High Availability Issues in DSM Systems: Research Opportunities

Zheng Zhang
Computer Systems and Technology Laboratory
HP Laboratories Palo Alto
HPL-2001-78
March 30th , 2001*

E-mail: zzhang@hpl.hp.com

shared memory multiprocessors, high availability

This report documents a first-cut understanding of the HA issues in DSM systems. We discuss the general HA strategy, advocate for minimizing fault propagation, system reconfiguration time and performance degradation as the distinctive goals for the three stages that the system goes through after the occurrence of a fault till full recovery. We show the possibility of estimating the impact of a fault through hierarchical component dependency analysis. We point out that coherent protocols should be extended and transactions be made closed in order to detect the fault and maintain data integrity. In particular, we propose source-buffering to augment dirty data transfer protocol in preparing for possible data loss and corruption. N+1 stand-by system is suggested as the ultimate HA solution. Further research opportunities are discussed. This report skims through a broad range of issues, but it does not attempt to treat each of them in depth.

1. INTRODUCTION 3
1.1 WINDOWS OF OPPORTUNITY 3
1.2 NEW COST/PERFORMANCE CONSTRAINTS 4
1.3 CHALLENGE OF FINDING RESEARCH OPPORTUNITIES 4
1.4 ORGANIZATION OF THE REPORT 5
2. STRATEGY OF HIGH AVAILABILITY IN DSM 5
3. DEPENDENCY ANALYSIS IS OF SYSTEM COMPONENTS IN DSM 7
3.1 HIERARCHICAL SYSTEM COMPONENTS DEPENDENCY ANALYSIS 7
3.2 UNDERSTAND THE IMPACT OF COMPONENT DEPENDENCY 9
4. IMPROVE HA WITHIN A NODE 12
4.1 INCREASE FAULT-TOLERANCE WITHIN A NODE 12
4.2 GRACEFULLY DEGRADED NODE ARCHITECTURE 12
5. IMPROVE HA OF THE SYSTEM 14
5.1 INCREASE FAULT DETECTION CAPABILITY USING THE COHERENT PROTOCOL 14
5.2 N+1 STAND-BY DSM SYSTEM 15
5.2.1 Rational 16
5.2.2 Recovery in N+1 System 17
6. CONCLUSION 18

1. Introduction

This report is a summary of several weeks' study of the High Availability (HA) issues of DSM system. In this I am integrating relevant materials that I learn from various literature resources and many notes that I have taken during the period. The intent is to give a first-cut understanding of the issues and discuss some potential research opportunities. They are by no means final. The report is written only in the hope of getting my thoughts more organized and to welcome critiques and suggestions from my colleagues, who have already helped me a great deal to get me started.

1.1 Windows of Opportunity

It is highly desirable that an HA system continues to provide correct services after faults have occurred, albeit with possibly degraded performance. Such fail-over property is currently easier to be reliazed in systems based on clustered-solutions. Examples of commercially available clustered-solutions are *ReliantHA* and *ReliantDLM* from SCO, *Wolf Mountain* from Novell, and *Wolfpack-NT* from Microsoft. Clustered-solutions, however, deliver only limited performance, especially for applications that demand response time as well as throughput. The fundamental reason behind the low performance of clustered-solution is its inefficiency of utilizing machine resources to match

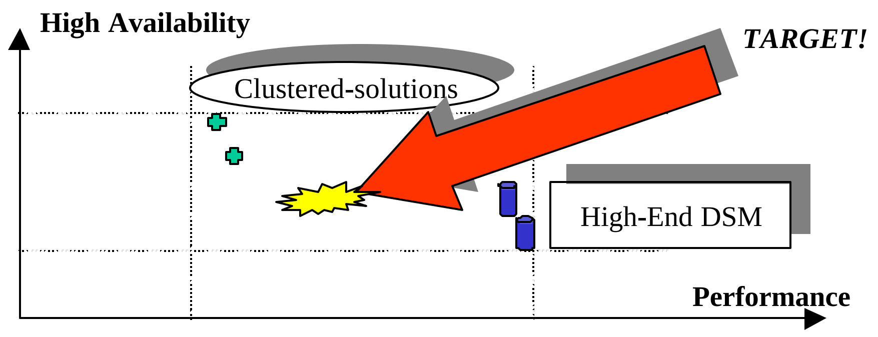


Figure 1 The Space of Performance and High Availability

parallelisms of the applications. In addition, issues such as load-balancing poses challenges as well.

At the other end of the spectrum is the high-end, shared-memory based DSM systems. These systems are capable of much higher performance, but on the other hand their HA abilities may not be as robust as clustered-solutions. Boosting the HA of DSM systems relies on matured software technologies such as multi-kernel Operating Systems. In addition, the software-hardware interface regarding HA support is no longer as cleanly defined as in the case of clustered-solutions. In fact, we can think of multi-kernel Operating System in DSM is approximating this particular interface of clustered-solutions.

This situation is depicted in Figure 1, with the target zone marked. Note that we have lower target performance, this is a direct result of using commodity components as building blocks for our DSM systems, not a necessary condition to achieve HA. Hopefully, the techniques that we research here will apply to high-end systems as well. What we are trying to provide, hopefully, is a much greater performance over clustered-solutions, with HA of the system as a whole approaching to that of clustered-solutions.

1.2 New Cost/Performance Constraints

The systems that we target at are low to mid-range DSM, and they are built on top of commodity boxes. It is for this reason that we will not expect them to compete at the same performance zone of those high-end DSM.

Building DSM out of commodity parts also poses greater difficulties to boost the HA, mainly because the system-HA is bound to the HA of the commodity components, of which we have no control.

The cost constraints also determine that many techniques employed in high-end fault-tolerant machines to be inappropriate. For example, the high-end fault-tolerant machines often use double or triple redundancy to detect and/or correct errors. Besides the fact that these redundancies linearly increase the system cost, there are non-trivial performance penalties associated with fault detection using lock-step comparison. However, we will see that some of the principles of full fault-tolerant machines can indeed be borrowed.

1.3 Challenge of Finding Research Opportunities

Finding research opportunities in HA of DSM is very challenging because of two problems. The first is the difficulty of assessing implementation complexity. Any HA optimization will likely to involve the software-hardware interfaces. If the interfaces are themselves fuzzily defined, it is hard to understand how many changes have to be made into the operating system in order to bring up the full benefit of the optimization.

The second reason is the difficulty of quantifying the result. In reality, system faults have long MTTF, running simulation as long is prohibitive. Many times one has to build a full-scale, event-driven simulator running operating system code as well as the applications, then inject faults and record system behavior. This characteristic makes the design/implementation overhead becomes the dominant portion of the research effort. Understandably, among the few interesting academia research projects that have been published so far, quantifying results are either in essence a design verification report, or not directly to the point of high-availability. A lack of relevant metrics contribute to the problem further, we will propose a set of metrics in Section 2.

Thus, it is one of the primary goals of this report to generate discussions of how to define adequate research opportunities. Whenever possible, potential research projects will be highlighted in the remaining of the discussions.

1.4 Organization of the Report

Section 2 reviews the general strategy that we propose to take on the high availability issues in DSM. Section 3 proposes the methodology of understanding the dependencies of system components in light of how a fault will affect the rest of the system. Following the foundation being laid out, Section 4 discusses how to improve HA within a node. Section 5 introduces the idea of how to enhance coherent protocol for better fault-detection capability, and then proceed to propose the N+1 stand-by HA DSM system, and the general algorithm of its recovery in case of fault. We conclude by filling out a table of research opportunities in Section 6.

2. Strategy of High Availability in DSM

In this report, we do not discuss HA issues in IO subsystem, and assume a robust and HA IO subsystem is already available. That is, disk accesses are independent of node failure. In the worst case, IO resources physically allocated at one node are available to other nodes via an IO interconnect. This assumption is made because the HA in IO subsystem is critical enough to warrant a

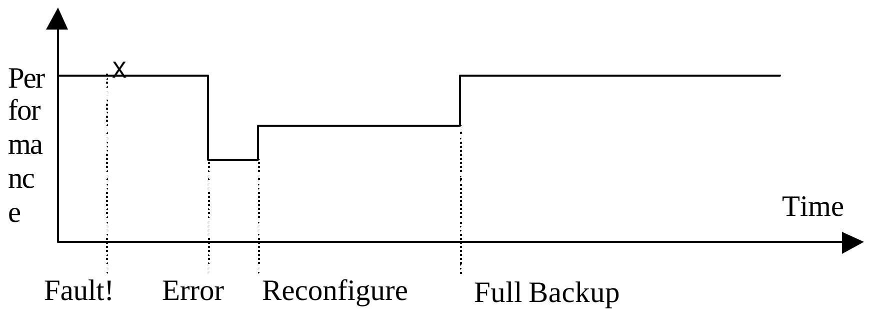


Figure 3 Decomposition of System Operation Period

separate study, and that it is largely orthogonal to the remaining of the system. In addition, an HA IO subsystem is already a essential requirements for all DSM and clustered-solutions alike.

We decompose the period starting from a fault occurrence till the system backs to full functionality into three separate phases, as shown in Figure 3.

Phase	**Goal**
Fault-to-Detected	• Preserve data integrity • Fast detection to prevent further machine states pollution
Detected-to-Reconfigure	Fast reconfigure to minimize service perturbation
Reconfigure-to-Full Recover	Graceful performance degradation

Table 1 Goals for Different Phases after Fault Occurrence

We define goals to be achieved in each of the three phases in Table 1. More details are given as follows:

1. ***Phase* 1**: Fault-to-Detected
   this is the phase when a fault has occurred till it is finally detected. A fault can potentially corrupt data. Thus the primary goal at this stage is to mask the fault if at all possible, through, for example, advanced ECC code and parity bits. If, however, the fault can not be masked, we should minimize the possibility that this fault further pollute other data of the system. One issue to be understood here is *how sensitive the fault-detection and recovery algorithm should be triggered*. This is so because many faults can be transient, and thus if a fault has been found *and* is likely to be of transient type, one might want to retry a number of times before engaging the recovery algorithm, which is usually quite costly. If the fault is not transient and we still blindly retry a number of times, we simply enlarge the window of further fault propagation and data pollution with no benefit in return.

2. ***Phase* 2**: Detected-to-Reconfigured
   this is the phase in which fault-detection algorithm is ran to isolate the faulty component, and reconfigure algorithm subsequently re-integrates the system. The goal here is to make this stage being accomplished fast enough so as to minimize service perturbation, and yet to preserve reconfiguration accuracy and maximize the utilization of surviving resources.

3. ***Phase* 3**: Reconfigure-to-Full Recover
   this is the period that the system possibly operates under degraded per-

formance. The goal is therefore to provide graceful performance degradation.

These three guidelines are the main principles that drive the rest of the investigation in this report. Although there have been a number of studies on HA in the DSM systems, none of them are proceeded by defining a set of goals that the HA must achieve. The outlines given above is the first attempt towards defining and measuring the effectiveness of various HA proposals. We need to investigate further on finding the metric for each of the goal, a set of first-cut metrics is listed below:

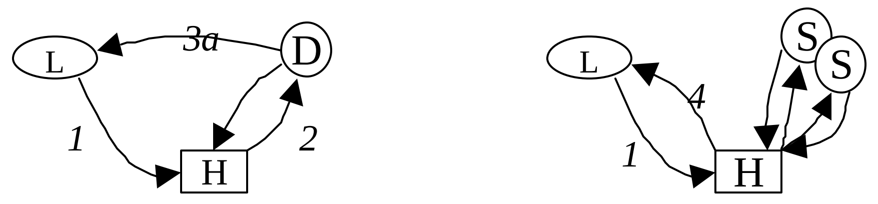
1. ***Phase 1***: a probability distribution of loss of data integrity and services
2. ***Phase 2***: services dropped during the time, and whether the surviving resources are maximally utilized after system re-integration.
3. ***Phase 3***: performance degradation as a percentage of full system power.

3. Dependency Analysis of System Components in DSM

The well being of the complete system relies on the correctness of interactions among components that participate in the transactions. It is thus instructive to understand the dependencies among the components.

3.1 Hierarchical System Components Dependency Analysis

Component dependency can be approached in a two-level hierarchical manner. The top-level dependency graph comprises nodes of the system, with dependency arcs being constructed among the nodes according to the coherent protocols. That is, protocol actions are the same node dependency graphs. Figure 6 describes two examples dependency graphs at this level. In the figure, *L* meaning the node that originate the transaction, *H* is the home node of the missing data, *D* is the node keeps the dirty copy, and finally *S* is the nodes that currently contain shared clean copy. In this graph three nodes are dependent on each other in both cases, with an implicit dependency on the interconnect that transmits messages back and forth. At a given time, a node may be involved for multiple transactions, therefore for any node we in fact has a *network* of dependency graphs, which will change dynamically as old transactions retire and new ones start.



```
Read miss on
Remote Dirty
```

```
Write Miss on
Remote Share
```

Figure 6 Example of node dependencies at the top-level

The next level of dependency analysis goes into details within each node. From any transaction point of view, the components engaged present a collection of state machines. These state machines are responsible for maintaining their internal states as well as the data that they own, if applicable.

Without loss of generality, Figure 8 shows the major components within a node of a DSM system. They are further summarized in Table 2.

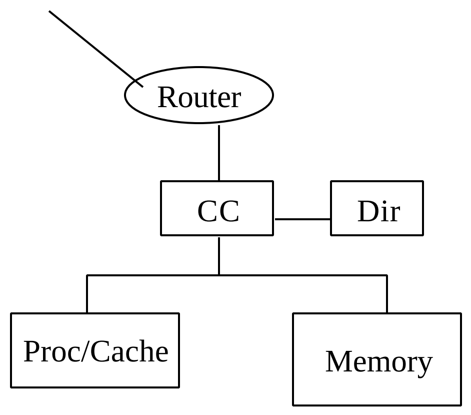


Figure 8 Components within a Node

Component	Faulty Behavior
Router	Erroneous message handling: drop, truncate, misroute etc.
Coherent Control-	Malfunction: arbitrary coherence action, directory infor-

ler	mation corruption
Directory	Malfunction: directory information corruption
Memory	Malfunction: not responding or data corruption

Table 2 Components with a node of DSM System

Dependency analysis in this level takes the same approach as we did for the node-level. That is, by examining how these components are involved in responding to different transactions. Table 3 gives a summary of the analysis of dependence set for major transactions. “Node” in this table corresponds to the role in which it participates in transactions of Figure 6. The result will drive the Section 4, where we discuss how to improve HA of a node.

Node	Transaction	Rtr	Dir	M	P/C
L	Originate transactions	X			X
D/S	Respond to recall	X			X
H	Read miss to idle line	X	X	X	
	Read miss to shared line	X	X	X	
	Read miss to dirty line	X	X		
	Write miss to idle line	X	X	X	
	Write miss to shared line (miss in L case)	X	X	X	
	Write miss to shared line (hit in L case)	X	X		
	Write miss to dirty line	X	X		
	Misc. requests/responses	X	X		
	Recall response	X	X	X	

Table 3 Dependence Set for Components within a Node

3.2 Understand the Impact of Component Dependency

Once we know how to obtain dependency graphs among system components, we can proceed to lay more theoretical foundations which will help us better formulate various problems. Although in practice these formulas might only give limited guides and insights, never the less they will provide the handle with which we can clarify at least some fuzziness in this field.

When a component *X* becomes faulty, the immediate result is that any transactions that engages it will not complete correctly. We call the *direct impact* of *X* at time *t*, *DI(X, t)*, to be the union of all these transactions. Mathematically, we have:

$$DI(X,t) = \bigcup_{T} \text{All } T \text{ that depends on } X \text{ at } t$$

To understand how *X*'s fault can propagate to the rest of the system, we need to define the *Dependence Set* of a component *X*. Each element in *DS(X, t)* is a tuple of two attributes, an id and a time. Altogether they identify how *X* impact which component and at what time. Specifically, we have:

$$DS(X,t) = \bigcup_{\substack{Y \text{ is touched} \\ \text{by } DI(X,t)}} [Y, T_Y(X)]$$

We further divide *DS(X, t)* into two subsets. The first subset, the *direct dependent set DDS(X, t)*, are components who will rely on the response from *X* in order to issue any future transactions. At the node level, the typical direct dependent set includes nodes that suffer misses and the corresponding transactions go through node *X*. In the case of read miss, the processor will halt for the data to return, and in the case of write miss, it will halt at synchronization release time when collecting invalidations. Impact of *X* made on the components in the second subset, the *indirect dependent set IDS(X, t)*, is more through data and directory dependence. For example, if there is a write-back transactions in *DI(X, t)*, component that will read this particular line will get a stale data if *X* fails to process this transaction. The feature of direct and indirect dependence set will be further explored in Section 5.1.

We can proceed to establish *DS(Y, Ty(X))* the same way we did for component *X* at time *t*. The *total dependent set* of *(X, t) or TDS(X, t),* is found by a breath-

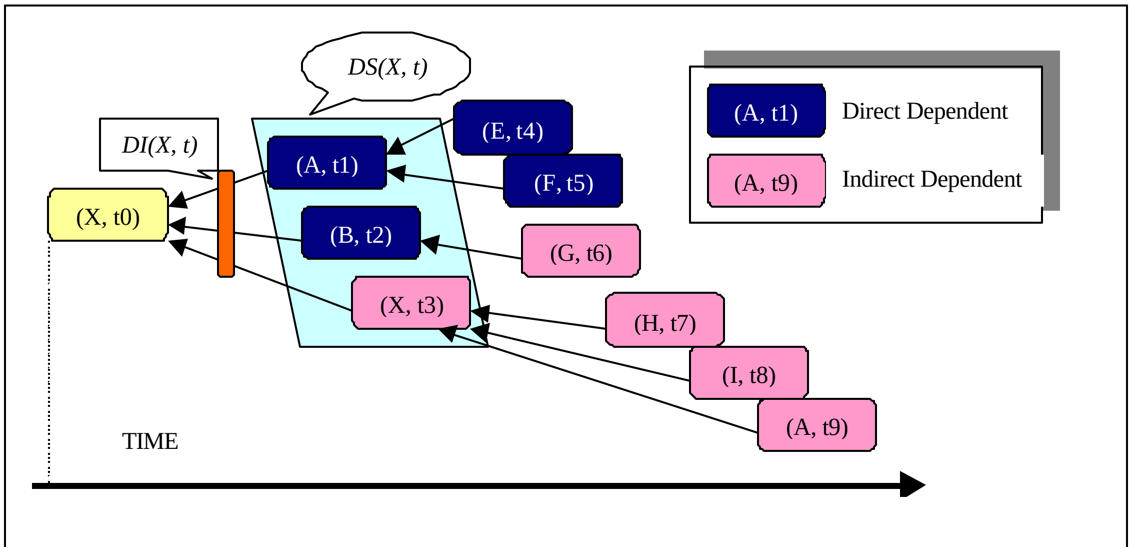


Figure 9 Direct Impact, Total Impact and Dependence Set

first search of the dependent network constructed, rooted at *(X, t)*. This is illustrated in Figure 9.

Notice that every element in *TDS(X, t)* is annotated with time, therefore the components affected by the faulty *(X, t)* at *T>t* are:

$$\bigcup_{Y} \forall Y \subset TDS(X,t) \text{ and } T_Y(X) < T$$

That is, we can answer questions such as "*how much a fault affect system after time* t_1?", at least this is made possible theoretically by formulating the problem as outlined above. Likewise, questions such as "*how does the fault propagate in the system*" can also be answered by traversing *TDS(X, t)* in an in-order walk (order here being the time).

There are still a few things need to be pointed out here. First of all, *TDS(X, t)* is always an approximation of the real situation. The sources of deviation are mainly because that *TCS(X, t)* will be altered, possibly arbitrarily, if a fault indeed occur. The components in *DDS(X, t)* will halt, and their presence in *TDS(X, t)* will be no more than the first time. An example of this redundancy is the *(A, t9)* in Figure 9: *A* belongs to *DDS(X, t)* and will thus issue no other transactions.

Furthermore, other components might simply behave unpredictably. What deserves further investigation is to detail out exactly *how* components is affected, and incorporate them into the analysis.

4. Improve HA within a Node

4.1 Increase Fault-Tolerance within a Node

Section 3 provides insights as how we will go about improve the fault-tolerance of a node. By far the most stressed component is the router, through which the node talks to the rest of the system. The second most stressed is the coherent controller, which also operates on the directory. Provided that failure rates of the router and the coherent controller might constitute a problem, one design that maximize HA of a node is shown in Figure 10:

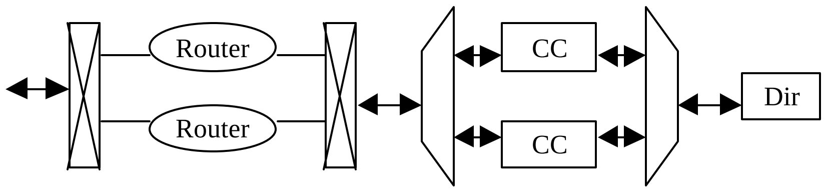


Figure 10 An HA Node Architecture

In this node architecture, two coherent controllers are hot-paired into operation. Their outputs are always compared before committed outside the boundary. The outputs of coherent controller are usually simple enough to ensure that a comparison will not add noticeable delay in the critical path. The two routers are operated in idle-standby mode, that is, the second router is only activated upon the failure of the primary one.

Whether this design is justified depends on field data on the failure rate of these components. When these data is available, we can decide where the redundancy should be added to maximize the cost-performance ratio.

4.2 Gracefully Degraded Node Architecture

Table 1 also shows that a gracefully degraded node architecture is possible by reduce the node to either a *computing* node or a *memory* node. Simply put it, a memory node is a node whose processor/cache component has failed, and a computing node is a node whose memory subsystem is not functioning. We now examine them in turn:

1. **Computing Node.**

   A computing node can still contribute to the computing power of the system with its processor/cache component. For a computing node to function, the cache controller is not required to be working. In addition, the performance of computing node can be greatly improved if part, if not all of the memory can be accessed by the local processor, though they will not and should not be accessible by other node because of the absence of the coherent controller.

   Degrade into computing node is relatively easy, if so desired. The only requirement is that the path of the processor/cache to router must still be open: it must not depend on the failure of the coherent controller.

2. **Memory Node.**

   A node can safely downgrade into a memory node if only the processor/cache component has failed. We assume that a processor/cache failure will not jeopardize the paths with which a normal home node needs to complete transactions.

   If the coherent controller has failed, the memory can still be exposed to the remaining of the system, provided that mutually exclusive access to this memory range is insured. One way to enforce mutual exclusive is to restrain access under operating system or application control. For example, some portion of file buffer can use this memory still(?).

   Another more costly solution is to couple with the memory with a reduced coherent engine in prepare for the failure of the full coherent controller. This reduced coherent engine only allows either *idle* or *dirty* state of the line. These two states require minimum coherent state that can be stored along the memory. However, the engine itself might still be quite complex.

We should point out that here we have not considered real implementation constraints which might render such degraded node architecture either unrealistic or insignificant. To give a simple example, consider the case where all of these components reside on one board: a design that minimizes inter-board communication latency. Although we do achieve graceful node performance degradation before repair, the unit of system reintegration will be the complete node regardless, and thus there will be further service perturbation when we bring the complete system to full power.

5. Improve HA of the System

5.1 Increase Fault Detection Capability using the Coherent Protocol

Fault detection is usually accomplished through a combination of different techniques. For example, high-end fault-tolerant systems usually use voting or comparison to mask or detect errors. Many current systems use ECC and parity bit mechanisms extensively in their components. The particular technique we will discuss here is time-out. It should be noted that it is often the case that one time-out will not immediately trigger fault-detection algorithm, this is so not only because of the case made by transient fault, but more for protocol timing extensions to cope with long delays induced by network congestion.

It is easy to see that any fault of *X* can be trivially detected by any *Y* that belongs to the direct dependence set of *X*. This is so because *Y*, by definition of direct dependence, is waiting for some response from *X*. In practice, however, a time-out at *Y* can only tell that *something* is wrong along with the path, which includes *X*, that the transaction has taken. By logic ***AND*** operation on all time-outs of all the direct dependence set of *X* will help to locate that the fault is *X* more quickly. Fault-detection algorithm needs to take advantage of this feature of direct dependence set.

Indirect dependent set, as pointed out in Section 3.2, is more hazardous because they are allowed to proceed erroneously, which will further pollute the correctness of the rest of the system. Further more, time-out can not be applied to indirect dependent set.

Therefore, to improve the fault detection capability of the system, we need to modify coherent protocol to minimize indirect dependent set. This can be accomplished by requiring all coherent transactions to be *closed*, that is, the unit of all transactions must be a request-response pair. Ironically, one of the class of open transactions in DSM is also one of the most vulnerable, namely, the write-back transactions. Write-back transactions can be made to be closed transactions by asking the home node to respond to whoever write-backs the data with a corresponding response. Other transactions can be enforced similarly. Notice that here we have a tradeoff: we add more protocol complexity and traffic to the system in a hope to maximize fault-detection capability.

In addition, we can further improve the data integrity of the system by a two-phase commit protocol for any transactions that involve a dirty data transfer. When a node is about to transfer a dirty data, it stores the data in its internal buffer, and only squashes it upon receiving of the response indicating that the dirty copy sent out has been safely delivered to destination. The following table uses write-back as an example to summarize the result of this section:

Normal write-back protocol	-
: made closed	Fault-detection
: made closed + buffering	Fault-detection + data integrity

Table 4 Improve Fault-Detection and Data Integrity of Coherent Protocol

This optimization we call ***source buffering***, which should apply to any dirty data transfer from the cache to the memory. The target here is to make sure that this transaction is safe. However, source buffering introduces a new possibility of inconsistency, which may arise when the acknowledgement is lost. In this situation, some one else may have modified the data before the fault is reported. When recovery, if we simply overwrite the memory with the copy buffered at the source, we are updating the memory with an obsolete copy. This inconsistency is still there even though we have reduced the possibility by closing the transaction.

There are several possibilities to handle this new inconsistency. The foremost rigorous approach is to have the home node, upon receiving the write-back, change the state of the line into a holding state that disables the service to any further requests. The source, on the other hand, will issue another acknowledgement to the home when the acknowledgement from the home is received. This second acknowledgement will then release the line back to the idle state.

A more relaxed approach is to do more checking at the recovery time. When the recovery algorithm carries the data back to the home node, it checks the state of the line. If the state is dirty and the owner is the source, it updates the memory accordingly. Otherwise it makes the optimistic assumption that the data has safely made to the home. This is a reasonable assumption since the state of the line can not be changed unless the write-back has succeeded.

5.2 N+1 Stand-by DSM System

In this section we propose N+1 standby system as the ultimate HA solution for DSM system. In this system, an extra node is in standby idle mode and only takes over when a faulty node has be identified and excluded from the system. This node is a full replica of any other node of the system, substantiated with equal memory/cache capacity and processing power. It is optional whether this extra node has its own dedicated disk storage. However, it is crucial that it has the same robust channel to the entire IO subsystem.

5.2.1 Rational

To explain the motivation behind the proposal, we proceed to examine the cost/performance metrics:

- **Cost**
  In systems being shipped today, the processor and memory resources combined amounts up to 30~40% of the total cost. Therefore, adding an extra node without dedicated disk storage will add 5~6% cost increase for a four node system. This fraction is even smaller for systems having larger number of nodes and can be justified if the HA is well realized.

- **Performance**
  The primary advantage of a N+1 standby system is high coverage. Studies in the literature have shown that single component failure is the dominating case. While multiple fault flags may be raised in case of a fault, chances are only one faulty component is responsible. Furthermore, by using a larger recovery unit (a node) instead of a specific component additionally increases the coverage and reduces the engineering cost of repair.

  The second advantage is *no* performance degradation after a system has successfully completed the reconfiguration. Because at all time the system is delivering power of an N node configuration. Thus, we have accomplished the goal of graceful performance degradation to the maximum.

  Lastly, an N+1 standby system has the potential of fast fault-detection and reconfiguration speed. In a full N node system where fault-containment approach is employed, each of the N-1 remaining node must first abort their on-the-fly transactions, save their semantics before dropping into fault-detection and reconfiguration phase. Then they must run a distributed consensus algorithm to agree upon which is the faulty node. This phase is characterized by intense communication and is based on the premise that the interconnect is error free, or else they have to use a serial link to accomplish this task. On the contrary, the extra node in N+1 node has not engaged before a fault has occurred and thus does not have any on-the-fly transactions to save. Upon a notification of a fault signal, the extra node rapidly executes a self-testing algorithm which does not generate outbound messages, and then proceed to detect the error. After the fault has been identified, it reprograms the routers in the system so that messages formally direct to the faulty node now goes to itself, and recover the contents of the faulty node and the operation resumes.

We need to investigate more on fault-detection algorithm of N+1 system. When fault-detection algorithm is called into action, the system may already be frozen and network totally jammed. We make two assumptions to simplify future discussions on fault-detection algorithm:

1. The network is congested but not corrupted: data and messages can be said temporarily "*stored*" in the network.
2. There are emergency lanes in the network reserved for fault-detection algorithm and are deadlock free for messages of fault-detection algorithm

In this report we outline the algorithm of the recovery algorithm in the next section.

5.2.2 Recovery in N+1 System

Given that the standby node is a full replica, we only needs to concern about retrieving cached contents (in memory and cache) on the faulty node. We first classify the contents into three categories: the *idle*, the *exported* and the *imported*. The first two refer to lines that are homed at the faulty node, the difference being whether they are cached by other nodes or not. The *imported* contents are those cached by the faulty node in its processor caches and are homed elsewhere. The lines that are both homed and cached in the faulty node can be arbitrarily classified. Here we say they belong to the *imported* class because the most updated versions are in the processor caches if they are dirty.

We discuss recovery algorithm for these contents in turn:

- **Recovery of idle and exported contents**
  The recovery of these contents depends how much architectural support there is to access these lines. Each line *l* has two attributes: $S_{mem}(l)$ and $S_{dir}(l)$, for whatever stored in the memory and directory, respectively. One thing we assume here is the availability of page table information for these lines to the recovery algorithm. Page table can be made fault-tolerant in N+1 system by keeping two copies in two distinctive nodes. The table entry will be invalidated before the recovery starts, and will only be re-established afterwards, during this period any access to the page will be hold.

  In the case when these contents and the two attributes are not accessible directly from the memory and directory of the faulty node, the algorithm first reads the page from the disk. If the page table indicates that the page is clean, the page can be safely loaded into the memory of the standby node. If however the page is dirty, we have no ways of knowing whether the dirty lines have been replaced from the caches (and thus lost permanently) or not, therefore these pages are not recoverable.

Total recovery is impossible for dirty pages. To ensure data recovery without resorting to means such as check pointing, we need to have battery backed-up memory and directory, at least for the duration that the recovery of the data is in operation. We have the following two cases:

$S_{mem}(l)$* and *$S_{dir}(l)$ both available:
This is a trivial case, all memory lines are either directly transferred to the standby node, or recalls are generated on behalf of the faulty node.

$S_{mem}(l)$* available, *$S_{dir}(l)$ unavailable:
In this case we first consult the page information, if the page is clean then all lines are transferred directly to the standby node. Otherwise we load the page from disk and generate recalls using broadcasting, and then merge the recalled lines with the page.

Slight architectural supports can further improve the recovery of this case by limiting the expensive broadcasting. One support is to add a bit stored along with the memory, indicating whether the line is clean or dirty, and generate broadcasting recalls only for lines that are dirty. A further optimization that requires more storage is to store the *Log(P)* pointer of the dirty owner along with the bit. This pointer can eliminate the broadcasting for good.

- **Recovery of Imported Contents**
  The situation here is divided into two cases, depends on whether the processor caches are accessible to the recovery algorithm.

  When the caches are accessible, dirty lines are written back to their homes. On the other hand, if they are not accessible, the algorithm scans the memory of other node and marks the lines that are dirty *and* owned by the faulty node: any future accesses to these lines are trapped and induce abortion of the corresponding processes. Note that the second case is quite expensive and may be better executed in parallel by all the nodes.

The above outlines the major part of the recovery algorithm in N+1 system. More details should be carefully looked into for corner cases.

6. Conclusion

The following summarizes the main points raised in this report:

1. Proposed and discussed the goals of an HA DSM system, and the metric to measure the success

2. Formulated the dependence analysis of fault occurrence.
3. Discussed HA node architecture and its gracefully degraded form.
4. Discussed enhancement of coherent protocols to improve fault-detection capability of the system.
5. Proposed N+1 standby system as the ultimate HA DSM.

The following table summarizes possible future research opportunities:

Research Opportunities	*Merits*
A better and more practical definition of HA metrics	Maybe. Possible leads to solving the difficulty of quantifying researches in HA.
Continue investigate the dependent analysis of fault-occurrence, identify what is the majority fault model of different components, incorporate into the theorem	Theoretical interest mostly.
Seek practical use of the theorem	
Detailed design of HA node architecture.	Design and verification. Possible opportunity for quantified result.
Detailed understanding of degraded node architecture	Design and verification. Possible opportunity for quantified result.
More investigation on fault-detection algorithm in N+1 standby system. Including architectural supports and sensitivity of recovery triggering.	Design and verification
More careful study on recovery algorithm in N+1 system	Design and verification

Table 5 Future Research Opportunities on HA issues in DSM

One *big* topic has not been covered yet is hardware support for HA in the case of software fault, we will address this issue in the future.