Chapter 1

Introduction

Early multiprocessors were designed with two major architectural approaches. For small numbers of processors (typically 16 or fewer), the dominant architecture was a shared-memory architecture comprised of multiple processors interconnected via a shared bus to one or more main memory modules, as shown in Figure 1.1. These machines were called bus-based multiprocessors or symmetric multiprocessors (SMPs), since all processors are equidistant from each main memory module and the access time to the centralized main memory is the same regardless of which processor or which main memory module is involved. Bus-based, shared-memory multiprocessors remain the dominant architecture for small processor counts.

To scale to larger numbers of processors, designers distributed the memory throughout the machine and used a scalable interconnect to enable processor-memory pairs (called nodes) to communicate, as shown in Figure 1.2. The primary form of this distributed address space architecture was called a message-passing architecture, named for the method of inter-node communication. (In the 1980s, a small number of architectures with physically distributed memory but using a shared memory model were also developed. These early distributed shared-memory architectures are discussed in Section 1.1.)

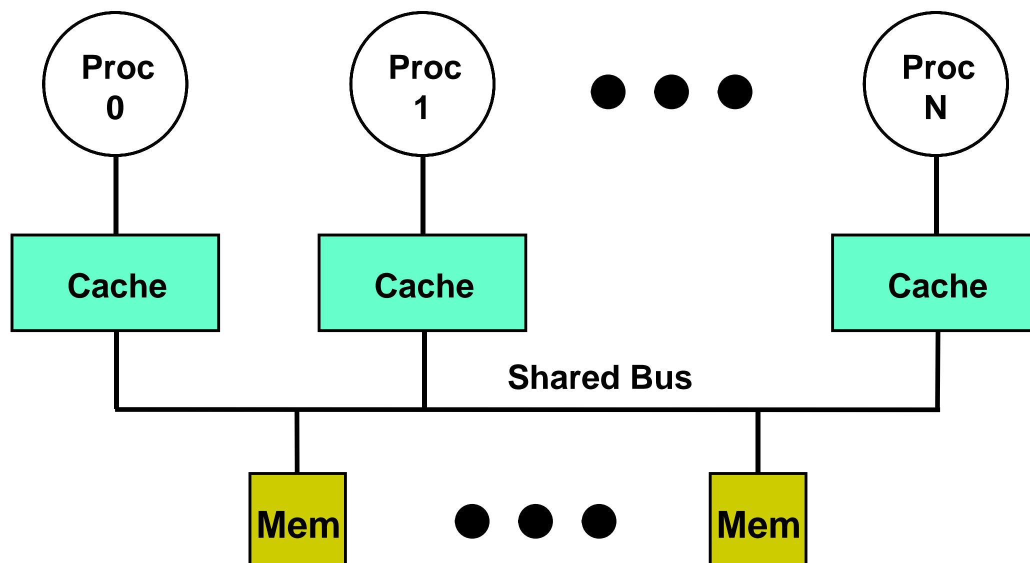

Figure 1.1. A small-scale, bus-based, shared-memory multiprocessor. This architectural configuration is also called a Symmetric Multiprocessor (SMP) or a Uniform Memory Access machine (UMA).

Each of these two primary architectural approaches offered advantages. The shared-memory architectures supported the traditional programming model, which viewed memory as a single, shared address space. The shared memory machines also had lower communication costs since the processors communicated directly through shared memory rather than through a software layer. On the other hand, the distributed address space architectures had scalability advantages, since such architectures did not suffer from the bandwidth limits of a single, shared bus or centralized memory. Despite these scalability advantages, the difference in programming model from the dominant small-scale, shared-memory multiprocessors severely limited the success of message-passing architectures, especially at small processor counts.

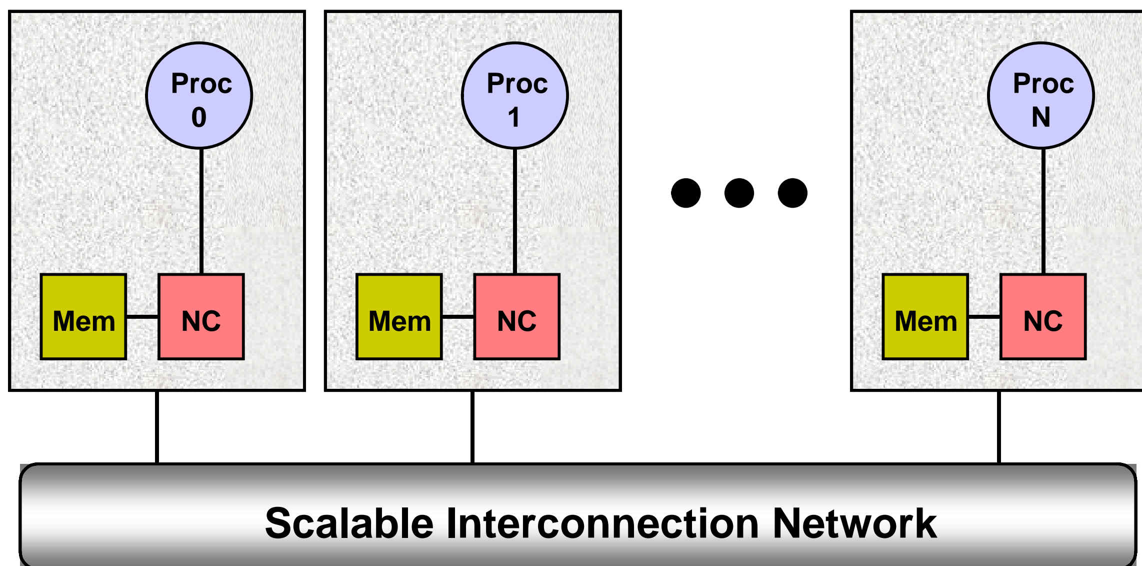

Figure 1.2. A distributed address space, or message-passing architecture. Each node has a node controller, NC, which handles communication via the network.

1.1 Distributed Shared Memory

Distributed Shared Memory (DSM) is an architectural approach designed to overcome the scaling limitations of symmetric shared-memory multiprocessors while retaining the traditional shared memory programming model. DSM machines achieve this goal by using a memory that is physically distributed among the nodes but logically implements a single shared address space. Like their bus-based predecessors, DSM architectures allow processors to communicate directly through main memory, and to freely share the contents

of any memory module in the system. DSM multiprocessors have the same basic organization as the machines in Figure 1.2.

The first DSM architectures appeared in the late 1970s and continued through the early 1980s, embodied in three machines: the Carnegie Mellon Cm* [54], the IBM RP3 [38], and the BBN Butterfly [5]. All these machines implemented a shared address space where the time to access a datum depended on the memory module in which that datum resided. Because of the resulting variability in memory access times, the name Non-Uniform Memory Access (NUMA) machines was also given to these architectures. Although the exact access time for a datum in a NUMA architecture depended on which memory module contained the datum, by far the largest difference in access time was between addresses in local memory and addresses in remote memory. Because these access times could differ by a factor of 10 or more and there were no simple mechanisms to hide these differences, it proved difficult to program these early distributed shared-memory machines.

In uniprocessors, the long access time to memory is largely hidden through the use of caches. Unfortunately, adapting caches to work in a multiprocessor environment is difficult. When used in a multiprocessor, caching introduces an additional problem: *cache coherence*, which arises when different processors cache and update values of the same memory location. An example of the cache coherence problem is shown in Figure 1.3. Introducing caches without solving the coherence problem does little to simplify the programming model, since the programmer or compiler must worry about the potentially inconsistent views of memory.

Time	Processor 0	Processor 1
0	x=0, y=0 (cached)	x=0, y=0 (cached)
1	x=1	y=1
2	y=y+1	x=x+1

Figure 1.3. The cache coherence problem. For simplicity assume that x and y are both initially 0 and cached by both processors. Without cache coherence the final values of x and y may be 1 or 2 or even worse, 1 in one cache and 2 in the other. A cache coherence protocol ensures that the final value of x and y in this case is 2.

Solving the coherence problem in hardware requires a *cache coherence protocol* that enforces the rules of the particular memory consistency model in use, and ensures that a processor will always see a legal value of a datum. There are two classes of cache coherence protocols: snoopy-based protocols [36] for small-scale, bus-based machines, and directory-based protocols [9][55] for more scalable machines. Chapter 2 details the cache coherence problem on both small-scale and large-scale shared-memory machines, and describes both snoopy-based protocols and the more scalable, and more complex, directory-based protocols.

In the late 1980s and early 1990s, the development of directory-based cache coherence protocols allowed the creation of cache-coherent distributed shared-memory multiprocessors, and the addition of processor caches to the original DSM architecture shown in Figure 1.2. The availability of cache coherence, and hence software compatibility with small-scale bus-based machines, popularized the commercial use of DSM machines for scalable multiprocessors. These DSM multiprocessors are also called Cache Coherent Non-Uniform Memory Access (CC-NUMA) machines, the latter characteristic arising from the use of distributed memory. All existing scalable cache coherence protocols rely on the use of distributed directories [2], but beyond that the protocols vary widely in how they deal with scalability, as well as what techniques they use to reduce remote memory latency.

1.2 Cache Coherence Protocol Design Space

Commercial CC-NUMA multiprocessors use variations on three major protocols: bit-vector/coarse-vector [20][30][60], SCI [7][14][32], and COMA [8]. In addition, a number of other protocols have been proposed for use in research machines [3][10][39][45][61]. The research protocols are for the most part similar, with an emphasis on changing the bit-vector directory organization to scale more gracefully to larger numbers of processors. From this list of research protocols, this dissertation adds the dynamic pointer allocation protocol to the set of commercial protocols, and quantitatively compares each of the protocols.

A cache coherence protocol can be evaluated on how well it deals with the following four issues:

Protocol memory efficiency: how much memory overhead does the protocol require? Memory usage is critical for scalability. This dissertation considers only protocols that have memory overhead that scales with the number of processors. To achieve efficient scaling of memory overhead, some protocols use hybrid solutions (such as a coarse-vector extension of a standard bit-vector protocol), while others keep sharing information in non-bit-vector data structures to reduce memory overhead (e.g., an SCI scheme). The result is that significant differences in memory overhead can still exist in scalable coherence protocols.

Direct protocol overhead: how much overhead do basic protocol operations require? This often relates to how directory information is stored and updated, as well as attempts to reduce global message traffic. Direct protocol overhead is the execution time for individual protocol operations, measured by the number of clock cycles needed per operation. This research splits the direct protocol overhead into two parts: the latency overhead and the occupancy overhead. In DSM architectures, the node controller contributes to the latency of each message it handles. More subtly, even after the controller sends the reply message it may continue with bookkeeping or state manipulations. This type of overhead does not affect the latency of the current message, but it may affect the latency of subsequent messages because it determines the rate at which the node controller can handle messages. This direct protocol overhead is controller occupancy, or the inverse of controller bandwidth. Keeping both latency and occupancy to a minimum are critical in high performance DSM machines [25].

Message efficiency: how well does the protocol perform as measured by the global traffic generated? Most protocol optimizations try to reduce message traffic, so this aspect is accounted for in message efficiency. The existing protocols vary widely in this dimension. For example, COMA tries to reduce global traffic by migrating cache lines, potentially reducing global message traffic and improving performance significantly. Other protocols sacrifice message efficiency (e.g., coarse-vector) to achieve memory scalability while maintaining protocol simplicity. Still others add traffic in the form of replacement hints (e.g., dynamic pointer allocation) to maintain precise sharing information.

Protocol scalability: Protocol scalability depends on both minimizing message traffic and on avoiding contention. In the latter area, some protocols (such as SCI) have explicit features to reduce contention and hot-spotting in the memory system.

The goal of this dissertation is to perform a fair, quantitative comparison of these four cache coherence protocols, and to achieve a better understanding of the conditions under which each protocol thrives and under which each protocol suffers. Through this comparison, this research demonstrates the utility of a programmable node controller that allows flexibility in the choice of cache coherence protocol. The results of this study can also be used to guide the construction of protocols for future, more robust, scalable multiprocessors.

1.3 Evaluating the Cache Coherence Protocols

The tradeoffs among these coherence protocols are extremely complex. No existing protocol is able to optimize its behavior in all four of the areas outlined above. Instead, a protocol focuses on some aspects, usually at the expense of others. While these tradeoffs and their qualitative effects are important, the bottom line remains how well a given protocol performs in practice. Determining this requires careful accounting of the actual overhead encountered in implementing each protocol. Although message traffic will also be crucial to performance, several protocols trade protocol complexity (and therefore an increase in direct protocol overhead) for a potential reduction in memory traffic. Understanding this tradeoff is critical.

Perhaps the most difficult aspect of such an evaluation is performing a fair comparison of the protocol implementations. Because most DSM machines fix the coherence protocol in hardware, comparing different DSM protocols means comparing performance across different machines. This is problematic because differences in machine architecture, design technology, or other artifacts can obfuscate the protocol comparison. Fortunately, the FLASH machine [28] being built at Stanford University provides a platform for undertaking such a study. FLASH uses a programmable protocol engine that allows the implementation of different protocols while using an identical main processor, cache, memory, and interconnect. This focuses the evaluation on the differences introduced by the proto-

cols themselves. Nonetheless, such a study does involve the non-trivial task of implementing and tuning each cache coherence protocol.

This research provides an implementation-based, quantitative analysis of the performance, scalability, and robustness of four scalable cache coherence protocols running on top of a single architecture, the Stanford FLASH multiprocessor. The four coherence protocols examined are bit-vector/coarse-vector, dynamic pointer allocation, SCI, and COMA. Each protocol is a complete and working implementation that runs on a real machine (FLASH). This is critical in a comparative performance evaluation since each protocol is known to be correct and to handle all deadlock avoidance cases, some of which can be quite subtle and easily overlooked in a paper-design or high-level protocol implementation.

1.4 Research Contributions

The primary contributions of this dissertation are:

- A framework for the comparative evaluation of cache coherence protocols, and a mechanism for carrying out that evaluation using the Stanford FLASH multiprocessor as the experimental vehicle.
- Efficient implementations of three full-fledged cache coherence protocols for the Stanford FLASH multiprocessor (bit-vector/coarse-vector, dynamic pointer allocation, and SCI). Each protocol has support for two memory consistency modes as well as support for cache-coherent I/O. The SCI implementation is particularly interesting, in that while it is based on an IEEE standard, the specific FLASH implementation presents new challenges, and implements many improvements that are not in the standard.
- The quantitative analysis of the performance, scalability, and robustness of four cache coherence protocols. This is the first study capable of performing an implementation-based evaluation, where the architecture, and the applications can be held constant, while changing only the cache coherence protocol that the machine runs. Insight into the scalability and robustness problems of the four protocols can guide the design of future protocols that may be able to avoid these shortcomings.
- A demonstration of the potential value of a programmable node controller in DSM systems. While there may never be a single cache coherence protocol that is optimal over a wide range of applications and machine sizes, a node controller that provides flexibility in the choice of cache coherence protocol may be the key to building robust, scalable architectures.

1.5 Organization of the Dissertation

This chapter began by describing the architectural history of multiprocessors and outlining the series of events that led to commercial DSM machines, most notably the development of directory-based cache coherence protocols. The design space of distributed shared-memory cache coherence protocols was presented next, along with a framework for analyzing coherence protocols. Section 1.3 described the problem of evaluating coherence protocols, and proposed a possible solution: the implementation of each protocol on the flexible Stanford FLASH multiprocessor, and their subsequent comparative evaluation. This quantitative evaluation is the focus of this dissertation.

Chapter 2 discusses the role of cache coherence protocols in shared-memory machines, and explains why the transition from bus-based snoopy coherence protocols to distributed directory-based protocols is necessary as the machine size scales. Each of the four DSM cache coherence protocols in this study are then introduced, with discussion of the directory organization, memory overhead, and high-level goals of each protocol.

Chapter 3 discusses the details of the Stanford FLASH multiprocessor architecture, particularly those architectural details that are exposed to the protocol designer. Implementing fully functional versions of four cache coherence protocols on a real machine is a challenge, and Chapter 3 discusses some of the issues in designing correct coherence protocols for FLASH.

With an understanding of the FLASH machine, Chapter 4 returns to each of the four cache coherence protocols and presents the particular FLASH implementation in detail. For each protocol, Chapter 4 details the protocol data structures, the layout and description of each field in the data structures, the network message types, the protocol dispatch conditions, and additional implementation considerations and resource usage issues. Example protocol handlers are presented for each protocol in Appendix A, highlighting some key aspects of that protocol. The chapter concludes with a table comparing each implementation in terms of handler counts, code size, and high-level protocol characteristics.

Chapter 5 describes the simulation methodology used in the experiments in this study. The results in this research come from a detailed simulation environment, and both the processor model and the memory system simulator are discussed in detail. Because both

the applications and the processor count affect the protocol comparison, this research employs a variety of different applications on machines ranging from 1 to 128 processors. Chapter 5 describes each application, and explains how each application is simulated in both tuned and un-tuned form. The application variety in this evaluation highlights different protocol characteristics, and evokes the relative benefits of some protocols.

Chapter 6 presents the results of comparing the performance, scalability, and robustness of these four protocols on machine sizes from 1 to 128 processors, using the applications described in Chapter 5. Chapter 6 begins by characterizing some of the basic performance metrics for these protocols (latency and occupancy for both local and remote accesses). Then detailed breakdowns of execution time are shown for each protocol and each application, complete with an analysis of the major issues for each application, including whether the most important characteristics are direct protocol overhead, message efficiency, or issues of inherent protocol scalability.

Chapter 7 summarizes the findings of this research and discusses both related work and future research possibilities.