Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-Layer Interconnects

Aske Plaat Henri E. Bal Rutger F. H. Hofman
aske@cs.vu.nl bal@cs.vu.nl rutger@cs.vu.nl
Department of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
http://www.cs.vu.nl/albatross/

Abstract

This paper studies application performance on systems with strongly non-uniform remote memory access. In current generation NUMAs the speed difference between the slowest and fastest link in an interconnect—the "NUMA gap"—is typically less than an order of magnitude, and many conventional parallel programs achieve good performance. We study how different NUMA gaps influence application performance, up to and including typical wide-area latencies and bandwidths. We find that for gaps larger than those of current generation NUMAs, performance suffers considerably (for applications that were designed for a uniform access interconnect). For many applications, however, performance can be greatly improved with comparatively simple changes: traffic over slow links can be reduced by making communication patterns hierarchical—like the interconnect. We find that in four out of our six applications the size of the gap can be increased by an order of magnitude or more without severely impacting speedup. We analyze why the improvements are needed, why they work so well, and how much non-uniformity they can mask.

1 Introduction

As computer systems increase in size, their interconnects become more hierarchical, resulting in growing bandwidth and latency differences in their interconnects. This trend is visible in NUMA machines and clusters of SMPs, where local memory access is typically a factor of 2–10 faster than remote accesses [17]. The gap in future large-scale NUMAs is larger, and the gap in meta-computers and computational grids is *much* larger.

For NUMAs with a small gap good performance has been reported with conventional numerical applications [17, 21]. On systems with a larger gap, such as clusters of SMPs and networks of workstations, it is harder to achieve good performance [19, 22, 26]. As gaps increase, it is likely that performance will continue to suffer.[1]

There is little insight in how a growing NUMA gap influences application performance, or how good performance can be achieved on systems with a large gap. This paper studies the performance of six nontrivial parallel applications, Barnes-Hut, Water, FFT, TSP, ASP, and Awari. We have built an experimental testbed using 64 Pentium Pros, a high-speed network (Myrinet) and an ATM network. The testbed can be configured as multiple Myrinet clusters that are interconnected by ATM links with different latencies and bandwidths. In this way, we can emulate a variety of NUMA/meta-computer configurations, where the gap between the fast local network (Myrinet) and the wide-area network (ATM) varies from 0 to 4 orders of magnitude. The parameter settings have been calibrated using a real wide-area system.

The contributions of the paper can be summarized as follows: We analyze the impact of a wide range of gaps between the slowest and fastest links of the interconnect, to see where performance of conventional parallel applications starts to deteriorate. We find that for gaps larger than those of current NUMAs (one order of magnitude) performance rapidly drops to an unacceptable level.

For five out of six applications we describe performance improvements. (Since the applications were originally designed for a machine with a uniform interconnect, it should perhaps not be surprising that there is room for improvement.) The applications are quite diverse, and so are the algorithmic changes, though all have in common that the application's communication pattern is made to fit the hierarchical interconnect—the changes do *not* improve performance on a uniform network. Commonalities between the improvements are described.

For the improved applications, the impact of the same range of gaps in the interconnect is analyzed, to see how the changes influence application behavior, and how well they work. Taking a speedup of 60% of speedup under uniform remote access as our criteria, we find that, for bandwidth, the acceptable NUMA gap is increased to two orders of magnitude, and for latency it is increased to three orders of magnitude.

We conclude that in many applications, with careful optimization there is room for growth to large architectures with highly non-uniform access times. The application im-

[1] This has prompted some to claim, perhaps whimsically, that non-uniformity is a bug, not a feature.

provements themselves are straightforward programming techniques—the challenge lies in understanding the interaction between interconnect and communication pattern. Further work is needed to make this easier, by expressing communication with higher level primitives, or by incorporating common traits of the improvements into coherency protocols. The experience with DSMs on SMP clusters suggests, however, that this will be challenging [26].

Gaps of two to three orders of magnitude correspond to differences between local area and wide area links. Most meta-computing projects currently use embarrassingly parallel (job-level) applications that barely communicate. Our results imply that the set of applications that can be run on large scale architectures, such as a computational grid, is larger than assumed so far, and includes medium grain applications. (Further research should study the impact of variations in latency and bandwidth, which often occur on wide area links.)

The remainder of the paper is organized as follows. Section 2 discusses related work. Sections 3 and 4 describe in detail the applications, improvements, and system that have been used in this experiment. Section 5 describes the results of our measurements, and analyzes them. Section 6 concludes the paper and discusses implications of this work.

2 Related Work and Background

In this paper, we try to understand how large differences in bandwidths and latencies in an interconnect influence application performance. For small gaps, several studies report good performance on hardware DSM NUMA systems [9, 17, 21, 29, 35, 28]. These systems have a gap of about a factor of 3 between the slowest and the fastest links. The picture changes for systems with longer access times. Papers by [10, 19, 22, 26, 31, 36] study local area clusters of SMPs, which have a gap of up to an order of magnitude. These studies tend to focus on coherency protocol issues, using software DSMs such as MGS, TreadMarks, SoftFLASH, CashMere and Shasta, to see how the presence of hardware shared memory improves performance. Performance results vary; earlier studies using partial simulation or tighly coupled hardware [10, 36] showed better results than studies using recent stock SMPs [11, 19, 22, 26, 31]. For SPLASH-like numerical applications, the experience with commercial SMPs is that the presence of hardware shared memory helps performance surprisingly little, due to bus contention and the cost of the hardware coherence protocol. Overall performance is somewhat disappointing, especially for applications that synchronize frequently. Soundararajan et al. attempt to improve NUMA performance through better data locality, with migration/replication protocols [30].

False sharing and disappointing performance in general is the reason for work on data structure and algorithm restructuring [15, 16]. Jiang et al [16] use a software DSM on top of a network of workstations. Even though here remote access times are uniform, the relatively large network overhead requires application changes for good performance. Their changes exceed simple padding or data-structure rearranging, requiring insight into both the application and key aspects of the SVM. In previous work we have experimented with a still larger gap, of two orders of magnitude, for which we also found that applications need communication pattern changes [3].

Wide-area systems typically have gaps of three to four orders of magnitude, which covers the end of the range that we study here. Compared to SMP clusters, they provide a more challenging environment in terms of latency and bandwidth gap, but also of fault tolerance and heterogeneity. Meta-computing research focuses on the latter two issues [12, 14]. Because of the high (and non-uniform) latencies, applications are typically embarrassingly parallel, unlike ours, which are of medium grain.

As NUMA systems scale up, it is inevitable that memory access times becomes less uniform. There is evidence that applications can be quite sensitive to non-uniform memory access [3, 16, 30], and we want to know how applications perform on such systems. So far, little attention has been paid to the effect of gap size. Previous studies use small, fixed, gaps. We are interested in how performance scales with different gaps; in our interconnect the gap is varied over a large range, from zero to four orders of magnitude. In addition, many studies focus on issues such as DSM protocols or message passing versus shared memory [8, 9, 30]. Again, our focus is the NUMA gap. We investigate where conventional applications break down, how communication patterns can be adapted, and how far performance improvement can be pushed. As an important aside, we want to know how difficult it is to implement such changes.

Differences in link speeds pose interesting challenges to programmers. This paper explores how serious these challenges are, and how we can deal with some of them.

3 Applications

Our application suite consists of six diverse programs. Barnes-Hut, Water, and FFT are numerical programs that originate from the Splash-2 suite [35], TSP and ASP are optimization codes, and Awari is a symbolic artificial intelligence program. The applications have diverse communication patterns. Table 1 summarizes the behavior of the applications on a single Myrinet cluster. For all applications, larger problems give better speedups. We use relatively small problem sizes in order to get medium grain communication. Medium grain is taken here as a total communication volume of at least 100 KByte/s on a single level cluster of 32 processors. All applications and problem sizes run efficiently on a single Myrinet cluster. Five of our six ap-

Program	Speedup	Speedup	Total Traffic	Runtime
	32 p.	8 p.	32 p. MByte/s	32 p, in sec
Water	31.2	7.8	3.8	9.1
Barnes-Hut	28.4	7.1	17.8	1.8
TSP	29.2	7.7	0.52	4.7
ASP	31.3	7.8	0.75	6.0
Awari	7.8	4.6	4.1	2.3
FFT	32.9	5.3	128.0	0.26

Table 1: Single-Cluster Speedup on 8 and 32 processors.

plications are written in the Orca parallel programming language [1], for ease of use of the wide-area system, and for ease of debugging (Orca is type-safe). For most programs, serial performance is comparable to serial C performance. Barnes-Hut is written in C with calls to the Panda [1] wide-area/local area messaging layer.

3.1 Application Characteristics

This subsection summarizes key application characteristics. The next subsection describes the improvements that were implemented to achieve good performance on the highly non-uniform system.

Water The Water program is based on the “n-squared” Water application from the Splash suite [35], rewritten for distributed memory [25]. Related distributed memory optimizations are described by [16]. We report on experiments with a medium sized input set of 1500 particles. The serial speed of the distributed memory program is about ten percent better than the original Splash code.

Barnes-Hut Barnes-Hut is an $O(n \log n)$ N-body simulation. The implementation in the Splash-2 suite has a fine coherency unit which causes inefficiencies on coarse grain hardware [1, 16]. In this experiment a new distributed-memory code by Blackston and Suel [4] has been used. Instead of finding out at runtime which nodes and bodies are needed to compute an interaction, this code precomputes where nodes and bodies are needed, and sends them in one collective communication phase at the start of each iteration. Stalls in the computation phase are thus eliminated. Related improvements have been reported by [13, 16, 34]. Using the same input problem, the serial program runs slightly faster than the Splash code (while computing the same answer). We used a set of 64K particles.

ASP The All-pairs Shortest Path program is a parallel version of the classic Floyd-Warshall algorithm. It uses a replicated distance matrix of 1500 by 1500 entries. Each processor iterates over rows in the matrix, and broadcasts result rows as they are computed. These have to be processed in order by the other processors before they can compute their rows. A designated node issues sequence numbers to achieve this ordering.

TSP The Traveling Salesperson Problem computes the length of the shortest path along n cities, by enumerating the possible paths. The program uses a centralized job queue which is filled with partial paths, from which workers get jobs. A 16 city problem is used as input; jobs consist of a partial tour of 5 cities, creating small jobs and a (for this application) fine communication grain, as Table 1 shows. Deterministic runs are ensured by using a fixed cutoff bound [1].

Program	Communication	Optimization
Water	All to Half Multicast	Cluster Cache, Reduct Tree
Barnes	BSP/Pers All to All	BSP-msg Comb Node/Clus
TSP	Centralized Work Queue	Work Q/Cluster + Work Steal
ASP	Totally Ordered Broadcast	Sequencer Migration
Awari	Asynch Unordered Msg	Msg Comb/Clus
FFT	Pers All to All	–

Table 2: Communication Patterns and Optimizations

Awari Awari, a retrograde analysis program, is a symbolic application that computes end game databases, of importance for programs playing games such as checkers. It is based on backwards reasoning and bottom-up search. Here we compute a relatively small 9 stone database for the African board game Awari. The program sends many small, asynchronous, packets of work to other processors [2]. These messages are combined into larger messages for performance reasons. The communication pattern of Awari is irregular asynchronous point-to-point messages.

FFT The FFT application computes a one-dimensional Fast Fourier transform, using the transpose algorithm [20]. The program is a rewrite of the Splash-2 code for distributed memory, and achieves an excellent speedup on a single Myrinet cluster, despite the short run time. The communication part of this program is very simple: it performs three transposes, interspersed by parallel FFTs. The problem size is 2^{20} complex floating point numbers, the largest that would fit in memory. FFT shows a small superlinear speedup, due to cache effects.

Table 2 summarizes the communication patterns and improvements. Figure 1 summarizes inter-cluster traffic of the original applications. The figures show data volumes in MByte/s per cluster and numbers of messages per second per cluster (for 6 MByte/s bandwidth per link and 0.5 ms latency, and 4 clusters of 8 processors, a configuration with

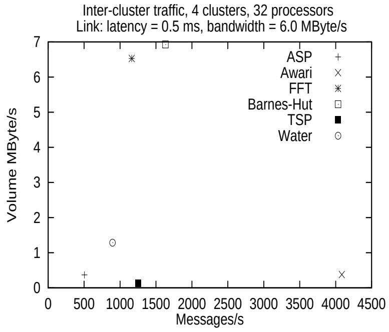

Figure 1: Communication Volume and Messages

12 wide-area links in total). TSP has an extremely low intercluster communication volume, 0.1 MByte/s, though a nonnegligible number of messages. Barnes-Hut and FFT have a high communication volume of nearly 7 MByte/s (note that the bandwidth limit in this case is 18 MByte/s per cluster, since with 4 clusters there are 3 links of 6 MByte/s out of each cluster). Awari can be found in the opposite corner of the graph, with a high number of tiny inter-cluster messages (more than 4000 per second per cluster). Water and ASP have a modest level of inter-cluster traffic, less than 1000 messages per second and less than 2 MByte/s per cluster.

3.2 Optimizations per Application

The applications are run on an interconnect whose bandwidth and latency difference ranges from small to large. The system consists of 4 clusters of 8 processors each. Inside the clusters the processors are connected by fast Myrinet links (0.020 ms application level latency, 50 MByte/s application level bandwidth). The clusters themselves are fully connected by slow ATM links through additional gateway machines (0.4–300 ms latency, 10–0.03 MByte/s bandwidth). Thus, we have a two level interconnect.

The applications have originally been developed with a uniform network in mind, where all links have the same latency and bandwidth. Performance suffers when the system is highly non-uniform, as in our system, where slow links have latencies up to 5000 times the latency of fast links. This subsection describes which problems have to be resolved to achieve good performance. Some of the improvements have been described previously in [3].

Water The Water program is a classical simulation program of the behavior of *n* water molecules in an imaginary box. Each of the *p* processors is assigned an equal number of water molecules. The computation of the $O(n^2)$ intermolecular forces is the most time-consuming part of the simulation. Forces between two molecules are computed by one of the two processors, the owner of that molecule. At the start of this phase, each processor gets the positions of the molecules of half of the other processors. As a force is calculated it is added locally to determine the total force acting on the local molecule, and is sent to the appropriate other processor so that it can compute the total force for its molecule. All individual molecule updates destined for a processor are combined into one message. The force update phase amounts to two “all-to-half” communications (one to distribute molecule positions, one to send force updates back). The total number of messages approximates $p \cdot \frac{p}{2} = O(p^2)$.

The Water program suffers from a severe performance degradation when inter-cluster links are much slower than intra-cluster links. With 4 clusters, 75% of all messages are between clusters—that is, slow. The two operations, *copying* of molecule positions and *adding* of force updates, are 1–*n* and *n*–1, reduction-like, operations. With the original program, the position of a given molecule is transferred many times over the same inter-cluster link, since multiple processors in a cluster need it. The optimization avoids sending the same data over the inter-cluster link more than once. For every processor *p* in a remote cluster, we designate one of the processors in the local cluster as the local coordinator for *p*. If a process needs the molecule data of processor *p*, it does an intracluster RPC to the local coordinator of *p*. The coordinator gets the data over the intercluster link, forwards it to the requester, and also caches it locally. If other processors in the cluster ask for the same data, they are sent the cached copy. A similar optimization is used at the end of the iteration for the force updates. All updates are first sent to the local coordinator, which does a reduction operation (addition) on the data and transfers only the result over the inter-cluster link.

Barnes-Hut Blackston and Suel’s distributed version of the Barnes-Hut algorithm precomputes where nodes and bodies will be needed in each iteration, and sends them in one collective communication phase at the start of the iteration. The program is coded in Valiant’s BSP style [33]. Communication takes place in so-called supersteps, which are separated by barriers. In each of these supersteps the program sends many small messages, which incur large overhead if sent indiscriminately over inter-cluster links. All efficient BSP implementations perform message combining of small messages for each recipient. To achieve good performance on the multi-cluster, two more optimizations have been implemented. First, each sender processor combines messages to different recipients in the same target cluster into one message towards the target cluster gateway, using the fact that the code precomputes which parts of the Barnes-Hut tree will be needed on other processors. These messages are dispatched by the receiving cluster gateway to the recipients. Second, the strict barrier synchronization is relaxed by using explicit sequence numbers. (BSP is a relatively young programming model. An active community exists working on efficient implementations of the model, see for example [32].)

ASP In ASP, processors iterate over rows in a distance matrix, generating result rows that are needed by the other processors before they can start new iterations. In the original implementation, a designated sequencer node is used to ensure that rows arrive in order at the processors. The sender of the row has to wait for a sequence number to arrive before it can continue. On a multi-cluster with 4 clusters, 75% of the broadcast requests will thus incur the intercluster penalty. This slows down the program significantly.

Communication in ASP is quite regular: first processor 1 computes and broadcasts its rows, then processor 2, etc. On the multi-cluster we take advantage of this regularity by migrating the sequencer to the cluster of the node that does the sending. In this way, sequencer request can be satisfied by a node in the local cluster. In a four cluster system, the sequencer has to migrate only three times, incurring inter-cluster latency only three times. (Another solution would be to drop the sequencer altogether, since processors know who will send which row. Again, this solution exploits the regularity of the ASP algorithm.)

The row broadcasts themselves are asynchronous, so the sender does not suffer from inter-cluster latency. Overall progress is, however, sensitive to inter-cluster bandwidth. Broadcasts are performed using a multicast tree, with point-to-point communication from the sender to the cluster gateways, and multicast primitives inside clusters.

TSP TSP uses a single job queue from which processors retrieve work when their current job is finished. On 4 clusters, 75% of the traffic is between clusters. Even though, compared to the other applications, TSP communicates infrequently, the level of traffic still limits performance considerably. The centralized job queue causes too much inter-cluster traffic. The multi-cluster optimization is to distribute the queue over the clusters. Each cluster now has its own queue, and workers perform work stealing only from their own queue. When the queue becomes empty it tries to steal work from the other cluster queues, to maintain a good load balance. In our system, the number of clusters is small compared to the number of processors. There are only as many queues as there are clusters, and inter-cluster traffic is solely influenced by the number of clusters, not by the number of processors per cluster.

Awari Awari performs a parallel search starting from known end states in a search space (for example, checkmate). States are hashed to processors. The values of all reachable states are computed and sent to the processors owning these positions, which start working on them by generating known values of their reachable states. This process results in many small messages. The original parallel program performs message combining for destination processors, to reduce communication overhead. The search progresses in stages; in each stage, the database for one more stone is computed. Too much message combining results in load imbalance since processors are starved of work at the end of the stages. The high volume of small messages combined with the larger overhead of the inter-cluster links limits performance. To reduce the impact of this overhead, we add another layer of message combining: cross-cluster messages are first being assembled at a designated local processor, are then sent in batch over the slow link, and are subsequently distributed by a designated processor at the other cluster to the final destinations. The extra layer of message combining reduces the impact of the large communication overhead of the inter-cluster links.

FFT The FFT application is renowned for its high communication volume. It is especially ill-suited for a system with long latencies and low bandwidths, or a highly non-uniform interconnect. The communication pattern is a matrix transpose, with little computation. No multi-cluster optimization was found. The purpose of this work is to gain insight in the limits of multi-layer systems with a highly non-uniform architecture. FFT serves as a reminder that there are programs that are unsuited for our interconnects.

3.3 Optimization Overview

Despite the small input problems, all applications except Awari perform well on a single Myrinet cluster (Table 1). Table 2 lists for each program the base communication pattern and its improvement. The main goal of the improvements is to match the communication structure of the application with the hierarchical interconnect; applications should reduce their communication over slow links—or at least reduce their dependency on that communication.

Like the programs, the communication patterns are quite diverse; the optimizations also appear to be quite varied. The general strategy in optimizing for highly non-uniform interconnects is to change the algorithm so that less traffic is sent over the slow links. If that is not possible, then we trade off latency sensitivity at the cost of increased bandwidth requirements. Four types of optimizable communication patterns can be distinguished in our applications; two are of a more algorithmic nature, and two are more related to communication parameters. The first optimization is the reduction operation, which was implemented in Water as a one-level tree. For the multi-cluster interconnect, it is implemented with a two-level tree, the cluster gateways accumulating intermediate results. The second optimization is the work queue, which is implemented in TSP as a single centralized queue. For the multi-cluster interconnect, it is implemented as a distributed queue, one queue per cluster gateway with work stealing among the clusters to maintain load balance. The third optimization is message combining, which is used in Barnes-Hut and Awari to reduce communication overhead for frequent small messages on high-overhead links. The fourth optimization is to exploit asynchrony inherent in the application, which is used in ASP to reduce the number of synchronization points. Furthermore, it can be argued that Blackston and Suel have performed just this kind of optimization in their rewrite of a traditional SPLASH-like shared memory Barnes-Hut code. The communication pattern of FFT is too synchronous and fine grained; no optimization was found.

It is interesting to contrast our changes to the restructuring by Jiang et al [16]. Their work is performed on a software DSM running on a network of workstations, a system with uniform remote access links. They focus on restructuring algorithms to reduce overheads caused by inefficient remote access patterns, fine-grain synchronization, and multiple-writer algorithms. Our system has non-uniform remote access links; we focus on restructuring communication patterns in a two-level system, reducing traffic over the slow links by clustered work stealing, message combining, removing synchronization points, and optimizing reduction operations. Our changes differ from Jiang's in that we make explicit use of the multi-level structure of the interconnect. Indeed, the changes are ineffective on an interconnect with uniform remote access links.

The algorithmic changes are applications of well-known techniques. The novelty lies not so much in the changes themselves as in the magnitude of the performance improvement they cause. The hard part is the understanding of the application behavior, and how it maps to the interconnect (see also [3, 16]). Efforts to express communication constructs on a higher conceptual level [6, 7, 23, 27], or to incorporate them in a cache coherency protocol, would ease this problem (Soundararajan et al describe work on protocol optimizations for a NUMA gap of one order of magnitude [30]). The next two sections describe in detail the impact of the improvements.

4 Experimental Setup

We use an experimental cluster-of-clusters system that consists of four local clusters of 200 MHz/128 MByte Pentium Pro machines connected by Myrinet [5], using LANai 4.1 interfaces. The peak bandwidth of Myrinet is 2.4 Gbit/s, and host-to-host latency is at least 5 μs. In our system, application-level bandwidth is 50 MByte/s, one-way application level latency is 20 μs. The clusters are located at four universities in the Netherlands. They are connected via dedicated gateway machines over ATM by 6 Mbit/s Permanent Virtual Circuits (application-level bandwidth is 0.55 MByte/s over TCP). The round trip latency is 2.5–3 ms. Three sites have 24 compute nodes each, one site has 64. The wide-area network is fully connected, the system-area networks are hypercubes. The operating system is BSD/OS version 3 from BSDI. The wide-area ATM links have a fixed latency and bandwidth. To allow for experimentation with different speeds, 8 local ATM links have been installed in the 64 processor cluster, using the same hardware as in the real wide-area system (ForeRunner PCA-200E ATM boards). The latency and bandwidth of the ATM links can be varied by delay loops in the cluster gateway machines. Except for the local ATM links, this experimentation system is identical to the real wide-area system; the same binaries are run in both setups, and except for the delay loops, there are no simulated parts. When the delay loops are set to the wide area latency and bandwidth, run times differ on average by 3.6% for our applications.

All runs on this experimentaion system showed very reproducible timings, except for Barnes-Hut, whose timings spread more than a factor of two over the runs. Since we suspect the TCP stack of causing this behavior, Barnes-Hut was run on a modified experimentation system where the ATM/TCP links were replaced by Myrinet links. The delay loops for this pure Myrinet system were calibrated to produce exactly the same behavior as the ATM/TCP system. On the pure Myrinet system, the timing anomalies disappeared. Therefore we present the performance on the pure Myrinet system for Barnes-Hut.

The system can be programmed through different libraries and languages, from message passing libraries such as MPI and Panda [1], software DSMs such as TreadMarks and CRL [18], to parallel languages such as Orca [1]. The Panda messaging layer has been adapted to support communication over both Myrinet (using Illinois Fast Messages [24]) and ATM (using BSD's TCP/IP).

5 Results

The goal of this work is charting the sensitivity of application performance to gaps in bandwidth and latency. This section discusses the performance measurements.

The speedup of a multi-cluster machine is bounded by the speedup of a single-cluster machine with the same number of processors, and the same execution schedule. (To put this differently, when some of the fast Myrinet links of the interconnect are changed into slow ATM links, our parallel applications will run more slowly.) Speedups are shown relative to the all-Myrinet upper bound.

5.1 Relative Speedup

Figure 2 shows speedup graphs for the six applications, for 4 clusters of 8 processors: unoptimized on the first and third row, optimized on the second and fourth row. Speedup is shown relative to the speedup of the 32 processor all-Myrinet cluster, for different inter-cluster bandwidths and latencies. It is computed as percentage $\frac{T_L}{T_M}$, where T_L is the run time on the single cluster and T_M is the run time on the multi-cluster. Startup phases are omitted from time and traffic measurements. Myrinet bandwidth is constant at 50 MByte/s, latency is constant at 20 μs. Inter-cluster bandwidth and latency are limited by the local OC3 ATM link of 155 Mbit/s—at the application level, over TCP, this yields 14 MByte/s bandwidth and 0.28 ms latency. On the *x*-axis the bandwidth of the ATM links is shown. Delays are set so that the resulting bandwidth is 6.3, 2.6, 0.95, 0.3, 0.1, and 0.03 MByte/s. The one-way ATM latency is set to 0.4, 1.3, 3.3, 10, 30, 100, and 300 ms.

The speedup profiles render performance relative to all-Myrinet speedup for latency/bandwidth combinations. The general shape of the graphs is as can be expected: higher inter-cluster bandwidth and lower inter-cluster latency improve performance, and multi-cluster performance is lower than single-cluster Myrinet performance. When we compare the optimized to the unoptimized graphs, the optimizations result in the graphs being shifted upward or to the

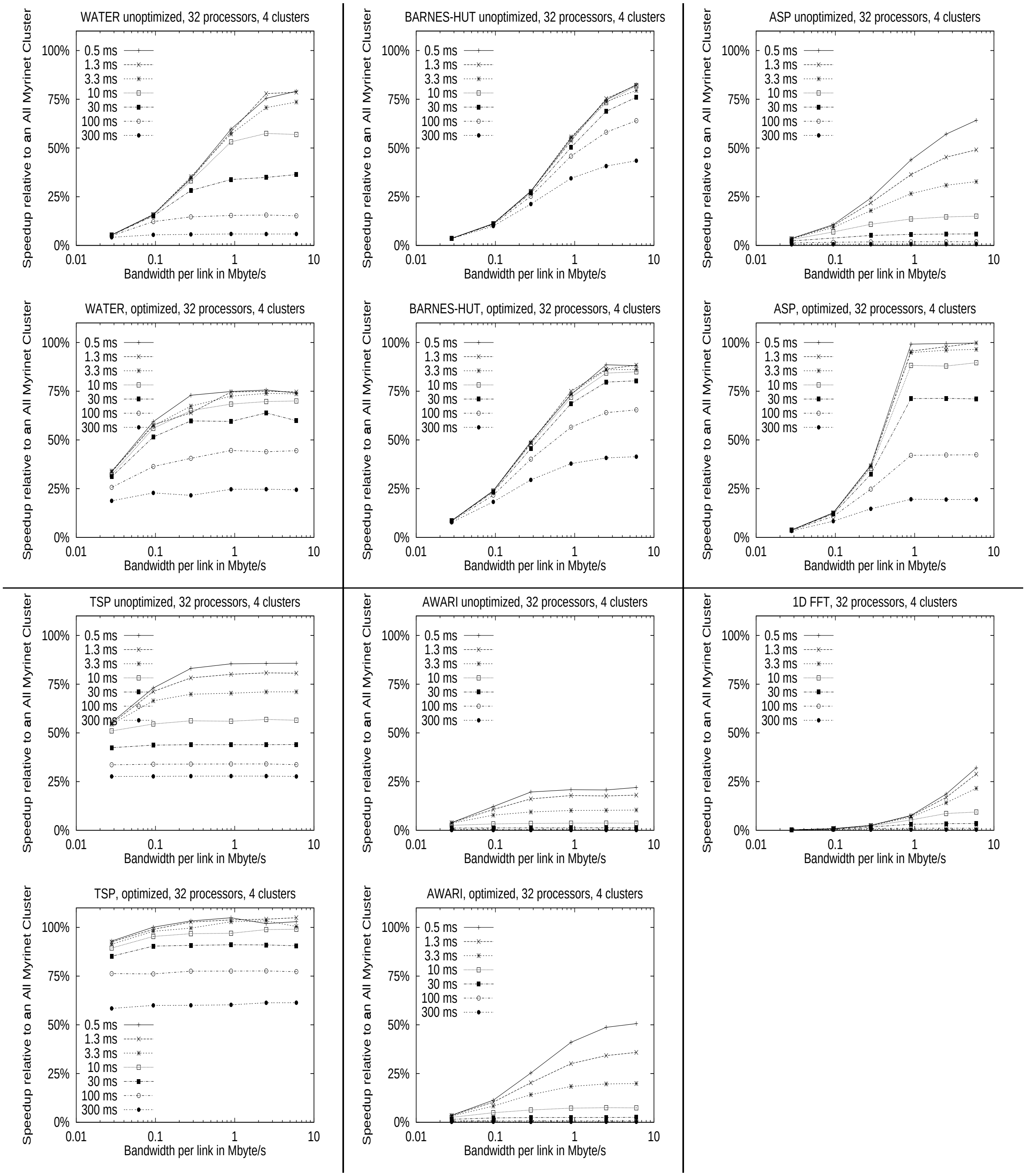

Figure 2: Speedup Relative to an All-Myrinet cluster

left. An upward shift indicates higher performance for the same inter-cluster latency. A shift to the left indicates that the same performance can be achieved at lower inter-cluster bandwidth. (In most applications both effects can be seen.) For Water, the optimizations extend the range of bandwidth where speedup is better than 60% of all-Myrinet from 1 MByte/s to 0.1 MByte/s. For the original program, performance decreases steadily from 10 ms latency or 1 MByte/s bandwidth; the performance of the optimized program is much more stable, and deteriorates seriously from 100 ms latency or 0.03 MByte/s. Overall, the NUMA gap for which good performance is achieved is improved by more than an order of magnitude for both inter-cluster latency and bandwidth. For the fastest inter-cluster links, however, the unoptimized program is faster: here, the increase in local communication is not (yet) outweighed by the reduction in remote communication. For Barnes-Hut the improvements have a similar effect, although overall performance is not as good. For Awari the message combining has more than doubled performance for latency up to 3.3 ms; the higher overheads can be masked by message combining, provided that there is enough bandwidth.

The improved version of ASP has a good performance for up to 30 ms latency, against 1 ms for the original program. Speedup shows a sharp sensitivity to bandwidth below 1 MByte/s, as explained in Section 3.2. TSP, on the other hand, is practically insensitive to bandwidth but is sensitive to latency. Its performance is increased by about 25% by the improvements.

For high bandwidth/low latency combinations, performance is good for the improved versions of four of the applications, Barnes-Hut, Water, ASP, and TSP. For inter-cluster latencies of 0.5–3.3 ms and bandwidths of 0.3–6 MByte/s multi-cluster speedup is well above 50% of single-cluster speedup. For bandwidths better than 1 MByte/s speedup reaches 60% for 30 ms latency, and about 40% for 100 ms latency. For extreme bandwidths and latencies (30 KByte/s bandwidth or 300 ms latency) relative speedup drops below 25%, which corresponds to the performance of a single Myrinet cluster of 8 processors. Thus, for these bandwidths and latencies, using extra clusters actually slows down the computation.

Performance for Awari and FFT is significantly lower. For FFT the 25% point is not even reached. The reason for the bad performance of Awari and FFT is that these applications have a higher level of inter-cluster communication. In Awari the extra level of message combining is moderately effective; too much message combining introduces load imbalance. In FFT no optimization has been implemented.

To summarize, for Barnes-Hut, Water, ASP, and TSP, the range for inter-cluster bandwidth and latency at which reasonable speedups are achieved is increased by an order of magnitude or more by the restructuring of the communication pattern. Reasonable speedup starts at an inter-cluster bandwidth of 0.1–0.3 MByte/s and an inter-cluster latency of 30–100 ms. Given a Myrinet bandwidth of 50 MByte/s and a latency of 20 μs, this corresponds to an intra-cluster/inter-cluster performance gap of 167–500 for bandwidth and 1500–5000 for latency, depending on whether 40% or 60% of single-cluster speedup is desired. The optimized applications allow a significantly larger gap for latency than for bandwidth.

In addition to the sensitivity to bandwidth and latency gaps, we have also performed experiments with different cluster structures. Performance increases as there are more, smaller, clusters: a setup of 8 clusters of 4 processors outperforms 4 clusters of 8 processors. This may seem counterintuitive, since replacing fast links with slow links ought to reduce performance. However, performance is limited by wide-area bandwidth, and our wide-area network is fully connected: in the multi-cluster, bisection bandwidth actually increases as more slow links are added, despite the loss of fast links. This effect can be traced to simple bandwidth sensitivity: speedup decreases as more processors compete for the same wide-area links. (Graphs show a straightforward effect, and are omitted for reasons of space.) In a larger system it is likely that the topology is less perfect. This effect will then diminish, and disappear in star, ring, or bus topologies. Future topologies will in practice be somewhere in between the worst case of a star or ring and the best case of a fully connected network.

5.2 Bandwidth and Latency Sensitivity

This subsection analyzes the inter-cluster traffic in more detail, to complement the speedup picture. We focus on synchronous versus asynchronous communication by examining inter-cluster communication for different bandwidths and latencies.

Performance is influenced strongly by inter-cluster traffic (high-traffic applications have a low speedup and vice versa). The speed of the interconnect influences communication and synchronization overhead of the programs. The left-hand graph in Figure 3 shows the percentage of runtime that is spent in communication over the inter-cluster interconnect as a function of bandwidth, for 4 clusters of size 8; one-way latency is set to 3.3 ms. The right-hand graph in Figure 3 shows the inter-cluster communication time of the interconnect as a function of latency; bandwidth is set to 0.9 MByte/s). The communication time percentage is computed as $\frac{T_M - T_L}{T_M} \cdot 100$, or the difference between multi-cluster run time and single cluster run time as a percentage of multi-cluster run time. These graphs represent no new data compared to Section 5.1, but they offer a different viewpoint to increase understanding of communication behavior.

These graphs indicate where applications are dominated by synchronous communication, and where by asynchronous communication. Purely asynchronous communication is limited by bandwidth (if we disregard startup time); it corresponds to a horizontal line in the latency graph. Purely synchronous communication (i.e., a null-RPC) is limited by latency; it corresponds to a horizontal line in the bandwidth graph. The graphs in Figure 3 show that the communication patterns of the applications contain both streaming of asynchronous communication and request/reply style synchronous communication.

It is interesting to note the differences among the applica-

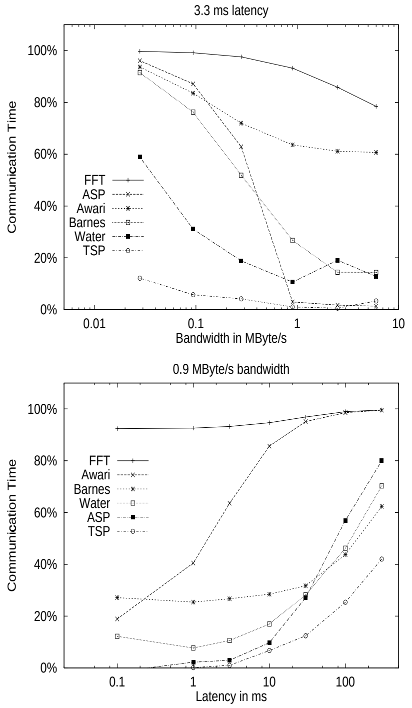

Figure 3: Inter-cluster Traffic—Bandwidth and Latency

tions. In both the bandwidth and the latency graph, communication time for FFT is close to 100%, indicating that run time is almost completely dominated by communication. Awari is a close second, although at latencies lower than 10 ms communication time drops sharply (at 3 MByte/s). For Barnes-Hut, Water, ASP and TSP communication time is significantly less at high bandwidth and low latency.

Latency: Up to 3 ms Barnes-Hut, Water, and ASP are relatively insensitive to latency; their lines are nearly flat. For longer latencies, communication becomes quite sensitive to latency. Apparently, up to 3 ms the data dependencies of the programs allow latency hiding. TSP is quite independent of bandwidth for latencies up to 10 ms.

Bandwidth: For a bandwidth of 10–3 MByte/s, Barnes-Hut, Water, Awari, and ASP are relatively insensitive to bandwidth. TSP is almost completely insensitive to bandwidth; its work-stealing communication pattern comes quite close to the null-RPC.

6 Conclusions

Current NUMA machines yield good performance for many applications [17]; however, there is little insight in application performance on interconnects with a larger gap between slow and fast link speed. Such a large gap will occur in large-scale NUMAs, or when, as in a meta-computer, wide-area links are added to the interconnect. As far as we know, this is the first study to examine performance of non-trivial applications over a large range of gaps, using a real system. Our main contribution is an analysis of when the difference between slow and fast links starts to affect performance. We describe ways to restructure the applications to make performance less sensitive to a large gap, and we analyze how well the improvements work. We do this by examining speedup relative to the gap in bandwidth and latency in the interconnect. We find that when the difference in speed in an interconnect grows larger than in current generation NUMAs, performance suffers dramatically. For these gaps (one order of magnitude and larger) communication becomes limited by the slow links in the interconnect. Once the bottleneck is identified, we can apply changes such as increasing the height of reduction trees, clustering work stealing, combining messages, and removing redundant synchronization points—changes that make explicit use of the multi-level structure of the interconnect, in contrast to the work by Jiang et al, who describe single-level changes [16]. The changes can speed up applications significantly. When acceptable performance is defined as 60% of the speedup on a uniform interconnect, restructuring the communication pattern increases the allowable gap in bandwidth and latency by more than an order of magnitude: in our system gaps of two orders of magnitude for bandwidth, and three orders of magnitude for latency, can be bridged by four of our six applications. However, some communication patterns, such as matrix transpose, resist optimization.

Interconnects are becoming increasingly hierarchical, making it harder to achieve high performance. Nevertheless, we believe that for many real applications it will remain to be possible to do so. We also believe that to achieve this level of performance, more effort is needed to assist programmers in identifying performance problems, to help them better to understand the characteristics of interconnect and program.

Acknowledgements

This research is supported in part by a PIONIER grant from the Dutch Organization for Scientific Research (NWO) and a USF grant from the Free University. NWO has partially supported the DAS project. Aske Plaat is supported by a SION grant from NWO. We are grateful to Torsten Suel for providing us with his n-body code. We thank Andy Tanenbaum and especially Raoul Bhoedjang for insight and inspiration. Together with Thilo Kielmann they also provided valuable feedback on previous versions of this paper. Mirjam Bakker implemented the Awari optimizations. Peter Dozy came up with and implemented the Water and TSP optimizations. Kees Verstoep kept the DAS alive.

References

[1] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1), February 1998.

[2] H.E. Bal and L.V. Allis. Parallel Retrograde Analysis on a Distributed System. In *Supercomputing '95*, December 1995. Online at http://www.supercomp.org/sc95/proceedings/.

[3] H.E. Bal, A. Plaat, M.G. Bakker, P. Dozy, and R.F.H. Hofman. Optimizing Parallel Applications for Wide-Area Clusters. In *IPPS-98 International Parallel Processing Symposium*, pages 784–790, April 1998.

[4] David Blackston and Torsten Suel. Highly portable and efficient implementations of parallel adaptive n-body methods. In *SC'97*, November 1997. online at http://www.supercomp.org /sc97/program/TECH/BLACKSTO/.

[5] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[6] J. Carter, J. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory System. *ACM Transactions on Computer Systems*, 13:205–244, August 1995.

[7] S. Chakrabarti and K. Yelick. Implementing an Irregular Application on a Distributed Memory Multiprocessor. In *ACM Symposium on Principles and Practice of Parallel Programming*, June 1993.

[8] S. Chandra, J. Larus, and A. Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs. In *ASPLOS-94 Architectural Support for Programming Languages and Operating Systems*, 1994.

[9] F. Chong, R. Barua, F. Dahlgren, J. Kubiatowicz, and A. Agarwal. The Sensitivity of Communication Mechanisms to Bandwidth and Lantency. In *HPCA-4 High Performance Communication Architectures*, pages 37–46, February 1998.

[10] A. Cox, S. Dwarkadas, P. Keheler, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: a case study. In *Proc. 21st Intern. Symp. Comp. Arch.*, pages 106–117, April 1994.

[11] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory. In *Proc. 7th Intern. Conf. on Arch. Support for Prog. Lang. and Oper Systems*, pages 210–220, October 1996.

[12] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Int. Journal of Supercomputer Applications*, 11(2):115–128, Summer 1997.

[13] A. Grama, V. Kumar, and A. Sameh. Scalable parallel formulations of the barnes-hut algorithm for n-body simulations. In *Supercomputing '94*, November 1994.

[14] A.S. Grimshaw and Wm. A. Wulf. The Legion Vision of a Worldwide Virtual Computer. *Comm. ACM*, 40(1):39–45, January 1997.

[15] T.E. Jeremiassen and S.J. Eggers. Reducing false sharing on shared memory multiprocessors through compile-time data transformations. In *Symposium on Principles and Practice of Parallel Programming*, July 1995.

[16] D. Jiang, G Shan, and J. Singh. Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors. In *PPoPP-97 Symposium on Principles and Practice of Parallel Programming*, June 1997.

[17] D. Jiang and J. Singh. A Methodology and an Evaluation of the SGI Origin2000. In *ACM Sigmetrics / Performance'98*, June 1998.

[18] K. Johnson, F. Kaashoek, and D. Wallach. Crl: High-performance all-software distributed shared memory. In *Symposium on Operating Systems Principles 15*, pages 213–228, December 1995.

[19] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *ISCA-24, Proc. 24th Annual International Symposium on Computer Architecture*, pages 157–169, June 1997.

[20] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin Cummings, November 1993.

[21] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *24th Ann. Int. Symp. on Computer Architecture*, pages 241–251, June 1997.

[22] S.S. Lumetta, A.M. Mainwaring, and D.E. Culler. Multi-protocol active messages on a cluster of SMP's. In *SC'97*, November 1997. Online at http://www.supercomp.org/sc97/proceedings/.

[23] MPI Forum. MPI: A Message Passing Interface Standard. *Int. J. Supercomputer Applications*, 8(3/4), 1994. Version 1.1 at http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html.

[24] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, San Diego, CA, December 1995.

[25] John W. Romein and Henri E. Bal. Parallel n-body simulation on a large-scale homogeneous distributed system. In Seif Haridi, Khayri Ali, and Peter Magnusson, editors, *EURO-PAR'95 Parallel Processing, Lecture Notes in Computer Science, 966*, pages 473–484, Stockholm, Sweden, August 1995. Springer-Verlag.

[26] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-grain software distributed shared memory on SMP clusters. In *HPCA-4 High-Performance Computer Architecture*, pages 125–137, February 1998.

[27] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology*, 1(3):85–96, August 1993.

[28] J. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole and radiosity. *Journal of Parallel and Distributed Computing*, June 1995.

[29] J.P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *ACM Computer Architecture News*, 20(1):5–44, March 1992.

[30] V. Soundararajan, M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, and J. Hennessy. Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors. In *ISCA-98, 25th International Symposium on Computer Architecture*, pages 342–355, June 1998.

[31] R. Stets, S Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software coherent shared memory on a clustered remote-write network. In *Proc. 16th ACM Symp. on Oper. Systems Princ.*, October 1997.

[32] T. Suel, M. Goudreau, K. Lang, S. B. Rao, and T. Tsantilas. Towards Efficiency and Portability: Programming with the BSP Model. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA 96*, pages 1–12, June 1996. See also www.bsp-worldwide.org.

[33] L. Valiant. A Bridging Model for Parallel Computation. *Comm. ACM*, 33(8):100–108, August 1990.

[34] M. Warren and J. Salmon. A parallel hashed oct-tree n-body algorithm. In *Supercomputing '93*, November 1993.

[35] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[36] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 45–56, May 1996.