Architectural Sensitive Application Characterization: The Approach of High-Performance Index-Set (HP-Set)

Zheng Zhang
Computer Systems and Technology Laboratory
HP Laboratories Palo Alto
HPL-2001-75
March 30th , 2001*

E-mail: zzhang@hpl.hp.com

shared memory multiprocessor architecture, performance evaluation

Good simulation tools that provide architectural relevant insights play vital roles in building complex system such as shared-memory multiprocessors. In this report, we discuss HP-Set, a simulation tool that takes the core scheduling component of CIAT and integrates it with a set of statistic gathering probes that generate the corresponding index. HP-Set stands for **H***igh* **P***erformance index*-**Set**. In a nutshell, HP-Set is a portfolio with its major indexes being the following: general statistics, coherent misses, data reuse and locality, granularity and the IO index.

The objective of HP-Set is to be *architectural sensitive* and yet not to evolve into the role of a full functional simulator. We achieve the goal by getting rid of fancy statistics and by actually implementing relevant protocols that aim at optimizing certain aspects of the index. By comparing the index with and without the perturbation of the protocols, we will know not only how big the impact the index has on the overall performance, but also how likely we can improve them architecturally.

Using HP-Set, we analyzed several commercial applications and obtained insights not available before. For example, our overall analysis points out that it's a common misconception that TPCC is more memory intensive than TPCD, the difference is rather due to their pressures on the memory system. Our communication index indicates that the third-party dirty hits dominate, and thus faster directory lookup and cache-to-cache transfer optimizations should be encouraged. On the other hand, significant number of false-sharing misses is and will continue to be a dominant performance factor. Our granularity analysis suggests that spatial localities of coherent objects are rather limited, and blind sequential prefetching might do more harm than benefit. Our IO-Memory analysis finds that IO contributes a non-negligible factor in total system traffic and is the major cause cache misses.

1. Introduction

Good simulation tools that provide architectural relevant insights play vital roles in building complex system such as shared-memory multiprocessors. CIAT [1] is one tool of popular use in HP-Lab. Unfortunately it does not deliver the application characteristics in such a way that immediately suggests relevant architectural improvement opportunities. In this report, we discuss HP-Set, a simulation tool that takes the core scheduling component of CIAT and integrates it with a set of statistic gathering probes that generate the corresponding index. HP-Set stands for ***H**igh **P**erformance index-**Set**.* The index set is composed of the set of statistics shown in Table 1. In a nutshell, HP-Set is a portfolio with its major indexes being the followings: general statistics, coherent misses, data reuse and locality, granularity and the IO index.

The objective of HP-Set is to be *architectural sensitive* and yet not to evolve into the role of a full functional simulator. We achieve the goal by getting rid of fancy statistics, and by actually implementing relevant protocols that aims at optimizing certain aspects of the index. By comparing the index with and without the perturbation of the protocols, we will know not only how big the impact the index has on the overall performance, but also how likely we can improve them architecturally. The following table gives the outline.

Table 1 : Implication of HPSet

Category	*Question*	*Implication*
Overall Analysis	What is the general characteristic of the applications, and what part of the system performance will it stress the most?	Identify system performance bottleneck and derive high-level engineering decision
Communication Index	What is the communication pattern between processes? How is the shared memory being used?	Understand the trade-off of several mature coherence protocols
Data reuse and locality	How effective are the cache hierarchies being used	Design the caching hierarchy, bandwidth requirement etc.
Granularity	What is the granularity of data objects?	Line size and sequential prefetch
IO index	How much the IO contributes to both traffic and cache misses	IO subsystem design issues

The analysis approach is hierarchical. The tool is flexible enough to allow the user select at which level the statistics should be collected. The top level is the coarsest, but it enables the designer to quickly pinpoint the specific problem. The middle level breaks the statistics into data types (when available), this gives feedback on what data causes the most problem. The bottom level breaks the statistics further into pages, and this helps to decide which particular piece of data may be of the greatest interest. In addition to specifying the level option at run time, one can always run HP-Set at the deepest level and let off-line processing aggregate the statistics towards higher levels. Furthermore, data can be either per-process or aggregated together. Unless otherwise stated, the results presented in the report are in the middle level and they are aggregated among all the processes.

We assume a general cc-NUMA type of architecture [2]. In our simulation, we assume every process run on one cache. Using the HP-Set, we analyzed several commercial applications and obtained insights not available before. The rest of the report is organized more or less in the order of the index presented in the above table.

2. Using HP-Set to Obtain an Overall Picture

In this section, we introduce the methodology of using HP-Set to obtain the overall performance pictures of the applications under study, namely TPCC16, TPCD-Q1 and TPCD-Q3. This part of HP-Set is itself composed of two separate components: 1) the memory footprint and access distribution and 2) the cache miss breakdown. We will see that although the footprint and access offers limited information, it can nonetheless answer a few key questions in a straightforward fashion. However, by far the most important analysis is the cache miss breakdown, through which we can immediately pinpoint the part of system that the application will stress the most and is thus likely to be the bottleneck when running this particular application. The ability of answering these questions quickly will help us to concentrate on the most critical part of the system by going into the relevant index data of HP-Set.

2.1 Static Memory Usage and Access Breakdown Analysis

It is important to breakdown analysis according to different data types, this is the only gateway that can lead us into any possible insight of the application – it is too often the case that analysis at the source code level is impossible. Listed in Table 2 is the data types that we are able to identify for the Oracle V7 database system. There are altogether seven types of data, five out of which is in the shared region. Note that even in situations where we can not divide data regions as detailed as we do here, it is *always* possible and useful to divide them into two general classes: the private versus shared data region.

Table 2: Data Types Breakdown

Name	Region	Meaning
Lit	Private	Literal Variables
Prv	Private	Private stacks etc.
IOB	Shared	IO staging area
Blk	Shared	Main database buffer pool
LkB	Shared	Special for Oracle: pages mixed with locks and database data

IBk	Shared	Special for Oracle: pages mixed with IO block and database data
Oth	Shared	Other unidentified data

The memory footprint and access distributions of the three applications are listed in Table 3, in both absolute and relative numbers. Memory footprint is in the unit of meg-bytes, while memory access is the fraction of instruction.

The procedure of analysis is outlined as follows:

1. ***The absolute footprint***. Obviously, the larger the footprint is, the more caching space *may* be required. However, strictly speaking, we still learn very little, unless we know exactly how data is being reused in the cache hierarchy. Moreover, while private data will be divided uniformly for each processor, the same may not hold true for shared memory region: there is always the possibility that all these memory lines will be accessed by every process. The second problem can be compensated by running HP-Set with the option of dumping statistics for each process, instead of the one aggregating over all processes

   Still, we can infer what cache conflict misses will look like by taking into account *both* the footprint and access weight. Assuming a class of data type *I* has access weight *R(I)* with total footprint *S(I)*, then under uniform access pattern, the reuse rate to any line of this data type is simply *R(I)/S(I)*. It follows that the higher this ratio is, the more likely that it will be found in caches.

2. *The access weight to shared memory portion*. The larger this number is, the more interesting optimal data placement problem *might* be, same is true for coherent misses resulted from inter-process communication.

Following the procedure outlined above, we have these observations:

1. **TPCC16**: this application has large footprint, and over 90% are in shared memory, access to which amounts to 46%. It is therefore more challenging from the cache hierarchy's point of view, in terms of both data placement that deals with remote conflict miss, and the potential large number of coherent misses. We can also infer that the data type *Blk* might not use caches as efficiently as others, for example the type *LkB*.
2. **TPCD-Q1**: this application has very small footprint, and less than half is shared memory, therefore conflict misses and data placement are unlikely to be a problem for shared data. Moderately sized caches should work well. In addition, most accesses (87%) are to private memory. Therefore, unless majority of shared memory accesses causes coherent misses, coherent misses should not be an issue either.
3. **TPCD-Q3**: relative to TPCD-Q1, TPCD-Q3 has a larger footprint. On the other hand, access weight to shared memory region is even smaller. Accesses to private memory account for about 85%. The conclusion is thus similar as TPCD-Q1.

Table 3: Memory Footprint and Access Breakdown

		Oth	Lit	Prv	IOB	Blk	LkB	IBk	ALL
Mem(M)	TPCC16	0.22	0.39	11.56	64.19	38.93	14.55	6.91	136.75
		0.2%	0.3%	8.5%	46.9%	28.5%	10.6%	5.1%	100.0%
	TPCD1	0.00	0.15	2.00	0.17	1.22	0.06	0.00	3.61
		0.0%	4.1%	55.4%	4.8%	33.9%	1.7%	0.0%	100.0%
	TPCD3	1.85	0.11	14.99	0.00	0.00	0.00	0.00	16.95
		10.9%	0.7%	88.4%	0.0%	0.0%	0.0%	0.0%	100.0%
Acc/Instr	TPCC16	0.02	0.00	0.19	0.02	0.05	0.06	0.00	0.34
		6.4%	1.0%	55.2%	5.1%	13.9%	17.7%	0.7%	100.0%
	TPCD1	0.00	0.00	0.30	0.00	0.04	0.00	0.00	0.34
		0.0%	0.1%	87.2%	0.0%	12.3%	0.3%	0.0%	100.0%
	TPCD3	0.00	0.00	0.36	0.00	0.06	0.01	0.00	0.43
		0.0%	0.2%	84.5%	0.0%	13.9%	1.4%	0.0%	100.0%

Comparing the access weights of all the three applications, we know that they are all around 0.3~0.4 range, that is, one memory access every three instructions. Therefore, it is clear that *all* these applications are memory intensive, and if there is anything that leads to higher performance of TPCD than TPCC, it is only because TPCD uses cache hierarchies more effectively.

2.2 Cache Miss Breakdown

We must first introduce the system configuration assumed by HP-Set in this study and address possible devaitions when going to other configurations. The default system has a multi-level cache hierarchy for *each* process, with inclusion property enforced. A process runs on one processor/cache set in its life time. The coherence of caches are taken care of by a invalidation based, full directory protocol. Conflict misses are measured by using a direct-mapped 1M cache with 32 bytes line size. Data placement and memory allocation follows the first-touch algorithm, i.e., the page is allocated in the node who touches it first. We will examine the effectiveness of this algorithm briefly later. Coherent IO is fully supported, when new lines are read into memory, all cached copies are invalidated; likewise, when lines are read out to the disk, recall transactions are generated for dirty lines, which will subsequently become shared instead. Event order is enforced by the dispatching engine of CIAT/CDAT, which states that *all* synchronization events are observed in sequential order. Obviously, this is a very conservative measure because only those synchronization operations that content for the *same* locks should be kept in order. It is therefore very likely that we over-estimate the execution time because processes are pre-emptied due to this aggressive scheduling. However, nailing down accurate timing information is at any rate very difficult. Therefore we did not attempt to modify the dispatching engine since it captures the inter-process communication very accurately, which is key to obtain a reliable measurement of coherent misses.

What happens when going to other configurations? The most important change occurs when several processes are scheduled to run in an interleaved fashion on one processor, and when several processors may share one bus. We now discuss the impact on coherent miss and conflict miss in turn:

1. *The impact on coherent miss*. It is clear that when multiple processes are scheduled to run on the same processor/cache set, some coherent misses reported by HP-Set will either disppear or short-circuited. The first case happens when the data a process writes is later picked up by another process that shares the same cache, the second case occurs for the same reason between processes that run on different processors but share the same bus. In any case, however, HP-Set is being conservative and reporting the up-bound of coherent misses.

   Process migration in real system in an attempt to achieve better load-balancing may alter the amount of coherent misses as well: when a process writes in one cache and then later migrates to another node, its access to that data will "appear" as a coherent miss. Therefore HP-Set will under-estimate this type of coherent misses. However, process migration should be discouraged at the first place if the benefit of load-balancing does not compensate the damage done by detroying cache affinity. Until a clear process management decision is made, we do not attempt to embed any decision here in an ad hoc manner. Besides, HP-Set's emphasise is to understand applications better, having dynamic process migration will make the analysis extremely difficult, if not entirely impossible.

2. *The impact on conflict miss*. With or without process migration, HP-Set is unlikely to give the *exacct* number on conflict misses. This is due to several reasons, the first is HP-Set's decision to simulate direct-mapped caches to simplify simulation, this approach is to give bounds on the miss rates and the insight of how caches are being utilized (will be discussed in Section 4). The second is due to the fact that we run one process per processor/cache set. When schedule multiple processes on one processor, they might be displacing working set from each other and thus increase cache misses. To compensate this, we use a relatively small L2 Cache (1M). This question is unlikely to be well answered until, again, a well defined process management has been designed and implemented.

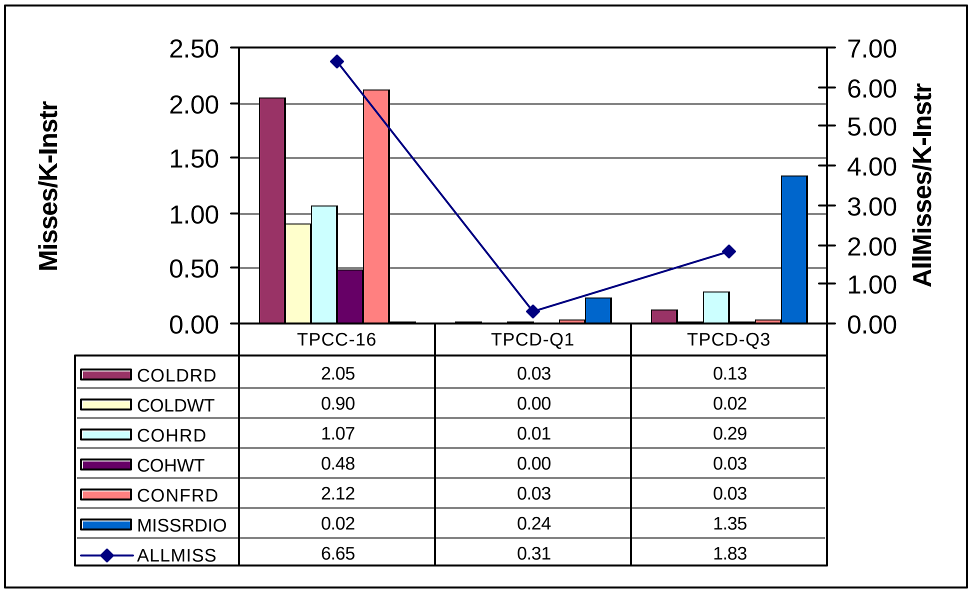
Misses are categorized into *cold*, *coherent* and *conflict* misses. The term *cold* miss deserves some discussions. A cold miss is the access miss when the process access the data the very first time. These cold misses can, in reality, be all types of outcomes: they can be true cold misses still, they can be coherent misses, and conflict misses and even hits. In fact, they are termed as *unkown* misses in cache sampling studies. While their true nature is difficult to predict, running a longer trace and collecting statistics after processing a certain amount of records will certainly help. Another thing to note is that in a system using direct-mapped caches, the cold misses reported minus the total number of lines in L2 caches will always be misses – it is just that we don't know whether they are true cold miss, coherent misses or conflict misses.

It is also important to distinguish read misses with write misses. In fact, the question set on stake is larger: with wider processors equipped with non-blocking caches, the equation of estimating the impact of memory stall becomes non-linear. While this is an expensive problem to tackle with (the only viable solution is to simulate the wider processor and non-blocking cache altogether), processors certainly need not to stall on writes as much on reads. In a multiprocessor configuration, the memory consistency model only states that stores are to be *performed* at the release point. We therefore separate load misses with store misses.

Misses are decomposed in Figure 1. Note the category *MISSIORD* is the load misses because the cache lines are invalidated due to IO operations. We report numbers in the convention of number of misses in 1K instructions. Please note the total misses are drawn towards the right side of the graph.

Examine Figure 1, we have the following observations:

1. **TPCC16**: this application is quite problematic, the total misses/K-instr is over 6, even with static affinity scheduling. Moreover, a large fraction of misses are due to conflict and coherent misses; on the other hand, misses caused by IO operations are minimum. Therefore, we should look into the coherent and data reuse index of TPCC16 in more detail, but probably pay less attention to the impact of IO (caveat discussed in Section 5).
2. **TPCD-Q1/Q3**: these two applications share one thing in common – they utilize caches very well, and their overall misses are small in numbers. However, IO causes the majority of misses. Therefore, if one is/cares to improve the performance of TPCD-Q1/3, the focus should be the IO subsystem and its interface to memory system.



	TPCC-16	TPCD-Q1	TPCD-Q3
🟥 COLDRD	2.05	0.03	0.13
🟨 COLDWT	0.90	0.00	0.02
🟦 COHRD	1.07	0.01	0.29
🟪 COHWT	0.48	0.00	0.03
🟥 CONFRD	2.12	0.03	0.03
🟦 MISSRDIO	0.02	0.24	1.35
◆ ALLMISS	6.65	0.31	1.83

Figure 1: Cache Miss Breakdown

3. The Communication Index

Coherent misses are results of inter-process communication, and they are *not* going to be eliminated no matter how cache sizes are increased. In fact, if there are significant coherent misses, increasing cache size is not only non cost-effective, but may adversely hurt the performance by attain dirty lines too long in caches, which leads to the increase of coherent miss penalty by other processes later.

The communication index of HP-Set is composed of several indices and will be discussed in turn. They are:

1. The fraction of 3rd party dirty hit
2. The fraction of false sharing misses
3. Producer-consumer pattern and update protocol investigation
4. Migratory data and the corresponding protocol handling

The first two indices are general. There are two reasons why the second two indices are chosen by HP-Set. The first being that producer-consumer and migratory communications are abundant in other applications, we would like to understand whether they exist in our applications also. Secondly, there are mature protocols proposed by the research community, and we want to see how much potential there is by incorporating them.

3.1 3rd Party Dirty Hit

When suffering a coherent miss, the miss penalty varies according to where the data is ultimately returned. For the system that HP-Set assumes, there will be two kinds of misses, often called as 2-hop and 3-hop misses, the later one is also know as 3rd party dirty-hit:

- 2-hop miss: the home node of the line has the line clean in memory
- 3-hop miss: the home node has a stale copy, and the most recent copy is dirty in the cache of the processor last wrote it.

In the case of infinite cache, the first processor that suffers the coherent miss will take a 3-hop miss and pull the data back to memory; all subsequent misses will end up as 2-hops. With a finite cache, dirty lines may be purged back to home node as a result of cache conflicts, and thus 3-hop miss will be reduced. It is also easy to see that average number of sharers can be calculated by using the 2-3hop breakdown:

$$number\ of\ sharers = 1 + \frac{number\ of\ 2\ hop}{number\ of\ 3\ hop}$$

This is so because between any two write operations, there will always be one and only one operation that suffers a 3-hop miss.

Table 4: 3rd Party Dirty-Hit and False Sharing Statistics

		IOB	Blk	LkB	lbk	All
TPCC16	All-Miss	0.9%	27.0%	72.1%	0.0%	100.0%
	3Hop	80.5%	76.9%	61.5%	97.3%	65.8%
	FALSE	79.5%	62.3%	69.6%	78.4%	67.7%
TPCD1	All-Miss	0.0%	52.2%	47.8%		100.0%
	3Hop		40.0%	14.20%		54.2%

	FALSE	79.5%	62.3%	69.6%		47.2%
TPCD3	All-Miss	0.0%	89.6%	10.4%		100.0%
	3Hop		98.7%	77.7%		96.5%
	FALSE		1.9%	65.7%		8.5%

The understanding of 3-hop and 2-hop miss breakdown is vital: they have different implications on the system requirement. 2-hop is served directly out of the main memory, therefore if the majority of misses are 2-hop misses, then speculative memory read in parallel with directory lookup as well as optimizing the path from main memory to network interface will be important. On the other hand, 3-hop is much more expensive – in terms of network bandwidth (more transactions) and latency (more hops in the network). If 3-hop misses are significant, we need faster directory checkup, no speculative read of memory (which will waste memory bandwidth anyway), faster network and cache-to-cache transfer support. The abundant amount of 3-hop also implies that the average number of active sharers is small in number, thus some saving of directory cost can be done by using shorter vector.

HP-Set over-estimate 3-hop by *not* taking into account for the write backs that turns 3-hop into 2-hop. In fact, the communication indices are all built upon infinite cache to catch the intrinsic communication pattern of processes. In reality, number of miss transactions stays the same, but with the clustering of caches, the hops will be reduced. Again, HP-Set emphasizes on application understanding, and over-estimating these hops gives a more conservative measure.

The number of 3-hop miss breakdown are listed in Table 4, the 4th row of each application gives the fraction of 3-hops in the percentage of total coherent misses, divided into different data types. The results show that across all applications, 3-hop misses are significant. Bearing in mind that the coherent misses in TPCD does not worth looking but those in TPCC does pose a performance threat, it thus tells us that engineering decisions such as those mentioned earlier optimizing 3-hop transactions can be rewarding, and it also highlights that directory savings is possible by using shorter vectors.

3.2 False Sharing

False-sharing [3, 4] happens when communications are generated because different processes are sharing the same cache line *without* sharing the same data. The exact definition will become clear when we discuss the algorithm that HP-Set uses to detect them. False-sharing was not at all paid enough attention when HP-Set was first started, it is only when a companion project aimed at reducing coherent misses surprisingly encounters *significant* false-sharing in Oracle V7's behavior that the index was designed and implemented.

To detect false sharing, HP-Set uses a bit vector, called *written*, for each line, each bit is for a subset of the bytes of the line. For simplicity, we assume that the line is 32-bytes, and there are 32 bits in the *written* vector – thus the false-sharing detection is at the granularity of byte; for each line we also has a flag called *is_false*, now the algorithm is as follows:

- **On writes**

```
If ( dirty hit ) {
   setBit( written, addr&0X1F); // keep track of written byte
```

```
    } else {
       clearVec( written );
       setBit( written, addr&0X1F);
    }
```

- **On reads**

```
If ( coherent read miss ) {
   If ( !isSetBit( written, addr&0x1F) ) {
          False_sharing++; this_line->is_false = TRUE;
   }
} else if ( isHit && this_line->is_false
   && isSetBit( written, addr&0x1f)) { // we have count a false-shr
   this_line->is_false = FALSE; False_sharing--; // but we shouldn't!
}
```

That is, we keep track of whether the coherent read miss was to the writing of the *exact* data. Note the use of the *is_false* flag to account for possible over-estimation of false-sharing.

The fraction of false-sharing reported in Table 4 is startling: 65% overall for TPCC. This effect is less pronounced in TPCD. This outcome may be partially due to the fact that tupples/attributes are packaged contiguously while accesses to are fine grained in nature.

It should be pointed out that having significant amount of false-sharing does not automatically imply that short-lines are necessarily better, even discount the fact that shorter lines will need more tag areas as well as increased directory entries all over the system. The reason being that large line size has the prefetching effect that conforms to data objects of good spatial locality – and we are only examining coherent misses here. What we do confirm is that in the realm of coherent objects, false-sharing is not trivial. What will happen when going to short lines? False-sharing will decrease, but misses reduced by prefetching with longer lines will also arise. The end result is a mixture of both trends. We use HP-Set to simulate 4-byte line size and find coherent misses are decreased by 20%.

What can one do to eliminate, or at least minimize the impact of false sharing? The responsibility largely rest on the programmers of these applications, HP-Set identifies where the false-sharing is happening and, if necessary, it can be run at page size to identify them in more detail. There are a couple of hardware optimizations one can apply as well, for example using write caches to combine writes, applying lazy releases consistency etc. However, it is not convincing, yet, that these optimizations will be cost-effective. The viable solution, if we indeed find significant false-sharing all across important applications, is to suggest *not* to push for a long cache lines too much, and use sectored L3 cache to alleviate the problem. With sectored L3 cache, it is possible to obtain the beneficial prefetching effect without the harm of false-sharing.

3.3 Producer-Consumer Type and Update Protocol

One special pattern of communication is both easy to detect and optimize: producer-consumer relationship. This can be made in a more general format as *xP-yC*, where *P* and *C* stands for producer and consumer respectively, *x* can be either *1* or *M*, stands for *one* or *multiple*. Obviously, when it is certain that one process produces the data consumed by another, a plain invalidate protocol will always produce 3-hop coherent miss which the consumer suffers (in the infinite cache case). This type of data is often seen in scientific applications at the boundary of data decomposition, or some global variables that all processes need to access. Update protocol, from simple ones such as competitive update protocols [5, 6], to sophisticated such as [7], which update the consumers when writing the data, can eliminate these misses.

Given infinite bandwidth throughout the system, an update protocol that always update no matter what will remove *all* coherent misses. The downside of any update protocol is the traffics associated with unnecessary updates. These spurious updates come from two sources. The first is when updates are brought to some one that will not use the data, or at least not immediately, this is the case for imperfect P-C detection. The another more general and also more severe problem comes from the artifact of long lines, we will see this more clearly when discussing the competitive update protocol.

The competitive update protocol of order *K* associates with each cache line a counter which can counts up to *K*. The counter is set to *K* whenever the processor access the line. When updates are sent to the current sharers, each of them decrement the counter, the line is invalidated if the counter reaches 0, or updated otherwise. With long cache lines, if there is spatial locality in the data objects, it is possible that the consumer side will overrun its threshold. And if this happens then we will be having all these update traffics without removing the actual miss.

Table 5: Producer-Consumer and Migratory Data

		IOB	Blk	LkB	Ibk
TPCC16	All-Miss	0.9%	27.0%	72.1%	0.0%
16.2%	UPRDT	10.3%	14.4%	23.0%	15.3%
	UPRD	65.0%	49.2%	37.5%	91.0%
	COHRD3	80.5%	76.9%	61.5%	97.3%
	MIG	59.4%	54.3%	72.5%	69.4%
TPCD1	All-Miss		52.2%	47.8%	
15.7%	UPRDT		1.1%	37.6%	
	UPRD		39.3%	46.2%	
	COHRD3		40.0%	14.2%	
	MIG		8.6%	52.1%	
TPCD3	All-Miss		89.6%	10.4%	
58.6%	UPRDT		0.8%	38.1%	
	UPRD		94.3%	95.2%	
	COHRD3		98.7%	77.7%	
	MIG		5.7%	75.7%	

To gain more insight of the competitive update protocol, let’s assume that there is a sequence of loads and stores from different processors to a certain line:

$$W_a R_b R_c W_a R_b R_d$$

Call the first group of readers the *last_sharers*, and the second group *this_sharers*. For simplicity, we let f stands for the ratio of the message size of an invalidate or update message over a read miss message, and also let S be the size of *last_sharers* and *this_sharers*. Denote h to be the success rate of the second update. We now proceed to compare the misses and traffics of the update protocol and competitive update protocol of order 1.

For the pure invalidate protocol, we have the followings:

$$miss_{inv} = S$$
$$traffic_{inv} = S(1+f)$$

And for competitive updates we have:

$$miss_{up} = hS$$
$$traffic_{up} = (1-h)S + (2-h)Sf$$

The second term of the traffic equation above is simply the product of the number of nodes that the protocol sent updates and the size of the message. For this simple analysis, an equal traffic breakdown is reached with the success rate of:

$$h > f/(1+f)$$

Thus, if update traffic comes for free (*h=0*), we always gain; and if the update message is as large as the read miss, then we need at least 50% success rate.

To investigate how much we can gain by using an update protocol, HP-Set implements two competitive update protocol. The first one is an hypothetical protocol, which collapses all writes by a single producer into one update, we denote this as UPDT in Table 5 as the up-bound measure. The second one, denoted as UPDTT is the default competitive update protocol with order of 1, which will update only the first time. There is no difference between the two protocols if the producer only write once before the consumer reads it. However, if the producer writes multiple times, even if the protocol is updating the right consumer, it may run over its threshold and decides to invalidate the copy instead. Therefore, the difference between the two protocols tells us the run-length of the producer, something interesting to obtain in any case.

Ignoring the results of TPCD and concentrate only on those of TPCC. Our result suggests a large difference of UPDT and UPDTT, which means that if competitive protocol is to be supported, there should also be some ways of lumping writes to the same line before they are sent to the potential consumers. The success rate of the baseline is around 20%, this means we should have enough bandwidth before we attempt this type of protocol, and only when coherent miss emerges as the major performance degradation factor. To simply put it, our results recommend that, at the present, the impact of update protocol is likely to be marginal and may not be promoted to higher priority of engineering decision. This is especially true in light of the significant false-sharing of the Oracle V7.

3.4 Migratory Data and the Corresponding Protocol Optimization

Migratory data refers to the type of communication pattern where a data changes hand in a load followed by store fashion. That is: $R_aW_aR_bW_b,\ldots,R_cW_cR_aW_a\ldots$

We will discuss the correlation of migratory data and producer-consumer data later. For now let's assume that it's largely unpredictable that who is the next node read/write the data. The migratory data is a very common pattern. For example, processes summing up a global variable, will certainly have this type of behavior. The presence of migratory data has been found in numerous applications [8] and it is by far one of the difficult communication pattern to be optimized.

Since the migratory data hops around unpredictably, their misses are very hard to eliminate – the only viable option will be using prefetching. However, there is the so-called *migratory data handling* optimization [9]. This optimization proceeds at several steps. The first one being detecting migratory data on the fly, using the regular expression outlined earlier. Once a line is marked as migratory data, the ownership will be transferred to the node that read miss on the data, with the old owner invalidated. The result is faster write completion which in turn leads to faster release as well, the second benefit is the reduction of traffics. There is downside of this optimization: if a data is wrongly detected as migratory data, 2-hop misses will be turned into 3-hops (example: $R_1W_1R_2R_3$, suppose this data is marked as migratory, R_3 will be a 3-hop miss), and there can even exist cases that hits will be turning into misses (substitute R_3 *with* R_1 in the same example). Therefore, unless migratory detection is successful, using this optimization could be risky.

One might expect, from the nature of commercial workloads, there should be significant instances of migratory data. However, the large quantity of false-sharing might just as well shield this effect from us. We report the result of migratory data in Table 5, the data is in the form of percentage of 3-hop miss (a clear instance of migratory data should always begin with a 3-hop load miss). We see indeed that migratory data consists the majority of communication pattern, 70% of 3-hops are of migratory pattern. While this is encouraging, the real implementation *must* use an adaptive migratory data handling algorithm, that is, the default protocol should not to rip off the ownership of a dirty line and transfer it to the node that load misses next. Otherwise it is almost certain that a fraction of 2-hop misses will be turned into 3-hops and endure larger miss penalty.

It is possible that some data can be both migratory data *and* producer-consumer. This happens when the data is shared between two processors, and each has a deterministic load-then-store fashion. This, in fact, happens quite often for some pages of TPCC.

4. The Data Reuse and Locality Index

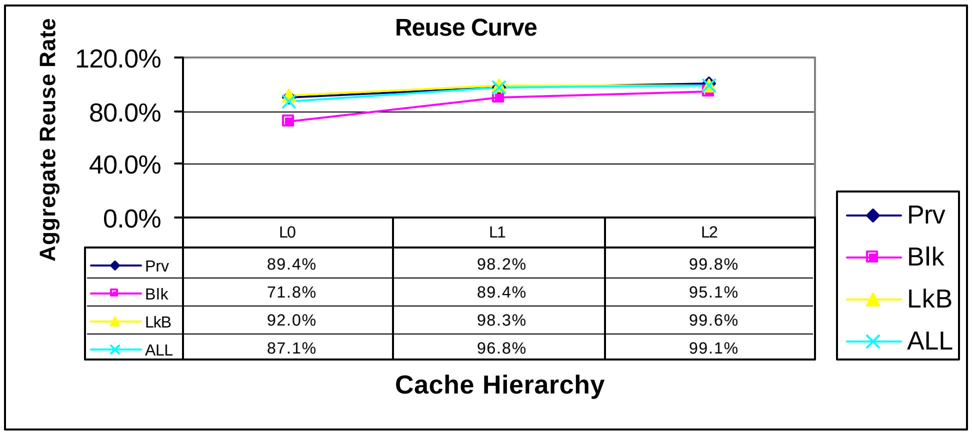
We have seen that there are significant conflict misses in TPCC. HP-Set also reports that, using the first-touch algorithm, 75%, 89% and 31% of these conflict misses are remote, for TPCC16, TPCD-Q1 and TPCD-Q3 respectively. This is indeed quite better than using a random algorithm, which will give instead 15/16, 14/15 and 22/23. However, HP-Set also indicates that first-touch algorithm might result in disparity of memory usage among different nodes. Some more investigation should be in order. Table 6 reports the conflict miss distribution of the three applications. With these data, and taking into account the projection that when we use larger L2 caches in the future, the database system might scale up their working set as well, it seems evident that architecture variation such as L3 cache should be recommended.

Table 6: Conflict Miss Distribution

	Lit	Prv	IOB	Blk	LkB	Ibk
TPCC16	1.78%	9.19%	2.45%	77.01%	9.11%	0.14%
TPCD1	2.25%	0.00%	2.14%	84.36%	11.13%	0.11%
TPCD3	1.47%	50.40%	5.03%	14.63%	23.38%	0.45%

How does the data use the caches? To understand this HP-Set simulates a vector of direct-mapped caches. There are a number of advantages by doing so. First, by looking at the reuse rate (defined as reuse captured by the level of cache in question over the total number of reuse in infinite cache) for different data types and/or pages, we will be able to identify the data using caches less efficiently and is thus the target of further improvement. Secondly, we keep track of what portion of the misses, in all levels of caches, is to remote node, and thus we can easily play with different combinations without the need to run multiple simulations. For example, suppose we choose the vector as 32K, 256K, 1M, 4M and 16M, then we can use the result of any three of these caches to estimate a three-level cache hierarchies.

The reuse curves of TPCC16 is presented in Figure 2. There are three levels of cache hierarchies, with L0 being 4K, L1 64K and L3 1M. We immediately identify that the data type *Blk* is the one that does not use the caches well – its reuse curve is below others. Therefore, software optimization should pay attention to improve, for example the temporal locality, of this type of data. There is also hardware optimization opportunity as well. If we know a certain data is not using cache well because its long reuse distance, we can have it bypass lower caches so as to leave more rooms for other data. This optimization we call as *smart-caching*. It is easy to note that for each instance of such by pass, one can at most prevent one future conflict miss. This optimization has been investigated, and the result is that conflict miss is reduced by 2%. This is largely due to the lack of the bypass instances. We will look into this in more detail in the future.



	L0	L1	L2
Prv	89.4%	98.2%	99.8%
Blk	71.8%	89.4%	95.1%
LkB	92.0%	98.3%	99.6%
ALL	87.1%	96.8%	99.1%

Figure 2: Reuse Curves for TPCC16

4.1 The Granularity Index

It is often important, and interesting as well, that we can understand the granularity of the data object that the application references. However, here we treat line size as a given and measure granularity in multiple of the default cache line size. This is mainly because we do not have control of various cache parameters. And if finer granularity is needed, we can always repeat this part of the simulation with shorter lines.

The granularity index goes hand in hand with sequential prefetching. And it helps to understand the potential of other optimizations [10, 11]. The larger the granularity is, the more likely that sequential prefetch should do well. Therefore, we design the algorithm of detecting granularity based on tagged sequential prefetching.

The algorithm of tagged prefetch is quite simple. Associated with each line of the cache there is a tag bit. This bit is reset when the line is fetched from below. Then, whenever a line that has the tag bit reset is referenced, a prefetch will be issued for the next line, and this line's tag bit will be set. One can imagine a sequence of blocks is being fetched, as long as their accesses are in the same order. This constitutes the base of granularity detection algorithm.

To detect granularity, we distinguish the prefetch that first starts a sequence, such instance is termed as *pref_lead*. When a prefetch is issued, we check to see if the previous line has the tag bit set, if not then this one is a *pref_lead*, otherwise this reference belongs to the sequence, we *reset* the previous line's tag bit. Using this algorithm, suppose we have a reference stream such as: *R(1)R(2)R(3)R((1)R(2)R(3)*, we will have 2 *pref_lead* for total of 6 prefetches. Then we can derive a set of statistics as follows:

$$gran = 1 + pref\_good / pref\_lead$$

$$effi = pref\_good / pref$$

The *pref_good* statistics are the portion of prefetches that are used before invalidated by other processes. For the above example, we have granularity as 3 and prefetch efficiency as 2/3, exactly what we expected.

These tags are built along with the directory entries, therefore there can be arbitrary number of sequences on the fly in the system at any time.

Next we can change the events that *triggers* the prefetching to detect granularity of different kinds of accesses. For example, we can allow only cold misses to enter the prefetch routine, and this should tell us the granularity of cold misses. By triggering the prefetching only on load access, we will then know the granularity of objects accessed by loads – something interesting theoretically but possess little practical interest. In Table 7 we report coherent object granularity, that is, we trigger prefetch with coherent misses only.

Ignore the data on *Prv* region for now, for the shared region, TPCC has very small granularity, something very nature for this application. TPCD has a somewhat larger granularity. In any case, however, coherent objects do not seem to be anywhere close to two lines. These traces are compiled for 32 bytes line size. A trace compiled for larger line size should be interesting to study. In the next section, we will see how the IO makes a significant difference as mentioned earlier.

Table 7: Coherent Object Granularity Distribution

		Prv	Shar	All
TPCC16	Gran	10.19	1.38	1.44
	Pref-Eff	90.3%	22.1%	24.8%
TPCD1	Gran	3.82	1.81	1.66
	Pref-Eff	74.2%	42.4%	73.9%
TPCD3	Gran	3.93	16.28	4.50
	Pref-Eff	75.0%	92.7%	78.0%

5. The IO Index

The goal of IO index of HP-Set is *not* to study the IO system behavior, but rather how the IO subsystem interacts with the memory system. This is first because that we fundamentally lack such knowledge, and is also due to some caveats that force us to backup from understanding the IO system using HP-Set altogether. The problem is largely associated with the tracing methodology. The dilemma is that to understand the interaction of memory with IO, we have to use load/store traces whose very nature determines that they are both expensive to collect and process. However, IO events are sparse, especially for systems with larger main memory, thus *unless* we collect longer traces, such as system call traces, we will not be able to get a complete picture.

Table 8: IO Index Breakdown

		Prv	IOB	IBk	ALL
TPCC16	INSTR/IORD(B)	1204.1	38.0	497.7	34.3
	IOINV/IORD	94.8%	0.0%	0.0%	2.7%
	INSTR/IOWT(B)	813.0	222.1	90834.5	174.1
	IORC/IOWT	93.7%	98.9%	100.0%	97.8%
	MISSIORD/ALLMISS	0.3%	0.0%	0.0%	0.3%
TPCD1	INSTR/IORD(B)	88.0	7648.7		87.0
	IOINV/IORD	72.9%	20.6%		72.3%
	INSTR/IOWT(B)	1023538.8			818831.0
	IORC/IOWT	100.0%			80.0%
	MISSIORD/ALLMISS	77.6%	0.1%	0.0%	77.7%
TPCD3	INSTR/IORD(B)	16.4	8203.0		16.4
	IOINV/IORD	73.1%	69.8%		73.1%
	INSTR/IOWT(B)	811351.4			649081.1
	IORC/IOWT	100.0%			80.0%
	MISSIORD/ALLMISS	73.2%	0.1%	0.0%	73.4%

To have a complete picture of IO subsystem's impact, therefore, involves two steps. We should consult researches that based on using long, system level traces to get the idea of sheer volume of IO, and then use HP-Set to gain the knowledge of how IO interacts with the rest of the system. IO indices of HP-set are listed in Table 8, here is the definition of terms used:

1. *INSTR/IORD(B)*: number of instructions per IO (inbound) bytes
2. *IOINV/IORD*: number of cache lines actually invalidated due to IO reads
3. *INSTR/IOWT(B)*: number of instructions per IO (outbound) bytes
4. *IORC/IOWT*: number of cache line recalls due to IO writes
5. *MISSIORD/ALLMISS*: fraction of cache misses due to IO reads over total L2 cache misses

Not shown here is the total IO traffics (weighted over different message size) over total system traffic.

We discuss the results of TPCC and TPCD in term:

1. **TPCC16**:
   - *Traffic*: it turns out that for TPCC16 the traffic contributed by IO can be significant, HP-Set reports an estimation of 12%. However, most of these IO traffics are a total waste from the point of memory subsystem: only about 5% of IO reads actually invalidate cache lines. IO writes however, have a very substantial recall rates.
   - *Miss*: misses caused by IO is minimum.
2. **TPCD**:
   - *Traffic*: IO traffic is significant for TPCD-Q3. However, since TPCD has low overall traffic because of its efficient usage of caches, it is less likely that IO traffic makes a big impact.
   - *Miss*: IO causes most of the them – well over 70%.

The results can be simply put it this way: IO affects TPCC mostly through traffic, and affects TPCD through cache misses – if processor stalls at all on memory access, chances are that this is due to IO reads.

There are a few points to make regarding these results:

1. ***Isolate the IO subsystem from the rest***. We have seen that IO traffics go on the bus in vain in the case of TPCC. For future applications, it is very likely that large amount of IO *can* by-pass the caches as well. Isolating IO can be done by instructing the DMA inbound traffic to examine the directory first, if there is no copies made in the caches, then there is *no* need to go further. This isolation also enables us to optimize the sub-component of the system without the need to worry about the interference and perturbation towards other components.
2. ***Update the caches in private region.*** The ratio of MISSRDIO over IOINV instances in the private memory region tells us how successful it will be if we were to update caches if IO traffic towards private region has to get on bus. We have 92%, 93% and 97% for TPCC16, TPCD-Q1 and TPCD-Q3 respectively. These numbers are strong support for one simple principle: if IO reads in private data region has to get on bus, let them go all the way to update the cache.

3. ***Prefetch for MISSIORD is effective***: for this we need to refer back to Table 7, the coherent misses in private memory are all caused by IO. Not with a surprise, we see granularity and prefetch efficiency both notches up. Therefore, if the underlying architecture does not support updating the cache, the next thing we can do is to prefetch on misses caused by IO. Note we need to distinguish misses by IO from other misses, as the small granularity and low efficiency of shared data region so indicates.

These suggestions made by analyzing the data of this set of application using HP-Set: isolate the IO subsystem, update or prefetch on misses due to IO reads, should largely address the necessary design issues of IO-memory interface.

6. Conclusion

In this report, we defined a set of indexes that gauge the performance of shared-memory multiprocessor. The tool, HP-Set, is based on the notion that application characterization must itself deliver architectural relevant insights. HP-Set achieves the goal be actually simulated optimizing protocols on the flight. By using HP-Set, we gain several insights of the applic ations as well as new research opportunities, these contributions are listed below, with new research opportunities underlined:

1. ***Overall Analysis***: we find that all three applications, TPCC16, TPCD-Q1 and TPCD-Q3 are memory intensive. However, TPCD-Q1 and TPCD-Q3 have small miss rates, they use caches efficiently and the only performance degradation comes from misses caused by inbound IO traffics. TPCC, on the other hand, has more aggressive demands on the cache/memory systems, in both conflict misses and coherent misses.
2. ***Communication Analysis***: we find 3-hop misses are dominating. Therefore faster directory lookup, cache to cache transfer and high-bandwidth low-latency network are required. Directory savings might be achieved by using shorter vectors, instead of saving total number of entries. There are significant false-sharing – enough to warrant both software and hardware optimizations. The contribution from more advanced cache coherent protocols to handle producer-consumer relationship can be marginal, especially in light of the false-sharing problem.
3. ***Data reuse and locality analysis***: we find that the first-touch data placement algorithm works well. However, the algorithm might result in disparity of memory allocation among participating nodes. More research should be done towards a better solution. We also identify a certain type of data that does not use cache efficiently. Smart-caching, whose aim is to determine which level of cache should retain a data deserves more research effort.
4. ***Granularity analysis***: our result indicates that the granularity of coherent object is rather limited. Therefore blind sequential prefetching might adversely degrade the performance if the system does not have enough bandwidth. Large granularity, however, has been found to be common for the private IO region.
5. ***IO-Memory analysis***: we find that IO contributes a non-negligible factor in total system traffic in TPCC, and is the major source of cache misses for TPCD. It is important to isolate IO subsystem from the rest of system as much as possible for a variety of reasons. We also believe that either updating the caches or sequential prefetch should work well to eliminate these misses, provided that a mechanism is derived to distinguish the private IO region apart from the rest of memory.

7. Acknowledgement

The author wishes to thank Sekhar Sarrukai for test-driving HP-Set and many of his valuable inputs. Tom Rokicki, John Janakiranman and many others have provided feedback; I am grateful for their time. Manu Thapar provides crucial management support and guidance.

Reference:

[1] G. Abandah, "Tools for Characterizing Distributed Shared Memory Applications," Hewlett-Packard Laboratories HPL-96-157, Dec 1996.

[2] D. L. e. al, "The Stanford DASH Multiprocesso," *IEEE Transactions on Computer*, vol. 25, pp. 63-79, 1992.

[3] J. Torrellas, M. S. Lam, and J. L. Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches," *IEEE Transactions of Computers*, pp. 651-663, 1994.

[4] M. Dubois, J. Skeppstedt, L. Riciullli, K. Ramamurthy, and P. Stenstrom, "The Detection and Elimination of Useless Misses in Multiprocessors," presented at the 20th Annual International Symposium on Computer Architecture, 1993.

[5] F. Dahlgren and P. Stenstrom, "Reducing the Write Traffic for a Hybrid Cache Protocol," *Journal of Parallel and Distributed Computing*, vol. 26, pp. 193-210, 1995.

[6] A. R. Karlin, M. S. Manasee, L. Rudolph, and D. D. Sleator, "Competitive Snoopy Caching," presented at the 27th Annual Symposium on Foundations of Computer Science, 1986.

[7] A. Raynaud, Z. Zhang, and J. Torrellas, "Distance-Adaptive Update Protocols for Scalable Shared-Memory Multiprocessors," presented at the 2nd Annual International Symposium on High Performance Computer Architecture, 1996.

[8] W. Weber and A. Gupta, "Analysis of Cache Invalidation Patterns in Multiprocessors," presented at the 3rd International Conference on Architectural Support for Programming Language and Operating Systems, 1989.

[9] P. Stenstrom, M. Brorsson, and L. Sandberg, "An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing," presented at the 20th Annual International Symposium on Computer Architectures, 1993.

[10] Z. Zhang and J. Torrellas, "Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching," presented at the 22nd Annual International Symposium on Computer Architectures, 1995.

[11] S. Palacharla and R. E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement," presented at the 21st Annual International Symposium on Computer Architectures, 1994.