Adaptive Granularity: Transparent Integration of Fine-Grain and Coarse Grain Communications

Daeyeon Park  
Computer Science Department  
University of Southern California  
Los Angeles, CA, 90089-0781  
daeyeon@cs.usc.edu

Abstract  

The granularity of sharing is one of the key components that affect the performance in distributed shared memory (DSM) systems. Providing only one or two fixed size granularities to the user may not result in an efficient use of resources. Providing an arbitrarily variable granularity increases hardware and/or software overheads. Moreover, its efficient implementation requires the user to provide some information on the granularity, sacrificing the programmability of the shared memory paradigm.

In this paper, we present a new communication scheme, called Adaptive Granularity (AG). Adaptive Granularity is a communication scheme that effectively and transparently integrates bulk transfer into the shared memory paradigm. Adaptive Granularity provides a limited number of granularities, but efficient enough to achieve gains from bulk transfer, without sacrificing any programmability of the shared memory paradigm and requiring any additional hardware. An Adaptive Granularity protocol consists of two protocols: one for fine-grain data and the other one for bulk data. For small size data, the standard hardware DSM protocol is used and the granularity is fixed to a cache line. For large array data, the protocol for bulk data is used and the granularity varies depending on the sharing behavior at runtime. Simulation results show that AG improves performance up to 43% over the hardware implementation of DSM (e.g., DASH). Compared with the equivalent architecture that supports fine-grain memory replication at the fixed granularity of a cache line (e.g., Typhoon), AG reduces execution time up to 35%.

1 Introduction

The shared memory paradigm provides a simple communication abstraction and thus greatly simplifies parallel programming, particularly problems that exhibits dynamic communication patterns or fine-grain sharing. Furthermore, most hardware distributed shared memory (DSM) machines [3, 6] achieve high performance by allowing the caching of shared writable data as well as read-only data and thus exploiting more locality. One disadvantage of that kind of
cache-coherent machines (e.g., Stanford DASH [6], MIT Alewife [3]) is that they are restricted to using only the fixed fine-granularity (i.e., a cache line for loads and stores) as a way of a communication. While this works well for fine-grain data, bulk transfer of data can sometimes be more effective for some applications. Bulk transfer has several advantages over fine-grain communications: fast pipelined data transfer, overlap of communication with computation, and replication of data in local memory [29].

To exploit the advantages of both fine grain and coarse grain communications, more recent shared memory machines such as Stanford FLASH and Wisconsin Typhoon have begun to integrate both models within a single architecture and to implement a coherence protocol in software rather than in hardware. In order to use the bulk transfer facility on the machine, several approaches such as explicit messages [12, 29] and a new programming model [13] have been proposed. With the explicit message approach, message passing communication primitives such as send-receive or memory-copy are used selectively to communicate coarse grain data and load-store communications are used for fine grain data communications in an application [29]. In other words, two communication paradigms coexist in the program and it is the user’s responsibility to select the appropriate model in each case. Another approach to exploit the advantages of bulk transfer is to use a new programming model. One example of this approach is a Hybrid protocol [13], in which programmer-supplied annotations are used to support the variable size granularity. The Hybrid protocol consists of a standard hardware DSM protocol that is used for fine grain data and a software (region-based) DSM protocol that is used for coarse grain data.

Though both approaches support an arbitrarily variable granularity and thus may potentially lead to large performance gains, they present several problems such as programmability and increased hardware complexity. As we can see in the above examples, the protocol that attempts to support arbitrarily variable grains either needs a new programming model (e.g., Hybrid protocol [13], object-based [4, 10] and region-based [22] software DSM protocols) or requires the user to use explicit message-passing paradigm. In other words, there is a tradeoff between the support of an arbitrarily variable granularity and programmability. The main reason for this tradeoff is that the support of an arbitrarily variable granularity makes its efficient implementation difficult without some information on the granularity from the user.

In this paper, we present a new communication scheme, called Adaptive Granularity (AG) and evaluate its performance in the context of a hardware- software DSM. Adaptive Granularity is a communication scheme that effectively and transparently integrates bulk transfer into the shared memory paradigm through a variable granularity and memory replication. Adaptive Granularity solves the tradeoff problem (programmability and the support of an arbitrarily variable granularity) by providing a limited number of granularities (i.e., $2^n$ cache lines up to a page), but efficient enough to achieve gains from bulk transfer, without sacrificing any programmability of the shared memory paradigm and requiring any additional hardware. For memory replication, AG assigns part of each node’s local memory for replicating remote data. In effect, AG uses this local memory as a large fully associative data cache, which eliminates much of the network traffic caused by conflict and capacity misses in smaller hardware caches.

---

1. A hardware-software DSM (HS-DSM) refers to the hardware DSM that implements coherence protocol on the separate node controller in software using handlers (e.g., Stanford FLASH [11], Wisconsin Typhoon [25]).
An Adaptive Granularity protocol consists of two protocols: one for fine-grain data and the other one for bulk data. For scalar data and array data whose size is less than some threshold, the standard hardware DSM protocol is used and the granularity is fixed to a cache line. For large array data, the protocol for bulk data is used and the granularity varies depending on the sharing behavior at runtime. When a command (e.g., load or store) is given to the node controller, it determines the type of communication (bulk transfer or standard load-store communication) depending on the data type² and sends a request to the home node without specifying any size even for bulk data request. When the home node receives a bulk request, it determines the granularity depending on the sharing pattern and sends back the data to the requesting node. To reduce false sharing, the home node splits a memory block into two when ownership change occurs. To exploit more spatial locality, two adjacent blocks are merged back into a larger block when both blocks are referenced by a single same node. When the data arrives, the node controller puts only the requested cache line onto the cache and the rest of the bulk data goes to the local memory.

In summary, for coarse-grain sharing, AG exploits the advantages of bulk transfer by supporting a spectrum of granularities. For fine-grain sharing, it exploits the advantages of the standard load-store communication by using cache line transfers. Simulation results show that AG changes most remote requests into local requests which in turn reduces the amount of network traffic significantly and improves the performance up to 43% over the hardware implementation of DSM (e.g., DASH). Compared with the equivalent architecture that supports fine-grain memory replication at the fixed granularity of a cache line (e.g., Typhoon), AG reduces execution time up to 35%.

The rest of this paper is organized as follows. Section 2 presents communication alternatives and the granularity for representative parallel systems and discusses the problems of the protocols that maintain coherence at the fixed or an arbitrarily variable granularity, mainly focusing on integrated DSMs. Section 3 describes AG and presents its memory replication mechanisms and protocols. Section 4 describes our simulation methodology to evaluate AG and Section 5 presents the experimental results and discusses the performance. Section 6 discusses the effect of architectural variations. Related work is given in Section 7 and Section 8 concludes with a summary.

2 Communication Alternatives and Granularity

A communication model can be presented to the user in different ways depending on the system by which it is supported. These alternatives have different performance, cost, and design characteristics. This section briefly reviews the communication alternatives and granularity for the message passing, hardware DSM, software DSM and integrated DSM models, and discusses the advantages and disadvantages of each approach. Table 1 summarizes the communication models and granularity for the different types of parallel machines considered here. As the table shows, most parallel machines support only one mode of communication except the integrated DSM.

²The type is saved on the node controller when page mapping is made to replicate the data.
### Table 1: Communication model and granularity for each parallel system

<table>
<thead>
<tr>
<th>Type</th>
<th>Connection</th>
<th>Example</th>
<th>Communication</th>
<th>Granularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP</td>
<td>LC</td>
<td>CM-5, Paragon</td>
<td>send-receive</td>
<td>-</td>
</tr>
<tr>
<td>HW-DSM</td>
<td>TC</td>
<td>DASH</td>
<td>load-store</td>
<td>cache line</td>
</tr>
<tr>
<td>SW-DSM</td>
<td>LC</td>
<td>IVY, Munin</td>
<td>load-store</td>
<td>page</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>send-receive</td>
<td>variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>memory-copy</td>
<td>variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>new program. model object</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>new program. model region</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>new program. model region</td>
<td></td>
</tr>
<tr>
<td>IT-DSM</td>
<td>TC</td>
<td>FLASH, Typhoon</td>
<td>load-store</td>
<td>cache line</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>send-receive</td>
<td>variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>memory-copy</td>
<td>variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>new program. model region</td>
<td></td>
</tr>
<tr>
<td>DSSMP</td>
<td>intra: TC</td>
<td>MGS</td>
<td>load-store</td>
<td>cache line, page</td>
</tr>
<tr>
<td></td>
<td>inter: LC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MP and IT-DSM denote message passing and integrated DSM, respectively; DSSMP refers to Distributed Scalable Shared memory Multiprocessors; LC and TC denote loosely-coupled, tightly coupled, respectively.

In this paper, a communication model refers to the high-level model visible to the user, not the low-level implementation, and the granularity refers to the unit of sharing. Load-store refers to the standard shared memory communication and a new programming model represents that programs should be written or annotated using the model that the system supports. A hardware DSM (HW-DSM) refers to the machine that implements a cache coherence protocol completely in hardware (e.g., Stanford DASH, MIT Alewife). A software DSM (SW-DSM) is a shared memory abstraction implemented on top of loosely coupled multicomputers such as workstations connected by a local area network (e.g., TreadsMarks [26], Midway [18]). The local node refers to the node that originates a given request and the home node is the node that contains the main memory and directory for the given physical address. A remote node is any other node.

### 2.1 Message Passing Machines and Hardware DSM

Message passing and hardware DSM machines are two well-known representatives of parallel machines and send-receive and load-store are usually used for each type, respectively. Although a message-passing machine usually provides more scalability than a hardware DSM, it has a disadvantage in programmability since the programmer must explicitly manage the communication for the data in an application. In contrast, a hardware DSM has an advantage of programmability and allows for efficient communications for fine grain sharing at the fixed granularity of a cache line. For coarse grain communications, however, the hardware DSM may not exploit system resources efficiently.
2.2 Software DSM

A software DSM provides a shared memory paradigm through a software coherence protocol and it can be divided into two categories, depending on whether the load-store or a new programming model is used for a communication. The software DSM that uses the load-store model (i.e., the standard shared memory programming model) typically maintains coherence at the granularity of physical memory pages (referred to as page-based). The fixed coarse granularity, however, can degrade performance severely in the presence of fine grain sharing, though some mechanisms such as lazy release consistency [16] can alleviate the false sharing problem to some extent. IVY [14] and Munin [19] fall into this category.

To alleviate the mismatch of the page-based software DSM, several new programming models such as an object-based and a region-based software DSM, in which a variable granularity is used in the protocol, have been proposed [4, 18, 22]. In an object-based approach a granularity is associated to each data objects [4, 10], while in a region-based approach it is allowed to be defined at the level of arbitrary programmer-defined regions. In summary, a software DSM with the fixed granularity (page-based) has the advantage of ease of programming, but the disadvantage of a poor match with fine grain sharing, while a software DSM with a variable granularity (object-based and region-based) exhibits the opposite characteristics.

2.3 Integrated DSM

For integrated DSM machines (IT-DSM) that support a bulk transfer facility in a cache-coherent shared address space, a programmer is presented with three mechanisms for communicating data: (i) standard load-store, (ii) explicit messages, and (iii) a new programming model with user annotations. As Table 1 shows, an integrated DSM with explicit messages provides several communication models so that the user can select the appropriate model for a communication: load-store, send-receive, and memory-copy model. Load-store and send-receive (or memory-copy) are usually used for fine-grain and coarse-grain communications, respectively. To explain the not-well-known memory copy model, a node sends bulk data from source address directly into destination address, rather than by performing a send to a processor that requires a matching receive [29]. When send-receive or memory-copy is used as the means of a communication for bulk transfer on cache-coherent shared memory machines, a granularity is variable and not necessarily cache-line aligned since the models usually support data transfer of any size.

Another approach to exploit the bulk transfer facility is to use a new programming model so that the user’s burden such as selecting an appropriate communication model is alleviated [13]. With this approach, a programmer provides information on the granularity and the compiler and runtime system use this information to select the best matching communication model. One example of this approach is a Hybrid protocol. With the Hybrid protocol, programmer-supplied annotations are used to identify shared data regions for which a specific granularity is used using software DSM techniques.

Although both approaches achieve the goal of providing a variable size granularity to the user, they present several problems. First, programs should be changed to effectively use the

---

3Note that they do not necessarily represent a new model, but a model with at least program annotations.
message passing paradigm [29] or a new programming model is needed for such integrated machines [13]. However, this kind of programming imposes a big burden on the user because the programmer should know both shared memory and message passing paradigms (or the new programming model), defeating the intended benefits of using shared memory machines. Second, because bulk transfer data can be cached, a coherence problem arise between the processors that cached the data. Implementing global coherence between arbitrarily sized bulk data and standard load-store data substantially increases the hardware complexity and/or software overhead [12]. Third, a data alignment problem arise because bulk transfer might include only a portion of a cache line rather than a full line. Supporting the data alignment increases hardware and software costs [12]. Fourth, there is a question that more performance gains can be achieved through the message passing paradigm on shared memory machines. Several researchers [5, 29] have shown that the bulk transfer using the message passing paradigm may not help in improving the performance of shared memory applications. The main reason for this performance result is due to the message passing overhead, even though this overhead is greatly reduced by allowing a user-level access.

### 2.4 Very Large Scalable DSM

To provide more scalability and offer better performance/cost, a new shared memory multiprocessor called DSSMP (Distributed Scalable Shared-memory Multiprocessors) has been proposed recently in MIT [17]. DSSMPs have two types of hierarchical networks: a tightly-coupled internal network that connects processors within each node of DSSMP, and a loosely-coupled external network that connects each node of DSSMP. DSSMP uses two different protocols (called MGS) for each network communication. For internal network communications, a hardware DSM protocol that provides a fixed size granularity of a cache line is used. For external network communications, a page-based software DSM protocol is used to amortize high cost of external communications. Because MGS provides transparently only two fixed size granularity (cache line and page), it has similar advantage (i.e., programmability) and disadvantage as in HW-DSM and page-based SW-DSM.

### 3 Adaptive Granularity

In this section we describe Adaptive Granularity in detail: overview, memory replication,
directory handling, and protocol. The granularity of sharing is one of the key components that affect performance in shared-memory machines. Theoretically, the granularity can have any value: word, cache line, page, complex data structure, etc. There are two main parameters in deciding the optimal granularity: false sharing and spatial locality. False sharing occurs when different processors access different location within the same granularity. This causes the data to bounce back and forth between the caches as if it were truly being shared. As the granularity increases, the probability of false sharing goes up while more spatial locality can be exploited. Since the program sharing can vary widely among programs and even in one program as the program executes, we may not obtain the best performance with the fixed granularity.

To solve this problem, Adaptive Granularity (AG) dynamically adjusts the granularity based on observed reference behavior as Figure 1 shows. When false sharing (i.e., by ownership change) is expected, the granularity is decreased at runtime. When more spatial locality is expected, the granularity is increased by merging two adjacent blocks. As the figure shows, the local node determines the transaction type (normal or bulk) and the home node determines the granularity.

AG has three main goals in exploiting both fine-grain and coarse-grain communications in one machine: programmability, high performance, and no additional hardware complexity. AG achieves the programmability by allowing the user to write programs using the standard shared-memory paradigm (even for bulk transfer). AG does not require the user to use message passing commands such as send and receive to exploit bulk transfer. AG achieves high performance through a different granularity depending on the data type and runtime condition. It results in exploiting the advantages of each model. AG provides these good features (high performance and programmability) without requiring any additional hardware for the AG.

3.1 Design Choices

When a variable size granularity is provided to exploit different communication patterns, several issues arise such as the destination of bulk data, and decision on the transaction type and granularity. We discuss the issues in this section.

3.1.1 Static or adaptive granularity

If the compiler could analyze the communication pattern accurately for a given application, static multi-granularity (i.e., determined at compile time) might be more efficient than the runtime decision because the runtime overhead could be reduced. However, the problem of predicting a good value of the granularity at compile time is inherently difficult due to several reasons. First, some critical information may not be available at compile time. Moreover this information usually depends on the input data. Second, the dynamic behavior of the program makes compile-time prediction difficult. Third, some aspects of parallel machines such as network contention, synchronization, cache coherency, and communication delays make it almost impossible for the compiler to predict the best value at compile time. Finally, the optimal value of the granularity tends to vary as the program executes, and this variance can be quite large on some parallel machines. Actually, the integrated DSM that support...
message-passing paradigm with global coherence passes this difficult task to the user. In AG, the determination on the granularity is delayed until run-time. At runtime, the granularity is decided on the home node based on the sharing pattern when the reference is made.

**Destination of bulk data** When bulk data is transferred, the final target can be a cache or local memory. Transferring bulk data into the cache can exploit more spatial locality without incurring a cache miss, but the bulk data may displace other useful data. Transferring it into a local memory can exploit the advantage of memory replication more easily, though subsequent data accesses are satisfied from the local memory instead of the cache. In AG, only the requested cache line is transferred to the cache and the rest of the bulk data goes to the local memory. This can be easily done since AG transfers bulk data as a sequence of cache lines.

**Determination on transaction type and granularity** Given that we have a variable size granularity and two types of transactions (e.g., standard read and bulk read), one of the issues is that who determines the transaction type and granularity: local or home node. For the transaction type, a local or home node can determine it if it is determined based on just a data type such as array and scalar. If the home node determines it, it can be implemented as follows. When memory is allocated on the home node, the flag is set in the directory to indicate a bulk type if the data size is greater than some threshold value. A local node sends a standard request even for bulk transfer because the node does not have the information. When the home node receives the normal request, it sends a bulk data assuming that the local node needs bulk data. With this approach, implementation is simple because only the home node contains the information. This approach, however, has a disadvantage that a local node cannot request fine-grain data even when it needs small amount of data.

To allow a local node to choose an appropriate transaction type, it must have the information on the data type. This scheme can be implemented as follows. When a local node first accesses a shared page on the remote node, a page fault occurs and appropriate actions are taken to map the page. At this time, a flag is set on the local node to indicate the bulk transaction if the home node says it is array data. When a command is given to the node controller, it checks the flag and sends a bulk request if the page is set as a bulk mode. If the local node does not want a bulk transfer (i.e., needs a small amount of data), it resets the flag and then sends a normal request to the home node. Because the local node can know better the transaction type this approach appears more reasonable. Thus we chose this scheme to give the user more flexibility.\(^4\)

For a granularity, we do not have such a choice as the transaction type. Even though the local node (i.e., consumer) can know better the amount of data needed for bulk transfer, it cannot know the sharing pattern on the home node. Thus, in AG, the home node determines a granularity. Another alternative is that the local node specifies just the maximum size needed and the home node determines the final granularity to send. To implement this scheme, the compiler gives the node controller the information on the size needed for bulk transfer (for

\(^4\)This approach has the same effect as the first one in our current implementation because the compiler does not give any information to the node controller.
Memory replication

search_dir(p,i)
{
    size = N; /* LINES_PER_PAGE; */
    d = 0;
    g = D[(p,d)].g; /* gran size */
    while (g!= size) {
        size = size / 2;
        if (i >= d+size) {
            d += size;
            g = D[(p,d)].g;
        }
    }
    return d;
}

(a) Memory replication

Algorithm for directory search

Figure 2: Memory replication and Directory search

example, before beginning of the loop) and the node controller uses it to request a bulk transfer.

3.2 Variable Grain Memory Replication

Memory replication is one of the techniques to tolerate long remote memory latency. The main purpose of memory replication is to change a remote access caused by conflict or capacity misses into a local access by mapping the remote virtual address to a local physical page and by copying remote data into the local memory so that the subsequent accesses are satisfied locally. To support memory replication with a variable granularity (i.e., $2^n$ cache lines up to a page), AG maps a virtual address of shared data to a local physical memory at the page granularity, but maintains coherence at a cache line level. To maintain consistency between the local cache and local memory, the node controller allocates a data structure called NCPT (Node Controller Page Table) for line access rights. NCPT has one entry for each page of physical memory that is mapped to the local node on behalf of the remote home node. It contains information about how the page is mapped. Each page table entry specifies the home node and includes various fields to enforce the access rights and to indicate how the transaction is requested to the home node (e.g., bulk read or normal read). When a cache line is displaced out of the cache and cached in again, the table provides the access right.

Figure 2,(a) illustrates how one virtual page is replicated on two different nodes with a different physical mapping on each node and various granularities. In this example, a granularity of two cache lines is replicated on the two nodes with the access right of READ_ONLY and a granularity of one cache line is replicated on one node with the access READ_WRITE. When a node first accesses a shared page on a remote node, the operating system maps the virtual address to a local physical page and gives information to the node controller such as a home node ID and page type (bulk or non-bulk data). This information has been stored on the page table when the memory was allocated on the home node. When the node controller receives a
<table>
<thead>
<tr>
<th>Local, Remote Node ⇒ Home Node</th>
<th>Home Node ⇒ Local, Remote Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>RREQ</td>
<td>Read Data Request</td>
</tr>
<tr>
<td>WREQ</td>
<td>Write Data Request</td>
</tr>
<tr>
<td>ACK</td>
<td>Acknowledge Invalidate</td>
</tr>
<tr>
<td>WACK</td>
<td>Ack. Write Inv. and Return Data</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Local Cache ⇒ Local Node Controller</th>
<th>Remote Node Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPR</td>
<td>Replaced Read Data</td>
</tr>
<tr>
<td>REPW</td>
<td>Replaced Write Data</td>
</tr>
</tbody>
</table>

Table 2: Message types used to communicate between local, home, and remote node

transaction on the page mapping from the operating system, it stores the home node ID and page type onto NCPT and initializes the line access tags to Invalid. When an access violates the access right of the cache line, the appropriate action is taken. (Access violation will be discussed in Section 3.4.) To distinguish bulk data from non-bulk data, we disallow both data from residing on the same virtual page.

### 3.3 Directory and NCPT handling

Given that we have a variable granularity ($2^n$ cache lines) and a fine grain memory replication, one of the issues related to the overhead of AG is how to handle the directory and NCPT efficiently. Because in one transaction several entries can be implied for both tables (i.e., 4 cache lines for the granularity of 4), we have to make either an entry for each minimum block, or a representative entry that covers the entire range of the blocks. There is a tradeoff between the two approaches: under the first approach search can be performed more efficiently with more overheads of making the entries. With the second approach (representative entry), the overhead of making the entries is small but search takes more time because the handler should find out the representative entry first. For NCPT, we adopted the first scheme for easy search because NCPT seems to be referenced frequently. Thus, NCPT[$(p,i)$] contains the access right information for the $i$th block of page $p$. For the directory, however, efficient updates such as split and merge are more important than search and the directory is not frequently accessed because many of cache misses are satisfied from the local node in AG. Thus we adopted the second scheme of making a representative entry for the directory. The representative entry for a memory block $(p,i)$ is the first directory covering the grain size of the block. The algorithm for searching the representative directory is shown in 2.(b).

### 3.4 Adaptive Granularity Protocol

An AG protocol consists of a hardware DSM protocol (HDP) and a bulk data protocol (BDP). For small size data in which reference behavior is always fine-grain, the standard invalidation-based hardware DSM protocol is used and the granularity is fixed to a cache line. For large size data in which the expected reference behavior is difficult to predict, BDP is used
<table>
<thead>
<tr>
<th>Arc</th>
<th>Msg-Type</th>
<th>Condition</th>
<th>Action</th>
<th>Out Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Cache ⇒ Node Controller</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>LOAD</td>
<td></td>
<td>determine type</td>
<td>RREQ ⇒ hn</td>
</tr>
<tr>
<td>2</td>
<td>STORE</td>
<td></td>
<td>determine type</td>
<td>WREQ ⇒ hn</td>
</tr>
<tr>
<td>3,4</td>
<td>LOAD,STORE</td>
<td></td>
<td>NCPT[(p,i)].f = YES;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>LOAD</td>
<td></td>
<td>NCPT[(p,i)].f = YES;</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>STORE</td>
<td></td>
<td>determine type</td>
<td>WREQ ⇒ hn</td>
</tr>
<tr>
<td>7</td>
<td>LOAD,STORE</td>
<td></td>
<td>NCPT[(p,i)].f = YES;</td>
<td></td>
</tr>
<tr>
<td>8,9</td>
<td>REPR,REPW</td>
<td></td>
<td>NCPT[(p,i)].f = NO;</td>
<td></td>
</tr>
<tr>
<td>Home Node ⇒ Local, Remote Node</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,11</td>
<td>RDATA</td>
<td>type == BULK</td>
<td>NCPT[(p,i)].f = YES;</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>RDATAM</td>
<td></td>
<td>NCPT[(p,s-e)].s = READ_TRANSIT;</td>
<td></td>
</tr>
<tr>
<td>13,14</td>
<td>WDATA</td>
<td>type == BULK</td>
<td>NCPT[(p,i)].f = YES;</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>WDATAM</td>
<td></td>
<td>NCPT[(p,s-e)].s = WRITE_TRANSIT;</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>INV</td>
<td></td>
<td>NCPT[(p,s-e)].s = INVALID;</td>
<td>ACK ⇒ hn</td>
</tr>
<tr>
<td>17</td>
<td>INV</td>
<td></td>
<td>NCPT[(p,s-e)].s = INVALID;</td>
<td>WACK ⇒ hn</td>
</tr>
<tr>
<td>18,19</td>
<td>INV</td>
<td></td>
<td>ACK ⇒ hn</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>BUSY</td>
<td></td>
<td>determine type</td>
<td>RREQ ⇒ hn</td>
</tr>
<tr>
<td>21</td>
<td>BUSY</td>
<td></td>
<td>determine type</td>
<td>WREQ ⇒ hn</td>
</tr>
<tr>
<td>22</td>
<td>WSHIP</td>
<td></td>
<td>NCPT[(p,s-e)].s = READ_WRITE;</td>
<td></td>
</tr>
<tr>
<td>Local, Remote Node ⇒ Home Node</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>RREQ</td>
<td>D[(p,0)].s == EMPTY</td>
<td>D[(p,0)].g = N; D[(p,0)].c = 1;</td>
<td>RDATA ⇒ ln</td>
</tr>
<tr>
<td>32</td>
<td>WREQ</td>
<td>D[(p,0)].s == EMPTY</td>
<td>D[(p,0)].g = N; D[(p,0)].c = 1;</td>
<td>WDATA ⇒ ln</td>
</tr>
<tr>
<td>33</td>
<td>RREQ</td>
<td>d = search_dir(i); D[(p,d)].c++;</td>
<td>RDATA ⇒ ln</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>WREQ</td>
<td>d = search_dir(i); (d1,d2) = split_dir(d);</td>
<td>INV ⇒ D[(p,d1)].m</td>
<td></td>
</tr>
<tr>
<td>35,36</td>
<td>WREQ,RREQ</td>
<td>d = search_dir(i); (d1,d2) = split_dir(d);</td>
<td>INV ⇒ D[(p,d1)].m</td>
<td></td>
</tr>
<tr>
<td>37,38</td>
<td>RREQ,WREQ</td>
<td>d = search_dir(i);</td>
<td>BUSY ⇒ ln</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>WREQ</td>
<td>D[(p,d)].c == 1;</td>
<td>d = search_dir(i);</td>
<td>WSHIP ⇒ ln</td>
</tr>
<tr>
<td>40</td>
<td>ACK,WACK</td>
<td>D[(p,i)].a == 1</td>
<td>D[(p,i)].a--; merge_dir()?;</td>
<td>WDATA ⇒ D[(p,i)].m</td>
</tr>
<tr>
<td>41</td>
<td>WACK</td>
<td>D[(p,i)].a == 1</td>
<td>D[(p,i)].a--; merge_dir()?;</td>
<td>(WDATA M,g) ⇒ D[(p,i)].m</td>
</tr>
<tr>
<td>42</td>
<td>ACK</td>
<td>D[(p,i)].a &gt; 1</td>
<td>D[(p,i)].a--;</td>
<td>(WDATA M,g) ⇒ D[(p,i)].m</td>
</tr>
</tbody>
</table>

Table 3: State transition table for the Adaptive Granularity protocol. <message> ⇒ <pid> represents that <message> is sent to <pid>. s–e denotes the range from starting to ending block for the bulk transfer. “ln” and “hn” denote local node and home node, respectively. NCPT[.].s: state; NCPT[.].f: cache present bit; N: number of cache lines per page; D[.].s: state; D[.].g: grain-size; D[.].c: count; D[.].m: member; D[.].a: account;
and the granularity varies depending on the sharing behavior at runtime. BDP is similar to HDP in a structure such as the number of states except that BDP is implemented using the buddy system [20] to efficiently support a variable granularity. The buddy system was adopted originally in a memory management algorithm to speed up the merging of adjacent holes when a memory block is returned by exploiting the fact that computers use binary numbers for addressing.

Table 2 shows the message types used for AG to communicate between local, home, and remote nodes. Most of them are similar to those of the standard hardware DSM protocol except that several extra message types (e.g., RDATAM) are introduced to support bulk transfer. Figure 3 shows the state transition diagrams on the local (and remote) node and home node assuming the sequential memory consistency, mainly focusing on BDP. Table 3 is a specification for the AG protocol (mainly focusing on BDP) and describes the actions to be taken and out messages for the transition arcs in Figure 3. Most of the message types and states of HDP have a corresponding message type or state on BDP. The message types and state are distinguished from their equivalent message types and states on HDP by the prefix BK. For example, RREQ and BK_RREQ are used for a read data request for HDP and BDP, respectively. In Table 2 and 3 and Figure 3, the message types and state are not distinguished to simplify the figure and tables. Messages for bulk transfer (except for the memory request messages such as BK_RREQ, for which the local node cannot know the range of bulk transfer) contain the starting address and size so that the handlers can deal with them efficiently.

**Local Node** A local node maintains consistency on NCPT and a state transition diagram for the local node is shown in Figure 3.(a). After the mapping of a virtual page to a local physical memory is made (discussed in Section 3.1), when an application attempts to access the block, the line-access-fault handler is invoked because all line access tags for the page were initialized
to *Invalid*. The handler retrieves the home node’s ID from the local table and sends a request (bulk or non-bulk depending on the page type) to the home node. This situation is represented by arcs 1 and 2 in Figure 3.(a). At the home node, the message invokes the handler that performs the appropriate coherence actions and returns the data (sends the requested memory block first and then the rest of them as a sequence of a cache line to utilize the memory and network bandwidth if the request was bulk transfer). When the requested block arrives from the home node, the message handler writes the data into the cache and changes the block’s access state. If the arrival data is the requested block (e.g., BK\_RDATA, not BK\_RDATAM), the handler also changes the state for the rest of the bulk data to a *transit* state so that a request to the transit data is suspended until it arrives without sending a request to the home node. This situation is represented by arcs 10, 11, 13 and 14 in Figure 3.(a). When the rest of the bulk data (e.g., BK\_RDATAM) arrives (arcs 12 and 15 in Figure 3.(a)), the message handler writes the data into local memory and changes the line’s access state.

**Home Node** When the home node receives a bulk request from the local node, it takes similar actions to the standard hardware DSM protocol, as shown in Figure 3.(b). The main difference between HDP and BDP is in the directory handling. For BDP there is one more field (grain size) in the directory to support the variable granularity. Initially, the grain size of a memory block is $N$ (the number of cache lines per page) and the representative directory for any address of the page is $D[(p,0)]$, regardless of the requested memory block.

When the home node receives BK\_RREQ or BK\_WRITE with the address $(p,i)$, the directory handler on the home node first searches the representative directory from $D[(p,0)]$, as shown in Figure 2.(b). With this scheme, the search can cost up to $\log_2 N$, but directory operations such as split and merge are efficient because only two representative directories are accessed to split or merge. Furthermore, only two message types (BK\_READ and BK\_WRITE) are needed to search the representative directory. For the other messages such as acknowledgment, the address for the representative directory is given with the size and thus the search is not needed.

After finding out the representative directory for the address, the handler takes some actions to send bulk data. Initially, the state for $D[(p,0)]$ is *EMPTY* and thus the handler sends to the local node a whole page (a requested block is sent first and then the rest of the page is sent as a sequence of a cache line), changes the state and sets the grain size field to $N$ (number of cache lines per page). This situation is represented by arcs 31 and 32 in Figure 3.(b). When a bulk request requires some ownership change (e.g., READ\_ONLY $\Rightarrow$ READ\_WRITE), the memory block is split into two sub-blocks (called *buddies*) with the same size if the granularity for the block is greater than 1. The handler sends BK\_INV to the remote nodes to invalidate the half of the original block containing the requested offset within a block and updates the representative directories for the two sub-blocks. The other half of the original block is untouched so that the remote nodes continue to access the sub-block. This situation is illustrated by arcs 34, 35, and 36 in Figure 3.(b).

When all acknowledgments for the invalidation requests arrive from the remote nodes (arcs 40 and 41 in Figure 3.(b)), the handler sends the sub-block to the local node. At this time, if the grain size for the block is 1 (minimum granularity), the handler checks the directories of
the two adjacent blocks to see whether merges are possible. The handler starts and continues
to merge the two blocks as long as both blocks have the same state and only a single same node
as a member for the blocks. As an example assuming a page size of 8 cache lines, in order to
migrate a page from one node to another, the page is split three times (into 4, 2, and 1 blocks,
respectively) and merged back into the original page when the last split is made. In summary,
a memory block is split into two sub-blocks when a bulk data request arrives with an ownership
change and the directories are checked for a merge at the time the ownership change completes
with granularity 1.

4 Simulation Methodology

This section presents the simulation framework used to study the performance of Adaptive
Granularity. In Section 4.1, we present the architectural assumptions and simulation environ-
ment, and in Section 4.2 we describe the benchmark programs.

4.1 Architectural Assumptions and Simulation Environment

We simulate a system of 16 processing nodes, which are interconnected by a bi-directional
mesh network, as shown in Figure 4. Each node consists of a processor, a cache, a portion of the
shared memory, and a node controller. The node controller contains a programmable processor
which processes all messages from the processor and network using corresponding handlers with
active messages [1]. The pages are allocated to the memory modules in a round-robin way. The
physical memory is equally distributed among the nodes and a sequential memory consistency
model [9] is enforced. Cache coherence for shared memory is maintained using a distributed
directory-based protocol. It is similar to Stanford FLASH and Wisconsin Typhoon except that
the bulk transfer facility for a message passing paradigm is not needed for AG.

Table 4 summarizes default machine parameters and values used in our simulations. With
these values, the machine has the following read latencies without contention: 1 cycle for a

Figure 4: NUMA multiprocessor

the two adjacent blocks to see whether merges are possible. The handler starts and continues
to merge the two blocks as long as both blocks have the same state and only a single same node
as a member for the blocks. As an example assuming a page size of 8 cache lines, in order to
migrate a page from one node to another, the page is split three times (into 4, 2, and 1 blocks,
respectively) and merged back into the original page when the last split is made. In summary,
a memory block is split into two sub-blocks when a bulk data request arrives with an ownership
change and the directories are checked for a merge at the time the ownership change completes
with granularity 1.

4 Simulation Methodology

This section presents the simulation framework used to study the performance of Adaptive
Granularity. In Section 4.1, we present the architectural assumptions and simulation environ-
ment, and in Section 4.2 we describe the benchmark programs.

4.1 Architectural Assumptions and Simulation Environment

We simulate a system of 16 processing nodes, which are interconnected by a bi-directional
mesh network, as shown in Figure 4. Each node consists of a processor, a cache, a portion of the
shared memory, and a node controller. The node controller contains a programmable processor
which processes all messages from the processor and network using corresponding handlers with
active messages [1]. The pages are allocated to the memory modules in a round-robin way. The
physical memory is equally distributed among the nodes and a sequential memory consistency
model [9] is enforced. Cache coherence for shared memory is maintained using a distributed
directory-based protocol. It is similar to Stanford FLASH and Wisconsin Typhoon except that
the bulk transfer facility for a message passing paradigm is not needed for AG.

Table 4 summarizes default machine parameters and values used in our simulations. With
these values, the machine has the following read latencies without contention: 1 cycle for a
Common

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory consistency</td>
<td>sequential</td>
</tr>
<tr>
<td>Memory access latency</td>
<td>35 cycles</td>
</tr>
<tr>
<td>CPU cache</td>
<td>direct-map</td>
</tr>
<tr>
<td>Cache line size</td>
<td>16 bytes</td>
</tr>
<tr>
<td>Cache size</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>Page size</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>TLB miss</td>
<td>25 cycles</td>
</tr>
<tr>
<td>Switch delay</td>
<td>2 cycles</td>
</tr>
<tr>
<td>Wire delay</td>
<td>2 cycles</td>
</tr>
<tr>
<td>Channel width</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

FG Only (Handler Overhead)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory request</td>
<td>14 cycles</td>
</tr>
<tr>
<td>Memory reply</td>
<td>30 cycles</td>
</tr>
<tr>
<td>Data arrival</td>
<td>20 cycles</td>
</tr>
</tbody>
</table>

AG Only (Handler Overhead)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory request</td>
<td>14 cycles</td>
</tr>
<tr>
<td>Memory reply</td>
<td>40 cycles</td>
</tr>
<tr>
<td>Data arrival</td>
<td>20 cycles</td>
</tr>
</tbody>
</table>

Table 4: Simulation parameters

Program | Description                                      | Input Data Set  |
---------|--------------------------------------------------|-----------------|
Cholesky | Cholesky factorization of sparse matrices       | bcsstk14        |
FFT      | Fast Fourier transformation                      | 64 K complex points |
LU       | Blocked dense LU factorization                   | 260x260         |
Radix    | Integer radix sort                              | 256K keys, radix=1024 |

Table 5: Application Characteristics

cache hit, an average of 120 cycles for a 2-hop global access and average of 140 cycles for a 3-hop global memory access. In Section 5, we will consider various architectural variations such as longer line size, larger cache size, 64 processors, and increased network latency.

The machine is simulated using an execution-driven simulator called Trojan [27]. Trojan is an extended version of MIT Proteus [21] and supports a virtual memory simulation. It also allows both process-based (e.g., SPLASH [23]) and thread-based (e.g., SPLASH2 [24]) applications to be simulated without requiring the program to be modified and provides a detailed model of the various hardware components. All hardware contentions in the machine are simulated, including the network. For the network, the model proposed by Agarwal [2] is used. Instruction references are assumed to take one cycle and virtual memory is enabled in all simulations.

4.2 Benchmark Applications

In order to compare the performance of Adaptive Granularity with other approaches, we use four scientific applications that have different communication patterns. The applications we study is shown in Table 5. Cholesky is taken from the SPLASH [23] suite and the rest of them (FFT, LU and Radix) are taken from the SPLASH2 benchmarks [24].

5 Experimental Results and Discussion
Figure 5: Simulation results for default machine parameters

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Type</th>
<th>Overall Miss-Ratio(%)</th>
<th>Local (R,W)(%)</th>
<th>Remote (R,W)(%)</th>
<th>Private RW(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cholesky</td>
<td>HW</td>
<td>9.6</td>
<td>4.2 (3.3, 0.9)</td>
<td>85.0 (71.1, 13.9)</td>
<td>10.8</td>
</tr>
<tr>
<td></td>
<td>FG</td>
<td>9.6</td>
<td>74.8 (73.1, 1.7)</td>
<td>15.2 (7.9, 7.3)</td>
<td>10.0</td>
</tr>
<tr>
<td></td>
<td>AG</td>
<td>9.6</td>
<td>76.3 (73.3, 3.0)</td>
<td>16.2 (8.2, 8.0)</td>
<td>7.5</td>
</tr>
<tr>
<td>FFT</td>
<td>HW</td>
<td>18.6</td>
<td>5.1 (3.1, 2.0)</td>
<td>82.2 (49.3, 32.9)</td>
<td>12.7</td>
</tr>
<tr>
<td></td>
<td>FG</td>
<td>18.6</td>
<td>51.8 (28.7, 23.1)</td>
<td>35.5 (23.6, 11.9)</td>
<td>12.7</td>
</tr>
<tr>
<td></td>
<td>AG</td>
<td>18.6</td>
<td>85.5 (51.2, 34.3)</td>
<td>1.8 (1.2, 0.6)</td>
<td>12.7</td>
</tr>
<tr>
<td>LU</td>
<td>HW</td>
<td>11.6</td>
<td>1.1 (1.1, 0.0)</td>
<td>96.7 (63.8, 32.9)</td>
<td>2.2</td>
</tr>
<tr>
<td></td>
<td>FG</td>
<td>11.6</td>
<td>90.8 (58.9, 31.9)</td>
<td>7.0 (6.0, 1.0)</td>
<td>2.2</td>
</tr>
<tr>
<td></td>
<td>AG</td>
<td>11.6</td>
<td>97.6 (64.7, 32.9)</td>
<td>0.2 (0.2, 0.0)</td>
<td>2.2</td>
</tr>
<tr>
<td>Radix</td>
<td>HW</td>
<td>8.8</td>
<td>3.5 (1.4, 2.1)</td>
<td>92.9 (41.5, 51.4)</td>
<td>3.6</td>
</tr>
<tr>
<td></td>
<td>FG</td>
<td>8.8</td>
<td>97.4 (35.4, 42.0)</td>
<td>18.6 (7.4, 11.2)</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td>AG</td>
<td>8.8</td>
<td>88.4 (38.9, 49.2)</td>
<td>7.9 (3.9, 4.0)</td>
<td>4.0</td>
</tr>
</tbody>
</table>

Table 6: Decomposition of cache misses. R: read, W: write
AG attempts to improve performance through the memory replication and bulk transfer. In order to quantify the performance gains from each scheme, we simulate three systems: a hardware DSM (HW), the hardware-software DSM that provides fine-grain replications with the fixed granularity of a cache line (FG), and the hardware-software DSM that supports Adaptive Granularity (AG). By comparing the performance of HW and FG (FG and AG), we can see the performance improvement (or degradation) that is achieved via memory replication (bulk transfer). Because we want to see the performance gains achieved mainly from the bulk transfer of AG (not from memory replication), we normalized the execution times of HW and AG to that of FG. Figure 5 shows the relative execution times of three types (HW, FG, AG). The execution time of an application is broken down as follows: (1) busy time spent executing instructions (busy), (2) read stall time, (3) write stall time, and (4) stall time due to synchronization (sync). The stall times for read and write are further decomposed into local, remote, and private. The local stall time represents the stall time satisfied from the local node for shared memory accesses. Thus, a cache miss that is satisfied from the replicated local memory is classified as a local cache miss. Table 6 shows the cache miss ratio for each benchmark and decomposes the cache misses into local, remote, and private misses. Note that each component (Local, Remote, and Private) does not represent a cache miss ratio, but the percentage over the total number of cache misses. Because FG and AG do not change the cache miss ratio itself directly (it can be changed slightly because of changes in cache mapping), we do not show it for each component in Table 6.

When we compare overall performance of FG with that of HW, FG outperforms HW for three applications (Cholesky, LU, Radix). For FFT, however, FG is slightly outperformed by HW, mainly due to the handler overhead. As the breakdows for the execution times and cache misses show, the gains from FG over HW are mainly due to the reduction in remote memory latency that arise from memory replications. They show that FG changes the cache misses caused by conflict or capacity misses to the local access in most cases by supporting fine-grain memory replication. However, FG has a limitation in reducing remote latency, especially the one caused by the migrating data type as shown in a FFT case. For FFT, though FG changes remote references from 82.2% to 35.5% in cache misses, the execution time is slightly increased for FG. This leads to the argument that supporting only fine grain memory replication is not enough in changing remote accesses to local accesses.

For the overall gains from AG over FG, Figure 5 shows that AG reduces the execution times of the programs over FG by 7.7%, 35.4%, 12.2%, and 11.5%, respectively. FFT showed the greatest improvement because it generates the most migrating memory references of the programs simulated. The gains are mainly due to the remote latency reduction that arises from bulk transfer. In order to migrate one page (N cache lines) from one node to another, FG communicates 3N times (INV, WACK, WDATA for each cache line) while AG performs the migration with only 3 log<sub>2</sub>N times communications. Compared with HW (i.e., normalized with HW), AG reduces the execution times of the programs by 18.1%, 35.1%, 42.7%, and 29.3%, respectively (this is not shown in Figure 5).

Another observation from the results in Figure 5 is that processor utilizations (i.e., the percentage of busy time over the total execution time) appears to be low: 19%, 16%, 26%, and 29%, respectively. The main reason for the low utilizations is that the applications exhibit high cache miss rates (see Table 6), mainly caused by a small cache (4K). Furthermore, the latency
due to write misses is not eliminated because sequential consistency is enforced. If we increase
the cache size, cache misses caused by conflict and capacity misses can be reduced, which thus
increases processor utilizations. However, the increased channel utilization does not necessarily
decrease the relative effectiveness of the Adaptive Granularity. The effects of the cache size on
the performance of AG is discussed in the next section.

6 Effect of Architectural Variations

In this section, we evaluate the impact of several architectural variations on the performance
of Adaptive Granularity. The variations we study include changes in the line size, cache size,
number of processors, and network latency.

Cache Size For our experiments so far, the cache size has been set to 4 Kbytes. In order
to see whether AG works well also for larger caches, we ran additional experiments with the
cache size set to 16 Kbytes. For larger cache size, we used the same size of data sets because
we want to see the performance changes with larger caches. The results of our experiments are
presented in Figure 6.(a). We see that for FFT, AG has a 39% shorter execution time with
larger caches than FG while it has a 35% shorter execution time with 4KB caches, which is
contrary to the intuition. The explanation for this difference is that the increase in cache size
may reduce conflict misses, but cannot reduce the true sharing misses. Because FG also provides
the memory replication, the effect of the reduced conflict misses is small. Thus increasing the
cache size cuts the execution time, which results in the increase of the component of remote
stall time (not the absolute value but the percentage of the stall time).

Line Size The next variation we study is to see how the effectiveness of AG is changed when
the line size is increased from 16 bytes to 64 bytes with a channel width fixed. Increasing
the line size has the advantage of providing better spatial locality and the disadvantages of
increasing false sharing and making remote accesses longer since a channel width is fixed. Thus
it is interesting to see how performance is affected by the line size change. Figure 6.(b) presents
the resulting execution times for FFT and LU. As the figure shows, the effectiveness of AG is
decreased. The main reason for this is that the component of remote-stall time is decreased
(from 58.4% to 48.7% for FFT and 13.3% to 8.6% for LU) due to the increase in busy-time
resulting from the increased spatial locality. This means that the benefits achieved by exploiting
more spatial locality more than offsets the overhead caused by false sharing, and therefore there
is less room for performance improvement by AG.

Number of Processors Next we study the performance effect on AG as the number of
processors increases from 16 to 64 processors. For this experiment, larger data sets are used.
For FFT, 256K complex points are used instead of 64K for 16 processors. For LU, 460 by
460 matrices are used instead of 260 by 260 for 16 processors. As the number of processors
increases, the variance of network latency is becoming large since the average hop count goes
up. In Figure 6.(c), we show the experimental results. We see that increasing the number
Figure 6: Effects of architectural variations
of processors reduces the effectiveness of AG for FFT and does not change it much for I.U. The main reason for the FFT case is that the cache miss ratio was increased a lot (from 19% to 41%) due to larger data sets with the fixed size. Because the increased cache misses are caused by cache conflicts (not true sharing misses), they are satisfied from the local node. As the figure shows, the component of local stall time is increased from 20.5% to 38.9% and AG cannot cover the increase. For I.U, the cache miss ratio was not changed much (from 11.6% to 11.9%) though a larger data set is used with the fixed cache size. Even with a little change in a cache miss ratio and longer network latency, the effectiveness of AG is reduced slightly with 64 processors. The explanation for this is that the data set size is not increased as much as the increase in the number of processors due to the simulation limitation of time and space. This resulted in the less allocation of work (and/or data) to each processor.

**Network Latency** The final variation we study is to see how the effectiveness of AG is changed by a variation in the network latency assuming the same number of processors. Until now, we have assumed that a switch and wire delay are each 2 cycles, which results in around 140 cycles for 3-hop remote memory access without contention. Figure 6.(d) shows the resulting execution time for FFT and I.U, assuming that the delay is doubled (i.e., 4 cycles each). With this assumption, it takes average 230 cycles for 3-hop remote memory access without contention. We observe in Figure 6.(d) that AG works better under a longer network latency. As a network latency increases as a result of increased wire (or switch) delays, the components of *remote-read* and *remote-write* are becoming larger and thus there is more room for improvement.

## 7 Related Work

Adaptive Granularity improves performance by reducing the memory latency through memory replication and variable granularity. Memory replication has been used differently depending on the system by which it is supported. Cache-only memory architectures (COMA) [7, 15] makes all the local memory into a cache and replicates all shared data into the cache. Software DSMs performs memory replication at the page granularity by the operating system, or at the granularity of user-defined objects or regions by the runtime system. Page-based memory replications, however, are poor-matched with fine grain applications and thus experience severe performance degradation. To solve this problem, Typhoon allows memory replication to be made at a much finer granularity (i.e., cache line) through a mechanism called Stache [25]. Stache consists of user-level handlers such as a page-fault handler, message handler, and line-access-fault handlers. Our particular implementation of memory replication borrows from that of the Typhoon system [25]. The main difference is that Typhoon uses tagged memory with the fixed cache line granularity while AG uses general memory with a variable granularity.

Several other approaches have been proposed to support multi granularities in the context of shared memory machines. Dubnicki and LeBlanc [8] describes a hardware cache coherent system that dynamically adjusts the cache block size based on the reference behavior. Cache blocks are split when false sharing occurs and merged back into a larger cache line when a single processor owns both of the sub-blocks. Compared to Adaptive Granularity, it cannot
not achieve a large granularity such as a page nor provide flexibility since a cache line is split and it is implemented in hardware. Galactica Net [28] and MGS [17] support two fixed size granularities of a cache line and a page. Both schemes select between cache line grain and page grain blocks, depending on shared memory reference behavior.

Another alternative approach for achieving benefits from bulk transfer is to sacrifice the programmability of a shared memory paradigm to some extent. With this approach, application annotations or explicit messages are used in an application to communicate bulk data. Hybrid protocol [13] and Shared Regions [22] allow coherence to occur at any granularity with user annotations to identify the regions in which a specific granularity is applied. With explicit message passing communication, a programmer is presented with two communication paradigms (load-store and message passing) that he has to select an appropriate model for a communication [29]. Even though application annotations and explicit messages may potentially exploit a variable granularity better than Adaptive Granularity does, they present several problems such as programmability and increased hardware complexity.

8 Conclusion

Several studies have shown that the performance in multiprocessor systems is sensitive to the granularity supported by the system and the sharing patterns exhibited by application programs. For fine-grain applications, a traditional hardware DSM provides an efficient communication through the fixed granularity of a cache line. For coarse-grain applications, however, this fine-grain communication can be less efficient than bulk transfer. The main advantage of bulk transfer is that we can reduce communication costs through fast data transfer and replications. A mismatch between the granularity and communication behavior can result in serious performance degradations.

In this paper we proposed a new communication scheme called Adaptive Granularity (AG) that achieves high performance without sacrificing the programmability of the shared memory paradigm, through the support of memory replication and use of different protocols depending on the data type. For small size data, a standard hardware DSM protocol is used so that fine grain communications are optimized through the fixed size granularity of a cache line. For large size array data in which it is difficult to find the best match between the granularity and sharing patterns, the protocol that dynamically adjusts the granularity according to the reference behavior is applied. A memory block is split when false sharing occurs, and merged back into a larger memory block when some conditions are met to exploit more spatial locality. Simulation results show that Adaptive Granularity eliminates a substantial fraction of remote cache misses through the memory replication and more spatial locality gained from bulk transfer. AG improves performance up to 35% over the equivalent system with the fixed size granularity and up to 43% over the hardware DSM.

Acknowledgments
We would like to thank Jacqueline Chame and Sungdo Moon for giving comments on a previous draft.

References


