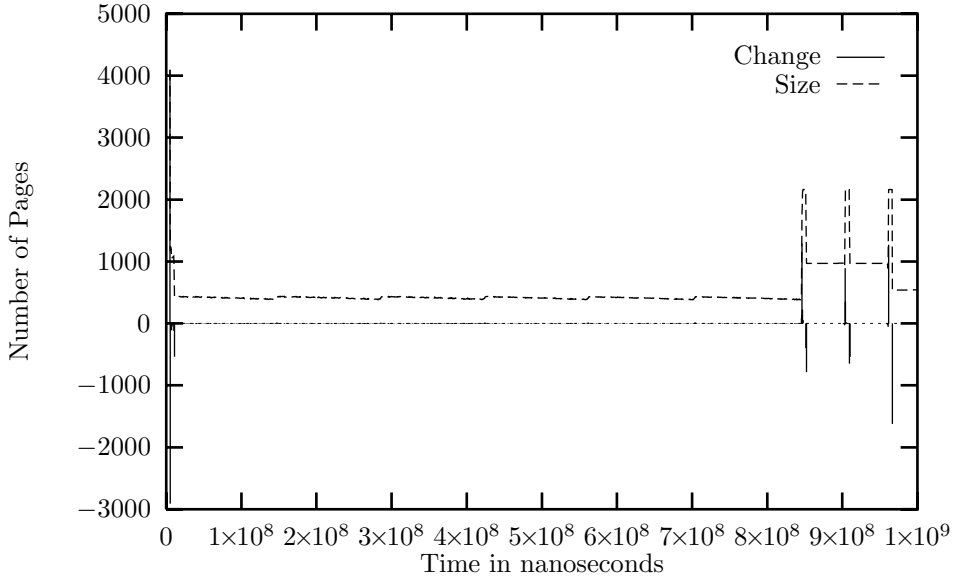

Figure 6.75: *Water working set size, 5ms window (U64)*

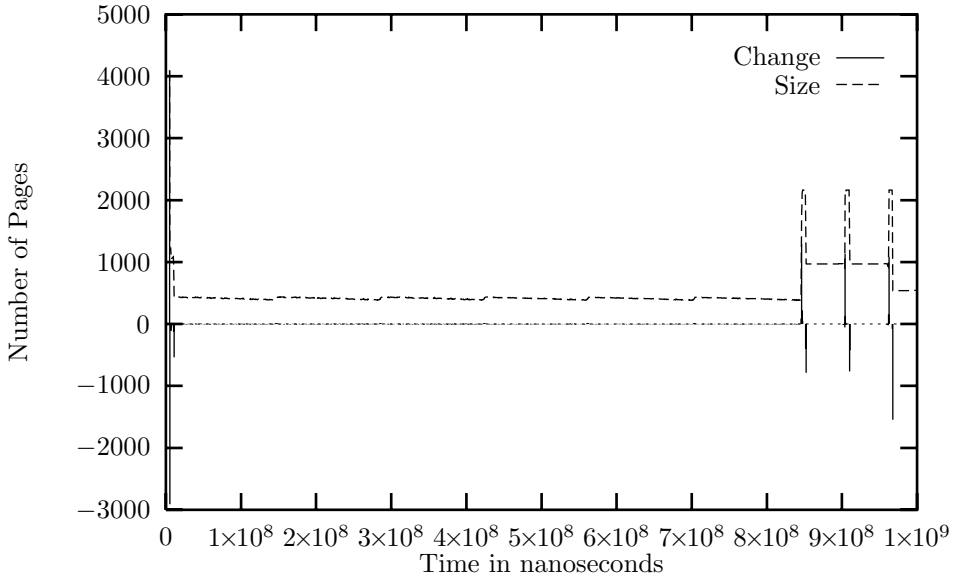

Figure 6.76: *Water working set size, 5ms window (I64)*

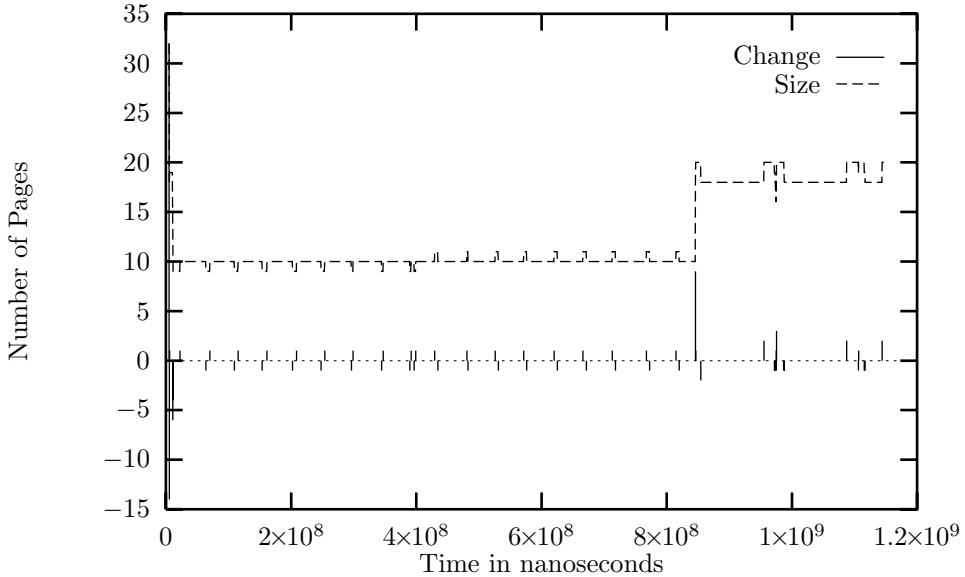

Figure 6.77: *Water working set size, 5ms window (I8k)*

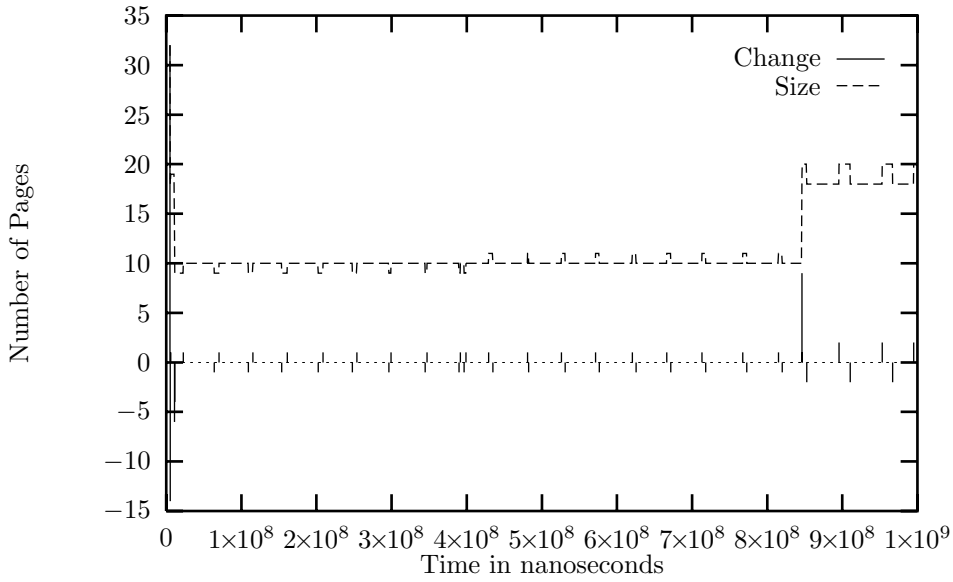

Figure 6.78: *Water working set size, 5ms window (U8k)*

end of the run, the number of active pages doubles. The difference at the end of the program between the invalidate (Figure 6.77) and update (Figure 6.78) is after the first major jump in working set size at 846140740ns in both plots. Up to this time, both the invalidate and update architectures proceed identically in terms of working set size. After this point, the invalidate case takes longer to finish, indicating contention for memory and active write sharing during the final output.

Shorter time window analysis

By dividing the execution of the program into time windows which correspond to the phases, we should be able to detect pages which exhibit false sharing, but not necessarily over the entire application. We use the lengths determined in the previous section for this analysis.

For the 64-byte page size architectures, we use a window of 150ms, which divides the run time into seven windows. During the first five windows, every page has $\mathcal{G} = 0.0$ and a false sharing impact of zero. The impact and $\mathcal{G}$ value for each page is plotted for the update case in Figure 6.79. Each window is plotted using a different symbol in the figure. The amount of false sharing during each interval is quite low. There is no correlation between $\mathcal{G}$ and impact. For the invalidate case, Figure 6.80, we get only one additional page with false sharing impact than we did when analyzing the entire run of the program. This may indicate we have chosen an interval that is too short or that there really is no false sharing. Since we know that this application has very little false sharing, we conclude that the interval is not too short.

The natural maximum window size for the 8k-byte page size architectures for this application is 50ms. For the update coherency case shown in Figure 6.81, the pages with the highest impact have $\mathcal{G}$ values in the middle of the range. Some pages with low measured impact have higher predicted impact than they did before. We are not better able to predict the impact than we were when analyzing the entire run. With invalidate coherency, we have very similar results (Figure 6.82). This is because of the private nature of the shared data references. Only at the end of the program are some pages shared among the processors that access them.

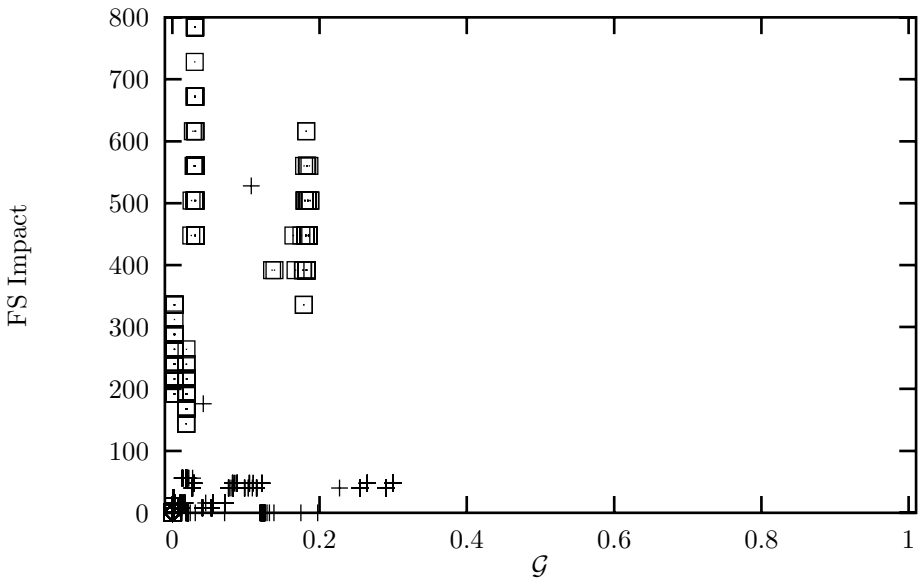

Figure 6.79: *Water, impact of $\mathcal{G}$, 150ms window (U64)*

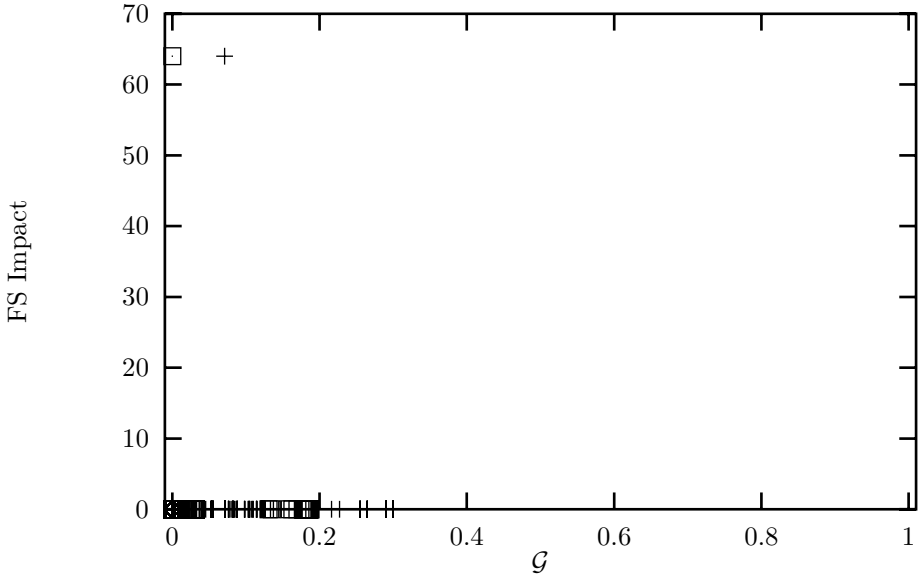

Figure 6.80: *Water, impact of $\mathcal{G}$, 150ms window (I64)*

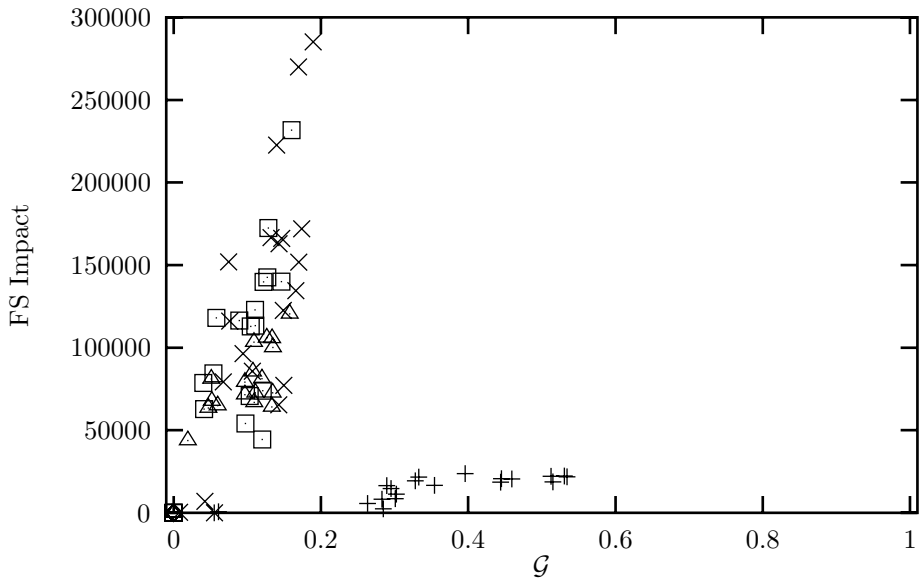

Figure 6.81: *Water, impact of $\mathcal{G}$, 50ms window (U8k)*

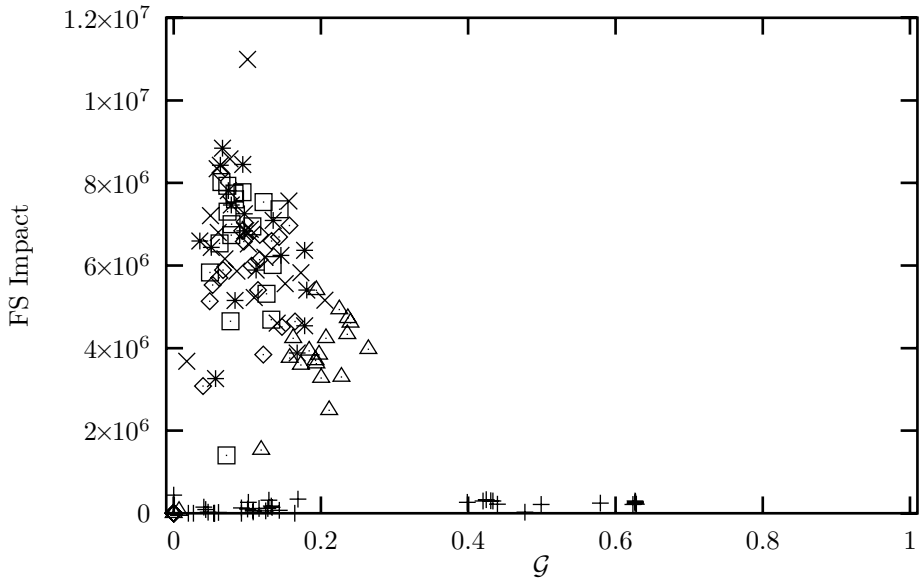

Figure 6.82: *Water, impact of $\mathcal{G}$, 50ms window (I8k)*

For all four of these cases, reducing the window size further does not substantially improve the correlation between $\mathcal{G}$ and the impact. Only in the 64-byte update coherency case do we get more pages exhibiting false sharing impact, with most of them having similar characteristics to the ones from the longer window interval which was presented.

6.4.5 Discussion

Evaluated in isolation, the plots of this section do not provide much more insight than did those of Section 6.1. However, when compared to those other plots, we see some of the effects of the shorter time window. The main point to notice is that the points in the plots are shifted to the right. This basically means that for the pages with higher impact, the $\mathcal{G}$ prediction is more accurate. The root cause of the shift is that the inter-phase sharing of the pages was masking the coherency operations charged to false sharing. With the shorter intervals, the references are limited to periods where the processors sharing the pages are *actively* sharing them. This is more in line with the assumptions in the design of $\mathcal{G}$, and results in slightly more accurate predictions.

6.5 Summary

Our original goal of using the $\mathcal{G}$ and $\mathcal{G}'$ metrics to predict false sharing impact is what we had hoped to prove useful. For certain programs with specific synchronization and reference patterns, we were successful in making the predictions; for the general case, we were not. The main result from these data is that the $\mathcal{G}$ metric alone is not generally useful for predicting the false sharing impact a page will have. In addition, even scaling the importance of a page by the number of modifications to it will not improve the predictive capability.

When a page was accessed by processors at different time localities the accuracy of $\mathcal{G}$ as a predictor was diminished, as we discovered in the previous chapter with the synthetic programs. This situation is what causes the appearance of many pages with zero false sharing impact but a high $\mathcal{G}$ measure in the SPLASH programs. The data on the page is truly shared, but at different times during the execution. This observation directed us to evaluate $\mathcal{G}$ for pages over a shorter time window.

To determine the proper size of the time window to analyze, we identified phase changes in the applications. Choosing arbitrary window lengths may have resulted in choosing windows which were too short, in turn resulting in many pages being classified as exhibiting false sharing when they should not. This would occur when the data on a page is required by several processors for a computation that took longer than the selected window. Alternatively, the time window could be chosen to be too long, possibly resulting in $\mathcal{G}$ indicating a reduced amount of false sharing. By using phase changes as boundaries for the windows, we have substantially reduced the possibility of these situations arising.

The sensitivity to window size is exemplified by the 8k-page size invalidate coherency architecture runs of Cholesky and Mp3d. In the Cholesky program, reducing the window size from 100ms to 50ms resulted in a decrease in the impact scale. The larger window size resulted in many pages being measured with higher false sharing impact than they should have been. In the Mp3d program, the smaller window size results in $\mathcal{G}$ having higher values, but the actual impact remaining basically the same as with the 40ms windows. In this instance, the smaller window size is too small for the program.

The effect of using shorter time windows for analysis is to reduce the importance of the exact ordering of references. A similar phenomenon occurs when the programs are synchronized explicitly. Both situations improve the correspondence of $\mathcal{G}$ with false sharing impact, but generally not enough to make it useful as a predictor. The conjecture we make from these data

and interpretations is that in order to accurately predict the false sharing impact of a page, we need to know the ordering of the accesses to the page from each processor.

To further examine these effects, we run an experiment which identifies the most important factors (of those defined in Section 2.2) that contribute to false sharing impact. The experiment and its results are the subject of the next chapter.

Chapter 7

Evaluation of factors

In the previous chapter, we determined that summary information about memory references appears insufficient for accurately predicting false sharing impact. The question remains unanswered as to which particular factor is the most important in predicting false sharing impact. To answer this question, we perform experiments which vary and control the various workload factors identified in Section 2.2, as well as certain architectural factors which contribute to the cost of sharing data across processors. The primary ones we are concerned with are the coherency protocol (update or invalidate) and page size (64 bytes, a typical cache line size, or 8192 bytes, a typical page size). The effect of other factors such as the time it takes to transfer data from one processor memory to another is left as future work.

7.1 Experiment design

7.1.1 Background

Not all of the factors defined in Section 2.2 are evaluated. We only consider factors that affect the amount of data transmitted across the interconnection network. The number of reads and writes to a page are not varied because these can be changed arbitrarily to require more data transfer (when more work is done). The value of the update timeout threshold is held constant, since it is not a factor for the invalidate architectures. This leaves us with the architectural and workload parameters and the values at which they will be evaluated listed in Table 7.1. The length of a run is kept constant, as is the length of the thrash cycle. The thrashing cycle length is equal to the number of processors in the experiment minus one. Note that $\mathcal{G}$ is used in this experiment merely as a quantification of the sharing participation — any proposed measure would agree that one case exhibits more sharing than the other.

Parameter	Values
number of processors	4, 16
page size (bytes)	64, 8192
coherency model	update, invalidate
interleaving order	run, thrash
sharing participation ($\mathcal{G}$)	0.5 and 0.75 (4 processors)
	0.5 and 0.9375 (16 processors)

Table 7.1: *Workload factors and the levels at which they are evaluated*

$\mathcal{G}$	Pattern
0.5	2 2
0.75	1 1 1 1

Table 7.2: *Memory reference patterns for four processors.*

word A	word B	word C	word D
P0	P2		
P1	P3		

Table 7.3: *Processor lists for reference pattern* `(2 2)`

The experiment design and evaluation is based on techniques described by Jain in [15]. We use a full-factorial design where every factor is evaluated at every level of each of the other factors. The response variable is the cost of copying data attributable to false sharing between processors. In these experiments, we only evaluate the coherency costs associated with references to one page by all processors. Data from the experiments is analyzed using statistical techniques to generate a model to describe the relationship between the factors and the response variable.

7.1.2 Details of the experiment

All of the factors have categorical values — they take on one of two possible values. The value for the sharing participation are computed from the sharing patterns used for the experiment. These patterns are selected from those used in the synth-FS program described in Section 4.1.1. By using these patterns, we can vary the sharing participation while keeping all other factors identical.

We map the distinct sharing patterns to values of $\mathcal{G}$. Using $\mathcal{G}$ allows us to capture our intuition that the reference pattern `(1 1 2)` has more sharing than the pattern `(1 1 1 1)`.

In Table 7.2 are listed the reference patterns that generate the corresponding sharing participation ($\mathcal{G}$) values for the four processor experiments. Computation of $\mathcal{G}$ here assumes each word in the page is referenced the same number of times. The numbers in the pattern indicate how many processors reference a given word in the page. Similar patterns are used for the sixteen processor experiments.

The selection of the value of the update threshold count is fixed at ten. This value was chosen because of the relatively low number of total references made during the experiments. Changing the value to five does not significantly alter the data. Turning off the update expiration does impact the data, but not the conclusions drawn from them.

The execution of the test program is as follows. First, the memory reference pattern is converted into a list of processors referencing each page. Each word is both read and then written 50 times, simulating a read/modify/write operation. If the interleaving order is *thrash*, each processor is allowed to make a single memory reference during each of the 50 cycles. If the order is *run*, then each processor performs all 50 references without any other processor making any references.

An example of the generated lists is shown in Table 7.3 for the pattern `(2 2)` for a four processor test. The first word, A, is referenced by processors 0 and 1, and the second word, B, is referenced by processors 2 and 3. The remaining words remain unreferenced. For the thrashing situation, the reference trace would look like this (limited to three references per word for brevity):

Factor				Impact
Page Size	Coherency	Order	$\mathcal{G}$	
64	invalidate	thrash	0.5	0
64	invalidate	thrash	0.75	12544
64	invalidate	run	0.5	0
64	invalidate	run	0.75	0
64	update	thrash	0.5	1584
64	update	thrash	0.75	2376
64	update	run	0.5	40
64	update	run	0.75	120
8192	invalidate	thrash	0.5	0
8192	invalidate	thrash	0.75	160563
8192	invalidate	run	0.5	0
8192	invalidate	run	0.75	0
8192	update	thrash	0.5	1584
8192	update	thrash	0.75	2376
8192	update	run	0.5	40
8192	update	run	0.75	120

Table 7.4: *List of all experiments for four processors*

$$A_0,\ A_1,\ B_2,\ B_3,\ A_0,\ A_1,\ B_2,\ B_3,\ A_0,\ A_1,\ B_2,\ B_3$$

where A_0 means word A was read and written by processor 0. The length of the thrashing cycle is 3, which is one less than the number of processors (as per the definition in Section 2.2). For the run interleaving, the trace would look like this:

$$A_0,\ A_0,\ A_0,\ A_1,\ A_1,\ A_1,\ B_2,\ B_2,\ B_2,\ B_3,\ B_3,\ B_3$$

The run length here is two, one less than the number of references per word.

This experiment is repeated with four processors and sixteen processors as separate cases.

7.2 Results

Because the response variable values differ by several orders of magnitude, we need to use a logarithmic transformation of these values for analysis. Additionally, physical considerations of the system in the way that the coherency model and page size are related to the data transfer impact lead us to use a model that is multiplicative, not additive. All of the analysis following uses log-transformed values of the impact, and the final model is generated by taking the anti-log of the additive model generated by the linear regression.

7.2.1 Four processor evaluation

The list of all experiments conducted using a four processor simulation is in Table 7.4. There are 16 experiments listed, which consist of all possible combinations of the factors. The result of running each experiment is labeled as *Impact*. It is this value that we wish to predict based on the other factors.

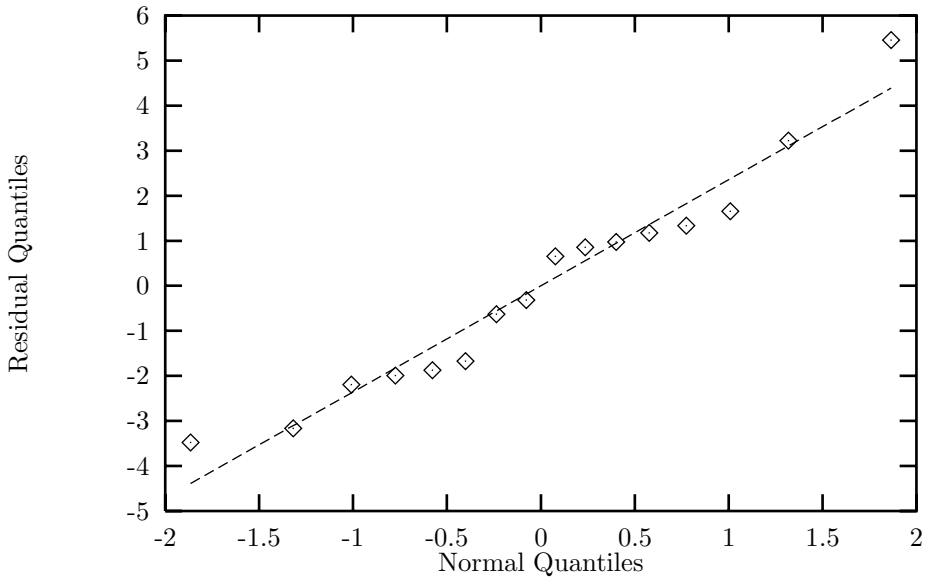

Figure 7.1: *Quantile-Quantile plot of residuals for four processor experiments*

Regression fit of all data

We use a linear regression fit of the logarithm of the impact against the factors listed in Table 7.1 and take the anti-log of the resulting equation to get our model. The computed model is

$$\begin{aligned} Impact &= 0.85274^{PageSize} \times 0.198493^{CoherencyModel} \times \\ &\quad 8.74078^{RefOrder} \times 0.217621^{\mathcal{G}} \times 73.3601 \end{aligned} \tag{7.1}$$

We interpret Equation 7.1 as follows: The most significant factor contributing to $Impact$ is the reference order (since this factor has the highest value for its effect) and the other factors are minor contributors to the impact. One test for the goodness of the fit of the model is indicated by the coefficient of determination (R^2) from the linear regression, which in this case is 0.6430. This means that the regression model only explained roughly 64% of the variation in the logarithm of the response variable. Since this is slightly low, we need to investigate further on the goodness of fit.

Visual tests for goodness of fit

To further evaluate the accuracy of the regression, we use visual tests to ensure the residuals (errors) are normally distributed and that they are homogeneously distributed with respect to the predicted values. The regression techniques assume that the residuals are normally distributed. If they are not, then the model is not valid.

To test the normality assumption of the residuals, we compare the quantiles of the residuals with quantiles of the unit normal distribution. The quantiles are computed by taking the inverse of the cumulative distribution function (CDF) of the expected distribution (in our case the unit-normal distribution). The residuals are computed by evaluating the formula in the regression equation and subtracting the result from the actual value. Precise instructions and formulas are found in [15, Section 12.10]. The resulting quantile-quantile plot will be linear if the residuals are normally distributed. To aid this analysis, we plot a line that indicates the linear fit of the

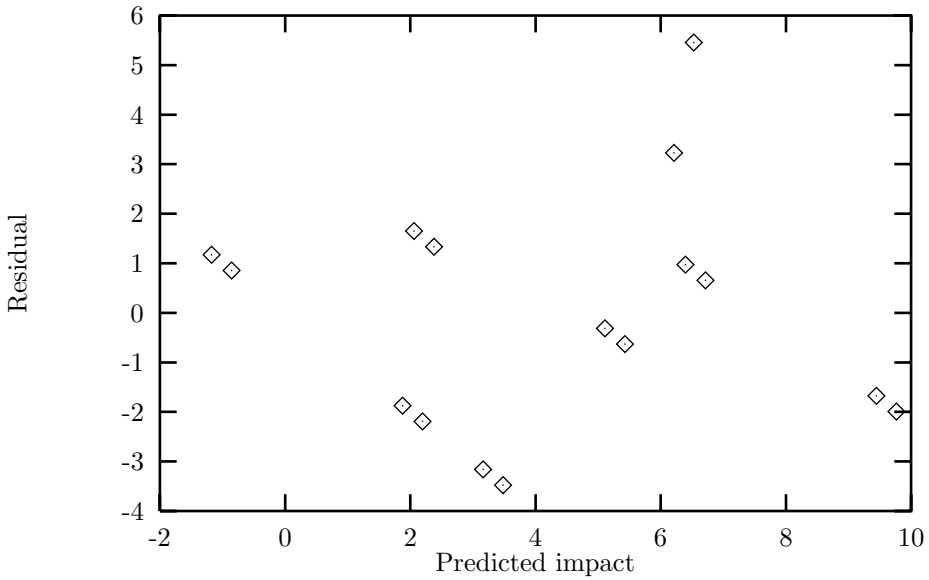

Figure 7.2: *Residual* vs. *Predicted value for four processor experiments*

points. The quantile-quantile plot for this regression, in Figure 7.1, shows only a slight deviation from the line. We consider this to indicate a normal distribution of errors, so the assumption of the model holds.

A lack of homogeneity in the residuals compared with the predicted values indicates that the residuals are still functions of the factors. The plot of the residuals against the predicted values in Figure 7.2 shows reasonably well scattered residuals, though there appears to be some slight pairing of the points. This could indicate that there are some interactions among the factors in the regression.

Testing for factor interaction

We present the full regression model for the experiment in a tabular form. This computation accounts for all interactions among the four factors. The effects for this regression are presented in Table 7.5. The factor A corresponds to the factor page size, factor B to coherency model, C to reference order, and D to $\mathcal{G}$. The factor BC corresponds to the factor describing the interaction between coherency model and reference order, and so forth. The regression equation is a product of these factors multiplied by the indicated factors. (The regression equation 7.1 could be summarized by a similar table consisting of only the first five lines. The error in that regression is explained by the additional factor interactions in this table.) It is quite clear from this analysis that reference order is the most important single factor in the model. The interaction of coherency model and reference order is significant, as is the interaction between page size and $\mathcal{G}$. The remaining factors are less important.

Since the interaction between the coherency model and the reference order is the more significant than the other interactions, we analyze the data by holding the coherency model constant. That is, we create two data sets, one with only invalidate coherency experiments and one with only update coherency experiments. This also reduces the number of factors in the regression by one.

Factor	Effect
I	73.3601
A	0.852713
B	0.198406
C	8.74113
D	0.217643
AB	0.852713
AC	0.852713
AD	1.17273
BC	1.66512
BD	0.315679
CD	0.285264
ABC	0.852713
ABD	1.17273
ACD	1.17273
BCD	0.240848
ABCD	1.17273

Table 7.5: *Regression results for four processor experiments with all interactions.*

The model describing only the invalidate-coherency experiments is

$$Impact = 0.727094^{PageSize} \times 14.556^{RefOrder} \times 0.0687004^{\mathcal{G}} \times 14.5549 \tag{7.2}$$

which indicates that reference order is by far the most important factor contributing to the impact. The regression explains 68% of the variation of the data. Interactions among the remaining factors are insignificant. The visual tests (not shown here) confirm the residuals are normally distributed, but have short tails.

The model describing only the update-coherency experiments is

$$Impact = 1.0^{PageSize} \times 5.2488^{RefOrder} \times 0.689423^{\mathcal{G}} \times 369.747 \tag{7.3}$$

again indicating that reference order is the most significant factor. This regression explains 99% of the variation of the data, which is excellent.

Based on the above tests and models, we can conclude that the most significant factor contributing to false sharing impact is indeed the global reference interleaving order.

7.2.2 Sixteen processor evaluation

The list of all experiments conducted using a sixteen processor simulation is in Table 7.6. As with the four processor case, we proceed by performing a linear regression on the log-transformed response variable *Impact*, and take the anti-log of the resulting equation to get the model.

Regression fit of all data

The model for the sixteen processor experiments is given by the equation

$$\begin{aligned} Impact &= 0.85274^{PageSize} \times 0.10208^{CoherencyModel} \times \\ &\quad 16.0868^{RefOrder} \times 0.14808^{\mathcal{G}} \times 201.595 \end{aligned} \tag{7.4}$$

Once again, we note that reference order is the most significant factor in this model. The R^2 test for this regression indicates that nearly 73% of the variation is explained by this model. This is good, but we still need to perform the visual tests.

Factor				Impact
Page Size	Coherency	Order	$\mathcal{G}$	
64	invalidate	thrash	0.5	0
64	invalidate	thrash	0.9375	50176
64	invalidate	run	0.5	0
64	invalidate	run	0.9375	0
64	update	thrash	0.5	19404
64	update	thrash	0.9375	31780
64	update	run	0.5	40
64	update	run	0.9375	600
8192	invalidate	thrash	0.5	0
8192	invalidate	thrash	0.9375	642252
8192	invalidate	run	0.5	0
8192	invalidate	run	0.9375	0
8192	update	thrash	0.5	19404
8192	update	thrash	0.9375	31780
8192	update	run	0.5	40
8192	update	run	0.9375	600

Table 7.6: *List of all experiments for sixteen processors*

Visual tests for goodness of fit

We perform the same visual tests as for the four processor experiments in Section 7.2.1. The quantile-quantile plot (Figure 7.3) shows good linearity. The residual *vs.* predicted value plot of Figure 7.4 also shows the same kind of scattering as the four processor case did.

Testing for factor interaction

As we did in the four processor case (Section 7.2.1), we compute the full model with all interactions among the four factors. The result is presented in Table 7.7. The only interactions are an order of magnitude smaller than the most significant primary factor, so they do not affect the model much.

7.3 Discussion

Performing the experiments detailed in this chapter has proved very insightful in pinpointing the major cause of false sharing impact. The statistical analysis done on these experimental data points conclusively to the reference interleaving order being the most significant contributor to the false sharing impact. By examining the raw data, it is clear that changing the reference interleaving order the impact can be changed dramatically—almost arbitrarily. Other reference order patterns can be made to generate even higher impact than the thrash patterns did. We did not use them here because they do not keep the other factors constant, which is important for our analysis.

The need for precise reference interleaving order is confirmed by the experiments of Chapter 5 and Chapter 6. The synthetic programs showed little correlation between impact and $\mathcal{G}$ in those tests where interleaving was not controlled. None of the analyses of the SPLASH programs showed promising results either.

In real programs the order of references cannot be changed as radically as we did here. However, the point of this exercise was to show that if the reference interleaving order is not

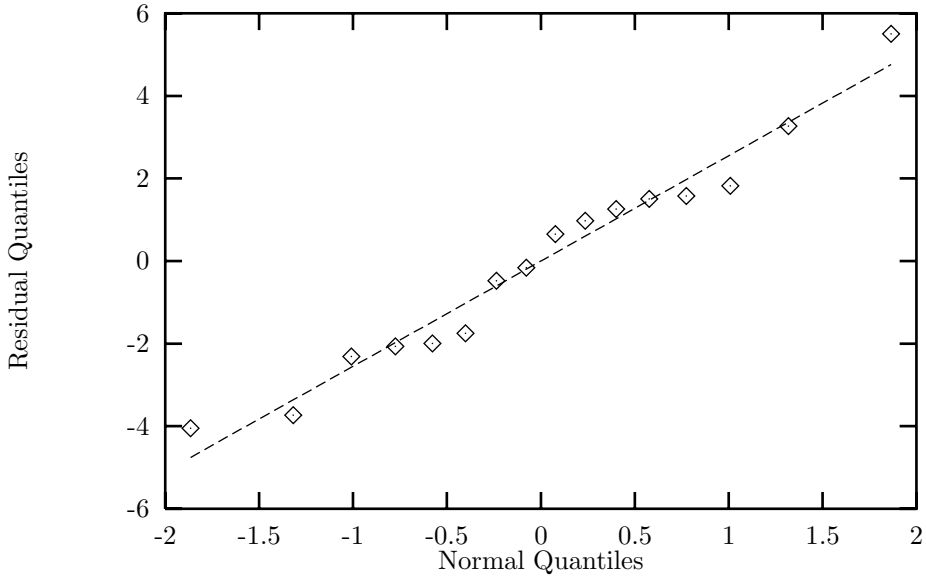

Figure 7.3: *Quantile-Quantile plot of residuals for sixteen processor experiments*

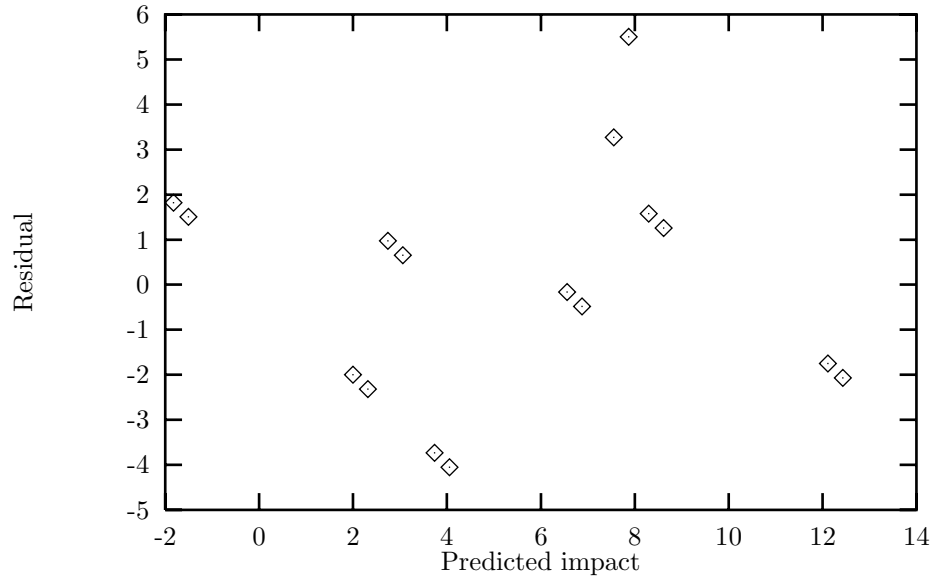

Figure 7.4: *Residual* vs. *Predicted value for sixteen processor experiments*

Factor	Effect
I	201.595
A	0.852709
B	0.102104
C	16.0904
D	0.148149
AB	0.852709
AC	0.852709
AD	1.17273
BC	1.27926
BD	0.327928
CD	0.289879
ABC	0.852709
ABD	1.17273
ACD	1.17273
BCD	0.167594
ABCD	1.17273

Table 7.7: *Regression results for sixteen processor experiments with all interactions*

known, the impact cannot be predicted: If we do not know the ordering, we do not know if it was beneficial or harmful in terms of its effect on false sharing impact.

The conclusion from these experiments is that knowledge of the precise interleaving order of references is vital to predict the false sharing impact for a page. Any metric that proposes to predict false sharing impact *must* take into account the reference interleaving order. The primary consequence of this is that accurate computation of false sharing impact requires an enormous amount of data to be collected and stored. Furthermore, the data that needs to be collected cannot easily be captured without perturbing the execution of the program.[1] If the program's execution is altered, the conclusions drawn from the data are no longer applicable. Even if we were able to capture all the necessary data, storing it would be a problem. A long-running parallel program can easily generate hundreds of millions of memory references, each of which needs to be recorded.

[1] We cleverly avoided this problem by using a simulation.

Chapter 8

Conclusions and future research

Our original motivation for this work was to detect the causes of excess data communication between processors in a shared memory machine. We demonstrated the existence of excessive data transfer by performing an experiment which compared the data traffic of programs running on an ideal memory architecture to more realistic architectures with large page sizes. We found that a significant portion of the data traffic between processors was unnecessary for the computation and could be attributed to poor data packaging, i.e., *false sharing*.

We discussed the difficulties of formally defining false sharing in a manner that has the following desired characteristics: (a) captures intuition, (b) is architecturally independent, (c) predicts performance impacts for various architectures, and (d) has a practical application in solving the false sharing problem. Our formulation has the first two of these desired traits by design. It is based on our intuition of the varying degrees of false sharing; we are not limited to the *de facto* false sharing discussed in previous work. The latter two characteristics are discussed in terms of our formulation in the next section.

8.1 Review

The different experiments with the synthetic workload programs demonstrate several points: (1) The details of the coherency protocol implementation interact with interleaving patterns so that the architecturally dependent false sharing cost measures yield values that may be too specific to a particular execution of an asynchronous program. (2) Because of the nature of the update protocol scheme being used in this study which masked some interleaving effects, the good correlations between false sharing byte traffic and $\mathcal{G}$ found in these cases are fairly convincing arguments that $\mathcal{G}$ captures something of the intuition of false sharing. (3) The loss of ordering information in the $\mathcal{G}$ false sharing measure, especially when calculated over the entire execution, can be significant. However, the pessimistic bias (pinpointing false sharing problems that don't necessarily translate into performance problems because of favorable reference orderings) make the measures useful as devices to explain observable poor performance.

Further investigations using real applications confirm the conclusions above. These additional experiments also reveal that the measures are most accurate at predicting false sharing impact when the pages are actively used by multiple processors during the entire interval under evaluation. By limiting the evaluation window to single *phases* of the applications, we are able to slightly increase the accuracy of the predictions. Limiting the window size also reduces the importance of the memory reference interleaving order, particularly when the program has short, regular phases.

The experiments conducted in Chapter 7 serve both to confirm and explain the previous experiments. The evidence gathered from all the experiments leads us to conclude that memory

reference interleaving order is a significant factor in the performance impact of false sharing. Looking back at the shorter interval analyses, we see a shift of the pages with higher impact to the higher predicted values. We can interpret this shift as being due to the elimination of the appearance of sharing when in fact the pages were used at distinct times in the execution. Shorter window durations afford us only limited relief from the impact of reference order.

We conclude that any proposed predictor that is based on summary information cannot accurately predict the impact. It can be at best provide an upper bound on the excess data traffic. On the other hand, measures that take into account precise interleavings are not likely to be practical in performance debugging tools. There is a delicate balance here that needs to be investigated.

8.2 Speculations on future work

There are several directions in which to proceed based on the results of this work. Foremost among these is to develop a measure that *will* accurately predict the false sharing impact of a page. This measure will need to account for the fine-grained information shown necessary for such a prediction. The need to track the detailed information of the memory references makes it unlikely that such a measure can be implemented as an on-line algorithm — the data storage requirements are just too great.

Even with an accurate predictive measure, the ability to detect false sharing and isolate the pages most affected by it does not solve the associated performance problems. In order to reduce the impact of false sharing in an application, we must first correlate the areas of memory causing the problems to the data structures in the program, then determine ways of eliminating or reducing the impact. A performance debugging tool needs to be able to provide the detection *and* a mechanism to identify the relevant data structures. Further, such a tool needs to be able to send hints to the compiler. As in any program debugging, performance debugging involves alternating cycles of execution and recompilation. Information learned from one execution should be able to guide the compiler to improve the program layout and performance on the next compilation in an iterative fashion. This type of technique has been used for uniprocessor performance enhancement. In the uniprocessor case, however, all of the information is automatically gathered and interpreted.

Techniques for reducing the effects of false sharing have been discussed in the literature. In general, knowledge of the suspected data structures is needed. Torrellas *et al.*, in [26], discuss methods of reducing false sharing related cache misses. Their primary methods are to rearrange data such that variables that exhibit false sharing are placed on different cache blocks and to put variables protected by a lock on the same block as the lock. Eggers and Jeremiassen [10] also propose techniques to eliminate false sharing in caches. Their transformations involve allocating data objects together that have similar sharing properties and the use of indirection. These methods address solutions when classic false sharing is involved. It is not entirely clear how to handle situations where data structures located on the same page are not shared by the same subsets of processors, such as those discussed in Section 2.1. Developing such techniques is a promising direction for future work.

Automating the program transformations is not yet possible — human decision making is still required. Using the techniques presented in this thesis and in some of the other literature, the best we can do currently is identify which data structures are causing problems; we cannot automatically determine which data structures have similar sharing patterns and should be co-located. Further, poor data structures cannot be redesigned automatically. In some situations, it would be more beneficial to redesign the algorithm than to rearrange the layout of the data. As an example of improper data structure choice, consider an array allocated in column-major order but accessed in a row-major order by different processors. The individual elements of the array would exhibit false sharing and would be candidates for reorganization.

To summarize, our future goals are to (1) refine our techniques of detecting false sharing at the page level, (2) develop tools to identify the data structures located on such pages, (3) generate a set of guidelines and heuristics to aid the programmer in refining the application to reduce false sharing impact, and (4) investigate compiler feedback technology to assist in the performance debugging process.

Bibliography

[1] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. Technical Report COMP TR89-98, Department of Computer Science, Rice University, P.O. Box 1892, Houston, Texas 77251-1892, November 1989.

[2] W. Bolosky, M. Scott, and R. Fitzgerald. Simple But Effective Techniques for NUMA Memory Management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19–31, December 1989.

[3] William J. Bolosky and Michael L. Scott. False Sharing and its Effect on Shared Memory Performance. In *Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems*, September 1993.

[4] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, April 1991.

[5] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfeld, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., New York, NY, 1987.

[6] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, May 1991.

[7] Yung-Syau Chen and Michel Dubois. Cache Protocols with Partial Block Invalidations. In *Proceedings of the International Parallel Processing Symposium*, pages 16–23, April 1993.

[8] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing, Volume II*, pages 99–107, 1991.

[9] Czarek Dubnicki and Thomas J. LeBlanc. Adjustable Block Size Coherent Caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[10] Susan J. Eggers and Tor E. Jeremiassen. Eliminating False Sharing. Technical Report 90-12-01, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, 1990.

[11] Susan J. Eggers and Randy H. Katz. A Characterization of Sharing in Parallel Programs and its Applicability to Coherency Protocol Evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 373–383, May 1988.

[12] Anoop Gupta and Wolf-Dietrich Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[13] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[14] Mark A. Holliday and Carla S. Ellis. Accuracy of Memory Reference Traces of Parallel Computations in Trace-Driven Simulation. Technical Report number unknown, Department of Computer Science, Duke University, Durham, NC 27706, May 1990.

[15] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, Inc., 1991.

[16] Richard P. LaRowe and Carla S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[17] Richard P. LaRowe Jr. Personal Communication.

[18] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[19] Stuart E. Madnick and John J. Donovan. *Operating Systems*. McGraw Hill, 1974.

[20] David Mosberger. Memory Consistency Models. *ACM SIGOPS Operating Systems Review*, 27(1):18–26, January 1993. See also [21] and [22].

[21] David Mosberger. Memory Consistency Models. Technical Report 93/11, Department of Computer Science, The University of Arizona, Tucson, AZ, 85721, 1993. Updated version of [20].

[22] Gil Neiger. Letter to the Editor. *ACM SIGOPS Operating Systems Review*, 27(3):1–3, July 1993.

[23] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. Technical report, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, August 1991.

[24] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[25] Josep Torrellas, Monica S. Lam, and John L. Hennessy. Measurement, Analysis, and Improvement of the Cache Behavior of Shared Data in Cache Coherent Multiprocessors. Technical Report CSL-TR-90-412, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, February 1990.

[26] Josep Torrellas, Monica S. Lam, and John L. Hennessy. Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates. In *Proceedings of the 1990 International Conference on Parallel Processing, Volume II*, pages 266–270, August 1990.

[27] Andrew W. Wilson Jr and Richard P. LaRowe Jr. Hiding Shared Memory Reference Latency on the Galactica Net Distributed Shared Memory Architecture. *Journal of Parallel and Distributed Computing*, 15(4):351–367, August 1992.

Biography

I was born in a small town in India in 1966. At the ripe old age of three, I moved with my family to New Jersey. After a year there, and another four in Indiana, we finally settled down in Rockville, Maryland, in 1975. Over the next nine years, I survived grade school, middle school, and high school. Then came college. I survived that, too, amazingly enough. After my four years at the University of Maryland at College Park (earning a BS degree in Computer Science), I moved to Durham to attend the Graduate School of Duke University. That was in 1988. In 1992, Duke awarded me the MS degree in Computer Science. It is now 1994, and I'm finally escaping the grip of the academic lifestyle.

While a graduate student at Duke, I published a few papers on topics totally unrelated to my dissertation. One was on using the Mathematica program to call external numerical libraries, and another was on the Duke Internet Programming Contest. The contest started out as a small fun experiment, and has grown to be nearly overwhelming in size. Hopefully, someone will continue running it when I'm gone.