RESOURCE MANAGEMENT IN SINGLE-CHIP MULTIPROCESSORS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Kelly A. Shaw
March 2005

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

William J. Dally
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Margaret Martonosi

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle A. Olukotun

Approved for the University Committee on Graduate Studies.

Preface

Technology advances will soon enable billion transistor chips, permitting large quantities of both logic and memory to be placed on a single die. Increasing on-chip wire delays, however, are shrinking the chip area reachable in a single clock cycle. In response, computer architects are redesigning on-chip structures to reduce the distance signals must travel in one clock period. Single-chip multiprocessors are one proposed architecture for dealing with the multi-cycle communication latencies affecting future chips. These chips are organized into a grid of nodes, where each node contains a processor and a portion of the total on-chip memory. Although these nodes function independently, they interact via shared memory through a network with the property that latency increases linearly based on Manhattan distance.

This dissertation examines how to efficiently map applications onto single-chip multiprocessors given these chips' constraints and opportunities. The small amount of per-node storage limits how much data can be placed on any given node, while the ample processing resources and high bandwidth, low-latency on-chip communication create a large number of quickly accessible locations where data and threads can reside. In order to achieve high performance on these chips, applications must balance the competing goals of improving locality and of distributing resource demands across the chips' many nodes. I present two symbiotic approaches for managing these chips' resources. The first technique, migration of data and threads, reacts to dynamic resource demands and communication patterns to avoid resource hot-spotting and improve locality. The second approach proactively eliminates communication by executing computation at the location of its most frequently accessed data, its anchor. Finally, I show how these techniques can be used in conjunction with well-established techniques, like caching, to further improve application performance.

Acknowledgement

This dissertation would not have been possible without the help and encouragement of a large group of people. I would like to begin by thanking my advisor, Bill Dally, for helping to shape my understanding of computer architecture research. One particularly valuable lesson Bill has taught me is the importance of considering technology's impact on the research you're proposing. Over the years, he has also helped me understand that research is really a progression of your ideas, where new ideas grow out of the understanding you gained from earlier ideas. Most importantly, I learned from Bill how to have confidence in my research.

Margaret Martonosi of Princeton University has been a wonderful mentor and friend to me since 1996. I cannot thank her enough for the time she has dedicated to helping me work through my thesis research. Her enthusiasm as we discussed my research results always motivated me to work harder and faster to get the next results. I will forever be grateful to her for always making time for me and for her encouragement, compassion, and understanding.

I would also like to thank Kunle Olukotun for serving on my Reading Committee and Mendel Rosenblum for serving on my Orals Committee. I had the good fortune of also working with Kunle, Mendel, and Christos Kozyrakis as a teaching assistant. I learned a great deal from all three of them about teaching and appreciate their willingness to let me develop my own teaching skills, using the students in their classes as my test subjects.

Carla Ellis of Duke University has been a huge supporter and friend to me since college. She has been there whenever I have been unsure of the best path to take, encouraging me to do my best and reassuring me of my abilities. Carla has also been a great hiking buddy and explorer; I am always curious if we're going to somehow or another get ourselves into trouble when we get together.

I also would like to thank Dan Sturman and Tushar Chandra, who I worked with at IBM's TJ Watson Research Lab. Although I only worked with them for a summer, they've both kept tabs on me throughout the years. I particularly want to thank Tushar for helping me realize the many choices open to me once I left Stanford.

Much of what I learned in graduate school was from the other students in the

CVA research group. I would like to thank Andrew Chang, Whay Sing Lee, Li-Shiuan Peh, Mattan Erez, and Abhishek Das in particular for their friendship and guidance. Andrew and Whay were my thesis (and life) coaches as well as being great friends. Graduate school was never the same once they finished. Not only did Li-Shiuan brighten up my days with gossip, she helped me gain perspective on graduate school and didn't mind me crashing at her place in Princeton. Mattan taught me tons about interacting with people different from me, spent many hours helping me with my research, and also livened up my afternoons with silly conversations in the office. Finally, Abhishek was the best officemate I could have had for my last years of graduate school. Thanks for all the cheers when things went well.

I was lucky to have three other graduate students to meet with on a weekly basis to discuss our research, life as a graduate student, and life in general. I owe a great deal to Beth Seamans, Bob Kunz, and Ayodele Thomas for their support through the last few years of graduate school. Their many suggestions improved my research and my abilities as a researcher. Their encouragement helped see me through to the end. Beth, in particular, helped me through each of the graduate school hurdles, starting my first year.

I need to thank Tim Purcell and Vicky Wong for getting me through my qualifying exams and not minding my dictatorial way of running our meetings. The same appreciation goes to Sameer Qureshi, Silas (Marner) Boyd-Wickizer, Howard Tsai, and Lance Hammond for putting up with me during my teaching stints at Stanford.

I could never have completed my research without help from the system administrators in the Computer Systems Lab. Not only did Charlie Orgish, Kevin Colton, and Joe Little keep things running, but they came to my rescue many times throughout the years. Somehow they would always find a way for me to get my work done.

Without the help of the many administrators in the Computer Science Department, I would never have managed to navigate the quagmire of Stanford bureaucracy. I'd like to thank Pamela Elliott, Kathi DiTommaso, Indira Choudhury, Peche Turner, Jam Kiattinant, Suzanne Bigas, Thea (the key lady), and the rest of the second floor administrators for all of their help and goodwill.

I also need to thank the many friends I made during graduate school who just made

life fun. Liadan Boyen, Greg Humphreys, and Jessica Humphreys have been steadfast friends for years. They've listened to me complain, laughed at my silliness and my foibles, and been there whenever I needed a shoulder to lean on. Rachel Weinstein brought hilarity back to the office hallway during my last years at Stanford. She also inspired me to make things better for younger women at Stanford. I've had some great times hanging out with Ravi Soundararajan, Janet Wu, Andrew Chang, and Whay S. Lee outside of Stanford. Aarati Martino, Diane Tang, Aaron Stump, Qi Sun, Suzanne Rivoire, Rebecca Schultz, and Marija Vrljic also brought levity and frivolity to my years at Stanford.

The people I met in California who were not affiliated with Stanford reminded me that life existed outside of graduate school. I cannot thank them enough for the reminders! In particular, I'd like to thank the Sun ladies - Val Henson, Val Bubb, and Tabriz Leman. They gave me a glimpse of being young, smart, *and* female in the Bay area. My yoga lady friend Heidi Kikiwada forever impressed me by how good a person she is; she also took me to Giants games! Volunteering with Lorraine Michelle at the Support Network for Battered Women gave me the opportunity to help others and reminded me that I had lots of useful skills. Although I only volunteered for three months, Lorraine's easy way of encouraging people and making them feel great has had a profound effect on how I try to treat people. Additionally, I'd like to thank my British neighbor, Micky Willmott, for just being herself on our daily train commute. Thanks also to Tom Kruse and Vikram Asrani.

I'm thankful to my college buddies for their friendship over the years. These people include Andy David (my trumpet buddy), Lenore Ramm, Eric Gramond (my Frenchman), Billee Jo Kelder, Jen Yates, Jeanette Bennett, Jon Snitow (my friend from Princeton), and Rob (Wob) Flowers. I also have to thank Rachel Pottinger and Steve Wolfman. I've known them both since the beginning of freshman year. Over the years, they've been my friends, roommates, work partners, and unofficial tutors. I've learned from observing them each individually as well as from observing them as a couple. They've been there for me every time I needed them, and they inspire me to try to be the person I want to be.

I'd like to thank my family for their support. My sisters and brothers have protected, supported, and encouraged me throughout my life. As they added people to our family, those people joined in and brought new joy to my life and taught me new things about myself and about life. There is not enough space to do justice to all that I have to be grateful for from my family, so I will just list my siblings from oldest to youngest and specify their families. My oldest sister Lisa Brideau and her husband Norman have two adorable children, Jessica and Justin. My brother Ronald DeGraw and his wife Heidi have a really cute daughter named Lizzie. My brother Russell DeGraw and his wife Tracy have two children, Skylar and Braden. My sister Cathy Smith has three sons, Freddie, Shahiem, and Billique. My twin sister Marianne Shaw is married to Steven Rubenstein. Steven and their two dogs, Greta (the Weimaraner) and Lola (the Redbone Coonhound), have added oodles of entertainment to my life. Steven's large and confusing family has also been extremely welcoming and supportive during my years of graduate school.

Finally, I have to add a very special thank you to my sisters Lisa and Marianne. They have been the constants in my life (along with my teddy bear). They have always encouraged me to be the best I can be, they have stepped in whenever I needed help, and they have let me be who I am even when they didn't necessarily approve. This dissertation is dedicated to them. Without them, I would not be here.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The computer industry has enjoyed doubling transistor counts every 18 months for more than 20 years. This growth in capacity has enabled manufacturers to place larger data structures and increasingly complex processors on individual chips. The resulting generations of processors run at faster clock speeds and require more chip area. These phenomenal advances, however, have begun to be plagued by two new trends, namely increasing on-chip wire delays and tapering performance gains from additional exploitation of instruction level parallelism. In a given clock cycle, smaller fractions of chips can be reached, a consequence of increasing transistor counts and wire delays. At the same time, dedicating larger chip areas to single, complex processor cores obtains only small performance speedups.

Faced with these new constraints, researchers have begun examining new ways to design chips. By strategically decomposing long wires into a series of short wires connected by repeaters, wire delays become proportional to the number of short wires traversed. However, even these constant delays inhibit the use of global control traditionally used in uniprocessor chip design. Designers, consequently, are creating chip architectures with decentralized control which are simply parallel architectures on individual chips.

Single-chip multiprocessors are one approach to using the abundant on-chip transistor counts while accounting for constant, across-chip wire delays. Instead of placing a few complex, superscalar cores on a single chip, many single-issue, in-order cores and memory populate the same total area. These chips are organized into a grid of *nodes*, where each node contains a processor and a portion of the total on-chip memory. Although these nodes function independently, they interact via shared memory through a network where latency increases linearly based on Manhattan distance.

The challenges facing users of these single-chip multiprocessors resemble those encountered in earlier multiprocessor systems, namely load balancing and locality. However, single-chip multiprocessors differ significantly enough from earlier multiprocessors in terms of per processor storage and remote communication latencies that solutions that improved performance in earlier systems have limited applicability in this new domain.

This thesis examines how to efficiently map applications onto single-chip multiprocessors given these chips' constraints, limited on-chip storage, and opportunities, ample processing resources and high bandwidth, low-latency on-chip communication. By analyzing application information about communication patterns and resource demands available at compile and runtime, we create techniques that improve application performance by making better decisions about the placement of data and threads across the chip. In particular, we present a strategy which simultaneously reduces communication demands and distributes resource demands. We also show that results from our application analysis can be used statically to create new thread decompositions which enable improved data and thread mappings on single chip multiprocessors. Finally, we show how these techniques can be used in conjunction with well-established techniques, like caching, to further improve application performance.

1.1 Technology Trends

Advances in chip transistor capacities have enabled increasingly complex, faster, and more powerful processors to be manufactured. Transistor densities, however, represent an important, but not comprehensive, factor in chip performance; wires connecting transistors influence performance as well. In fact, the relative performance impact of wire delays is growing. As wire widths have decreased, wire resistances have increased, creating larger signal propagation delays. Combined with growing transistor capacities and chip sizes, the fraction of chip area reachable at a given clock rate is decreasing as seen in Figure 1.1.

One approach to reducing the impact of wire delays includes breaking long wires into a series of short wires connected by repeaters. Instead of delay growing quadratically with wire length, it grows linearly. This design approach to on-chip interconnect results in multi-cycle chip latencies. Consequently, global control of chips becomes more difficult to achieve, making uniprocessor architectures less appealing.

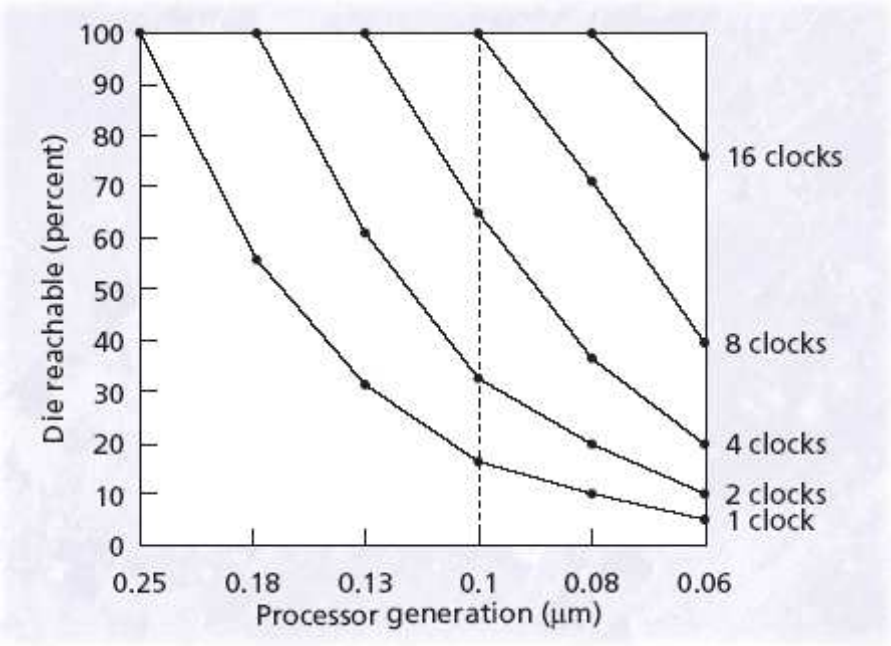


Figure 1.1: Chip area reachable in single clock cycle taken from Matzke's "Will Physical Scalability Sabotage Performance Gains"[36]. ©1997 IEEE

Table 1.1: Ongoing projects assuming multi-cycle chip latencies.

Project	Description
Smart Memories	A single chip is divided into tiles which each include a processor, reconfigurable memory, and a network interface. The project is exploring the usefulness of providing reconfigurability on a larger scale. They have made the on-tile memory reconfigurable and allow multiple adjacent tiles to function together as a single processing entity.
MIT RAW	The RAW project divides a chip into many small tiles, each containing a processor, memory, configurable logic, and a programmable switch. Communication between tiles is fast and generally orchestrated by the compiler although a slower dynamic network exists when communication can not be scheduled statically.
TRIPS	This project at the University of Texas divides a single-chip into two large grid processors. Each grid processor contains an array of executing nodes. Nodes in the array are connected to their nearest-neighbors but can send messages to any other node in the array. The goal of this architecture is to allow these arrays to be reconfigured in order to provide the appearance of different types of architectures suited for different application types.

1.2 Single-Chip Multiprocessors

Computer architects must accomodate these multi-cycle latencies in their designs. One approach being taken is to decompose chips into multiple regions, where all structures in a given region can be reached in a single cycle. Communication between regions takes multiple cycles depending on the distance between these regions. The intuition behind this chip design is that each region operates independently, enabling the region to run at a fast clock rate. In order for this approach to achieve high performance, however, multi-cycle communication latenices must be either tolerated or minimized.

Table 1.1 lists three projects currently incorporating this type of chip design. In

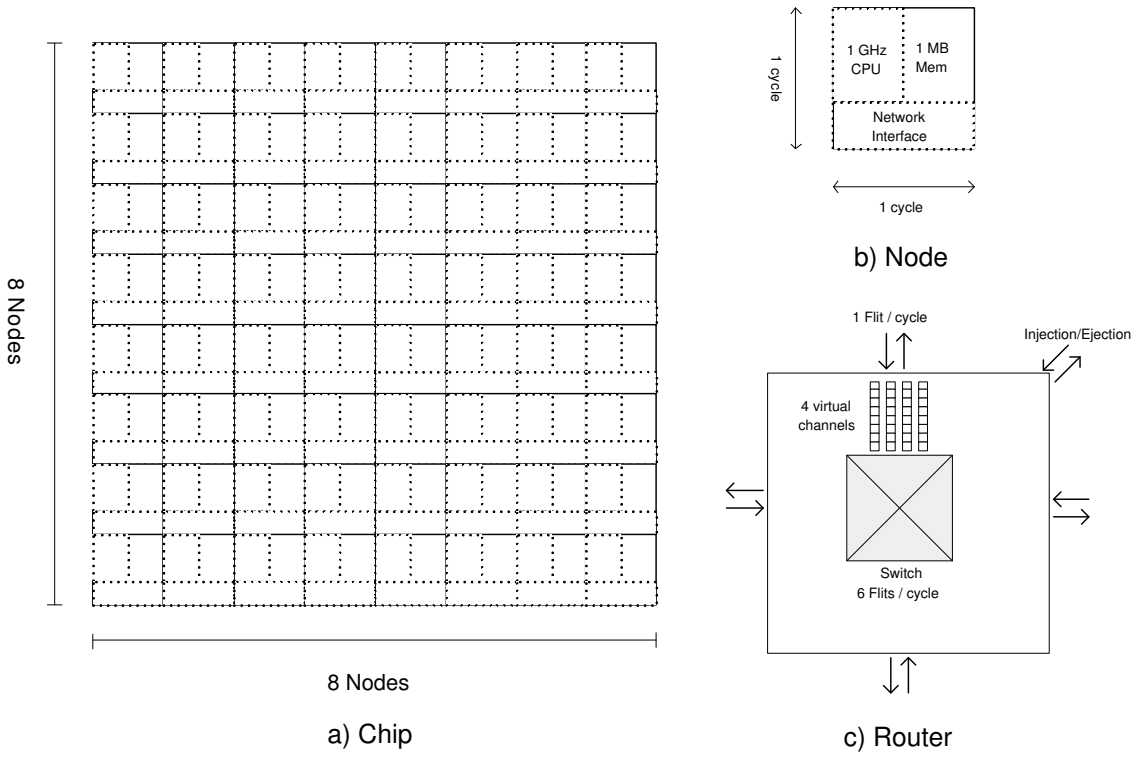


Figure 1.2: Baseline single-chip multiprocessor

this thesis, we define a single-chip multiprocessor to include the recurring features of these architectures, namely

- chips consist of multiple, independent processing nodes,
- each processing node contains limited storage which is globally accessible, and
- communication takes 1 cycle between adjacent nodes and increases linearly with Manhattan distance.

Our conclusions apply to all of these chips regardless of additional architecture dependent features.

Figure 1.2 and Table 1.2 depict the parameters of our baseline architecture. A single chip contains 64 nodes organized in an 8x8 grid. Each node contains a processor core similar to a MIPS R5000, a portion of the on-chip memory, and a network

Table 1.2: Baseline architecture for single-chip multiprocessors

Parameter	Value
Nodes	64
Processor	Single-issue, in-order, 1GHz CPU CPI=1 for all non-memory instructions
Hardware Thread Contexts	8
Per Node Memory	1 MB
Node Memory Access Time	0 cycles
On-chip Network	8x8 mesh
Flit Size	8B (equal to read request)
Per Node Physical Channels	6 (N, S, E, W, injection, ejection)
Injection Channel Buffers	16 Messages
Ejection Channel Buffers	Infinite
Virtual Channels Per Physical Channel	4
Per Virtual Channel Buffers	8
Hop Latency	1 cycle
Off-chip Memory Access Time	50 cycles

interface to the on-chip network. Specifically, each node includes a single-issue, in-order processor with 8 hardware thread contexts. The processor clock rate is relatively slow, 1 GHz. We assume a zero cycle latency for switching between thread contexts. 64MB of memory are divided equally among nodes on the chip, and each processor can access the memory on its node within a single cycle.

A mesh network connects the nodes together. Consequently, each node has six channels, one to each of its neighbors and one each for injection and ejection. Nodes can buffer up to 16 messages to be injected into the network. The channels connecting nodes are 8B wide, the size of a memory request, and operate at the same rate as the processor. Table 1.2 specifies the additional parameters determining network bandwidth. With respect to latency, the mesh network allows adjacent nodes to communicate in one processor cycle and the communication latency to remote nodes equals the number of nodes traversed times the clock cycle, without contention. Although not modeled in detail, off-chip DRAM can be accessed with a fixed latency of 50 cycles.

Table 1.3: Key differences between multi-chip and single-chip multiprocessor systems

Parameter	Multi-chip	Single-chip
Cache	Multiple Levels (32KB L1 / 1 MB L2)	Single Level (32-64KB)
Local Memory	32+ MB	1MB
Remote Memory Access	100s cycles	10s cycles (distance dependent)

1.3 Single-Chip Multiprocessor Challenges

Single-chip multiprocessors resemble the multi-chip multiprocessors examined extensively in the 1990s. Obtaining high performance in both of these systems requires distribution of work across all of the available processors. Additionally, reducing remote memory access times is essential to keeping these processors busy.

Despite these similarities, single-chip multiprocessors exhibit several distinct characteristics from multi-chip multiprocessors. First, the amount of storage available at each processor is significantly smaller than in a multi-chip system. Because a fixed area can be reached in a single clock cycle, a fixed amount of storage can be reached in a single cycle. Second, the on-chip communication network connecting processing nodes differs substantially from the off-chip networks used in multi-chip designs. Nodes on a single chip interact via a high-bandwidth, low latency network. Communication between neighboring nodes is fast (1 clock cycle) and latencies between distant nodes increase linearly with Manhattan distance. In contrast, multi-chip remote accesses took 100s of clock cycles regardless of the destination node's physical proximity to the requesting node. Finally, although single-chip multiprocessors have a high-bandwidth network, their network usage frequency is likely to outpace between-chip transfers simply due to the limited amount of per node storage. Nodes will need to collectively share all on-chip memory. Contention for network resources, therefore, will become a concern in these systems.

Table 1.3 summarizes the key differences between single- and multi-chip multiprocessors. These differences motivate re-exploration of techniques that distribute

resource demands and reduce remote memory latencies. Techniques that improved performance in the multi-chip domain do not necessarily suit the single-chip environment. For example, extensive caching enables multi-chip multiprocessors to reduce memory access latencies, however, the limited per-node storage in single-chip systems makes this approach infeasible on the same scale.

1.4 Contributions

In this thesis, we examine the two main challenges to achieving high performance in parallel architectures: distributing resource demands and reducing communication latency. These goals frequently conflict, making their simultaneous fulfillment difficult. The large number of nodes on a single chip combined with the linearly increasing latency function for communication among those nodes insure this conflict will strongly influence single-chip multiprocessor performance.

Consequently, the first component of this research examines how the underlying architecture impacts the interactions between application *objects*, a generic term used to encompass both threads and data. We create a framework for characterizing the resource demands of these objects in terms of computation, communication, and storage and characterizing the inherent communication patterns among these objects. This framework provides an architecture-independent view of the possible application traits. We use this framework to pinpoint which application traits will perform poorly on single-chip multiprocessors.

The second component of this research presents a simple migration strategy which simultaneously distributes resource demands and reduces communication latency. The technique builds specifically on the distinctive communication properties of single-chip multiprocessors. In the network we consider, communication latencies between adjacent nodes are small but increase linearly with Manhattan distance. Interacting data and threads can therefore be placed on nearby nodes without incurring large communication latency penalties. We use this property to design a migration strategy which moves data and threads to nearby nodes in order to alleviate nodes' resource demands and/or reduce an object's communication distance. Our strategy represents

these two conflicting goals as directed forces. When combined, the resulting migration force specifies the neighboring node which best satisifies both of these goals.

Because our migration strategy relies on dynamically visible communication patterns, it cannot always improve an object's locality despite the presence of locality inherent in the application; the mapping of objects onto the architecture obscures the relationships between objects. Often this occurs because different threads use the same set of data at different points in time; thus, there are connections between the data being used, but the observable connections are between each data object and the requesting threads, not among the data objects. The best solution for obtaining good performance, therefore, requires avoiding these types of mappings. We, therefore, analyze which application traits and mapping strategies obscure inherent application locality.

In the third component of our research, we present and evaluate a technique called anchors which statically modifies the application to prevent obfuscation of these inherent communication patterns. Specifically, application code is changed to expose these patterns. Threads are decomposed into sub-threads based on the clusters of data those threads use over time. Each sub-thread then executes at the location of a representative, or *anchor*, chosen from the associated data cluster. Doing so insures that communication patterns among data, not just between data and threads, become visible to potential optimizations like our migration strategy. A larger set of applications can therefore be shaped to perform well on single-chip multiprocessors.

1.5 Roadmap

The presentation of our research begins with the introduction of our application characterization framework in Chapter 2. We present the basic terminology used throughout the remainder of the thesis. In particular, we specify the resource demands exhibited by both threads and data and delineate the different types of communication relationships exhisting among these objects. In Chapter 3, we describe the synthetic benchmarks and applications used in our studies and explain how each application fits into our framework.

Chapter 4 describes our force-based migration strategy. After giving a detailed description of the migration algorithm, we use synthetic benchmarks to analyze the performance of this migration technique on different application characteristics and mappings. We then proceed to analyzing the performance of full applications. Based on our analysis, we explain the limitations and possible improvements for this technique.

Our anchor approach provides a solution for limitations of the migration strategy. Chapter 5 presents simple equations to explain the intuition behind anchors and then explores how different parameters impact the costs of applying this technique. We use full-sized applications to show how static approaches like anchors become even more important to improving performance as processors become relatively faster than communication networks.

Finally, we show how these two techniques compare to existing techniques for reducing communication latency. We show that these techniques can be used to improve performance on single-chip multiprocessors incorporating caches on each node in addition to being used on cacheless systems. Chapter 6 uses full-length applications to explore the impact of the different system designs and application characteristics on the usefulness of the three approaches. We finish by discussing related work in Chapter 7 and concluding in Chapter 8.

Chapter 2

Application Characterization

The single-chip multiprocessor architecture explored in this work exhibits several characteristics that need to be accounted for when decomposing applications into threads and data and when distributing those threads and data across the nodes of a chip. First, these chips offer large quantities of parallel processing resources. Not only can a single thread execute independently on each node, but executing additional threads per node will mask remote memory access latencies. Second, individual nodes contain limited memory but can quickly access remote on-chip memory via the fast and high bandwidth communication network. Third, these architectures can create lightweight threads quickly (around 11 cycles) and can switch between threads quickly [28].

In this chapter, we present a framework used to help decompose applications into threads and data and then distribute the threads and data across the chip. We describe the information available in each application that can be used to create informed data and thread placements. For each application we examine, we extract the resource demands exhibited by individual threads and data and the architecture-independent communication links among data and threads. This information is used throughout the rest of this dissertation to improve data and thread placements so that unevenly distributed resource demands are avoided and communication demands are reduced.

2.1 An Example Application: *barnes-hut*

Throughout this chapter, we use the *barnes-hut* application to illustrate which information we extract from applications and explain why we use that information in our framework. We first describe the application's organization and then discuss its resource demands and its inherent communication patterns.

2.1.1 Description

barnes-hut is an optimized version of the n-body problem. N-body problems simulate the influence that each body exerts on all of the other n-1 bodies in a system. For

example, galaxy simulation calculates how different bodies' masses and velocities affect one another's locations and velocities. The further bodies are apart from one another, or the less mass they have, the smaller impact they have on one another. *barnes-hut* exploits this property to reduce the amount of computation performed; multiple distant bodies are treated as a single body. Consequently, the effects of fewer bodies on a specific body's location and velocity must be computed than in the n-body case.

Figure 2.1 graphically depicts this difference between the applications for a given body G. Despite the distance between G and bodies E_4, E_5, and E_6, Figure 2.1(a) shows computation must occur between G and all six other bodies when a generic n-body algorithm is applied. In Figure 2.1(b), however, only 4 computations must be calculated; the bodies E_4, E_5, and E_6 are treated as a single body due to their distance from G.

barnes-hut organizes the n bodies into a tree structure, called an octree, that allows each node in the tree to have up to eight children. The octree's internal nodes represent a portion of the entire application space that includes several bodies; the octree's leaves represent individual bodies in the system. Bodies are positioned in the octree as children of the internal nodes that encompass their locations. Figure 2.1(c) shows the octree organization of the bodies in Figure 2.1(b). Internal nodes (I_i) represent the total mass, location, and velocity of the bodies in their subtrees. When a single internal node will have the same effect as the combination of all the elements in its subtree, the values at the internal node are used for computation.

2.1.2 Resource Demands

We now look at how our organization of *barnes-hut* into threads and data determines the computation and communication demands generated.

For each body E_i, we compute its next location and velocity based on all of the bodies with which it interacts. We quantify a body's computational demands as the number of bodies with which it interacts. In Figure 2.1(c), body G must calculate its position by looking at E_1, E_2, E_3, and I_2. E_3, on the other hand, must incorporate

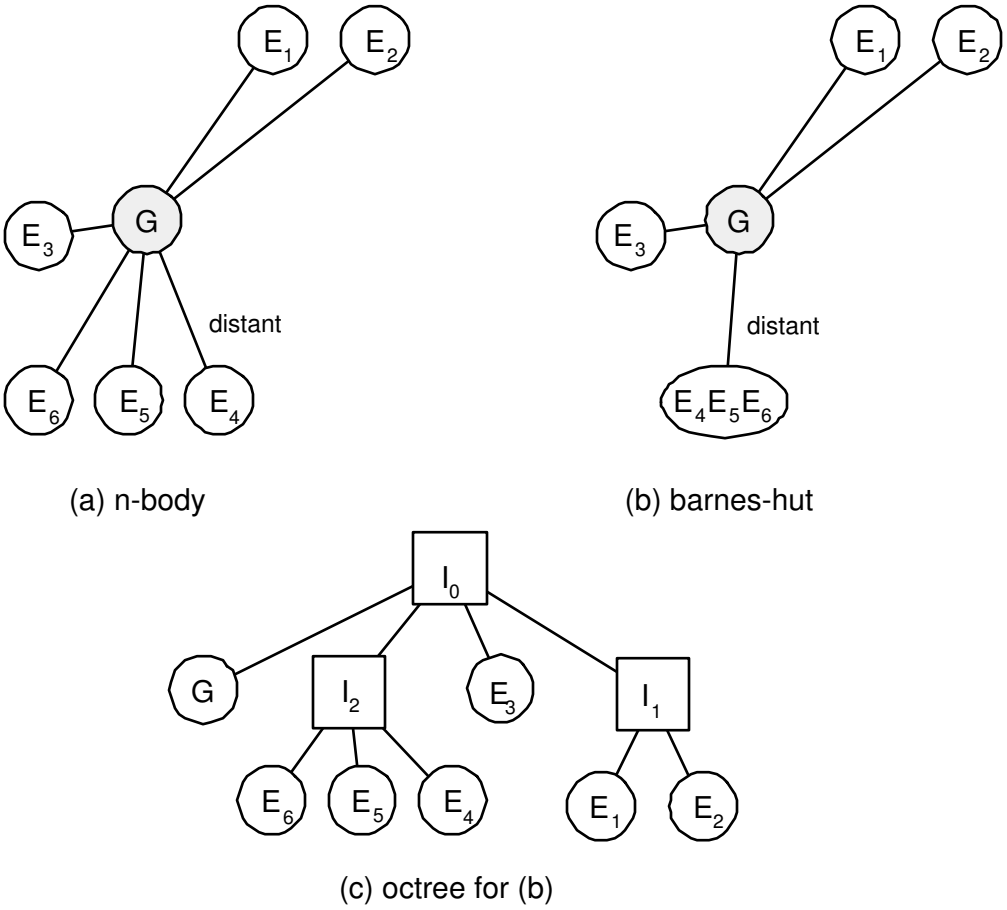


Figure 2.1: (a) shows an example graph for an n-body problem. The impact of all bodies (E_i) on body G must be calculated. Because bodies E_4, E_5, and E_6 are distant from body G, *barnes-hut* calculates their effect on G as a single entity as seen in (b). (c) shows the organization of (b) into an octree which includes internal nodes, I_i.

all of the bodies into its calculations because it is not particularly distant from any of the other bodies in the system. Because the amount of processing required to calculate each body's location and velocity at the next timestep varies for each body, it is difficult to equally distribute work across processors.

The number of data accesses, and therefore the communication demands, for computing a body's new position and velocity depend on the number of bodies included in its calculation. The amount of communication incurred, therefore, varies across the n bodies. Because of the tree structure placed on the n bodies, the inter-body communications exhibit temporal locality; as each thread traverses the tree, the elements of the tree will be accessed in the same order, adding predictability to the data accesses.

2.2 Two Program Decomposition Styles

Before proceeding with our extraction of information about resource demands and locality, we first need to discuss how single-chip multiprocessors impact decisions about decomposing applications into threads. Computation can be decomposed into small numbers of heavyweight threads that execute for long periods of time or into many lightweight threads that exist for short periods of time. The method used for attributing resource demands and communication to individual threads and data differs depending on which decomposition style is chosen.

2.2.1 Heavyweight threads

One way of decomposing a program into threads is to create one thread per processor and divide the work equally among those threads. Computations that share similar data can be computed by a single thread to improve locality. The n bodies in *barnes-hut* can be divided equally among the processor threads; bodies near one another in the octree are computed by the same thread. This approach is particularly useful when thread creation, context switching, and inter-processor communication latencies are large.

Several limitations exist for this approach. First, distributing work equally may be difficult in applications where sharing patterns change over time or are unknown statically. For example, as bodies move over time in *barnes-hut*, they will move away from previously close bodies and towards previously distant bodies. As a result, threads will have unequal amounts of work and may end up having reduced data locality despite good initial static placements. Second, attempts to take advantage of idle resources will be thwarted by the limited number of threads and the threads' associated data footprints. A single thread may have a large amount of work remaining, but it cannot be doled out to other processors. Additionally, these threads frequently accumulate large quantities of state (data); this state must be moved with the thread to obtain the best performance.

2.2.2 Lightweight threads

An alternate approach to creating one thread per processor is to create one thread per independent computation. An independent computation is defined as a series of instructions and data accesses that complete one logical action in the application. In *barnes-hut*, we can associate one thread with a single body's computation for a given timestep. In this scenario, N threads, where N represents the number of independent computations and may be larger than the number of processors, can execute in parallel. These threads can be moved between processors to take advantage of idle resources. Additionally, because lightweight threads' data footprints generally remain small, the cost of moving thread state remains manageable.

Applications written in this fashion, however, perform poorly on systems with high thread creation and communication latencies. Limited communication bandwidth may restrict thread movement. Additionally, the locality inherent in the original application can easily be lost if associations between threads and their data are not maintained. For example, the data locality created by the octree structure in *barnes-hut* can be easily lost as each body is assigned to its own thread.

2.2.3 Comparison

The single-chip multiprocessors used for our studies can execute applications decomposed either way. In order to take advantage of the available on-chip parallel resources, however, we focus on applications with many lightweight threads, each associated with a single independent computation. The remainder of this chapter focuses on extracting information about resource demands and locality for a set of such applications. We can use this information to determine how to both initially place and dynamically move data and threads on a single-chip multiprocessor to achieve high performance.

2.3 Resource demands

In single-chip multiprocessors, both computation and communication resource contention can significantly degrade application performance. An imbalanced thread distribution limits the overall speedup gained by using parallel processors. Hot spots in the network can cause communication latencies to explode, causing applications expecting small on-chip latencies to flounder. Because data and threads contribute to the overall demands for these resources, we discuss how to attribute resource demands to both of these program abstractions.

2.3.1 Computation

The use of lightweight threads that only exist long enough to complete a single independent computation poses problems for detecting and responding to dynamic knowledge about processor demands. When threads are associated with independent computations, multiple threads may be used to perform the same calculations at different times during the application's execution. In *barnes-hut*, for example, different threads will perform movement calculations for a single body during different time steps. At each new timestep, new threads are created and placed according to the application's original thread placements; consequently, all previous thread redistributions will be lost between timesteps. To obtain the benefits of thread migration

experienced by long-running threads, we must explicitly make connections between short-lived threads that perform the same computation on the same data at different points in the application's execution.

The data operated on by these threads outlives thread creations and deletions, making it an ideal mechanism for creating connections between these threads. Consequently, we associate independent computations with the data they use. This association is similar to the associations created between computation and data in object-oriented code. Consider the scenario in which an object *Foo* contains private data *bar* which is modifed by *Foo's* method *UpdateBar()*.

```
object foo {
    ...
    UpdateBar(...){
        ...
        bar = ...
    }

    private:
        bar;
}
```

The computation in *UpdateBar()* is clearly associated with a specific instance of object *Foo* and, in particular, that instance's data element *bar*. Similarly, in *barnes-hut*, we associate the computation performed for a given body with the data representing that body. This establishes a single entity that represents the total work performed by all of the body's threads over time.

One way of deciding where to execute these short-lived threads is to execute each thread at the location of the data with which the computation is associated. We can use the information gathered about the amount of computation associated with data to distribute data across nodes. Furthermore, we can insure that data is distributed so that the total amount of computation per node is roughly equal. In *barnes-hut*, the amount of computation performed to calculate each body's next location and velocity

varies depending on the body's relative location to other bodies in the system. Placing several bodies that require less computation on the same node as a body that requires the maximum amount of computation prevents that node from being assigned more than its share of total computation.

2.3.2 Communication

In addition to processor demands, both threads and data also have communication demands. Because communication is dependent on data storage locations, we decompose communication demands into two components: network resources and memory. Memory stores data while network resources enable communication of that data between threads and data.

Any time a thread performs a load or store, creates a new thread, or performs synchronization, it may require network resources. In contrast, data does not perform actions but instead has actions performed on it. Just as data can be associated with the computation that accesses it, we associate data with the network resources used to access that data. Consequently, every load or store to a specific data element may contribute to that data's network demands.

Both threads and data require memory to store their associated data. For a thread, the quantity of storage needed is the size of its stack and any additional private data. For data, the storage demands are equal to its size.

When executing applications, we can use communication resource demands to avoid resource contention caused by co-locating many frequently communicating data and threads on the same node. For example, the bodies and internal nodes at the top of *barnes-hut*'s octree are likely to be accessed frequently; to avoid network contention, they should not all be placed on a single node. Similarly, distributing threads and data with large memory sizes across multiple nodes improves the likelihood that all of the data associated with these objects will fit in local memory.

Table 2.1: Explicit object relationships created by direct communication between objects.

Object	Object	Mechanism
T	T	Synchronization
T	T	Thread creation
T	D	Memory reference

2.4 Communication patterns

The preceding resource-centric characterization of threads and data provides information which can help avoid overloaded architecture resources. However, it fails to describe the complete set of relationships between these entities. These relationships define the inherent locality within an application and, therefore, enable us to evaluate the communication demands resulting from a given placement of data and threads. In this section, we characterize the communication patterns that result from both explicit and implicit relationships among data and threads. For the remainder of this thesis, we will refer to data and threads generically as *objects*.

2.4.1 Inherent application communication patterns

Whenever communication occurs between two objects, an explicit relationship exists between these objects. Table 2.1 summarizes the instances of explicit communication between objects. When two threads synchronize with one another or a thread accesses data, these objects are explicitly linked.

However, threads and data are also implicitly linked in many more complex ways, as shown in Table 2.2. Threads that use the same data have an implicit relationship via that data. Similarly, data used by the same thread are connected to one another via the thread. These implicit relationships shape the global communication structure of an application.

Table 2.2: Implicit object relationships created by communication to two objects via an intermediate object.

Object	Object	Mechanism
T	T	Data used by multiple threads
D	D	Data used by same thread

Table 2.3: Explicit and implicit relationships in Figure 2.2 at cluster i

Relationship	Object	Object
Explicit	T	b_i.velocity
	T	b_i.mass
	T	b_i.location
Implicit	b_i.velocity	b_i.locality
	b_i.velocity	b_i.mass
	b_i.locality	b_i.mass
	b_0	b_i

2.4.2 Description of object interaction parameters

Having enumerated the set of possible implicit and explicit relationships between threads and data, we can now see how these relationships interact. In particular, we are interested in how these relationships define communication links among objects.

In general, a thread can be viewed as a sequence of instructions executing on a series of data. During different parts of a thread's execution, it may operate on different groups of data. For example, Figure 2.2 shows a thread, T, executing computation for a single body, b_0, in *barnes-hut*. The thread calculates the impact of four distinct bodies on b_0. We call the data used in each body calculation a *cluster*. Clusters represent data used together for a given time interval. In Figure 2.2, each cluster lasts for the interval associated with calculating the impact of a body b_i on b_0.

Decomposing threads in this manner exposes the different relationships and communication patterns between objects. The explicit thread/data relationship exists between a thread and the data in its associated cluster for the duration of a given

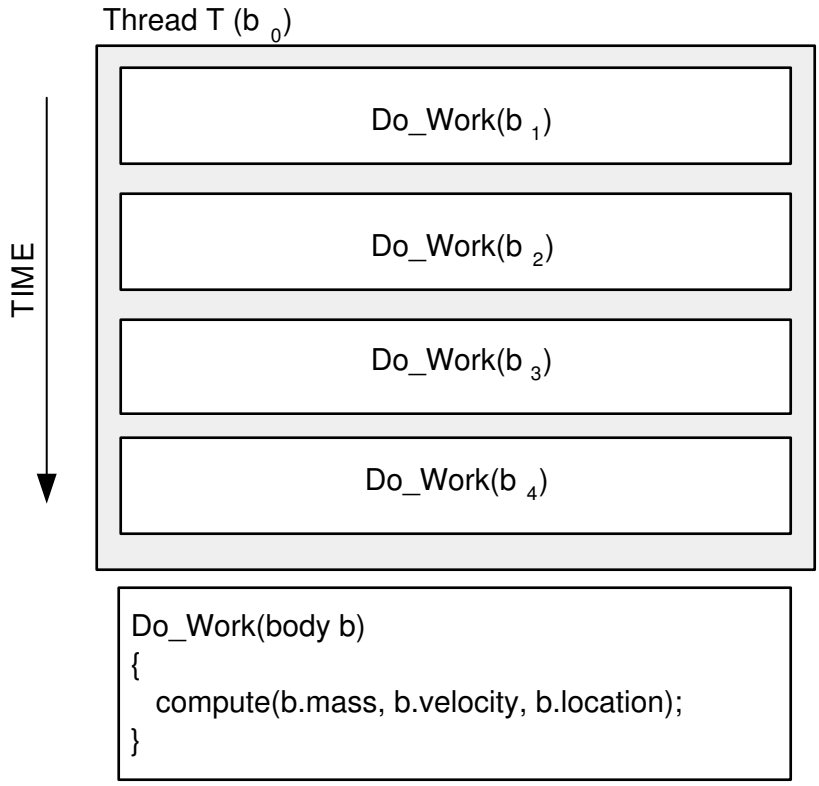


Figure 2.2: The thread associated with body b_0, thread T, accesses four different bodies when calculating b_0's next location and velocity. Because the thread accesses these bodies at non-overlapping time intervals, we say the thread uses four distinct sets, or clusters, of data.

time interval. Table 2.3 depicts the explicit relationships in Figure 2.2. Additionally, the cluster concept allows us to recognize which data have implicit relationships due to use by threads in a single time interval. In our example, there are implicit relationships among all data within a cluster as shown in Table 2.3. There also implicit relationships among body b_0's and body b_i's mass, velocity, and location data. Finally, the overlap of data in distinct clusters exposes the implicit relationships between multiple threads via shared data. Implicit relationships would exist between thread T and any other threads that access any of these four bodies.

2.4.3 Graphing Object Relationships

Based on these associations, programs can be described with a graph showing both implicit and explicit edges connecting the vertices that represent data and threads. This graph of object relationships can be used to guide placement decisions. We define four dependent parameters that help characterize the structure of a program's relationship graph. The parameters are cluster size, object access frequency, time interval size, and correlation between objects. For each of these parameters, there is a sweet spot. Making these parameters too large results in patterns being obscured; when they are too small, little useful association information can be gleaned.

Time interval

The time period used to classify clusters impacts the number of items in a cluster. The larger the interval, the more data accessed by a thread and, consequently, included in the cluster. Fortunately, programs frequently have existing abstractions, like threads, function calls, loops, and conditional statements, that create boundaries between data set changes. In our example, shifts to new elements of the octree delineated by calls to *DoWork()* mark possible cluster boundaries.

Cluster size

The size of the cluster determines both the likelihood of data being shared by multiple threads and the amount of data that must be placed on nodes near one another to

improve locality. For example, a time interval equal to a single thread's duration in *barnes-hut* creates a single cluster, effectively discarding temporal locality created by the octree. All data accessed by the thread would be included in the cluster, making it difficult to fit the entire cluster in a single node's memory.

Communication frequency between objects

The communication frequency or percentage of communication suggests the strength of relationships between objects and can be used to weight edges in our relationship graph. Objects that communicate frequently with one another have a strong relationship while infrequent communication implies a weak relationship. Exploiting stronger relationships will reduce communication demands more significantly than optimizing locality between objects with weak relationships.

Correlations between objects

The correlation between objects across multiple time intervals/clusters describes how tightly woven the objects in a program are. For example, if all threads access all data, as in a naive version of the n-body problem, implicit relationships exist between all threads and between all data; this is in addition to explicit thread/data relationships. This close interconnection does not expose particular links between objects that can be exploited when making placement decisions. In contrast, some data may always be used with another set of data and may only be accessed concurrently by a small set of threads; here, the correlations among the data and between threads and data would be high. For example, the impact of all bodies in the same galaxy would be individually calculated, while all bodies in a separate, distant galaxy would be aggregated into a single force calculation. Consequently, good placement strategies can reduce communication demands.

By graphing the relationships among data and threads, we can determine which placements minimize the number of edges that will cause inter-node communications. The graph becomes a tool that enables us to see patterns beyond a single thread/data pair; these patterns can be exploited to reduce communication demands.

2.5 Conclusions

In this chapter, we have described the information available for use in optimizing performance of applications on single-chip multiprocessors. The next chapters describe when this information is available for use, either statically or dynamically, and how we can incorporate this information to improve application performance.

Chapter 3

Application Descriptions

Throughout this thesis, we use four applications and multiple synthetic benchmarks to evaluate both the single-chip multiprocessor environment and our performance optimizations. In this chapter, we describe each of these applications based on the framework presented in Chapter 2. We describe each application's decomposition into threads, its resource demands, and its communication patterns. Our analysis enables us to pinpoint in later chapters which application traits are well-suited for this architecture and our optimizations.

3.1 Application Description

We begin by describing the problem each application solves. Table 3.1 presents a high-level description of the work performed by each applicaton. The first application, *raytrace*, renders a picture from a given viewpoint by sending rays of light through a scene. The second application, *nbody*, simulates the movement of particles over time, where the particles can be any number of logical entities such as molecules or galaxies. *barnes-hut* solves the same problem as *nbody*, however, it exploits key properties of the problem to reduce the amount of computation performed compared to *nbody*. The last application, *equake*, is taken from the SpecOMP 2001 benchmark suite [40]. It uses a finite element method to simulate the propagation of seismic waves resulting from an earthquake.

3.1.1 *raytrace*

The *raytrace* application takes as input a file describing the locations and colors of geometry in a given scene and the position of the viewer looking at the scene. The application then creates a file which depicts the portions of the scene visible from the viewer's position. The application sends rays out from the viewer's location in order to detect what the viewer can see; the color associated with the ray's origin in the final picture depends on which scene geometry the ray intersects with as it travels through the scene.

The 3-dimensional scene being viewed is divided logically into a set of small three

Table 3.1: Application descriptions

Application	Description	Input	Output
raytrace	Computer graphics application which renders a picture representing the scene components visible from a specific eye location.	Scene geometry and eye location	128x128 pixel image
nbody	Models the impact of n particles on one another's location and velocity over time. For each particle, it calculates interactions with all n-1 other particles, regardless of distance or particle mass.	1024 particles with randomly assigned mass and velocity. Particle locations have a uniform distribution.	Particle locations after 1 timestep
barnes-hut	Like *nbody*, models the impact of n particles on one another's location and velocity. It speeds up computation by grouping distant particles together and approximating their combined effect as a single particle.	1024 particles with randomly assigned mass and velocity. Particle locations have a normal distribution.	Particle locations after 1 timestep
equake	Simulates the propagation of seismic waves through large basins by using a finite element method.	Uses the test input distributed with SpecOMP2001 which includes an unstructured grid topology, seismic event characteristics, and the structure of sparse system matrix.	Displacements at the earthquake epicenter and hypocenter after 4 timesteps

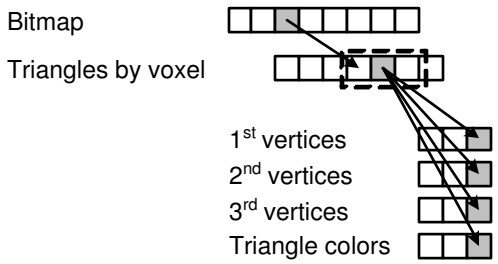


Figure 3.1: The picture shows how the different data structures in *raytrace* are accessed when the ray intersects with a triangle.

dimensional volumes called voxels. Information about the scene geometry is stored as triangles, where each triangle has three vertices and a color associated with it. The application stores this information in four arrays indexed by triangle number; three arrays store the triangles' vertices and one array stores the triangles' colors. A bitmap specifies whether any triangles are located in each voxel. Another list keeps track of all triangles within each voxel. An array stores the pixels for the final two dimensional picture.

Rays traverse the scene by moving from one voxel to another in their path. Figure 3.1 shows how the data structures are accessed. As a ray enters a voxel, the bitmap is used to determine if any triangles are located in this voxel. If the voxel includes any triangles, the list of triangles in this voxel is traversed, checking to see if the ray intersects with any. If the ray does intersect with a triangle in the voxel, the color from the geometry is written into the output data array. If the ray does not intersect with any triangles, it proceeds to the next voxel in its path. By organizing the scene and data in this manner, intersections only need to be computed for voxels that the ray travels through.

The input scene geometry used for our experiments is taken from the game Quake; we generate a 128x128 pixel image.

3.1.2 *nbody*

A brief overview of *nbody* problems was presented in Chapter 2 when we introduced the *barnes-hut* application. In this application, the movement of particles is simulated over time. Each particle has mass, location, and velocity components. Every particle affects every other particle's location and velocity in the next timestep. In general, particles that are further away in space and particles that have less mass have a smaller impact on a given particle's movement. In this application, the location and velocity of every particle are recalculated at discrete timesteps based on the interaction of all other particles. Although this computation can be performed in $nlog(n)$ time, where n is the number of particles in the system, the most straight-forward implementation, which is used in our application, takes n^2 time.

Each logical particle is represented by a particle data structure; this structure contains fields specifying the particle's location in the three dimensional space, its three dimensional velocity vector, and its mass. The data set for the entire application consists of two arrays: one array of particles for the current timestep and another array for the next timestep. The computation uses the current timestep's array of particles to calculate the next timestep's array values.

For our experiments, we model a single timestep for a 1024 particle system. The particles' initial locations are distributed uniformly.

3.1.3 *barnes-hut*

We presented the *barnes-hut* application in Chapter 2. Like *nbody*, *barnes-hut* simulates the movement of bodies over time. Unlike *nbody*, however, *barnes-hut* exploits the fact that the strength of particle interactions falls off dramatically as the distance between particles increases. Consequently, a group of distant particles can be viewed as a single particle representing the location, mass, and velocity of all of the particles in the group; we do this by using a multipole expansion of the potential field for the distant particles. Because a single group particle will represent many particles, fewer computations must be performed during each timestep than in the *nbody* application.

In order to use this problem-dependent observation, the work per timestep is broken into two unequal phases: computing the effects of particles on one another and building the structures needed to group particles together.

Like *nbody*, each particle in *barnes-hut* is represented by a particle data structure which includes the particle's three dimensional coordinates, three dimensional velocity vector, and mass. There are two arrays of particles, one for the current timestep and one for the next timestep. Unlike *nbody*, however, there are also two additional tree structures, called octrees, that organize the particles in the current and next timestep. The top node in each of these trees represents the entire application space, with each of its (up to) eight child nodes representing one eighth of that space; at each level of the tree, the space associated with a node's subtree is one eighth of the size of its parent node's space. An internal node in this tree provides a single representation for all of the particles located within its subtree. When a particle calculates the impact of all other particles on its location and velocity, it traverses the octree starting at its current location. At each non-leaf node, the computation decides whether or not the particles contained in that node's subtree can be represented as a single body or if each of the particles' impacts must be calculated separately.

In this application, we simulate a single timestep for 1024 particles. We randomly assign particles' locations from a normal distribution. This approach creates a distribution of particles such that groups of particles are distant enough from certain particles to be accounted for collectively.

3.1.4 *equake*

equake simulates the propagation of seismic waves caused by an earthquake. It consists of a single large loop which corresponds to timesteps. This loop contains a series of inner loops. The inner loops tend to occur in pairs: one to initialize data and one to perform computation on that data. Although the loops require synchronization among themselves, the individual computations in all but one of the inner loops are independent of one another. (This one non-parallel loop merges results created by earlier parallel threads into the global data arrays.) Data in *equake* is organized into

large global data arrays of relatively small elements. In addition to these global arrays, each thread of control is allocated local arrays to store their temporary results; these results are then merged back into the global arrays after all threads of control complete.

We use the test input accompanying the SpecOMP2001 code distribution. We only execute four timesteps in order to limit our simulation times to twenty-four hours.

3.2 Decomposition into Threads and Data Objects

The decomposition of applications into data and thread objects impacts the number of possible thread and data mappings onto our architecture. In order to insure sufficient thread parallelism to keep all processors busy, we assign each logical computation to its own thread. These threads are distributed equally among all processors. Additionally, depending on which is more appropriate within each application, we perform data placements based on divisions of data into logical entities or individual memory lines. Our goal is to keep associated data (for example, data in a class instance) together while limiting the size of each datum that must be mapped; our architecture's small per-node memory makes it imperative that we not waste storage space on unused data. In the remainder of this subsection, we describe the thread and data decompositions used in each application.

In the *raytrace* application, rays travel through space, intersecting with scene geometry. Each ray's computation can be performed independently. Consequently, each ray's computation is performed by exactly one thread. *raytrace*'s data set can be decomposed into a small number of logical structures discussed in Section 3.1.1. Because of the large number of elements stored in these structures and because the individual elements are relatively small, we use arrays for the underlying storage of data in these logical structures. Hence, we map data onto the architecture based on memory lines versus logical structures, and we refer to memory lines as the data objects in this application.

The work performed in the *nbody* application corresponds to calculations of each

particle's location and velocity based on interactions with all other particles in the system. The computation of a given particle's next location and velocity is completely independent of the computations performed for all other particles in a given timestep. Therefore, we create a new thread for each particle's calculation at each timestep. Each thread traverses the particles in array order starting at the particle following its associated particle. Unlike the *raytrace* application, data in *nbody* is structured as particle objects. The size of each particle object can exceed typical memory line sizes, causing us to map data based on particle object boundaries. In *nbody*, therefore, data objects are defined to be the particle data structures.

Because *barnes-hut* closely resembles *nbody*, our decomposition of it into threads and data resembles our decomposition of *nbody*. For each particle's location calculation in a given timestep, we create a new thread. During the octree rebuilding phase of each timestep, we also associate a unique thread with each particle's insertion into the new octree. The data in the application consists of particle data structures and tree node data structures. Both of these data structures include enough data to require multiple data lines; however, they are not allocated as arrays of data structures. The tree nodes are created on an as-needed basis as particles are inserted into the tree. Consequently, we use these tree nodes and particle data structures as the elements, or data objects, that must be mapped onto the architecture.

Because the control in *equake* consists of a series of inner loops, where each loop iteration can be performed independently, we assign a new thread to each inner loop iteration. The application's data is decomposed based on memory lines since it is predominantly organized as arrays or primitive types (e.g. doubles).

3.3 Resource Demands

The next component in our analysis requires examination of each application's computation, communication, and synchronization characteristics. In this section, we want to establish two things. First, we want to know which resource demands may act as performance bottlenecks in each application. Second, we want to know if variance in these resource demands might lead to imbalanced resource demands on a

Table 3.2: Cumulative breakdown of application events

Event	*raytrace*	*nbody*	*barnes-hut*	*equake*
instrs	136,608,131	224,610,302	223,073,339	202,418,788
memory instrs	6,064,376	14,692,352	13,566,436	40,534,324
comp-to-mem	22.5	15.3	16.4	5.0
synchronization	0	0	28,738	100,956
comp-to-synch	NA	NA	7,762	2,005

single-chip multiprocessor.

3.3.1 Cumulative Application Demands

Table 3.2 shows cumulative breakdowns of events for the four applications. By comparing the number of memory instructions to total instructions executed, we identify which applications may potentially suffer from remote communication latencies. *raytrace* executes a memory instruction to read-only data that is neither on the stack nor global (e.g. pointers to global arrays) every 22.5 instructions. In contrast, *barnes-hut* and *nbody* issue memory requests every 15 and 16 instructions respectively, and every fifth instruction in *equake* performs a memory request. Consequently, we expect *raytrace* to be the least negatively impacted by communication demands compared to the other applications, and we expect *equake* to be severely affected by communication latencies.

Similarly, due to communication latencies accrued by accessing remote synchronization primitives, frequent synchronization events may negatively impact performance due to communication latencies accrued by accessing remote synchronization primitives. This performance penalty is in addition to any processor cycles left idle due to serialization of threads on these primitives. Table 3.2 includes the number of synchronization events executed in each application when a single thread executes on each processor; these numbers do not reflect synchronization required for thread creation and deletion. Only *barnes-hut* and *equake* include user-level synchronization. Both the distance of communication between synchronizing threads and any

Table 3.3: Variation in thread lifetimes

	raytrace	*nbody*	*barnes-hut*	*equake*
Threads	16,384	1,024	2,048	117,728
Instructions				
Avg Thread Lifetime	8,337	219,346	214,212 / 3,633	1,719
Min Thread Lifetime	3,028	219,346	54,947 / 485	58
Max Thread Lifetime	48,521	219,346	326,094 / 12,195	65,672
Std. Dev. Thread Lifetime	4,015	0	87,082 / 1,243	6,071
Memory References				
Avg Thread Lifetime	370.1	14,348	12,664 / 584	344
Min Thread Lifetime	100	14,348	3,574 / 90	6
Max Thread Lifetime	3,096	14,348	19,089 / 1,202	14,588
Std. Dev. Thread Lifetime	262.2	0	4,888 / 102	1,354

network contention will increase this latency, potentially making synchronization a performance bottleneck for these two applications.

3.3.2 Variability

While cumulative statistics provide insight into the relative performance impact of communication, synchronization, and computation, they do not describe whether demands are distributed equally across the chip's nodes. We examine the number of threads created in each application and the variance in thread lifetimes to determine whether distributions that randomly assign equal numbers of threads to nodes will result in processor load imbalance; the larger the variability in thread lifetimes, the more likely processor load imbalance will ensue. Similarly, we examine the reference frequencies of all data objects to determine whether a random distribution of data across nodes would result in imbalanced communication demands.

Threads

Table 3.3 presents the minimum, average, maximum, and standard deviation of thread lifetimes in terms of both instructions and memory references. (Note: Because threads

perform only one independent computation during their lifetime, these numbers are equivalent to the number of instructions and memory references between barrier synchronizations when heavyweight threads are used.) The number of threads created and their respective lifetimes differ significantly among the applications. *raytrace* consists of a large number of relatively short-lived threads in comparison to *nbody*'s and *barnes-hut*'s small number of long-lived threads. *equake* creates the largest number of threads which, on average, live for relatively short periods of time. The variance in thread lifetimes and thread data references, expressed by the standard deviations values in Table 3.3, distinguish which applications may suffer from unequally distributed processor demands. All threads execute the same number of instructions and memory references in the *nbody* application; consequently, all processors will initially be assigned equal quantities of computation. In contrast, the remaining three applications display high variance in instruction and memory reference counts. Figures 3.2- 3.4 show histograms for these applications' thread instruction lifetimes.

Threads in *raytrace* range from executing slightly more than 3,000 instructions to executing more than 48,000 instructions. Figure 3.2 shows that despite this wide variance, three quarters of the threads execute fewer than 9,000 instructions. If the remaining one quarter of threads are disproportionately allocated to a small number of nodes, processor load imbalance would surely result.

Although the threads in *barnes-hut* also display a range of thread lifetimes, it is important to note a key characteristic created by the application's structure. Figure 3.3(a) shows that threads in *barnes-hut* can be divided into two distinct groups. Table 3.3 shows that threads in the first group execute 3,633 instructions on average while the average for the second group is 214,212 instructions. These two groups coincide with the two phases of computation per timestep: rebuilding the octree and determining particles' next timestep values. Since these two phases do not overlap, we must isolate the threads in each phase to determine whether thread variability exists within a given phase. Threads created for calculating particles' next timestep values have larger variances in instruction count lifetimes than those created in the rebuilding phase. In Figure 3.3(b), we use a larger x-axis scale to illuminate the variance in thread lifetimes for these threads. Consequently, while both phases of

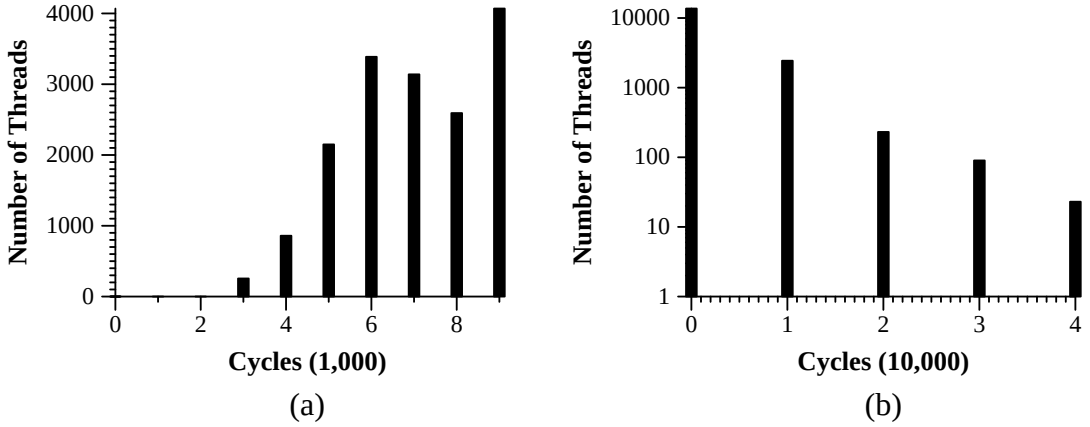


Figure 3.2: Of the 16K threads created in *raytrace*, one quarter execute more than the average number of instructions per lifetime as seen in (a). However, (b) shows that some threads execute more than 40,000 instructions.

execution may suffer from processor load imbalance, the first phase of computation - calculating next timestep values - will result in larger imbalances.

Like *barnes-hut*, *equake* can be decomposed into five distinct phases. The first phase consists of 7,294 threads which each execute 58 instructions. The second phase includes 256 threads that execute 65,672 instructions. Like these two phases, the fifth phase contains threads that execute identical numbers of instructions, 7,294 threads executing 1,012 instructions. Threads in the third and fourth phases have variable instruction lifetimes; however, the variance is small. Consequently, while *equake* may have some processor load imbalance in these two phases, the third phase in particular, the application is less likely than *raytrace* and *barnes-hut* to suffer from imbalanced processor demands.

Data

To examine the communication demands created by the placement of data across the nodes of a single-chip multiprocessor, we examine the reference frequencies of data objects in the four applications. Applications with large discrepancies in the number of accesses of data objects may suffer from network contention if a large fraction of highly accessed data reside on a small set of nodes.

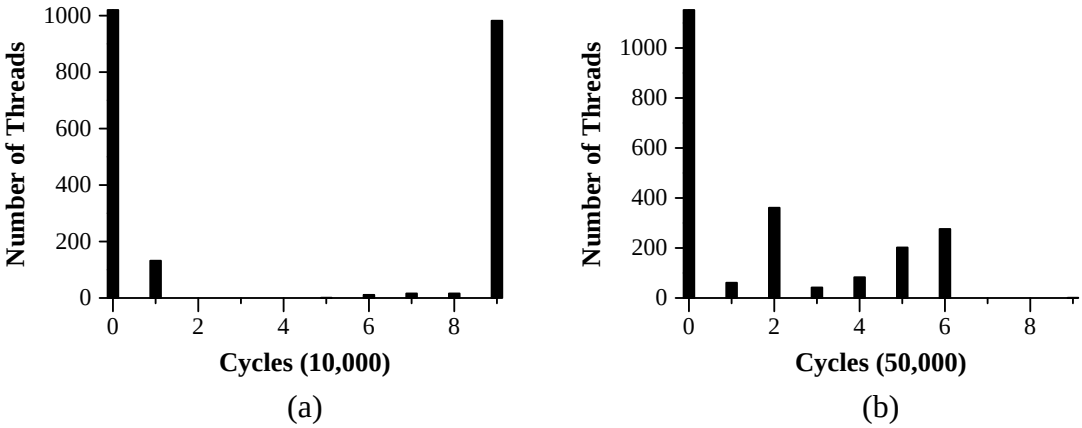


Figure 3.3: Threads in *barnes-hut* can be divided into two groups as seen in (a), where each group is associated with one of the two phases of computation in the application. In (b), we use a larger x-axis granularity to identify the instruction variance across threads belonging to the more computationally intensive phase.

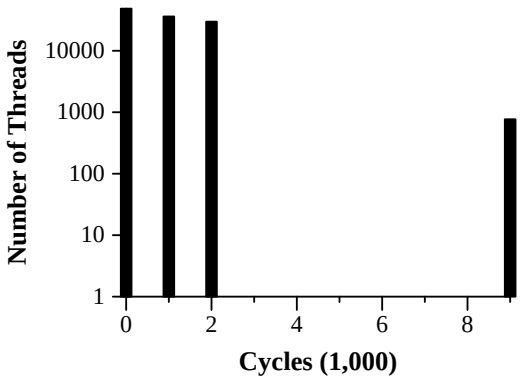


Figure 3.4: *equake* consists of five phases. Three of these phases do not have variance in thread instruction lifetimes; all threads execute 58, 65,672, and 1012 instructions respectively. In the two remaining phases, threads execute between 500 and 3,000 instructions, creating a small amount of variance.

Table 3.4: Cumulative data reference frequencies

	raytrace	*nbody*	*barnes-hut*	*equake*
Total data objects	116,914	2,048	3,279	1,137,421
Total referenced data objects	9,745	2,048	3,007	157,674

We begin our examination by looking at the percentage of the applications' data sets that are actually accessed. Table 3.4 shows that while the *nbody* application accesses all of the data in its data set, the other three applications access their data sets selectively. In particular, the *raytrace* and *equake* applications only access small fractions of their entire data sets.

Figures 3.5-3.7 depict the reference frequencies for objects accessed at least once in the different applications. We do not show histograms for the *nbody* application because all particles are referenced an equal number of times; consequently, all nodes receive equal numbers of requests, as long as particle objects are equally distributed among the nodes. The histograms show that in the remaining three applications, variability exists in the number of accesses to data objects.

As seen in Figure 3.5, most data objects in *raytrace* are requested fewer than 1,000 times over the course of the application. A select number of data objects are accessed more frequently, including some experiencing more than 9,000 references. When we consider this variability with the fact that 9,745 out of 116,914 objects are not referenced at all, we conclude that imbalanced communication demands and network contention may result from data object placements. Figure 3.7 shows that *equake* resembles *raytrace* in that the majority of data exhibit identical reference frequencies. However, the minority of objects with different reference frequencies may generate unequal communication demands across the nodes. Finally, while *barnes-hut* accesses most of the data in its data set, similar to the *nbody* application, the frequency of accesses to its data varies greatly as seen in Figure 3.6. Therefore, unlike *nbody*, it may suffer from unevenly distributed communication demands.

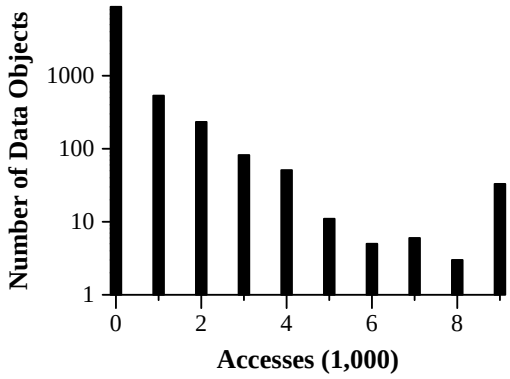


Figure 3.5: Most accessed data objects in *raytrace* are referenced fewer than 1,000 times. However, a small fraction of the objects are accessed substantially more times.

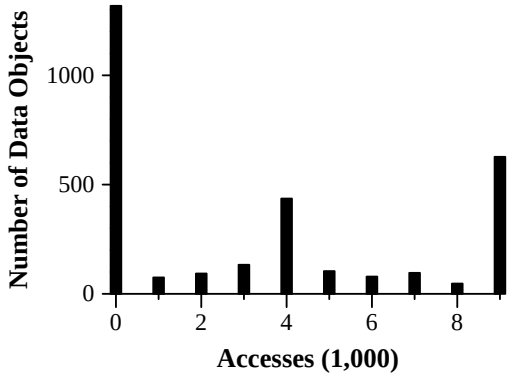


Figure 3.6: The number of references to data in *barnes-hut* varies greatly. More than 500 objects are accessed more than 10,000 times.

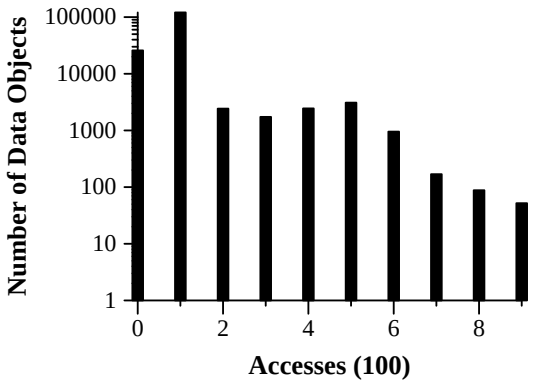


Figure 3.7: Like *raytrace*, most objects in *equake* have identical reference frequencies; however, a fraction of the data objects are accessed at significantly larger rates.

3.4 Communication Patterns

In the last component of our application analysis, we examine the communication among data and thread objects. We wish to understand how closely connected these objects are to one another. Depending on the relationships between these objects, different mappings of objects onto a chip may result in significantly different inter-node communication demands. When small groups of objects are strongly connected to one another and have looser ties to the rest of the objects in the application, we can reduce communication demands by placing objects in the connected groups on neighboring nodes. Therefore, we want to determine whether these applications contain these small, tightly-connected groups of objects.

For this analysis, we determine the number of clusters each thread consists of, as well as the similarities between threads' clusters. In particular, we want to ascertain whether or not small sets of data are consistently used across different threads and across time. Additionally, we want to know how large this group of data is. Table 3.5 presents the numbers used for the following discussion.

Direct Connections

The direct connections between data and threads can easily be described by two numbers: the average number of unique data referenced by a single thread and the

average number of threads that reference each data. To gather this information, we collect information about either the number of unique memory lines or the number of data structures accessed, depending on how data was decomposed for the given application. In Section 3.2, we specified that data in *raytrace* and *equake* was tracked on a per line basis, while it was tracked on a data structure basis for *nbody* and *barnes-hut*.

According to Table 3.5, threads access an average of 35.2 data objects in *raytrace*, 1025 in *nbody*, 565 in *barnes-hut*, and 82.5 in *equake*. We can generate a graph where every thread and data object is a vertex and edges connect threads to the data objects that they access. Each data vertex has edges connecting it to 59.2 thread vertices in *raytrace*, 512.5 in *nbody*, 197 in *barnes-hut*, and 59.2 in *equake*.

In general, the more edges each vertex has, the more closely knit the graph becomes. It becomes more difficult to divide the graph into smaller subgraphs connected via a small number of edges; this is because all vertices will be connected to a large number of other vertices. In terms of data placement on a single-chip multiprocessor, when the graph is more closely-connected, it becomes difficult to distribute data and thread objects across nodes without having large quantities of communication (edges) between those nodes. Therefore, an initial observation based on the direct connections among data and threads is that it may be harder to partition *barnes-hut* and *nbody* across a chip without introducing large quantities of global communication. It should be easier to perform the same procedure on the *raytrace* and *equake* applications.

Grouping Implicit Connections via Clusters

A graph of connections among data and threads presents a global view of how all threads and data are connected, but it does not include temporal information; it does not tell us if all data used by a thread are all being used at the same time. In Chapter 2, we presented the concept of a cluster; clusters group data together based on implicit connections created via an accessing thread and time. We now estimate the size of the logical clusters among data used within threads.

We begin our analysis by examining the average interval of time between a thread's

Table 3.5: Application communication patterns

	raytrace	*nbody*	*barnes-hut*	*equake*
Threads	16,384	1,024	2,048	117,728
Avg. Thread Lifetime (instrs)	8,337	219,346	214,211 / 3,644	1,719
Average no. unique data referenced per thread	35.2	1,025	565 / 12	82.5
Average no. of threads referencing each data	59.2	512.5	197	59.2
Average interval of references (instrs)	1081	301.8	2704	57.8

first and last reference of a single data object. Our goal is to discover the number of data clusters comprising each thread. If individual data objects are only referenced for a short duration, this can indicate that the thread accesses different data sets over time. Table 3.5 shows the average interval length for all data across all threads. By simply dividing this interval by the average thread lifetime, we create a rough estimate of the number of logical clusters contained within the thread lifetime. For example, the average reference interval for data is 1/8 of the average thread lifetime in *raytrace*, 1/100 in *barnes-hut*, 1/1000 in *nbody* and 3/100 in *equake*. These numbers imply that *raytrace* consists of a small number of clusters, but the threads in the other applications are composed of large numbers of distinct logical clusters. Based on the average number of unique data accessed by threads in each application, we can also conclude that the average logical cluster size is quite small. The cluster size ranges from a single data object in *nbody* to five data objects in *barnes-hut*.

Our analysis suggests that many of the implicit connections between data accessed by the threads may be disregarded because the data are not used during overlapping time intervals; the data only share a connection because computation was grouped together by the applications' current thread decompositions. For example, each thread in *nbody* accesses each of the data structures representing the other n-1 particles in the application. However, the calculation of particle i's impact on a thread's particle is completely independent of the calculation of particle $i + 1$'s impact on the thread's

particle. Particle i and particle $i+1$ have an implicit connection due to the thread that accesses them both, but the lack of overlap in their accesses implies this connection is tenuous. Connections between these loosely connected data may be ignored when partitioning the data; however, communication between these data and their corresponding thread will continue to exist and must be accounted for in partitioning strategies.

Repeatability of clusters

We are also interested in determining whether or not clusters recur. When a thread accesses a set of data together, will other threads that access a piece of the data in that set access all of the other data in that set as well? To answer this question, we examined each data object and compared the data sets used by their accessing threads. We counted the number of those threads that accessed more than 90% of the same data; we wanted to find out if these threads had more than 90% similarity in their data sets. Trivially, a large number of the objects in *barnes-hut* and *nbody* met this criteria simply because an average thread accesses more than half of the entire data set in those applications. Of greater interest are the results for the two applications where threads access a small fraction of the total data set. In Figure 3.8, we show the percentage of a data object's accessing threads in the *raytrace* application that have at least 90% data set similarity with the other threads accessing that same data object. We can see that more than 4,300 (out of 9,745) data objects have 70% of their associated threads meeting this criteria. Figure 3.9 shows that more than half of the data objects in *equake* are accessed by threads that have 90% similarity across their data sets. This implies that data is repeatedly accessed in well-defined sets across different threads and across time. The resulting data clusters can be treated as single entities which have strong connections to one another and weaker connections to other data and thread objects.

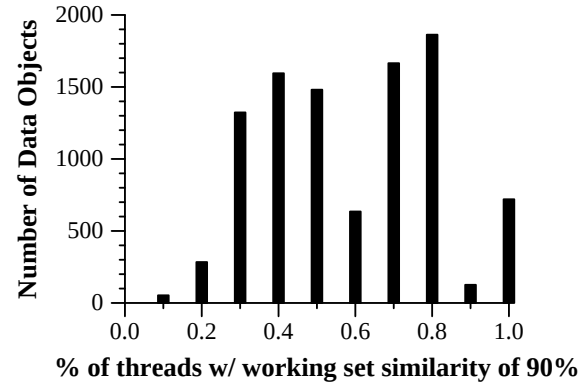


Figure 3.8: For each data object in *raytrace*, we determine the fraction of its accessing threads that have 90% or greater similarity among their working data sets.

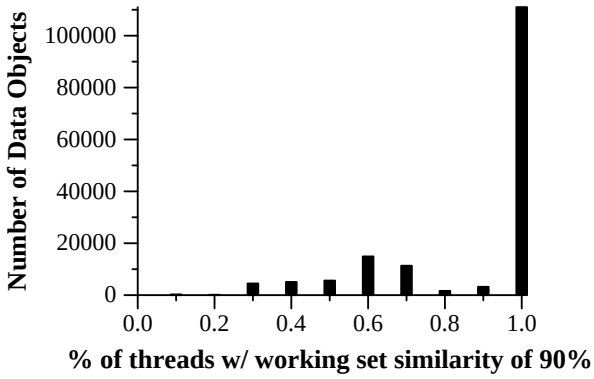


Figure 3.9: For each data object in *equake*, we determine the fraction of its accessing threads that have 90% or greater similarity among their working data sets.

Table 3.6: Summary of application characteristics

	raytrace	*nbody*	*barnes-hut*	*equake*
Decomposition - thread	many	few	few	many
Decomposition - data	memory lines	objects	objects	memory lines
Performance bottlenecks	proc	proc / synch	proc	memory / synch
Variability - processor	yes	no	yes	some
Variability - communication	yes	no	yes	yes
Number of logical clusters per thread	8	1000	100	33
Data similarity across threads	yes	n/a	n/a	yes

3.5 Summary

Table 3.6 presents a brief summary of the application characteristics described in this chapter. In future chapters, we explore how to overcome potential bottlenecks such as imbalanced resource demands and how to exploit the knowledge exposed by the cluster concept to reduce global communication demands.

Chapter 4

Reactive Approach - Migration

In the preceding two chapters, we describe the inherent characteristics of applications in general and of the particular applications used in this dissertation. This information provides an overview of the types of object interactions and resource demands exhibited by an application. The real bottlenecks for an application, however, cannot be known until runtime. An application's performance on a given architecture depends on how its constituent components, threads and data, are distributed on the architecture. Only then can we determine which resources become overutilized and how object interactions manifest themselves into network communication.

In this chapter, we explore how to optimize application performance using only the information available at runtime. We present a technique that uses runtime information to quickly and frequently migrate data and threads away from overloaded resources and towards the objects with which they communicate. Our technique represents the two competing goals of distributing resource demands and reducing communication distance as two directed forces. When these forces are combined, the resulting directed force specifies the best migration destination that simultaneously satifisfies both goals.

Before presenting our migration strategy in detail, we first discuss the application characteristics that can be observed and quantified at runtime.

4.1 Impact of Object Placement on Runtime Information

The placement of objects on a given topology determines the demands experienced by individual nodes' processor, memory, and network resources. Because placement determines the frequency and distance of communication between objects, it also dictates the demands placed on different parts of the on-chip network. In this subsection, we examine how, for a given topology, different object placements change the information that can be detected and exploited by runtime optimizations. The nodes in the single-chip multiprocessor that we study are organized as a mesh, with messages routed along the x- and y- dimensions.

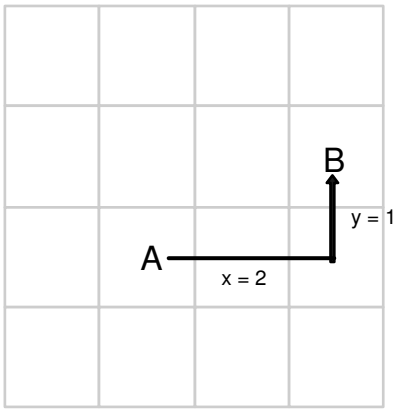


Figure 4.1: In a mesh, Manhattan distance expresses communication distance. Each communication can therefore be decomposed into its x- and y- dimension components.

4.1.1 Observable communication patterns

Our ability to dynamically detect the inherent communication patterns between objects depends on the amount of state we are willing to collect in hardware. Although every message contains both the sender and receiver as well as these entities' respective node locations, retaining all of this information for every object over a given time interval would require a great deal of storage. Additionally, processing this amount of state to make migration decisions would take a significant amount of time.

Instead of retaining all of this state, for each object we collect the frequency of communication along the x- and y- dimensions. Because the nodes of our single-chip multiprocessor are organized as a mesh, we can consider each communication to exhibit directional characteristics. For example, each communication from object A to object B in the 4x4 mesh depicted in Figure 4.1 traverses two hops to the east and one hop north. Each communication between objects can be described by its origin and its x- and y- dimension components. By collecting the frequency of communication along each dimension, we can infer the relative location of most of the objects with which a given object interacts.

Combined, this information about the frequency and direction of communication supplies a simple view of each object's communication patterns. However, the penalty

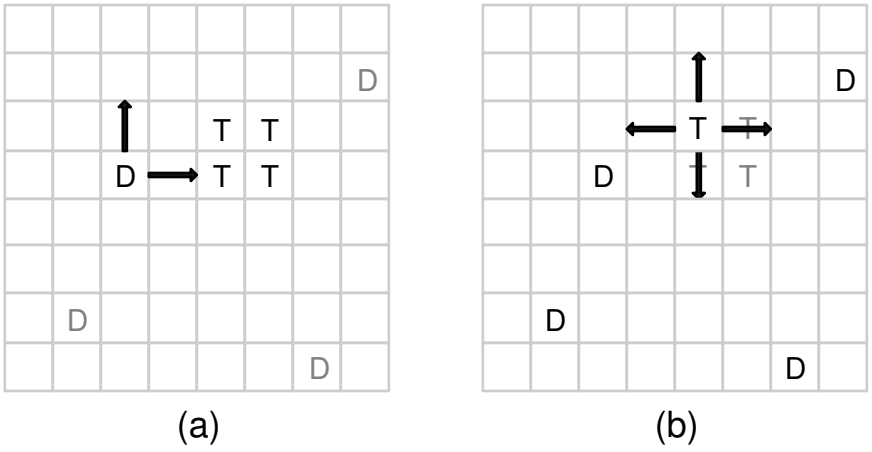


Figure 4.2: Communication Patterns: The highlighted data in (a) exhibits clear locality preferences; it is pulled towards the four threads. In contrast, the highlighted thread in (b) accesses data in all four directions and, therefore, has unclear communication patterns.

for using only these two parameters, direction and frequency, to describe an object's communication patterns is that we cannot determine and therefore must sacrifice much of the locality information observed in the original application. Instead of being able to determine which specific objects communicate explicitly or implicitly, we can only determine the relative physical position of objects that communicate with one another.

Using this simple collection technique, we can only make one observation about an object's locality preferences - whether or not an object has a clear communication pattern. An object that communicates predominately in one direction of a given dimension can be viewed as having a *clear* communication pattern while an object that communicates equally in both directions of a given dimension has an *unclear* communication pattern. When objects have clear communication patterns, we can determine which part of the chip with which they communicate; the object's locality preferences can be distinguished. For example, in Figure 4.2(a), all four threads access the highlighted data. Consequently, the data is pulled towards those threads, exhibiting a clear locality preference. When objects have unclear communication patterns, they

communicate in conflicting directions in a given dimension. The highlighted thread in 4.2(b) accesses all four data. The thread is pulled N, E, W, and S; there is no clearly defined chip direction in which the thread communicates.

4.1.2 Nodes' resource demands

Because processor, memory, and network contention can all significantly impact performance in future chips, we want to collect runtime information for all three of these resources. There are a number of possible metrics for gauging processor load; in this work, we simply track the number of threads concurrently executing on a node. The communication resource load is broken into several components: the number of attempted injections into the network, the number of successful injections, and the number of cycles the processor or memory were unable to send a message. Each node's memory load is designated by the percentage of used memory versus the total amount of memory available.

4.2 Migrating Based on Directed Forces

Given the runtime information described in Section 4.1, how do we use it to optimize execution time? In the approach presented here, locality and resource load distribution are viewed as competing forces: attraction and repulsion forces. *Attraction forces* pull interacting data and threads together, improving locality. *Repulsion forces* push objects away from locations with high resource demands, reducing individual nodes' resource demands. To obtain the best application performance, our technique migrates objects in a way that simultaneously improves locality and distributes resource load.

Figure 4.3 shows a simple example where the attraction forces pull a thread north and west towards the data it is accessing (Figure 4.3(a)) while the repulsion forces created by high resource demands on the shaded nodes push the thread to the north and east (Figure 4.3(b)). Combining these forces results in the thread migrating north (Figure 4.3(c)), improving locality while redistributing load.

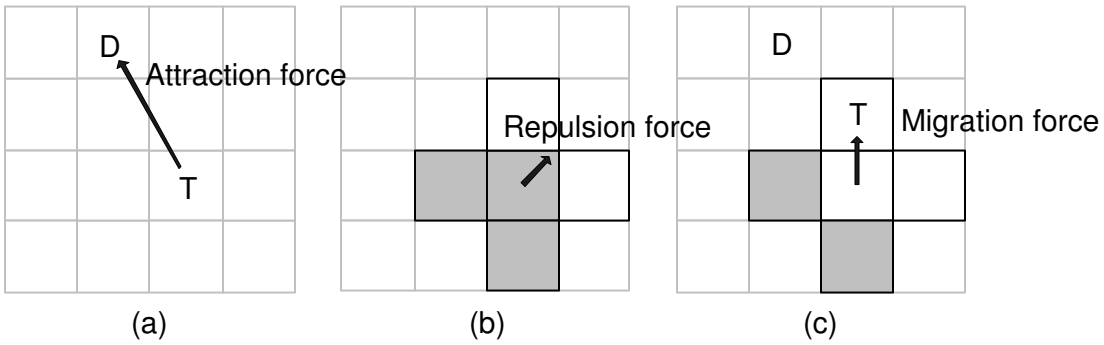


Figure 4.3: Thread T is attracted towards data D as shown in (a). In (b), the three shaded nodes' resources are overloaded, causing a repulsion force to the north and east. When we combine these forces, the conflicting horizontal components cancel out and thread T moves north as seen in (c).

In the remainder of this chapter, we show that by modeling locality and resource demands with directed forces, we can still improve performance using the limited and topology-dependent runtime information. We describe our technique in terms of two distinct components: the movement policy and the invocation policy. The movement policy determines *where* to migrate objects; the invocation policy determines *when* and *which* objects to consider for migration.

4.2.1 Movement Policy

A movement policy both defines an object's possible migration destinations and chooses one destination from that set. The more nodes included in the destination set, the more state collected, stored, and processed. Consequently, considering larger destination sets takes more network, memory, and computation resources than smaller sets.

Depending on the topology and the size of the destination set, state exchange messages and object migration messages may travel over multiple channels in the network. For example, a migration set of 4 nodes in a mesh can be reached in one channel traversal while multiple channels must be traversed to reach any larger set of nodes. For a given topology, the larger the destination set, the more network

bandwidth used collecting state and transmitting objects. Additionally, the more channels traversed, the longer an object spends migrating instead of doing useful work.

The choice of the destination set impacts the construction of attraction and repulsion forces. Given our destination set, we only need to consider the state of possible destinations. The movement policy we use assumes a mesh topology and permits only single hop migrations to any neighboring node: north, south, east, or west.

Force Calculations

In general, reducing the number of hops traversed by a message reduces the distance the communication travels and, hence, its latency. Because we do not have complete locality information available statically at runtime, we use the direction of each communication to specify where to migrate an object to improve locality.

We combine the directional information associated with each communication and the frequency of communication information to create *attraction forces* for each object. By adding the number of communications in each dimension (where communications in opposing directions cancel out) for a given object, we construct a measure indicating how much that object's cumulative communication distance would decrease if the distance for each communication in one dimension was reduced by one hop. This measure is the magnitude of the attraction force for the given object in that dimension. For each object, we calculate the attraction force for both the x- and y- dimensions.

To determine these attraction forces at runtime, we collect communication statistics for each object using four counters. Two of these counters are assigned to the x- and y- dimensions. Because communication in opposite directions of a given dimension cancel out one another's pull in that dimension, only one counter is necessary. Communication along one direction in a dimension will increment this counter while communication in the dimension's opposing direction will decrement it. For example, assuming east is the positive direction, if an object communicated 50 times to its east and 30 times to its west, the counter would be set to 50-30=20.

The remaining two counters collect statistics for communication from nodes that

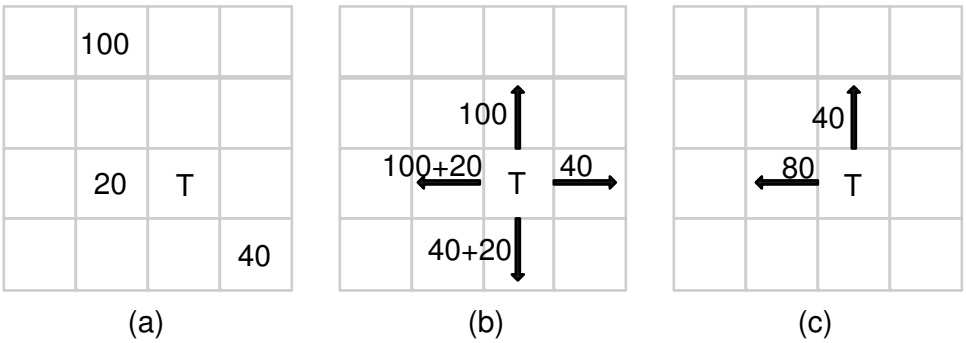


Figure 4.4: (a) depicts thread T's communication frequency with objects on 3 other nodes. This communication leads to the forces in (b) pulling on thread T. Thread T's overall attraction forces for each dimension are shown in (c).

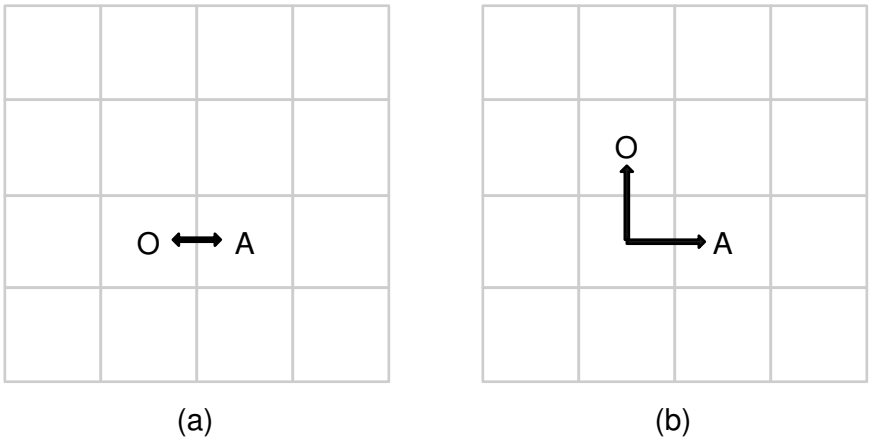


Figure 4.5: In (a), objects O and A communicate with one another, incurring a communication latency of 1 hop. If object O moves in the y dimension as seen in (b), this latency increases to 2 hops.

reside in the same row or column as the object. In Figure 4.5(a), object A resides in the same row as the object O. Communication between these objects takes one hop latency along the x dimension; there is no y dimension component. Consider how the communication latency changes when object O moves one hop in the y dimension as seen in Figure 4.5(b). Each communication between the two objects will take one additional hop latency. Consequently, object A wants to discourage object O from moving in the y dimension. Therefore, we collect statistics for the number of communications in the same row and column as the object O. The values for these counters will be combined (added or subtracted depending on sign) with the counter values for their appropriate dimension counter so that the absolute value of the dimension counter decreases by the absolute value of the row or column counter; the row counter gets combined with the y dimension counter and the column counter is combined with the x dimension counter.

Figure 4.4 illustrates the calculation of attraction forces for thread T. Because migration away from the current row will increase the latency of the 20 communications to the west, a force with a magnitude of 20 opposes vertical migration. Figure 4.4(b) shows that the combined forces pull thread T north with a force of 100 and south with a force of 20+40=60. The western and northwestern nodes also combine to pull thread T west with a magnitude of 20+100=120, while the eastern node exerts a force of 40. Figure 4.4(c) depicts the overall attraction forces.

While attraction forces are associated with specific objects, *repulsion forces* are associated with regions of nodes; in our case, this region is a node and its four neighbors. Nodes within a region exchange resource demand statistics at specified time intervals. If there is contention for a node's resources, the node is considered overloaded. Based on the relative loads of nodes in the region, a non-zero magnitude is assigned to nodes with overloaded resources. For each direction, we then create a repulsion force which weights (w_{res}) and combines the loads (l_{res}) for several resources and then multiplies them by a unit vector ($\overrightarrow{v_{dir}}$):

$$\overrightarrow{repulsion_{dir}} = \left(\sum w_{res} l_{res}\right) \times \overrightarrow{v_{dir}} \tag{4.1}$$

The magnitude of this force represents the strength of the force pushing objects away from the center node to the node in the specified direction.

In the strategy used for our studies, nodes exchange information about processor and communication resources and weight them equally. In general, we only want to push work away from a node if that node is experiencing contention for one of its resources. For our studies, we did not run into contention for memory so we opted to focus on processor and communication resources. We set thresholds to determine if these resources are overloaded and then add in some hysteresis. In order to prevent oscillations between nodes with equal numbers of threads, we only consider a neighboring node's processor resources to be below our threshold if it has at least two fewer threads than the center node in the region. Only in this case are the center node's processor resources considered overloaded. A node's communication resources are considered overloaded if 15% of the node's injections fail or if the processor or memory stall due to network contention more than 50% of the time.

Figure 4.6 depicts a 4x4 chip with a single region of five nodes highlighted. The first two figures show the processor and communication demands for the nodes in the region. The processor demands are specified in terms of the number of threads on each node; communication demands are specified in terms of the percentage of failed injections. The two nodes with processor demands of one thread are considered underutilized while the two nodes with injection failure percentages higher than 15% are considered overloaded in terms of communication demands. The last picture shows that when both processor and communication demands are considered, only one of the nodes (the non-shaded node) is considered underutilitized.

Force Combination

We combine attraction and repulsions forces together using vector addition, where the resulting vector is projected along the x- and y-axes. However, we add one additional constraint. We only want to move objects when a performance advantage will result, i.e. decreased communication distance and/or reduced resource contention. Therefore, we sometimes limit the impact of neighboring nodes' repulsion forces when the center node of a region does not exhibit resource contention.

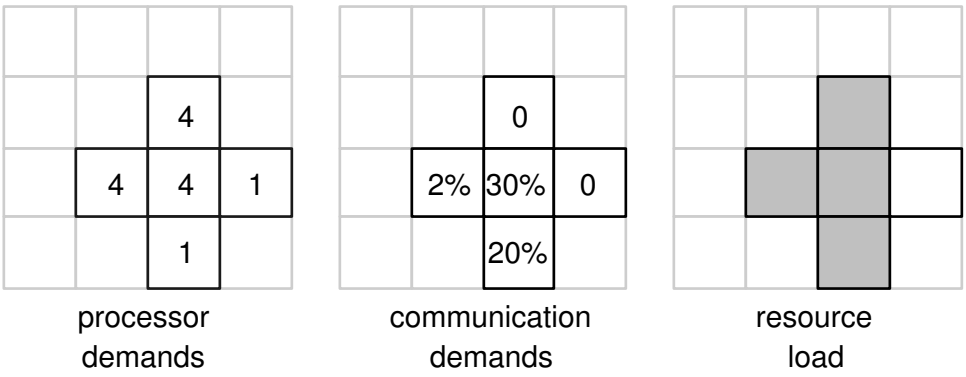


Figure 4.6: For the given region, we show the processor demands in terms of number of threads and communication demands in terms of the percentage of injections that fail. The final figure shows that four of the nodes are considered overloaded (shaded) when both processor and communication demands are accounted for.

The four possible combinations of attraction and repulsion forces are shown pictorially in Figure 4.7. For simplicity, we show calculations only for the vertical dimension. Forces are shown by arrows and overloaded nodes are shaded.

When only one of the two forces in a dimension has non-zero magnitude, the migration force pushes in the direction of the non-zero force as seen in Figures 4.7(a)-(b). When both forces have non-zero magnitudes and their directions are the same, they combine to push the object in that direction as seen in Figure 4.7(c). When the forces conflict as seen in Figure 4.7(d), the repulsion force is subtracted from the attraction force to obtain the migration force. Depending on the relative strengths of the two conflicting forces, the object may migrate to either of its neighbors or remain at its current location. This strategy allows a strongly attracted object to move to an overloaded node if the attraction force is stronger than the repulsion force.

After the migration forces for each dimension are calculated, the force with larger magnitude is chosen as the migration direction. The object is migrated in that direction if the expected benefit of moving the object exceeds the communication latency incurred to move the object.

Figure 4.8 shows a simple example where tradeoffs are made between locality and load balance. Both the thread T_1 and the data D_1 in Figure 4.8(a) want to migrate to

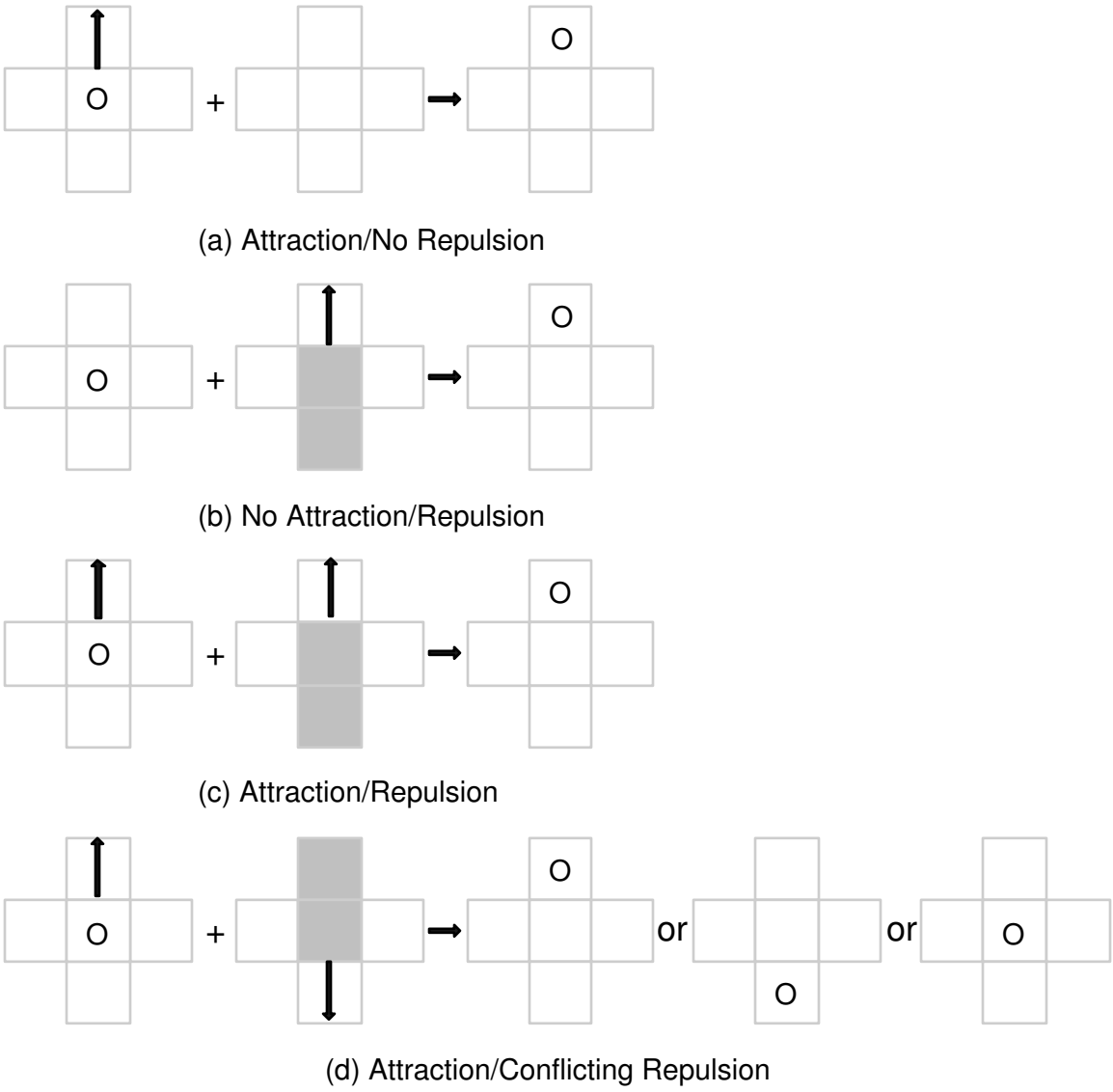


Figure 4.7: When attraction and repulsion forces do not conflict, object O is pushed in the direction of those forces as shown in (a)-(c). However, when forces conflict, their respective strengths determine the object's migration direction as seen in (d).

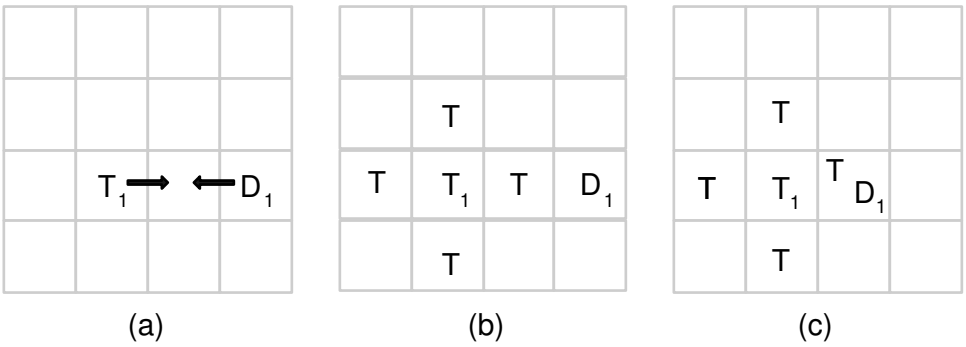


Figure 4.8: Thread T_1 and data D_1 are pulled towards the intervening node in (a). As shown in (b), processor demands in the region are already balanced. Consequently, only D_1 moves in (c).

the node between them to improve locality. However, the region's processor demands are already balanced as seen in Figure 4.8(b) and any migration of T_1 would create an imbalance. Figure 4.8 shows that while thread T_1 does not move, locality still improves through D_1's migration.

4.2.2 Invocation Policy

In addition to determining where to move objects, migration schemes must specify which objects to consider for migration and how often to examine them. A *static invocation policy* considers objects for migration at specified time intervals, cycling through a list of objects. A drawback of this approach is that objects that are not used frequently use decision slots, increasing the time between the migration of objects that would improve performance. In contrast, a *time varying policy* dynamically determines when to migrate data and which data to consider. For example, every 100th remote memory access could trigger consideration of the data accessed. Unused objects do not waste migration opportunities with this policy because they do not activate the trigger mechanism.

Table 4.1: Description of synthetic benchmarks

Name	Description
resource-imbalance	Threads not on the center 8 nodes repeatedly access and immediately use 4 data. For 3 of the 4 threads on each node, data is local and unshared; for the 4th thread, data is shared and is placed at the chip's center. On the 8 center nodes, 4 threads execute 3 times as many instructions and half as many loads as other threads.
single-unshared	Each thread repeatedly (1000 times) accesses and then uses a single piece of unshared data.
several-shared	Each thread repeatedly (1000 times) accesses 4 shared data and then immediately uses the data. Each group of 4 pieces of data is accessed by the same 4 threads.

4.3 Exploring Migration Potential

In this section, we use a set of synthetic benchmarks to examine the potential impact of our migration strategy. Using these benchmarks allows us to isolate specific application characteristics that may interact in unique ways with our migration strategy. The intuition gained from this study allows us to understand the effect of migration on real applications in Section 4.4; based on these applications' composition of characteristics exhibited in the synthetic benchmarks, we can both anticipate and explain how migration impacts each application's performance.

Table 4.1 summarizes the basic characteristics of these synthetic benchmarks. Four threads are executed on each node for these benchmarks. The resource demands and relationships among threads and data, however, differ across the benchmarks. For example, threads in *resource-imbalance* require different amounts of computation and communication. In contrast, the threads in the remaining two benchmarks all execute the same instructions as the other threads in their respective applications, but they access different data sets. By using this set of benchmarks with different object placements, we can study the impact of migration on both varying resource demands and relationships among objects.

To isolate the impact of migration, we vary the simulation environment from the baseline architecture presented in Chapter 1 for these experiments. Because

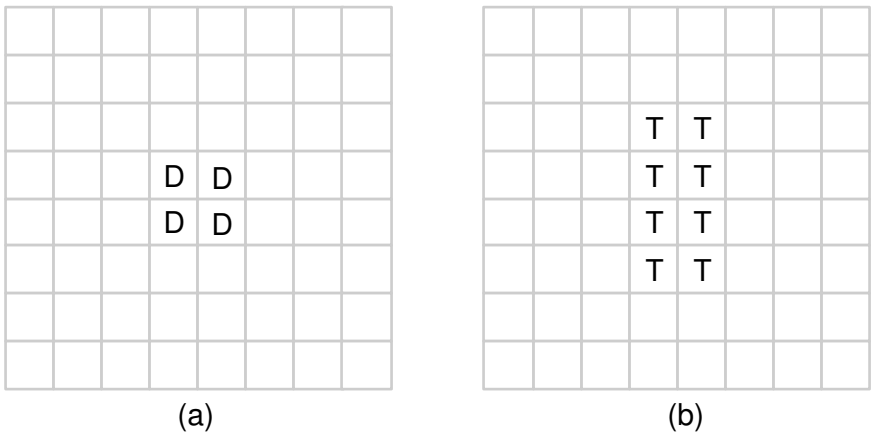


Figure 4.9: In the *resource-imbalance* synthetic benchmark, the placement of data at the chip's center as depicted in (a) minimizes overall communication distance but creates large network demands at those nodes. Similarly, the placement of computationally intensive threads on center nodes as depicted in (b) creates high processor demands.

the synthetic benchmarks are relatively small and short-lived, their data sets will completely fit into the architecture's caching structures; therefore, we remove caches and directories completely to emphasize the effects of migration. To see the impact of migration on network demands, we reduce the network's capabilities by reducing its ability to buffer data; we reduce the number of virtual channels and buffers per virtual channel to two each. Furthermore, we examine the impact of increasing processor speeds relative to network speeds by reducing the physical channel bandwidth; we accomplish this by increasing the latency between successive uses of physical channels. Finally, the small number of objects in these benchmarks permits us to use a static invocation policy; in this policy, nodes exchange state information every 5000 cycles and each node initiates a new migration decision every 100 cycles.

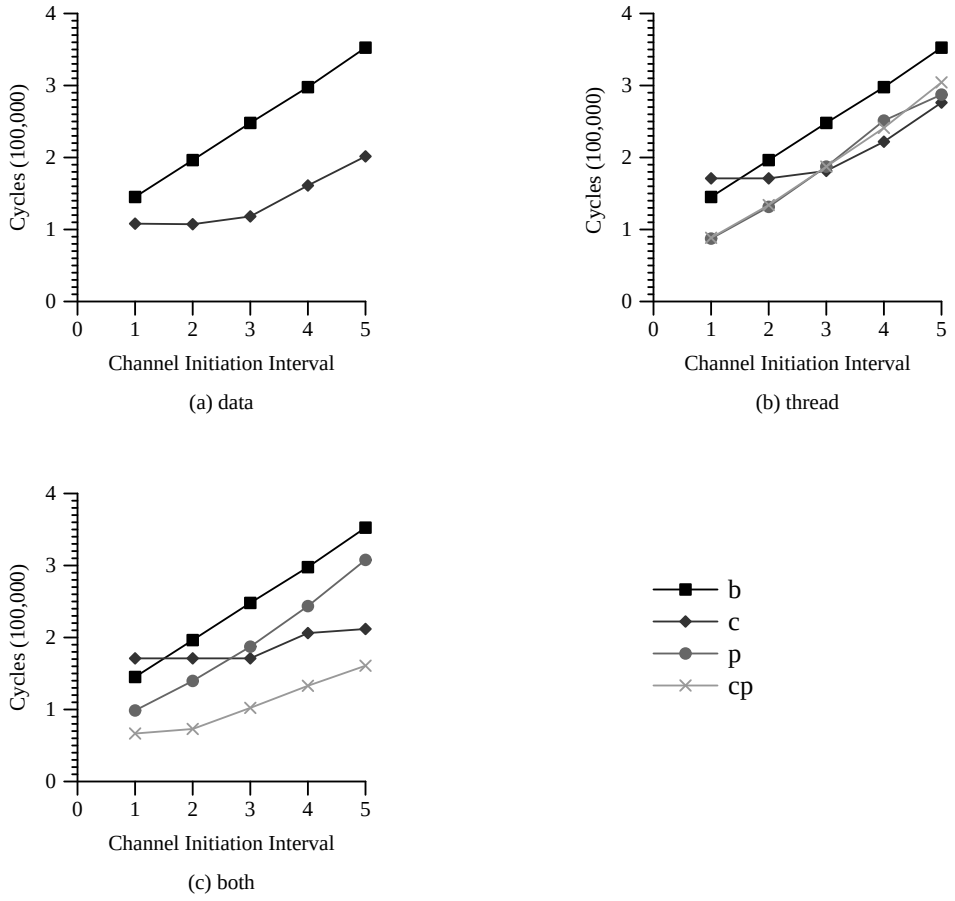


Figure 4.10: Execution times for the *resource-imbalance* benchmark. Migrating both threads and data based on processor and communication resources reduces execution time the most.

4.3.1 Repulsion Forces

The *resource-imbalance* benchmark allows us to examine the usefulness of incorporating both processor and communication demands in repulsion forces. In this benchmark, the four center nodes shown in Figure 4.9(a) experience high communication demands due to data requests while the eight nodes depicted in Figure 4.9(b) experience high processor demands. Performance improvements, therefore, require redistribution of processor and communication demands.

Figure 4.10 contains graphs depicting the execution times of the *resource-imbalance* benchmark when data, threads, and both data and threads are migrated. In each graph, we show the baseline (b) execution time when migration is not used and the execution times when the repulsion force is composed of communication resource load (c), processor resource load (p), and both communication and processor resource loads (cp).

Each graph also depicts the impact of migration as processor speeds become relatively faster than network speeds; we achieve this effect by reducing the available network bandwidth in our system. The *channel initiation interval* depicted on the x-axis specifies the amount of time required between subsequent uses of a network channel. As this channel initiation interval increases, the channel bandwidth decreases since data must be sent across the channel at a slower rate; a 1 cycle interval allows data injection every cycle, whereas a 5 cycle interval allows injection every fifth cycle. When the channel initiation interval is 5 cycles, the processor can complete 5 instructions in the time it takes to send data across a single channel, making the processor relatively faster than the network.

Figure 4.10(a) shows that migrating data in response to communication resource demands, a novel feature of our migration strategy, significantly reduces execution time. Thread migration also reduces execution time when incorporating communication resources (c) as seen in Figure 4.10(b). Adding processor resources to thread migration decisions (cp) further reduces execution time; this is particularly true at slower processor speeds (channel initiation interval less than 3) because computation resources constrain performance. The largest performance improvement, however, results from migrating both data and threads based on both communication and

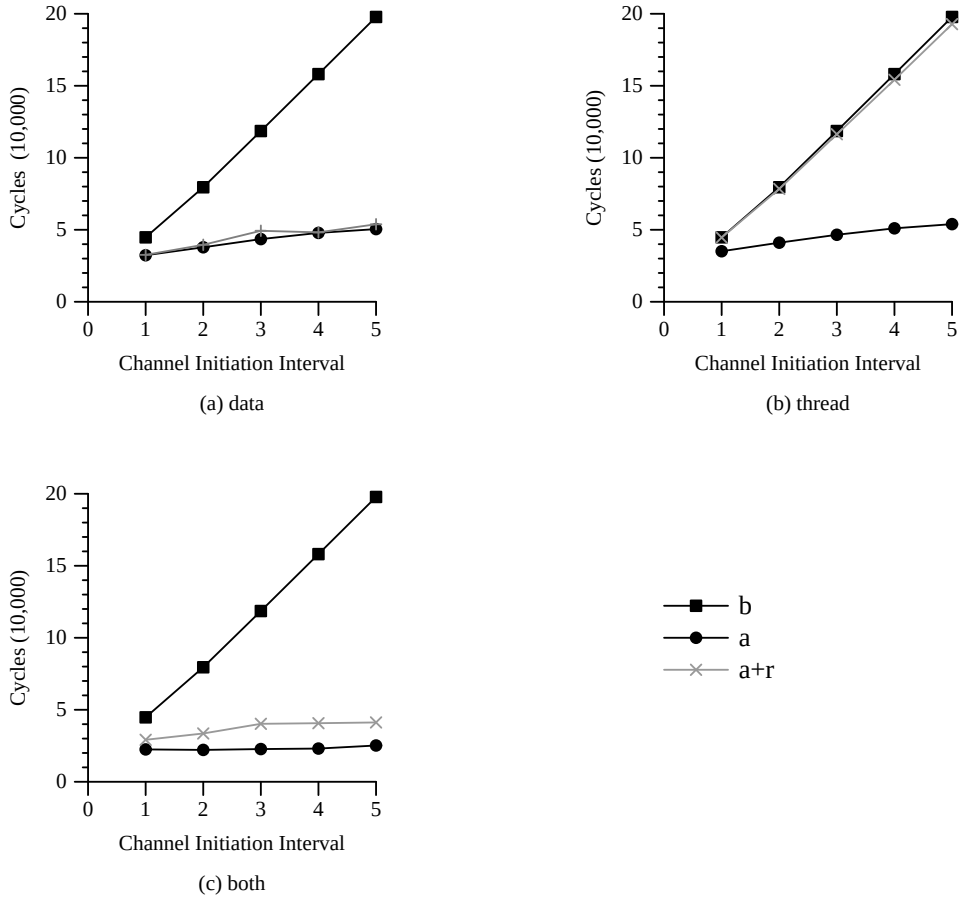


Figure 4.11: Execution times for the *single-unshared* benchmark using migration based on attraction and repulsion forces. Repulsion forces restrict thread migrations to prevent overloading intermediate nodes.

processor resource demands (*cp*) as seen in Figure 4.10(c).

4.3.2 Adding Attraction Forces to Repulsion Forces

In this section, we explore the impact of incorporating locality into our migration strategy. Because attraction forces are based on the direction and frequency of communication between interacting objects, we vary communication patterns by changing the relative positions of objects in the two locality benchmarks.

Clear Data / Clear Thread

In the *single-unshared* benchmark, all objects have clear communication patterns; each thread is pulled towards the datum it accesses and each datum is pulled to the thread that accesses it. When only threads or only data are migrated, migration based solely on attraction forces (a) reduces execution time more than other migration schemes as seen in Figures 4.11(a) and 4.11(b). This is because the sooner interacting threads and data are co-located, the less time threads waste waiting for remote accesses. When only data or threads are migrated, execution time suffers with the addition of repulsion forces (r) because migrating objects can be temporarily stalled by intermediate nodes with high resource demands as seen in Figure 4.11(c). However, when both data and threads migrate, the combination of attraction and repulsion forces ($a + r$) reduces execution time more than attraction forces alone.

Clear Data / Unclear Thread

In this section, we arrange the data and threads in *several-shared* to create clear communication patterns for data but unclear communication patterns for threads. Threads using the same data are located on four neighboring nodes, while data is randomly distributed. Consequently, data are pulled in a single direction while threads are pulled in multiple conflicting directions. As in the *single-unshared* analysis, Figure 4.12(a) shows that migrating data based on attraction forces (a) improves performance. However, the addition of repulsion forces (r) does not hurt performance; it actually improves performance at slower network speeds (channel initiation interval of 5) by preventing network contention. The repulsion forces cause data to be distributed across the four accessing nodes and, therefore, prevent one node's communication resources from becoming overloaded.

Because threads do not have clear communication patterns, attraction forces move them towards the chip's center to improve locality; this results in unbalanced processor load and decreased performance as shown in Figure 4.12(b). Despite the lack of benefits from thread migration, Figure 4.12(c) shows that migrating both data and threads based on both attraction and repulsion forces reduces execution time as much

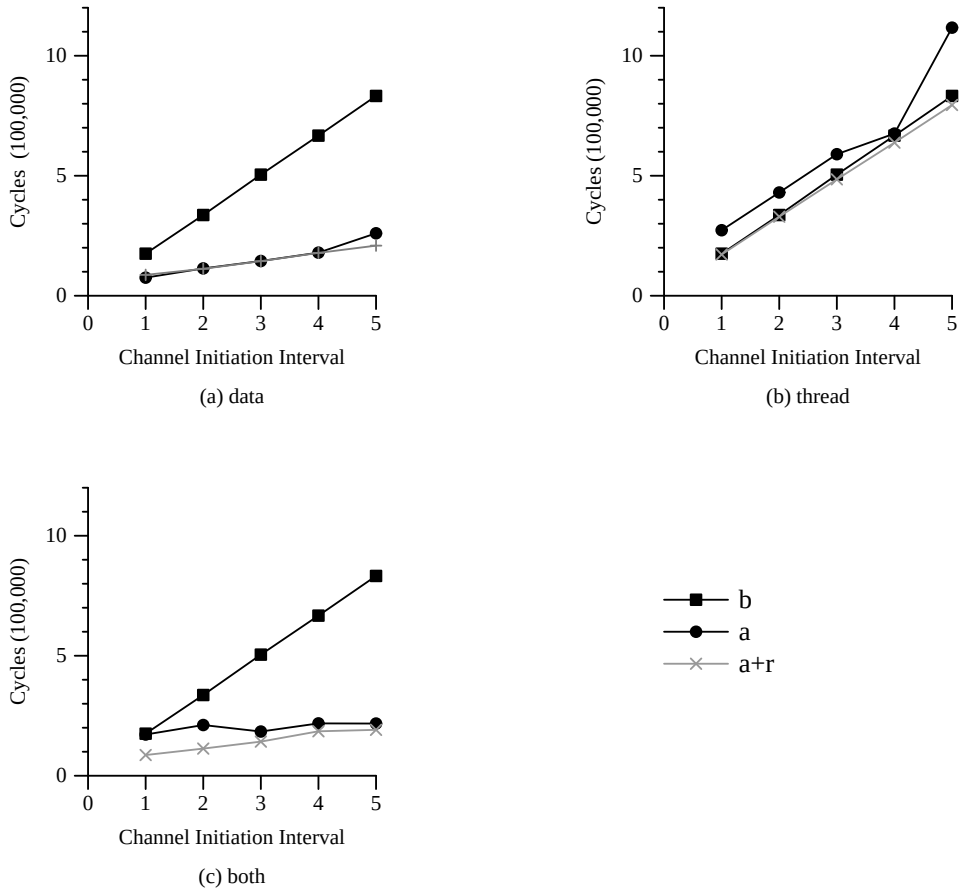


Figure 4.12: Execution times for *several-shared* when threads do not have clear communication patterns but data do. Migrating both data and threads obtains the same benefits as migrating data alone.

as data migration alone; processor resources inhibit thread migration while locality and communication resources migrate data.

Unclear Data / Clear Thread

In our second placement of data and threads in the *several-shared* benchmark, data accessed by the same threads are placed on four neighboring nodes while threads are randomly distributed; this creates an initial placement in which threads have clear communication patterns but data do not. Because threads have clear communication patterns, thread migration based on attraction forces (a) improves performance as shown in Figure 4.13(b). Unclear communication patterns pull data towards the chip's center, thus creating network contention. Figure 4.13(a) shows that adding repulsion forces to the attraction forces used for data migration prevents this network contention.

Figure 4.13 shows that, when threads have clear communication patterns, migrating both data and threads improves performance more than migrating either alone. However, Figures 4.13(b)-(c), show that these benefits are greatest when threads are migrated based solely on attraction forces; repulsion forces prevent threads from migrating to improve locality if intermediate nodes' resources would be overloaded in the process.

Unclear Data / Unclear Thread

When data and threads in the *several-shared* benchmark are both randomly distributed, all objects communicate in conflicting directions. Because objects have unclear communication patterns, migrating only data or only threads obtains limited performance improvements as seen in Figures 4.14(a) and 4.14(b).

Migrating both object types based on attraction and repulsion forces achieves larger reductions in execution time as seen in Figure 4.14(c). The combined forces cause objects to move closer to one another while avoiding overloading nodes' communication resources. Because communication latency is more important than processor resources at higher channel initiation intervals, imbalanced processor demands do not

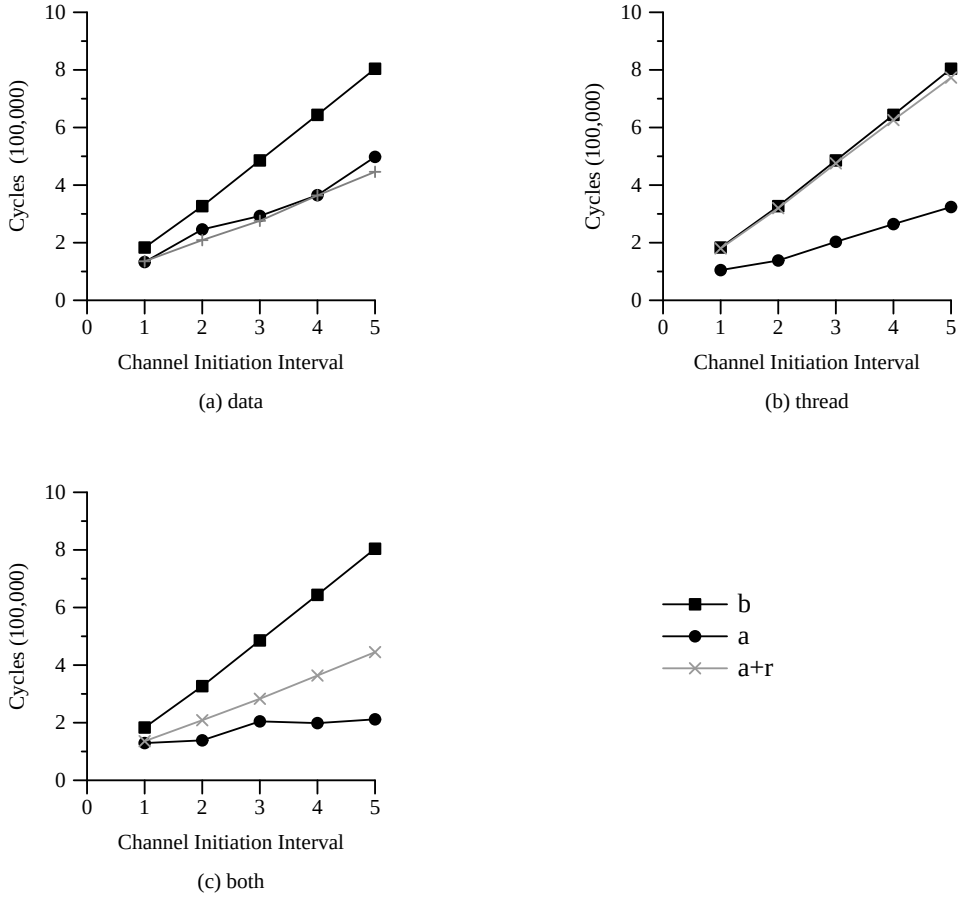


Figure 4.13: Execution times for the *several-shared* benchmark when threads have clear communication patterns but data does not. Thread migration is inhibited by repulsion forces.

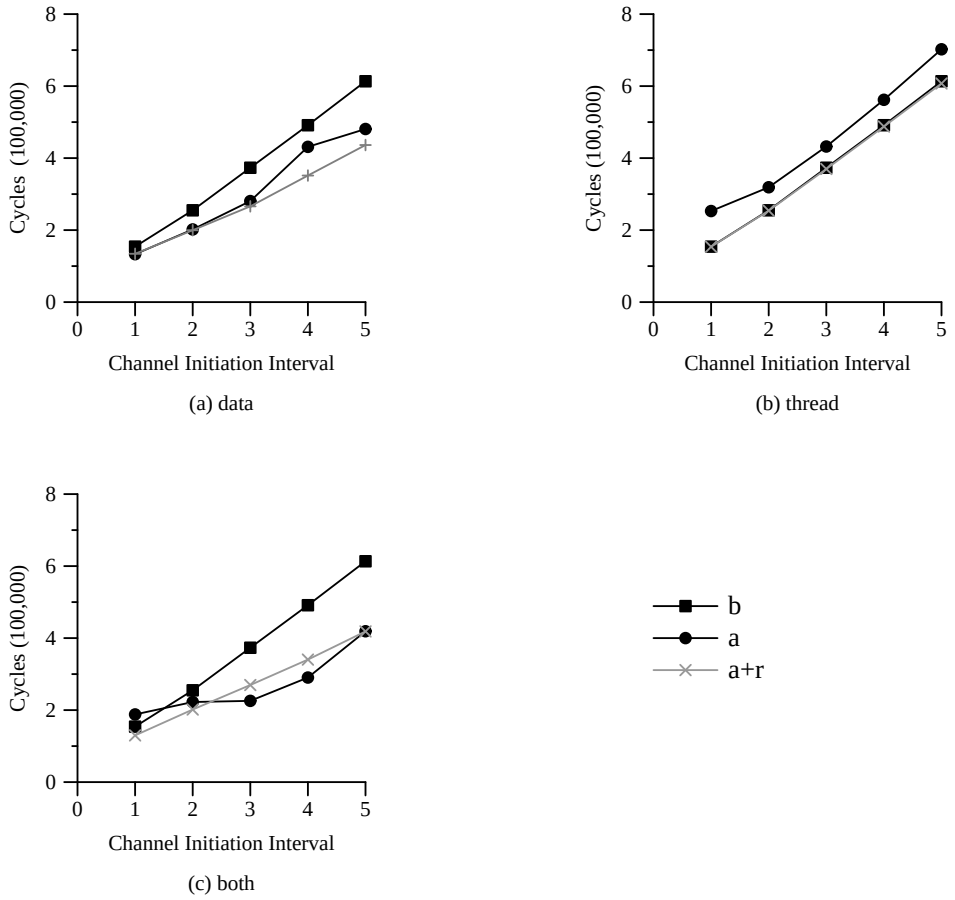


Figure 4.14: Execution times for the *several-shared* benchmark when all objects have unclear communication patterns. Once communication latency impacts performance more than processor demands, migrating both threads and data based on attraction forces achieves the best performance.

negatively impact execution time. However, when the availability of processor resources impacts performance (channel initiation interval is 1), migrating without the restraining effects of processor resources hurts performance.

4.4 Examining Larger Applications

In the preceding section, we showed that migrating both data and threads improves performance as much as or more than migrating only data or only threads. However, we also found that application characteristics determine which combinations of attraction and repulsion forces create the largest performance improvement. Depending on the frequency of these different characteristics, our migration strategy may or may not help performance.

In this section, we evaluate our migration strategy on the applications introduced in Chapter 3. We use the architecture presented in Chapter 1 with one modification. The architecture used for our simulations does not model directory traffic; in Section 4.5, we discuss the implications of this assumption.

In order to observe the impact of thread migration, we execute *raytrace*, *barnes-hut*, and *nbody* with a multithreading level of eight threads and *equake* with a multithreading level of four threads. Table 4.2 summarizes our applications' key characteristics with respect to resource load and communication patterns.

Table 4.2: Summary of application characteristics

Name	Resource Imbalance	Data	Threads
raytrace	Lots	Clear	Unclear
barnes-hut	Some	Unclear	Unclear
equake	Limited	Clear	Unclear
nbody	None	Unclear	Unclear

4.4.1 Real applications: execution time improvement

For these studies, nodes exchange state every 10,000 cycles. Upon state exchange, threads are considered for migration. We use a time varying invocation policy for data migration because the data set sizes are large and may be sparsely accessed. On every 100th remote memory access, a node considers the currently accessed data for migration. Finally, we assume that the migration decision computation takes 100 cycles and does not take away processing resources from application computation.

Figures 4.15(a)-(c) show the applications' execution times when attraction forces (a), repulsion forces (r), and attraction and repulsion forces ($a + r$) form the basis for migration. For comparison, the baseline (b) and oracle execution times are also shown. As a reminder, the *per-m* oracle emulates all memory accesses being handled locally; the *bal-p* oracle models all memory accesses to be local and performs perfect processor load balancing.

The migration strategies that rebalance processor demands improve *raytrace*'s performance by 41% by migrating 3,727 threads and 26,973 data objects. Because data have clear communication patterns, migration reduces the average distance of communication from 5-6 hops to 2-3 hops. However, data migration by itself does not reduce execution time despite this reduction in communication distance; communication latency has little effect on the baseline execution time as shown by the equivalent heights of the baseline and *per-m* bars in Figure 4.15(a). Like the results shown in Figure 4.12(b), migrating threads based on attraction forces alone increases execution time because processor load becomes imbalanced.

In *barnes-hut*, all objects have unclear communication patterns. Hence, data migration by itself does not decrease execution time. The large increases in execution time seen in Figure 4.15(b) resulting from migrating threads based solely on attraction forces make it clear that computation resources clearly define performance for this application. For these reasons, performing 979 thread migrations and 71,056 data migrations reduces execution time by only 7%.

The *nbody* application exhibits good processor load balance so we do not expect and do not obtain any benefits from thread migration based on repulsion forces. Like *barnes-hut*, *nbody*'s objects have unclear communication patterns which make

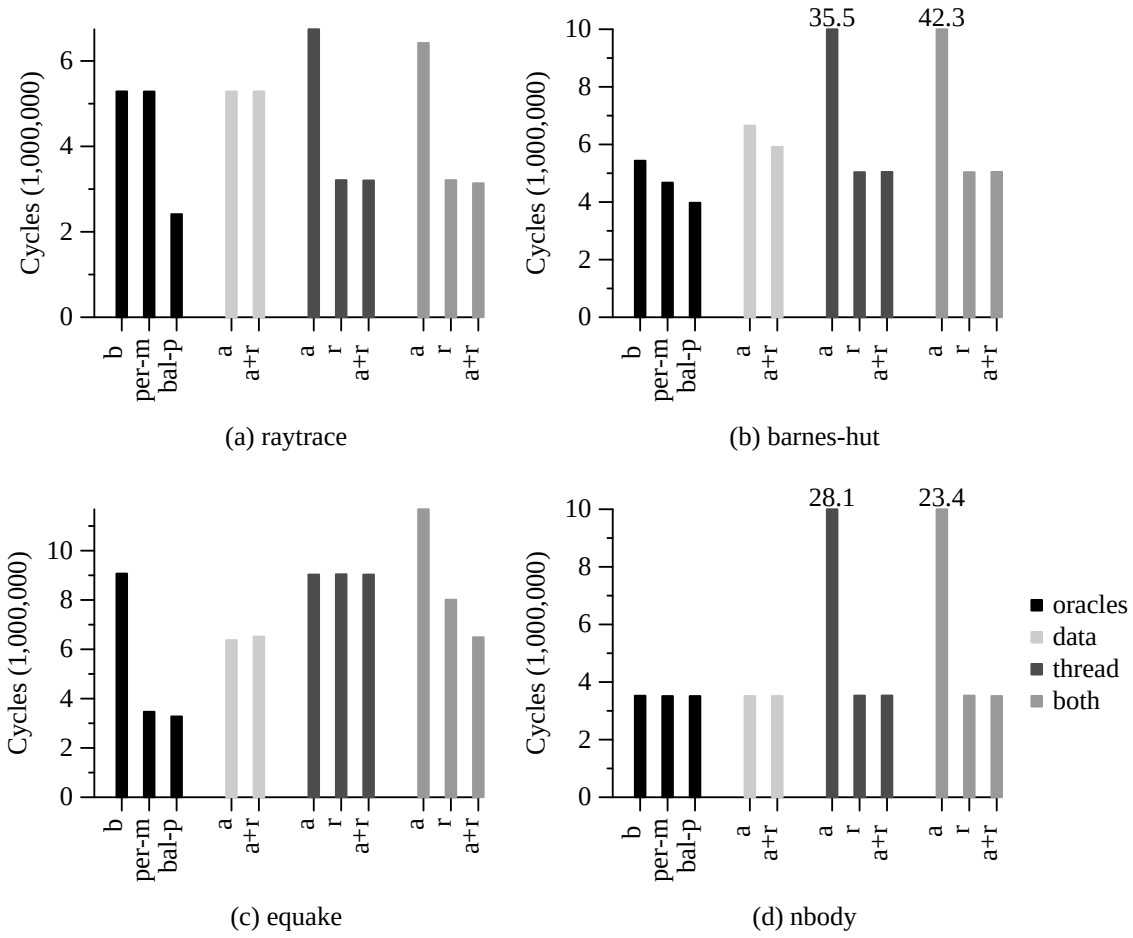


Figure 4.15: The application execution times depict the benefits accrued by different combinations of attraction and repulsion forces when data, threads, and data and threads are migrated.

migration based on attraction forces pull all objects towards the center of the chip. For applications with these characteristics, our migration strategy is unable to improve performance; however, it does not hurt performance either.

Because data have clear communication patterns in *equake*, data migration based solely on attraction forces reduces the execution time by 29%. As with the *single-unshared* benchmark, some of this performance gain is lost when communication resources are incorporated in the migration strategy via repulsion forces. Because threads have unclear communication patterns, thread migration does not improve performance. Despite this, migrating both object types obtains the same performance benefits as data migration alone.

4.4.2 Performance impact of migration as processor speed increases

To conclude this examination of our migration strategy, we execute the computationally intensive application *raytrace* when processors are five times as fast as our baseline architecture. Figure 4.16 shows the execution time of *raytrace* when objects migrate. Unlike in our earlier analysis, making all memory accesses local reduces the baseline execution time; the *per-m* oracle shows that communication latency increasingly impacts performance at faster processor speeds. Data migration based on attraction forces is now able to improve performance by 26%. Thread migration by itself, however, provides limited performance improvement. Thus, increasing processor speeds move applications into the realm where communication latency, both communication distance and contention, must be reduced to improve application performance.

4.5 Directory Traffic

The simulator used in this chapter does not model directory traffic. Instead, perfect knowledge about data's location in memory is assumed. In general, directory traffic can have two effects on application execution times. First, the latency of remote memory accesses may increase because a message must first be sent to the directory

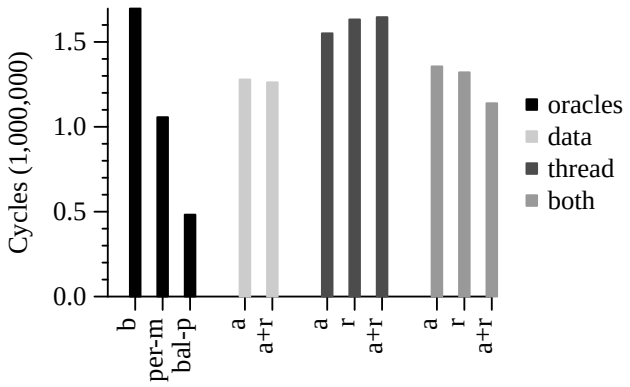


Figure 4.16: Increasing processor speed results in communication impacting *raytrace*'s execution time.

before proceeding to the data's memory location. Second, these additional messages to the directory will increase network demands and may increase latency due to network contention.

In Chapter 6, we execute each of the four applications on the architecture presented in Chapter 1, which models directory traffic. We briefly present information here about those executions in order to understand how directory modeling affects our migration results. *equake* is the only application whose execution time changes significantly with the addition of directory traffic; the addition of 16 million messages due to directory traffic causes the execution time to quadruple. There are several reasons that *equake* is affected and the other applications are not. First, the remote cache of memory locations at each node reduces the need to go to a directory. The hit rate for remote requests is only 54% for *equake* while it is 72% for *barnes-hut*, 92% for *raytrace*, and 97% for *nbody*. Second, the level of multithreading used in *equake* is smaller than the other applications (only 4 versus 8 threads), making it less latency-tolerant. Third, the frequency of memory instructions is much higher in *equake*, again making it more difficult to tolerate remote memory accesses.

Given that directory traffic does not impact three of four applications, we believe that this simplifying assumption does not affect our conclusions. For the remaining application, *equake*, we expect the benefits of migration to be even larger when directory traffic is modeled. In addition to reducing communication distance and balancing

processor demands, we expect that the improvements in locality and distribution of resource demands created by migration will reduce increases in network contention created by directory traffic. This reduction in network contention will further improve performance. For example, migrating data and threads based on attraction and repulsion forces reduces the execution time of *equake* by 56% when directory traffic is modeled compared to only 28% when directory traffic is not modeled.

4.6 Conclusions

Our application and benchmark studies indicate that migration improves the performance of applications with clear communication patterns and/or resource load imbalance, but has limited improvements on applications with unclear communication patterns. In Chapter 5, we explore a technique called *anchors* which turns unclear communication patterns into clear ones, enabling our migration strategy to work on a larger set of applications. Additionally, we consider how our strategy complements the use of caches in Chapter 6.

Chapter 5

Proactive Approach - Anchors

In Chapter 2, we described the static and dynamic information available for optimizing an application's performance. The migration strategy presented in Chapter 4 exploits dynamically collected communication and resource demand statistics to improve locality and resource usage. Unfortunately, this strategy has two limitations. First, information about inherent communication patterns between objects can be lost or obfuscated at runtime. Second, our gradual migration of objects to their best location takes time and, therefore, limits the achievable performance improvements. In this chapter, we address these two issues by introducing and evaluating a technique called *anchors*. By using statically available information about inter-object communication patterns to create new thread decompositions, the anchors technique helps expose communication patterns at runtime and enables execution of computation near the data it accesses.

5.1 Moving Computation to Data

The key to retaining locality information is to include it in either of the program abstractions: threads or data. We choose to retain temporal locality information by incorporating it into our thread invocations. The cluster concept introduced in Chapter 2 provides a technique for recognizing temporal locality among data objects and associating these objects with their computation. Figure 5.1 depicts the execution of a thread T; its data accesses to data A-F are specified and divided into three clusters. Specific data may be used in different clusters, however, these clusters can be distinguished from one another by the core set of data they access. The most frequently accessed data in a cluster constitutes a *core data set*. For example, in Figure 5.1, the core data sets are A for cluster 1, A and D for cluster 2, and C for cluster 3.

By associating each cluster's computation with its core data set, we can specify the locality association between the thread and the data it uses. One element of the core data set, called the *anchor*, acts as an approximation for the core data set. We then exploit this association by remotely executing the computation at the location of its anchor, thereby eliminating remote communication between the computation and its anchor.

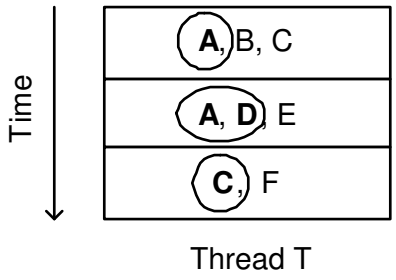


Figure 5.1: Thread T's data usage is decomposed into 3 clusters. In each cluster, the core data set is circled.

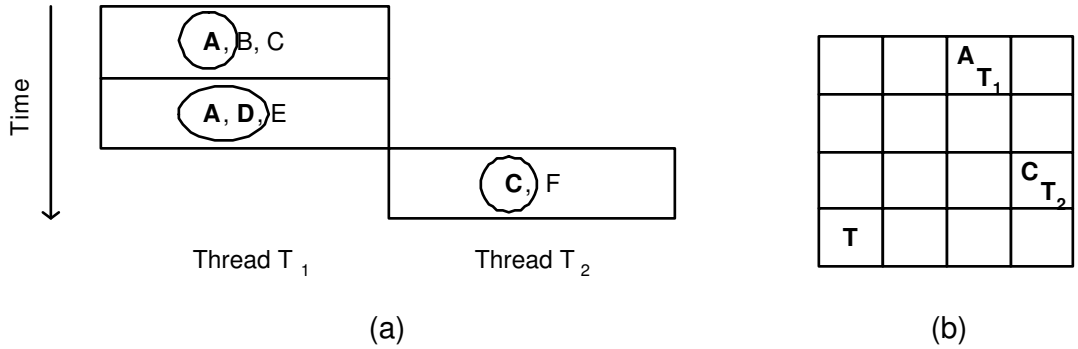


Figure 5.2: Anchors are chosen for each cluster and then two subthreads, T_1 and T_2, are created as depicted in (a). A single thread includes clusters 1 and 2 because their core data sets are similar. Figure (b) shows the execution of the two subthreads at their respective anchor locations.

The anchor technique divides threads consisting of multiple clusters into subthreads; each subthread is associated with its respective cluster and executed at the location of the anchor chosen from that cluster's core data set. Figure 5.2 shows how thread T from Figure 5.1 is decomposed into two subthreads, T_1 and T_2, based on their clusters and then executed at their respective anchor locations. It is important to note that these two threads are run serially; finding places to increase concurrency is not the objective of this work.

5.2 An Example: *Barnes-Hut*

In Chapter 3, we describe the inherent locality between particles created by the octree structure in *barnes-hut*; particles likely to strongly influence one another are situated close to one another in the octree. This locality information is lost at runtime, making it extremely difficult for our migration strategy to reduce communication distance without moving all data towards the center of the chip. This loss of information derives from the decomposition of *barnes-hut*'s computation into threads. A single thread calculates how all other particles impact a single particle. Consequently, threads access a large fraction of the application's data set, and many threads access overlapping data sets. This destroys any locality created by the octree.

In *barnes-hut*, each thread, t_i, associated with a particle, p_i, accesses some number of other particles, p_k, to determine p_i's new location and velocity. When threads are decomposed into clusters, there is no overlap in the accesses of particles p_k and p_{k+1}; each cluster contains accesses to a single particle p_k. Therefore, when we apply the anchor technique, we divide the thread t_i into many new subthreads where each new subthread, s_k, executes the computation that accesses particle p_k. Because the particles are organized in a tree structure, this means that subthreads will create new subthreads for their associated particles' children in the tree. The anchor for each new subthread, s_k, will be its associated particle, p_k.

Several improvements are gained by using anchors. First, all communication between subthread s_k and particle p_k is eliminated because they reside at the same location. Second, accesses to all other data used with p_k originate from the anchor's

location. Data in subthread s_k's data cluster, therefore, has a clear direction to move towards to improve locality; it is pulled towards p_k's location. Third, the locality created by the octree between a node, its parent node, and its children nodes is exposed by the communication needed to create new subthreads for accessing these nodes in the octree. By creating subthread s_k, particle p_k's association with its parent node in the octree is exposed. Our migration strategy can potentially exploit these clarified communication patterns to reduce communication demands.

5.2.1 Benefits: Reduced and Clarified Communication

The primary goal of using our anchor technique is reducing remote communication and remote communication latency. In general, the latency for a single remote communication, L_{remote}, equals the sum of sending a request, L_{req}, receiving a reply, L_{reply}, and data serialization, L_{data}, as expressed in Equation 5.1. For an architecture where communication latency increases depending on the number of hops traveled, the time for sending a request or reply message depends on the number of hops, H, and hop latency, L_{hop}, as seen in Equation 5.2. Note that this assumes that a request message with no data is equivalent to the size of a single channel width or flit. Additionally, any network contention would be added to this latency.

$$L_{remote} = L_{req} + L_{reply} + L_{data} \tag{5.1}$$

$$L_{req} = H \times L_{hop} \tag{5.2}$$

The total remote request latency for accesses to a particular data object can be expressed as the sum of all the remote request latencies between that data and the requesting thread. Because placing a thread and its anchor on the same node reduces the number of hops to zero, this value also represents the maximum amount of communication latency that would be eliminated if the thread and data resided on the same node. Therefore, the overall decrease in cumulative communication latency is simply the sum of the latencies for all remote requests, k, between a thread and its associated anchor. This is seen in Equation 5.3.

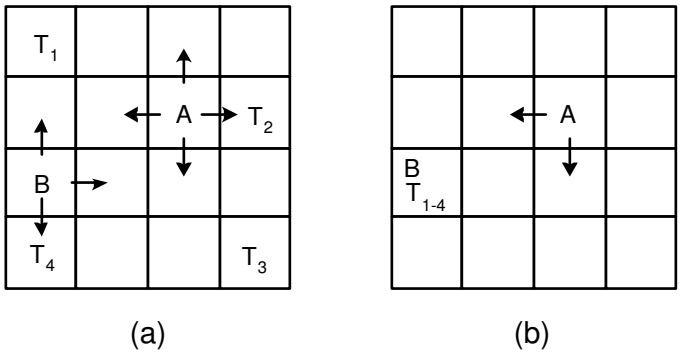


Figure 5.3: Figure (a) shows the communication directions for data A and B when threads do not execute at the location of the data that they access most frequently - data B. Figure (b) shows data A's clear communication direction towards data B when threads execute at the location of data B.

$$\sum_{k=1}^{m} L_{remote}(k) \tag{5.3}$$

An additional benefit of the anchor technique is that it clarifies the communication patterns between threads and data. As seen in Chapter 4, the physical placement of threads and data on a chip can obfuscate the logical communication patterns in an application. For example in Figure 5.3, if data A is always used when data B is used, then threads that use B will want A to be nearby. If, as in Figure 5.3(a), the threads that use B are distributed across the chip, A will be communicating in all directions. However, if all threads that use B execute at B's location, A will only communicate in one direction. Given the placement in Figure 5.3(b), migration schemes that use communication direction to improve locality can then move A towards B.

The anchor technique clarifies the inter-object communication patterns exhibited by *barnes-hut*. It enables us to capture the relationships among the data accessed when calculating a single particle's impact on a given particle. Additionally, it exposes the structure imposed by the octree. As threads recursively execute on a particular element's children, we can observe the communication between the parent thread and the children's data.

5.2.2 Performance Limiting Overheads

There are, however, costs associated with decomposing a thread into subthreads and executing these subthreads remotely. Invoking a remote thread requires any state that is needed for execution to be transferred. Consequently, the communication latency for creating a remote subthread, called the remote invocation latency, L_{inv}, is the sum of the latencies for sending a request and a reply as well as any necessary state, L_{state}. This set of latencies is captured in Equation 5.4.[1] In this work, we assume all threads are executing the same code so we do not include an explicit additional cost for code migration.

$$L_{inv} = L_{req} + L_{reply} + L_{state} \tag{5.4}$$

In addition to the remote invocation latency, there are also two main processor costs for our approach. First, invoking a remote thread requires additional instructions, L_{instr} to be executed on both of the processors involved. On the calling processor, additional instructions may need to be executed to send the thread invocation message. Creating a new thread at the anchor's location requires the remote processor to execute additional instructions to set up the new thread for execution. Both of these actions increase latency by taking time to execute the new instructions and by taking processor resources away from other useful computation.

Second, processor demands may become unbalanced when threads' execution locations are dictated by anchor locations. If many threads use the same anchor or have anchors residing at the same location, the processor resources at that location will be overwhelmed. The cost for this imbalance, L_{imbal}, is the sum of the number of instructions that overlap; this is because independent threads that could execute concurrently are serialized when they use the same processor, causing their execution times to increase. Consequently, care must be taken in the assignment of anchors to insure processor imbalance does not subsume the benefits gained by decreasing communication.

[1] If the program can be organized as a series of tail-end function calls or continuations, the reply message is not necessary.

5.2.3 Cost-Benefit Analysis

The anchor technique should only be used when the possible performance benefits for an application are larger than the costs discussed in Section 5.2.2. Our technique is advantageous when the change in execution time due to eliminating messages to/from anchors is larger than the combined costs of thread creation overhead and processor load imbalance as seen in Equation 5.5.

$$\sum_{k=1}^{m} L_{remote}(k) > L_{inv} + L_{instr} + L_{imbal} \tag{5.5}$$

From this simple equation, we can make two observations. First, a key benefit of remotely executing a thread is the elimination of message overhead. Instead of having many request and reply messages, a single thread invocation message is sent, containing all the state needed to execute the thread. This means that the overhead of sending messages is only incurred once instead of many times. Second, Equation 5.5 implies that as processor speeds increase relative to network speeds, the negative effects of these processor overheads on the obtainable performance improvements will be reduced.

Finally, it is important to note that this model is simplistic in that it assumes that the latency for a remote request cannot be masked via some other computation. When latency can be masked, the overall benefit of removing remote requests decreases; however, the decrease in messages may also reduce latency due to network contention.

The remainder of this chapter examines the impact of these theoretical costs and benefits on the performance of a set of synthetic benchmarks and a set of applications. The synthetic benchmarks enable us to analyze each of the potential costs in detail while the applications show us the full potential of our technique.

5.3 Analyzing the Impact of Changing Costs

In this section, we use a set of synthetic benchmarks to examine the potential benefits and costs of using the anchor technique. We study a range of costs in order to understand how sensitive application performance is to each overhead. Additionally,

we examine how the relative impact of these costs changes as processor speeds increase relative to network speeds.

5.3.1 Synthetic Benchmark Description

Each of the synthetic benchmarks consists of 64 threads that repeatedly access and then immediately use data. The differences between the benchmarks are specified in Table 5.1. Every thread in the benchmarks has a specific number of clusters in which it accesses core data objects equally; any other data in that cluster is accessed 25% as frequently.

Benchmarks with multiple clusters are candidates for the creation of subthreads, each with their own anchor. We decrease the importance of communication relative to computation in the *single-nosubthreads-compute* benchmark by decreasing the number of memory references; we use these benchmarks to explore the benefits of our technique on more processor-centric applications. Finally, we vary the number of nodes data reside on in order to examine the impact of processor load imbalance.

For each of the benchmarks, we vary the network's hop latency. We use the variable hop latency to approximate several possible increases in communication latency: divisions of chips into larger numbers of nodes, network contention, and increasing processor speeds. By increasing the hop latency, we change the relative processor speeds; larger hop latencies imply relatively slower networks. By modeling increasing hop latencies, we can see the performance trends caused by increased network congestion, faster processors, or increased average hop distances (caused by subdividing chips into larger numbers of nodes).

5.3.2 Communication Benefits from Using Anchors

Executing Computation at an Anchor

We use the *single-nosubthreads* benchmark to highlight the benefits accrued by executing a thread at the location of its anchor. We assume minimal overheads for thread creation and that the benchmark's processor and memory demands are distributed evenly across the chip. This benchmark places data within the same cluster

Table 5.1: Synthetic benchmarks' defining parameters.

Name	*single-nosubthreads*	*single-nosubthreads-compute*	*multiple-nosubthreads*	*multiple-subthreads*
Clusters	1	1	4	4
Core Data	4	4	1	1
Other Data	0	0	1	1
Ref.	400	200	125	125
Instr	700	700	250	250
Placement	Locality	Locality	Random	Random
Nodes w/Core Data	64	64	64	64/32/16/8

on neighboring nodes, with data distributed evenly across the nodes. Consequently, when a thread resides near one of the data in its core data set, it resides near all of them. In the no-anchor case, threads do not execute at their anchor's location; instead, the benchmark randomly assigns threads to nodes. When anchors are used, threads execute at the location of their anchors.

Figure 5.4 depicts a dramatic difference in execution time for the no-anchor and anchor versions; as hop latency increases, this disparity grows. Figure 5.5 depicts the number of flits and messages sent during execution, as well as the cumulative distance these messages travel. Fewer flits and messages are sent when anchors are used. Because threads in the anchor version execute at the location of their respective anchors, messages between these anchor/thread pairs are eliminated; only messages to data on remote nodes are seen.

The large decrease in the anchor case's message distance, however, contributes more significantly to the execution time disparities than message elimination. In the no-anchor case, threads do not necessarily execute near the data they access. In the anchor case, however, threads execute near their entire core data set. Because the data in the core data set resides near that set's anchor, threads communicate only short distances. Figure 5.6 shows that the distance between data and the center of its accesses decreases when anchors are used. These shorter distances result in

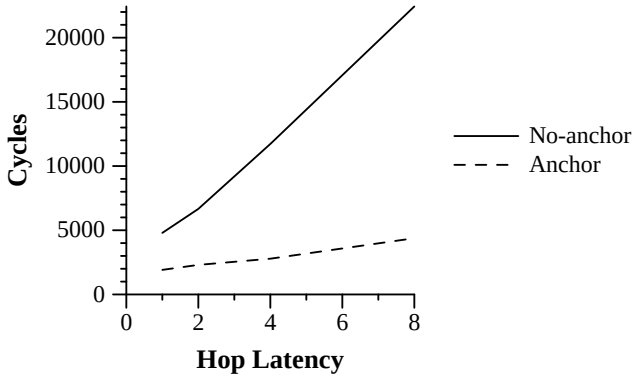


Figure 5.4: We execute the anchor and no-anchor versions of single-nosubthreads, varying network hop latencies from 1 to 8.

shorter memory stalls which create the large changes in execution time. Threads in the no-anchor version are unable to exploit its core data set's locality and, therefore, incur larger stalls.

Finally, Figure 5.7 depicts one further benefit of using anchors. As described in Section 5.2.1, anchors can help make the inherent communication patterns between objects clearer despite initial object placements. Figure 5.7 shows the optimal locations for data placement when overall communication distance is minimized. Darker colors imply more data would be placed on these nodes. Without the use of anchors, data is attracted towards the center of the chip; this implies that each data is pulled in conflicting directions as discussed in Chapter 4. When anchors are used, however, optimal data placement distributes data across the chip. Data is accessed by threads whose locations are near the data's location instead of by threads distributed across the entire chip. Consequently, the data will remain in the area of these threads instead of all of them being pulled towards the chip's center. By using anchors, we avoid the obfuscation of communication patterns that stems from physical placements of data and threads on the chip; anchors allow us to clearly see the communication patterns between data and threads regardless of placement.

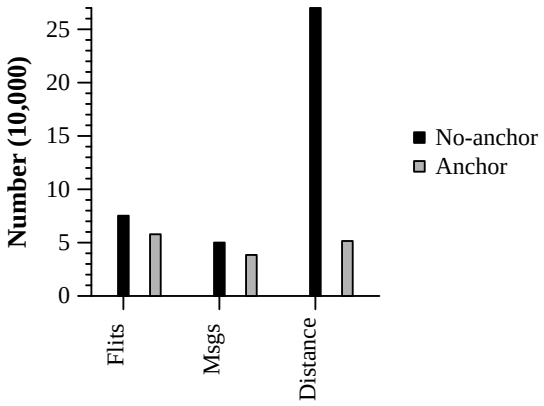


Figure 5.5: The cumulative number of messages and flits sent, as well as the overall distance these messages travel, for the single-nosubthreads benchmark is shown.

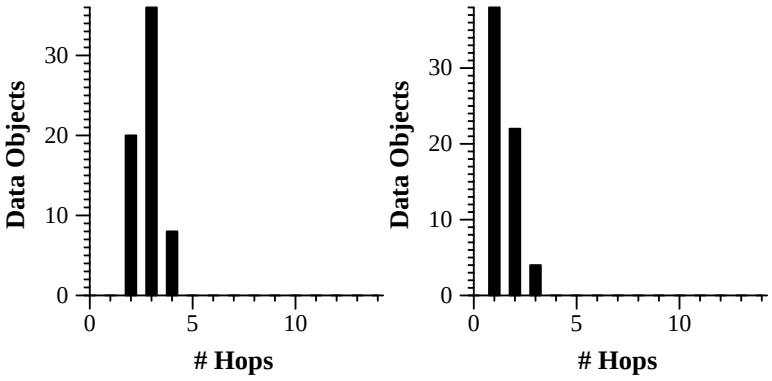
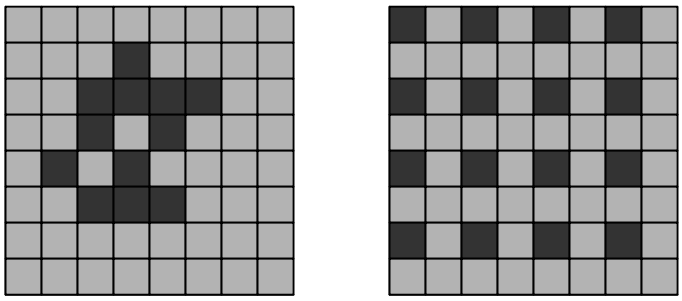


Figure 5.6: The distance between a data object and the center of its references is shown.



(a) no-anchor (b) anchor

Figure 5.7: These temperature graphs show the optimal locations for data given their references' node origins. Darker colors imply more data are pulled to that location. When anchors are used, objects would optimally be distributed across the chip, reducing the likelihood of imbalanced resource demands.

Evaluation of Creating New Subthreads

We use the *multiple-nosubthreads* and *multiple-subthreads* benchmarks to examine the impact of subthreads on communication latency and execution time. The *multiple-nosubthreads* benchmark accesses four primary data and one other piece of data. However, it accesses each of the primary data for an interval and then switches to use a different one. In *multiple-subthreads*, we create four subthreads at each of these clusters with the primary data designated as the thread's anchor.

Figure 5.8 shows the change in execution time when we simulate both of these benchmarks using minimal anchor overheads. Similar to our example in the preceding section, we observe large differences in the two cases' execution times resulting from decreased communication latency. Figure 5.9 shows the large differences between the two cases in terms of both the number of flits and of messages and the cumulative communication distance. By taking advantage of the different data clusters, anchors transform the many read requests into a single remote thread invocation request and many local data accesses.

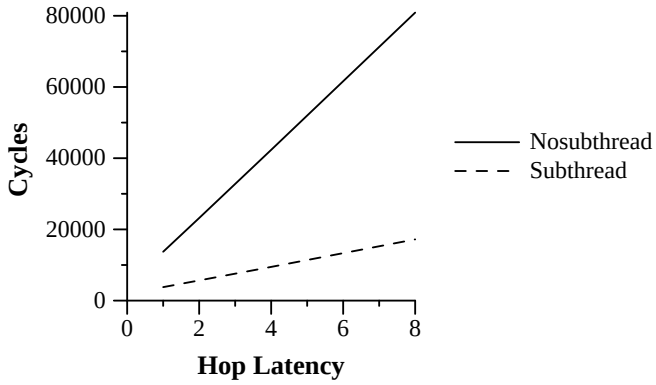


Figure 5.8: Execution times for the *multiple-nosubthreads* and *multiple-subthreads* benchmarks are shown as hop latency is varied from 1 to 8 cycles.

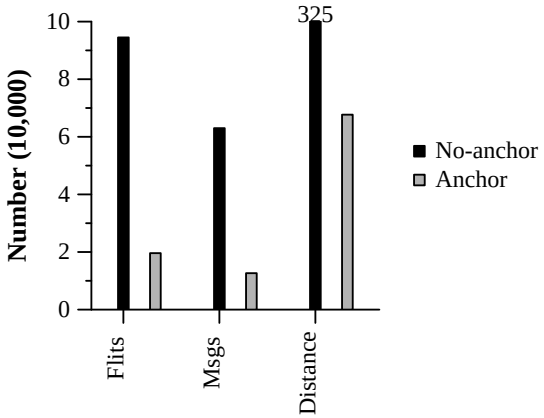


Figure 5.9: The number of messages and of flits, as well as the distance messages travel, are shown for the *multiple-nosubthreads* and *multiple-subthreads* benchmarks.

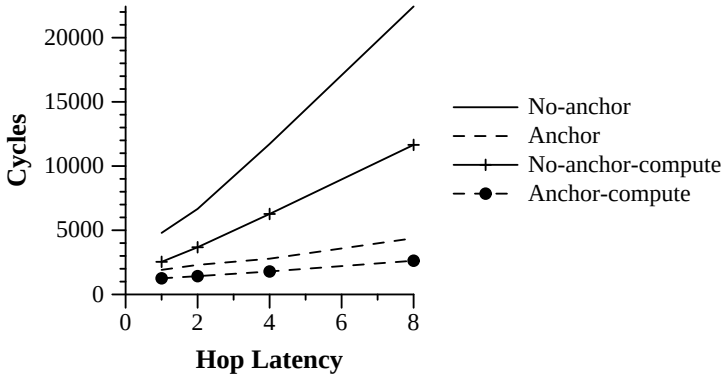


Figure 5.10: The execution times for the noanchor and anchors versions of the *single-nosubthreads* and *single-nosubthreads-compute* benchmarks are shown as hop latency and thread invocation cycles are varied. The execution times for the *single-nosubthreads-compute* benchmark show the effect that reducing data references (and hence the relative impact of communication) has on the improvements gained by using anchors.

Effects of communication frequency

To understand how communication frequency affects how well our approach performs, we decrease the total frequency of data accesses (and thus communication) in our benchmarks and repeat our experiments. The threads in the *single-nosubthreads-compute* benchmarks perform 50% fewer data references than threads in the *single-nosubthreads* benchmark. Figure 5.10 shows the execution times for both sets of benchmarks. The differences in execution times for the two versions of the *single-nosubthreads-compute* benchmark are smaller than in the more communication-bound benchmark. At smaller hop latencies, in particular, only small performance gains appear. The costs of using anchors discussed in Section 5.2.2 would likely overcome these small improvements. However, as hop latency increases, even these more compute-centric benchmarks experience significant execution time reductions with the use of anchors.

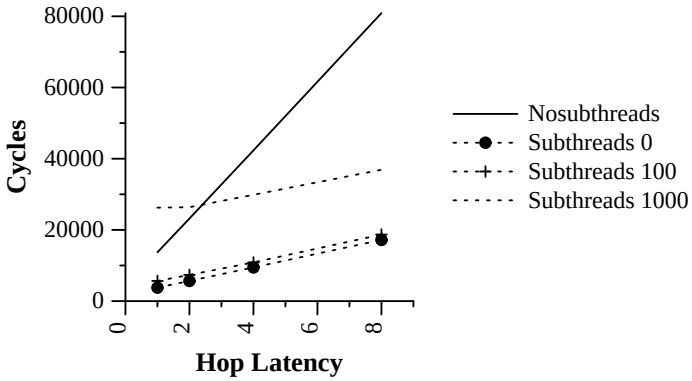


Figure 5.11: The effects of hop latency and state size on a remote thread invocation are depicted for the *multiple-nosubthreads* and *multiple-subthreads* benchmarks. The legend specifies state size in terms of flits.

5.3.3 Impact of Remote Invocation Costs

Up until this point, we have focused on the benefits of anchors assuming minimal overhead of applying this technique. We now look at the negative impact of additional instructions that must be executed on remote invocation, of state that must be transferred on remote invocation, and of increased processor load imbalance.

Communication Resources: Transferred State

The state needed to execute a subthread must be included in remote thread invocation messages; this increases both network bandwidth demands and the latency of remote thread invocations. At the very least, a program counter is required, but generally the transferred state will be similar to that passed on a function call. Larger state quantities increase the message's transit time and increase network demands. If the state size is too large, the costs may offset the benefits from anchors.

Figure 5.11 shows the effects of increasing the state transferred on each subthread invocation. We vary the amount of state transferred in terms of flits (8B) from 0 to 1000 while also varying hop latency. As the number of transferred bytes increases by 100 flits from 0 flits to 1000 flits, the execution time gradually increases; there are no severe jumps in execution times as the transferred bytes increase by increments

of 100 flits, so we only show execution times for 0, 100, and 1000 flits. Figure 5.11 shows that state sizes of less than 100 flits do not strongly impede anchors from improving performance. However, when the state size is 1000 flits, anchors only improve performance at higher hop latencies. When the hop latency is less than four cycles, the overhead of transferring a thread's state overpowers any obtainable anchor benefits. Therefore, subthreads should not be created when the state that must be transferred is large and hop latencies are small. For the applications used in our study, at most 40 bytes (5 flits) of state must be transferred in addition to the 8 bytes (1 flit) of state (including the program counter) necessary for creating any remote thread.

Processor Resources: Invocation Time and Load Imbalance

As mentioned earlier, processor resource demands increase and may become unbalanced when using our anchor technique. The number of cycles it takes to create a new thread will reduce the potential performance benefits gained by the introduction of subthreads. To understand the magnitude of this effect, we vary the invocation time for subthreads from 0 cycles to 1000 cycles. Figure 5.12 shows the total execution cycles for the *multiple-nosubthreads* and *multiple-subthreads* benchmarks as both hop latency and invocation time are varied. As hop latency increases, the benefits of using subthreads is sustained despite increasing invocation times. Additionally, as hop latency increases, the differences in execution time for different invocation costs decrease. This trend implies that invocation time will diminish in importance as processors become relatively faster than the network.

Finally, we note that there are points when the number of invocation cycles are too high to allow the use of subthreads to reduce execution time. In Figure 5.12, this happens when each invocation time is 1000 cycles and hop latency is 1. The interval for which anchors will be too expensive depends on the relative speeds of the network and processor, the application's communication frequency, and remote thread invocation costs. In [28], they showed that new threads could be created in 11 cycles at a remote processor. When more state is transferred, we expect the time for creating a new thread will increase; however, the small application state transferred

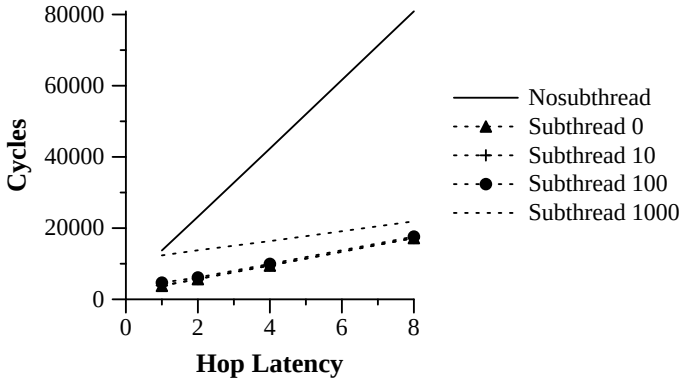


Figure 5.12: The execution times for the *multiple-nosubthreads* and *multiple-subthreads* are shown when hop latency and remote invocation latency are varied. The remote invocation latency is specified in the legend.

on our applications' thread creations is unlikely to increase invocation times by large quantities. We expect the thread invocation cost to be larger than 10 cycles but less than 100 cycles.

In addition to increasing the number of instructions executed, executing subthreads at the location of their anchors can destroy any initial processor load balance created by the programmer. We use the *multiple-nosubthreads* and *multiple-subthreads* benchmarks to explore this issue. By varying the number of nodes (64, 32, 16, or 8) on which data resides, we vary the number of subthreads that execute on each node in that set. This is because threads are co-located with their anchors.

Figure 5.13 shows the effects of varying hop latency and the number of nodes on which data resides. When subthreads are not used, threads are evenly distributed across all 64 nodes. Data, however, resides on a subset of these nodes. The smaller the set of nodes with data, the larger the number of messages sent to each of these nodes. Consequently, network contention increases, causing execution times to increase.

The results differ considerably when subthreads execute at their anchor locations. At smaller hop latencies, the use of fewer nodes can double execution times because threads must compete for the limited processor resources. These performance losses, however, diminish as hop latency increases since communication resources have a

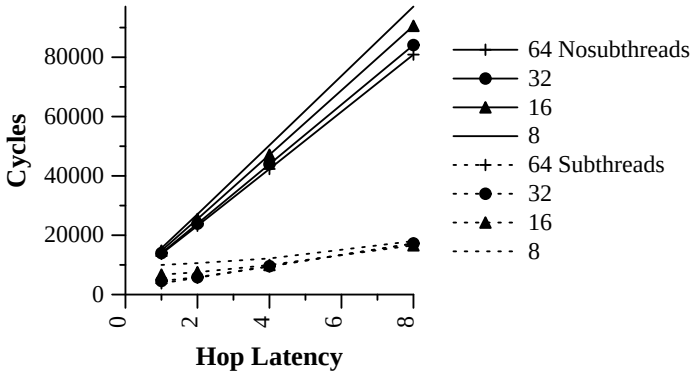


Figure 5.13: Execution times for the *multiple-nosubthreads* and *multiple-subthreads* benchmarks are shown when hop latency and the number of nodes on which data resides vary.

stronger influence on performance than processor resources. The convergence of execution times at a hop latency of eight implies that as the speed of the processor increases relative to the network speed, imbalanced processor demands no longer inhibit the use of anchors.

5.3.4 Summary

The overheads we examined in this section detract from the performance improvements achieved by using anchors when large overheads are assumed. However, as communication speed grows relatively slower than processors, the impact of invocation time and processor load imbalance diminish. Large state transfers have the largest impact on the use of subthreads, but, even with substantial state transfer amounts, benefits will be obtainable as the network becomes a larger bottleneck.

The applications we study do not incur these large overheads; they transfer small quantities of state (<100 flits) on each thread creation, and the invocation time required for creating new threads should be on the order of tens of cycles. Consequently, these overheads should have limited impact on the applications' performance when the anchor technique is applied.

5.4 Benefits of using anchors on full applications

Having analyzed anchors with synthetic benchmarks, we now apply anchors to three applications introduced in Chapter 3. For each application, we manually determine where threads should be broken into subthreads and which data should be used as anchors for the anchor versions. We also calculate the amount of state that must be transferred on remote thread invocations.

5.4.1 Adding Subthreads and Anchors to Applications

raytrace is a computationally intensive application which obtains little benefit from reducing remote memory access latencies. By enabling a multithreading level of up to eight threads per processor, we further decrease the need for reducing remote memory latencies and the potential benefits obtained by using anchors. We can, therefore, use it to examine the impact of applying our anchor technique to an application that is not limited by communication latency. In the original version of *raytrace*, threads walk through a variable number of voxels, using only a small subset of application data. In order to employ the anchor technique, we modify the application by creating new subthreads each time a thread begins accessing a new voxel and its associated geometry. The voxel becomes that subthread's anchor.

In the *barnes-hut* application, each thread walks through large parts of the octree, using the data at its current tree position and then not using that data again until the next timestep is simulated. In general, when multithreading is not used, the application is processor load balanced and memory latency sensitive. By modifiying this application to use both our anchor technique and no multithreading (to prevent any masking of communication latency), we can focus on how our technique impacts processor load balance and remote communication latency. Our modified version creates a new subthread when a thread begins accessing a different tree element; that tree element is designated the subthread's anchor.

n-body does not have a tree structure that imposes locality. Instead, threads access all particles, creating large quantities of remote communication and balanced work distributions. Although this application is simple, it provides insight into applications

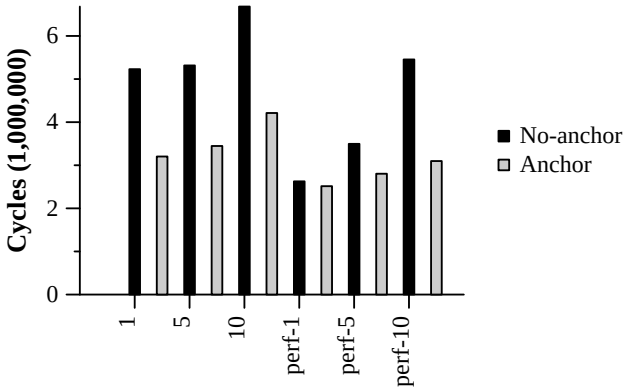


Figure 5.14: Despite being computationally-intensive, the size of *raytrace*'s increase in execution time decreases with the use of anchors as hop latency increases.

that experience periods of all-to-all communication. Our modified version of *n-body* creates a new subthread each time a thread begins accessing a different particle and defines that particle to be the subthread's anchor. Only one thread executes on each node at a time.

5.4.2 Application Results

For each application, we simulate hop latencies of 1, 5, and 10 cycles. Additionally, we have created an oracle that models memory delay while it also performs perfect processor load balancing. This oracle, called *perf*, executes one instruction per timestep from at most 64 threads, regardless of those threads' locations. We also vary hop latency for this oracle. The *perf* oracle provides a way of distinguishing between processor imbalance and increasing communication latency.

Figure 5.14 shows that *raytrace*'s performance is improved by the use of anchors. The benefits, however, are not from reductions in communication latency at small hop latencies; changing the hop latency from one to five cycles does not significantly increase execution time. The performance benefits come from better processor load balance; this can be seen by comparing the execution time of *perf-1* with the execution time for a hop latency of one. Finally, we note that at a hop latency of ten cycles, communication latency starts to significantly impact performance. The anchor version

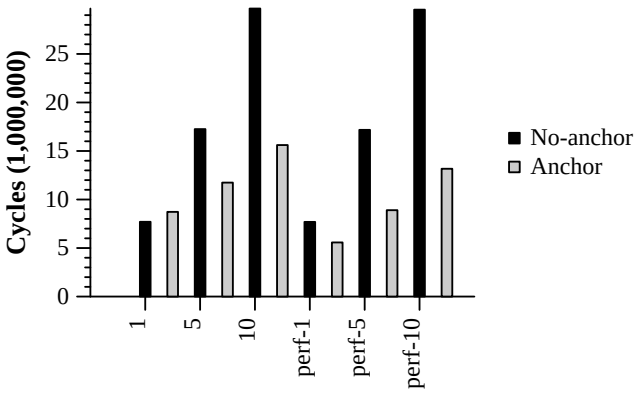


Figure 5.15: The size of *barnes-hut*'s increase in execution time decreases with the addition of anchors as hop latency increases.

recoups some of this performance loss and processor load balancing recoups even more. Hence, we observe that even computationally intensive applications can benefit from the use of anchors.

At a hop latency of 1, anchors actually slightly degrade the performance of both the *barnes-hut* and *nbody* applications as seen in Figures 5.15 and 5.16. The cost of additional instructions and processor load imbalance introduced by anchors exceeds any benefits. However, as hop latency increases, the anchor version outperforms the no-anchor version for both of these applications. The addition of processor load balancing to the anchor versions further improves performance.

We can use the temperature graphs in Figure 5.17 to understand the difference between the anchor and no-anchor versions of *barnes-hut*. When anchors are not used, data is accessed from all parts of the chip. As a result, the optimal location for data with respect to locality is the chip's center. When anchors are used, however, data remain distributed across the chip. By using anchors, the locality imposed by the tree structure is preserved; data are referenced from a small set of nodes instead of all nodes. This reduces the application's overall communication frequency and communication latency, resulting in reduced execution times.

nbody's all-to-all communication patterns make it impossible for a single thread to exploit locality. Yet, using anchors improves execution time as discussed above.

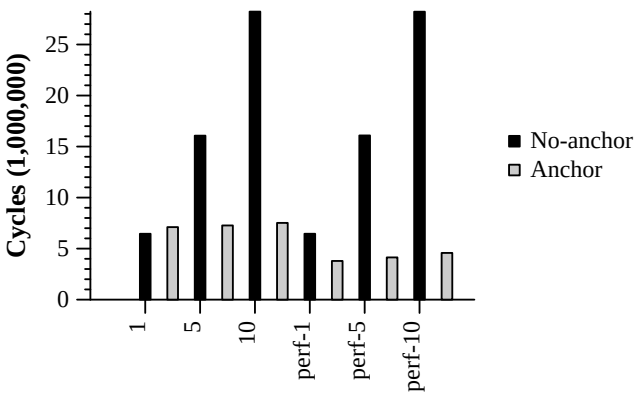
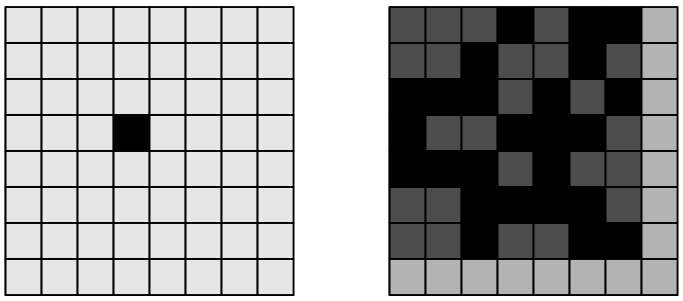


Figure 5.16: Using anchors decreases *nbody*'s execution time by more than 70% at a hop latency of 10 cycles.



(a) No-anchor (b) Anchor

Figure 5.17: The temperatue graphs show the optimal placement of data in terms of locality when the noanchor and anchor versions of *barnes-hut* are executed. Darker squares have more resident data than lighter colored squares. The use of anchors causes data to be distributed across the chip, distributing resource demands across more nodes.

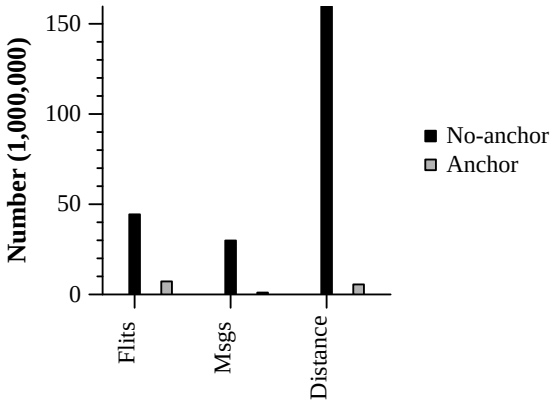


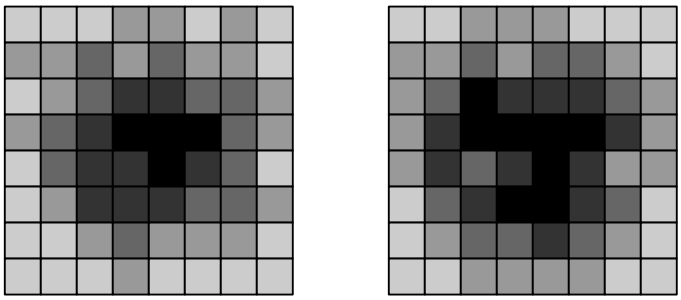
Figure 5.18: We show the change in communication demands for the anchor and no-anchor versions of *nbody*. Not only do the number of messages and distance decrease with the addition of anchors, but the amount of transferred data decreases as well. As hop latencies increase, this reduction in communication translates into substantial reductions in execution time.

Figure 5.18 shows that when anchors are used, the total amount of communication decreases significantly. In addition to reducing the quantity of communication and distance traveled, anchors also reduce network congestion. Hence, anchors can significantly improve performance on applications with all-to-all communication.

5.5 Anchors Plus Migration

We now examine how our anchor technique interacts with migration. We showed in Chapter 4 that applications with unclear communication patterns obtain only small benefits from migration because the strategy cannot detect which direction to move data to improve locality. In this section, we use the *barnes-hut* and *nbody* applications to explore our conjecture that applying the anchor technique to applications with unclear communication patterns can help clarify communication patterns, allowing our migration strategy to improve locality and reduce communication demands.

In Chapter 4, we showed that the optimal locations for data objects in *barnes-hut* and *nbody* were at the center of the chip. This placement minimized the overall communication distance for the applications. When anchors are used, the optimal



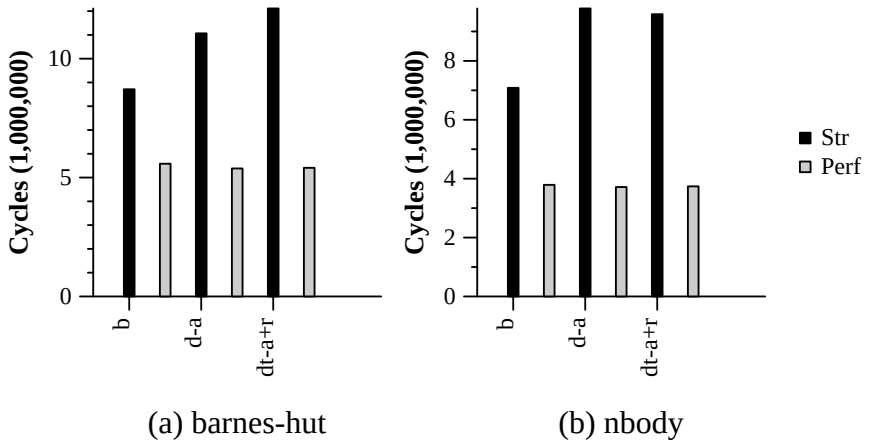
(a) barnes-hut (b) nbody

Figure 5.19: The temperatue graphs show the final locations of data when migration based solely on attraction forces is applied to the anchor versions of *barnes-hut* and *nbody*. Darker squares have more resident data than lighter colored squares. Data are distributed across the nodes of the chip instead of residing on a small number of center nodes.

placement of data objects changes; as seen in Figure 5.17, the optimal placement of *barnes-hut*'s data objects is distributed over many more nodes when the anchor technique is applied compared to the no-anchor case. We present similar temperature graphs in Figure 5.19. These graphs represent the final locations of data objects for *barnes-hut* and *nbody* when data are migrated based solely on attraction forces. Migration based on attraction forces results in data objects being distributed across many nodes of the chip when anchors are used; this is in contrast to the discovery in Chapter 4 that data migrate to a small number of center nodes when anchors are not used.

An additional benefit of applying migration based on attraction forces is the reduction in the overall communication distance. The cumulative distance traveled by messages is reduced from 25 million to 15 million hops for *barnes-hut* and from 55 million to 27 hops for *nbody*. The result is that the average distance of communication decreases from 5.2 to 3.2 hops for *barnes-hut* and from 5.3 to 3.1 hops for *nbody*.

Despite these reductions in communication demands, the overall impact of applying migration to the anchor versions of these two applications is an increase in execution times. Figure 5.20 shows the execution times for these applications when anchors are used with and without migration. We show results for two migration



(a) barnes-hut (b) nbody

Figure 5.20: We show the execution times for the anchor versions of *barnes-hut* and *nbody* with (d-a and dt-a+r) and without migration(b) when executing on the baseline architecture. Additionally, we show the execution times when processor demands are perfectly load balanced by the *perf* oracle.

strategies: data migration based solely on attraction forces (d-a) and data and thread migration based on both attraction and repulsion forces (dt-a+r). In order to show the impact of processor load imbalance, we also show the execution times when the *perf* oracle is used.

Figure 5.20 shows that both applications experience processor load imbalance; this can be seen by the large difference in the baseline and *perf* execution times. When migration is applied, execution times increase even further. However, this increase is not reflected in the *perf* execution time when migration is used. This implies that migration is exacerbating the processor load imbalance.

Unfortunately, the inclusion of repulsion forces in the migration strategy does not significantly reduce execution times by redistributing processor demands. Part of the reason the migration strategy is unable to improve processor load balance has to do with the frequency of state exchange information among the nodes. The use of anchors causes the frequency of thread invocations to increase while the thread lifetime decreases. This means that many threads begin and finish executing in a single interval between state exchanges. Consequently, the state exchanged becomes stale much faster, and our migration strategy cannot respond quickly to changes in

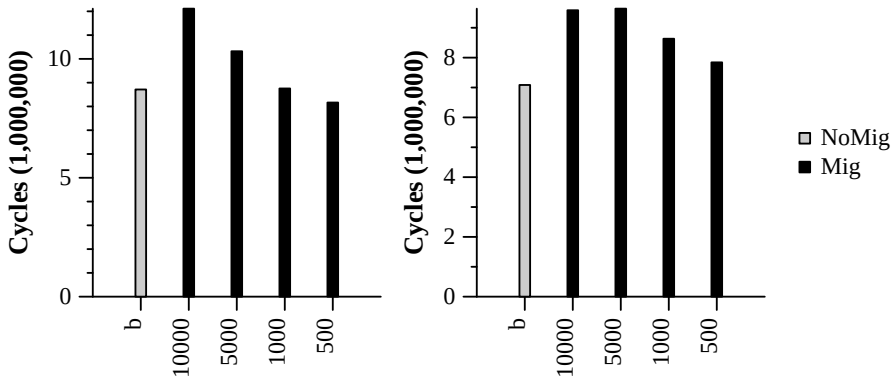


Figure 5.21: We show the execution times for the anchor versions of *barnes-hut* and *nbody* with migration as the interval between state exhanges varies. When the interval decreases, more opportunities become available to migrate objects in response to resource demands.

resource demands.

In Figure 5.21, we show the execution times of the two applications when data and threads are migrated based on attraction and repulsion forces. We vary the number of cycles between each state exchange. The interval used in Figure 5.20 is 10,000 cycles. As the number of cycles between state exchanges decreases, the migration strategy obtains more opportunities to migrate objects to redistribute resource demands. This results in execution time decreasing.

The implication of this result is that when anchors are used, nodes must exchange state information frequently. However, frequent state exchanges increase communication demands and are not necessarily sufficient to obtain well-balanced resource demands. For example, the execution time of *nbody* with migration is always worse than not using migration. The reason is that resource demand distributions are lost as soon as threads finish executing. Future threads will execute at the location of their associated anchor, not at a nearby node where a previous thread had been migrated to.

As future work, we suggest that information about processor demands must be associated with data when anchors are used. A migration strategy could then migrate

data based not only on communication demands, but also on processor demands. Recurring overloading of nodes' processor demands created by using anchors could then be avoided.

5.6 Conclusions

In this chapter, we presented a technique which reduces remote communication demands and helps expose inter-object communication patterns. Unfortunately, the overheads of using this technique place a larger burden on processing resources, namely additional instructions and load imbalance. As processors become relatively faster than network speeds, however, techniques like anchors that reduce the impact of communication on performance will become increasingly important. In Chapter 6, we explore how the anchor technique compares to caching in terms of both improving performance and reducing communication demands.

Chapter 6

Comparison to Caching

Caching is a well-established technique for improving processor performance. Caches improve memory system performance by reducing latency and by reducing bandwidth demands. First, they reduce the latencies of data accesses to data they contain. Second, by filtering accesses that hit in the cache, they reduce the bandwidth demands placed on the next level of the memory hierarchy. As dataset sizes have increased, larger and more levels of caches have been used to maintain low latencies and limit bandwidth demands.

Modern processors contain multiple levels of caches. These caches decrease the number of requests that reach shared resources, such as the network connecting processors to remote memory. Assuming the number of transistors allocated to a single node of a single-chip multiprocessor remains small due to increasing wire delays, the total amount of storage, and hence cache, on a single node can be expected to be smaller than in current processors. Consequently, increasing the size and number of individual processor caches is only part of a solution to reducing memory latencies and bandwidth demands in future single-chip multiprocessors. Additional mechanisms, such as our migration and anchor techniques, must manage the on-chip shared memory and communication resources efficiently.

In this chapter, we examine how caching impacts performance, and we examine how caching interacts with our migration and anchor techniques. We study whether data migration can still reduce communication demands when caching is used. We also consider whether the cost of losing data stored in a local cache exceeds the benefits gained by using idle processor resources when threads migrate. Our study of the anchor technique and caching includes comparing caching to anchors as well as exploring how these two techniques function together.

We begin by describing the cache coherence protocol used in our study. Next, we compare the performance of four applications on a system using caching to their performance on a cacheless system. We execute each of the applications on configurations where the cache size varies from 1KB to 32KB and the multithreading level changes from one to eight threads (four in the case of *equake*). We present both execution times and communication demands throughout this chapter; this is because the two are closely linked and communication demands are likely to become increasingly

important in future chips. For the remaining studies in this chapter, we choose two cache and multithreading configurations, where caching both helps and hurts performance, and examine how caching and our anchor and migration techniques interact at these configurations.

Our studies show that our techniques can complement caching depending on application characteristics. Migration can reduce execution times more than caching alone by both reducing communication distance and balancing resource demands. We also show that the use of the anchor technique can reduce the communication demands in a cacheless system to below those for a system using caching, and we show that the addition of anchors to a system using caches can reduce communication demands.

6.1 Communication Produced by Caches

Caches rely on spatial and temporal locality in data accesses to reduce both memory access times and the number of requests sent to the next level of memory. When a reference hits in the cache, no requests exit the cache for lower levels of storage and the access latency remains small. However, when accesses miss in the cache, requests are sent to the next level of the memory hierarchy and latencies grow. In this section, we briefly describe the communication demands generated by remote memory accesses in both cacheless and caching systems.

6.1.1 Communication Description

Figure 6.1(a) depicts a 4-by-4 single-chip multiprocessor with no first-level caches. In this system, a request for remote data must first travel to the directory associated with that data to discover the current location of the data in memory. A static mapping function decides which node's directory contains information about a given data line. Once the location of the data is ascertained, the request proceeds to the memory location. Finally, the requested data is sent from the data's memory node to the node that initiated the request.

The benefits accrued from using caches include the elimination of these three

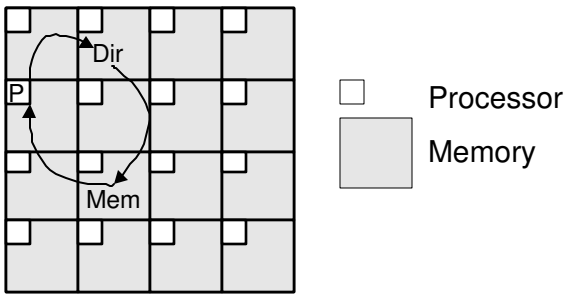


Figure 6.1: In a cacheless system, all remote requests first go to the directory and then proceed to the data's memory location before returning to the requesting node with the data.

communications on a cache hit. This reduction in communication, however, entails three primary costs. First, information about each cache line's current state must be stored either at the directory or in memory where the data resides. Second, caches introduce additional communication to maintain cache coherence as depicted in Figure 6.2. Dirty cache lines must be written back to memory and sent to the requesting node as seen in Figure 6.2(b). Invalidate messages must be sent to all caches sharing a cache line when another processor wants to write to the data as seen in 6.2(c). Third, entire cache lines are transferred on a request, even if some of the data is not needed, increasing the amount of data being sent through the networks.

As long as the communication demands eliminated by cache hits exceed the coherence traffic, the on-chip shared network will be less of a bottleneck in a caching system than in a cacheless system. This premise, however, requires nodes to request a relatively small amount of frequently-accessed data that can be reused before data is evicted due to conflict or capacity misses. In single-chip multiprocessors, the caches on each node will be smaller than modern-day, on-chip caches in order to keep clock cycle times small. Consequently, it is unclear whether these caches will always be able to contain a node's data working set. This concern is particularly valid if multiple threads create destructive interference when executing on a single node.

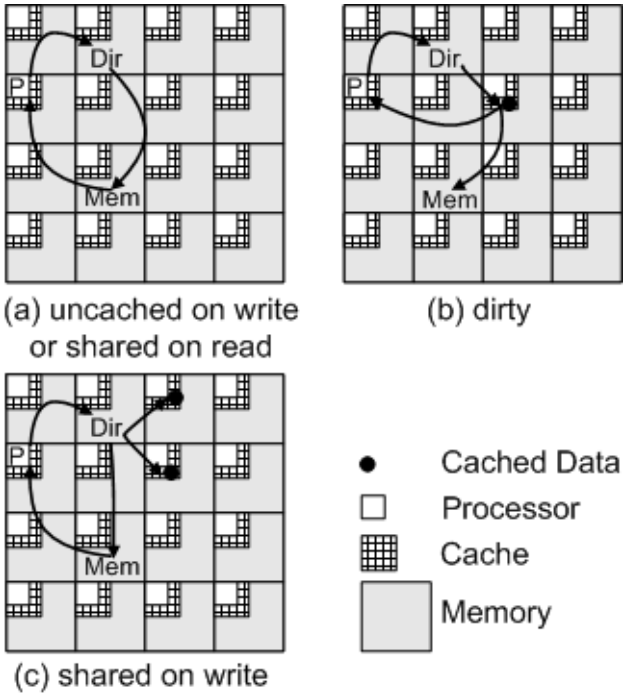


Figure 6.2: In a system using caches, three scenarios are encountered depending on the request type and the cache line's current state.

6.1.2 Directory Overhead

In the previous section, we indicate that every request that does not hit in the requesting node's memory must go to the directory to discover the current location of the requested data. In a cacheless system, the frequency of these directory requests can be reduced by retaining a small cache of memory locations for recently accessed data. When requests hit in this cache, the two messages involving the directory can be eliminated and the request can proceed directly from the requesting processor to the memory location found in this cache.

Eliminating messages to the directory can be more difficult in caching systems. For each line of memory, information about its current state and sharing nodes must be stored. The logical place for storing this cache state is either in the directory or in memory at the data's location. If cache state is stored at the directory, the directory requests cannot be eliminated as they were in the cacheless system because each request must check or modify the cache state. If the cache state is stored with the data in memory, then the migration of data would require the migration of cache state; however, we could eliminate requests to the directory in a manner identical to the method used in a cacheless system.

For the results presented in this chapter, we assume that a 128-entry, remote-location cache exists on each node in a cacheless system. This directory cache contains the memory locations of recently accessed remote data. Remote requests that hit in this cache do not need to go to the directory, eliminating one of the three messages on each remote request. In a system with caches, we assume that cache state is stored at the directory; therefore, a similar cache of remote data locations is not useful.

6.1.3 Potential Improvements beyond Caching

Looking back at Figure 6.2, we can see how our migration and anchor techniques can help reduce the number of misses as well as their latencies and bandwidth demands. In a cacheless system, migration moves data closer in memory to the processors accessing it. The anchors technique takes an alternate approach to minimizing the distance between computation and the data it uses; it moves the computation to the

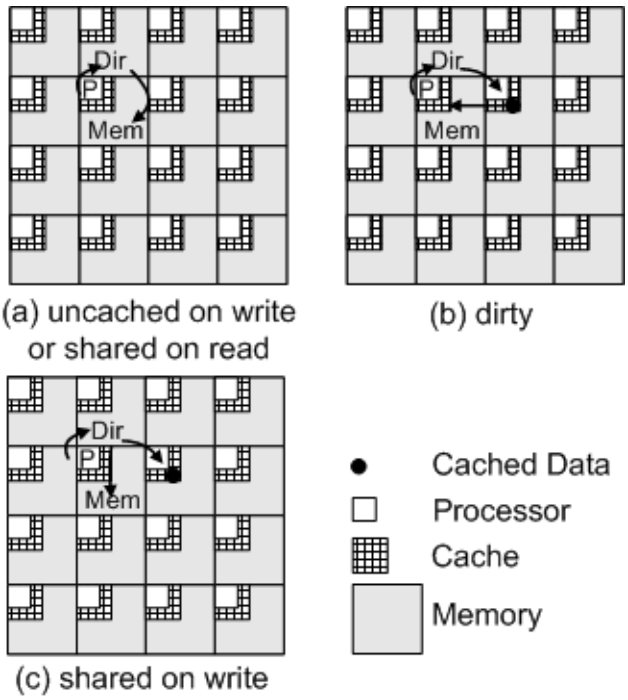


Figure 6.3: We show one possible transformation of Figure 6.2 when anchors and/or migration are applied. The distance between threads and data is reduced, and the distance between threads accessing the same data is reduced.

data it accesses, eliminating misses to the data. Both of these techniques are trying to reduce the number of nodes sharing data and the distance between those nodes.

When caches are used, the same techniques can be used to reduce the number of nodes sharing data as well as the distance among these nodes and the data's memory location. Figures 6.2(a)-(c) would be transformed into Figures 6.3(a)-(c). In Figure 6.3(a), we eliminate the memory's reply to the requesting processor because the accessing thread and data reside on the same node. The distance of the writeback message to memory decreases and the separate forwarding message to the processor is incorporated into the writeback of data to memory in Figure 6.3(b). Finally, Figure 6.3(c) shows that the number of sharing nodes that must be invalidated are reduced by the convergence of implicitly connected threads and data to a small set of nodes.

In the remainder of this chapter, we explore how migration and anchor techniques

can be used with and in place of caches to reduce the on-chip communication demands and improve performance through redistribution of resource demands.

6.2 Workload Characterization

Cache characteristics and application data access patterns strongly influence the performance of a cache on a given application; they affect the likelihood of cache hits and the quantity of coherence traffic. We begin our examination of caching by looking at how two factors affect our applications' performance: cache size and multithreading.

For our four applications, we examine how each application's execution time and communication demands vary when we use six different cache sizes: 0KB, 1KB, 4KB, 8KB, 16KB, and 32KB. In general, we expect execution times and communication demands to decrease with larger cache sizes. The goal of our study is to understand how caching interacts with the migration and anchor techniques in situations where caching helps and where caching hurts performance. Using small cache sizes enables us to understand how the interactions are affected when the data working set does not fit into the cache. For our applications, a 1KB cache allows us to create this environment; for applications with larger data working set sizes, this same scenario can be created despite the use of much larger cache sizes. Similarly, the use of large caches enables us to understand how the techniques interact when caching captures most of the available spatial and temporal locality. Therefore, by using the range of cache sizes, we can examine the entire spectrum of interactions.

We also examine how increased multithreading impacts these results. Multithreading allows applications to hide larger latencies at the expense of possibly increasing the number of simultaneous inflight memory accesses; multithreading may issue requests that conflict in the cache, causing the number of off-node requests to increase.

In addition to trying to understand how different cache sizes and multithreading levels affect our applications' performance, this study allows us to identify which cache and multithreading configurations positively and negatively influence performance. We can then focus our examination of interactions between caching and migration and anchors to a small set of configurations - those configurations where caching

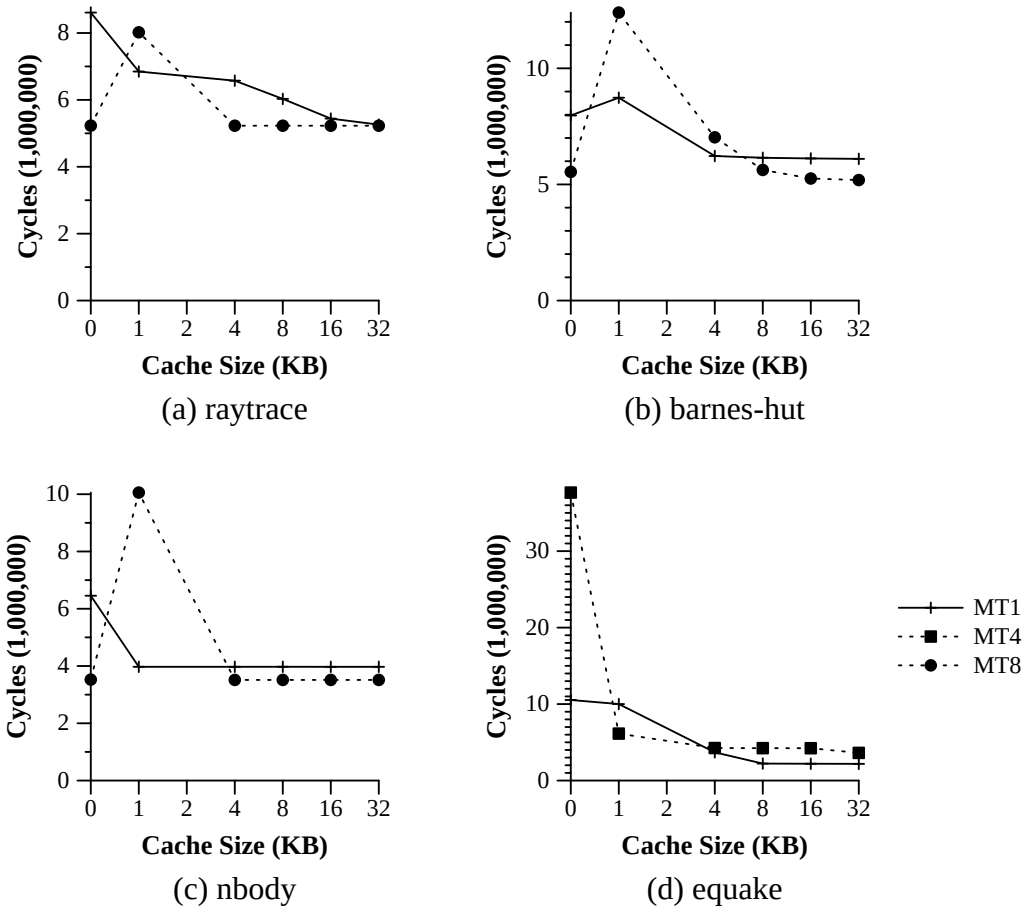


Figure 6.4: Execution times at different cache sizes and multithreading levels

benefits are limited and those configurations where caches perform well.

Figure 6.4 depicts the execution times for our four applications as we vary cache size. In this figure, we show curves for *raytrace*, *barnes-hut*, and *nbody* when only one thread executes on each processor (MT1) and when eight threads execute on each processor (MT8). Our analysis of *equake* includes evaluation of a multithreading level of one thread (MT1) and a multithreading level of four threads (MT4); increasing the level of multithreading above four threads significantly increases the amount of data tracked in our simulator, exceeding our simulation capability.

In general, the addition of larger caches reduces execution time; however, there are key configurations which do not adhere to this trend. When the cache size is only 1KB,

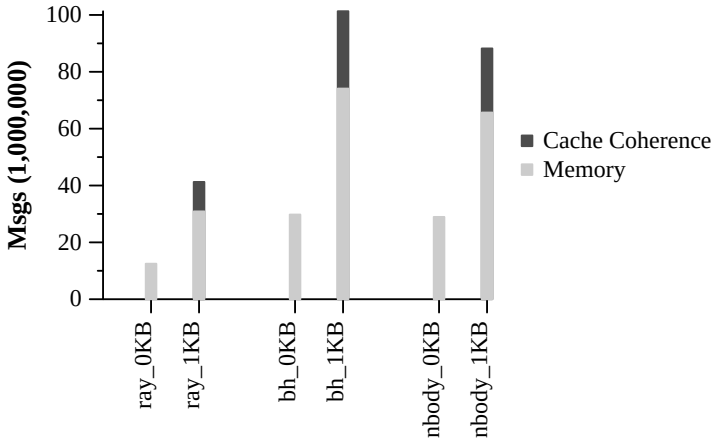


Figure 6.5: Breakdown of total number of messages when 1KB caches are and are not used (0KB). We only present the applications that have increased execution time when caches are used.

raytrace and *nbody* experience increases in execution time when the multithreading level is eight threads. *barnes-hut* experiences increases in execution time at a cache size of 1KB regardless of the level of multithreading. These configurations, therefore, can be classified as positions in which caching hurts performance.

In determining the reasons behind these performance changes, recall that in the preceding section, we discussed coherence traffic as a potential bottleneck in systems using caching. Figure 6.6 shows the number of flits sent into the on-chip network for the different configurations. Configurations where caching improves performance have decreased the number of flits traveling in the network; the reduction in communication demands due to cache hits exceeds any overhead created by caching. In contrast, the three applications that suffer from increases in execution time at a cache size of 1KB also suffer from large increases in flits sent into the network. This increase is due to both increases of memory messages and increases in coherence messages as seen in Figure 6.5. The inability of the cache to handle conflicting, or just too many, memory accesses at each node results in these increased communication demands.

These larger communication demands created by caching increase network contention, causing longer latencies for all messages. When no caches are used in these

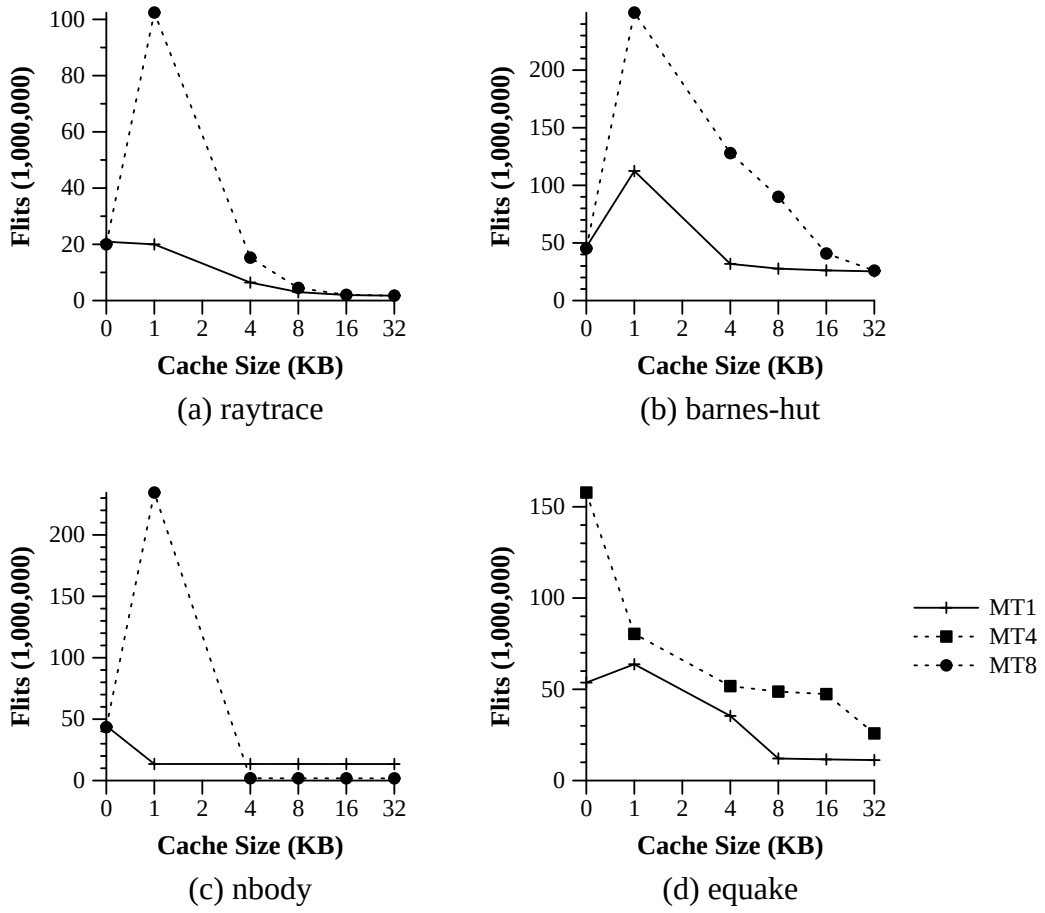


Figure 6.6: Total number of flits transferred during execution for different multithreading levels as cache size changes.

applications, each message experiences additional latency above the physical wire delay due to network contention: 1.9 cycles on average for *barnes-hut*, 1.3 cycles for *nbody*, and 0.9 cycles for *raytrace*. When a cache size of 1KB and multithreading level of 8 threads are used, these numbers rise to 54.3, 67.2, and 26.9 cycles. These larger latencies contribute to the rise in execution times.

In this section, we identified caching and multithreading configurations in which caching helped (32KB cache and multithreading level of 1 thread) and hurt (1KB cache and multithreading level of 8 threads) performance. The following sections

combine these configurations with migration and anchors to evaluate how each technique interacts with caching, paying attention to changes in communication demands and latency as well as execution times.

6.3 Combining Migration with Caches

Although caches eliminate large numbers of remote requests, they cannot reduce communication on compulsory cache misses, and they still experience capacity and conflict misses when the data set size exceeds the cache size. Additionally, they do not perform any resource load redistribution. In this section, we examine how caching interacts with both the communication distance reduction component of our migration technique as well as our technique's ability to move resource demands away from areas of high resource demands. In order to understand how migrating data and migrating threads interact with caching, we examine these two actions separately.

6.3.1 Data Migration

Our migration strategy moves data based on both locality and resource demands. In this section, we focus on how caching affects our ability to migrate data to improve its locality and, therefore, decrease communication distance. Because little benefit was gained from data migration in the *barnes-hut* and *nbody* applications, our discussion focuses on the *raytrace* and *equake* applications.

Recall from Chapter 4 that we collect locality statistics over time based on requests received at the data's memory location. We then select which data to consider for migration simply by selecting the data associated with every 100th remote memory request at a given node. Systems that use caching affect both of these steps in our migration strategy. Caching reduces the number of messages sent to memory. Consequently, it decreases the number of messages used in our statistics collection at a datum's memory location. It also increases the interval of time between each migration decision as it takes longer for 100 remote memory requests to be received at a given node.

Table 6.1: Average number of references per data object that reach the data object in memory. As we increase the cache size, the number of references decrease.

	Average References Per Object			
	raytrace		*equake*	
Cache Size	MT=1	MT=8	MT=1	MT=4
0KB	1238	1210	396	345
1KB	158	395	111	58
4KB	66	97	71	40
8KB	23	34	19	38
16KB	8	8	18	37
32KB	4	4	17	18

The first part of our evaluation on *raytrace* and *equake* examines how the information collected about communication patterns varies with cache size. Table 6.1 gives an example of how the number of references observed by our migration strategy changes with increasing cache sizes for *raytrace* and *equake*. The table depicts the average number of references that reach individual data objects at memory. As can be seen in the table, as cache sizes grow, the number of requests that reach data objects in memory, meaning they miss in local caches, decrease.

The second part of our evaluation examines how changes in the interval between migration decisions impact the benefits gained from data migration. Figures 6.7 shows the effect of migrating data based on both location preferences and communication demands for the two applications' execution times when a single thread executes on each node using a 1KB cache. On the x-axis, we vary the number of remote memory requests received at a node before another migration decision is made. We also show the applications' execution times when data migration is not used, *no*.

From the figure, we see that *raytrace* obtains little benefit from data migration regardless of the frequency of migration decisions. The reason that execution times do not decrease is that data migration does not reduce the average communication distance. When caching is not used, data migration decreases the average distance of memory requests to 3.6 hops (from 5.6 hops), while it can only reduce the distance

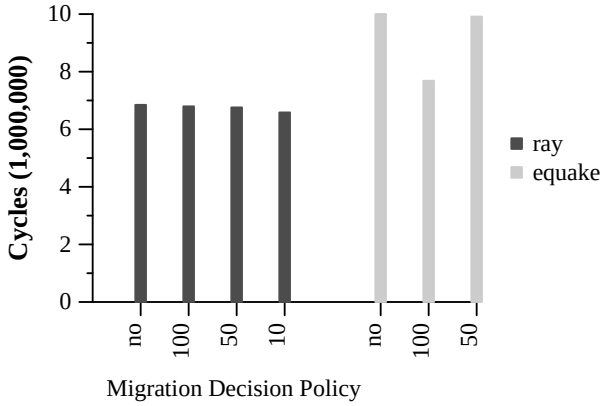


Figure 6.7: Execution times for *raytrace* and *equake* when a single threads runs on each node, each node has a 1 KB cache, and the number of remote requests between migration decisions is varied. We migrate data based on both location and communication demands.

to 5.0 hops (from 5.6 hops) when caching is used. Part of the reason for the reduced benefits of data migration on *raytrace* when caching is used is that fewer requests are reaching memory. Additionally, the smaller numbers of requests that miss in the cache and reach memory are spread out over long intervals of time, making them less likely to trigger a migration to improve locality. For example, more than 4,000 of the data objects in *raytrace* have their accesses from a given node spread out over more than 90,000 cycles.

Unlike *raytrace*, equake still obtains benefits from data migration when caching is used. When every 100th remote request instigates a migration decision, *equake* experiences a 24% decrease in execution time compared to caching alone. However, as the frequency of migration decisions increases, these performance benefits are lost. Migration decisions occur every 6,000 cycles on average when every 100th remote request triggers a decision; when every 50th remote request triggers a decision, migration decisions occur every 3,000 cycles on average. This shortened interval allows for more migrations but not necessarily better migration decisions. Additionally, since migrations make the data unavailable for use and increase communication demands, more frequent migrations contribute to the loss in migration benefits.

Depending on the application characteristics, data migration can still reduce the

communication distance for those memory requests not satisfied by the cache. However, data migration cannot always improve performance as seen in the case of *raytrace* in Figure 6.7. It is important to note that our migration strategy does not significantly hurt performance. When insufficient data is available for making decisions, it does not migrate data. Zero data migrations occured in *raytrace* when it was executed on a 32KB cache configuration executing one thread per processor, and only 90 migrations occured when *equake* was executed on the same configuration; neither caused any significant change in execution times. When data is available but overall communication demands are high, the inclusion of communication resources in our migration strategy insures that no negative effects result from migration. Without this repulsion force, migration can improve locality but hurt execution time. For example, when migration based solely on locality is used on the *raytrace* application as it executes eight threads per node, execution time increases by 37%. The use of communication resources in our migration strategy keeps this increase to 1%. Because of the balancing effect of repulsion forces, migration can be used with caching without any loss of performance.

6.3.2 Thread Migration

When caches are used there is an additional cost to migrating a thread. All of the data stored in the cache and used by the thread must be reacquired or a copy of the data must be forwarded to the thread's new node. The larger the amount of data stored in the cache, the larger the penalty encountered when migrating a thread. Our migration strategy forwards stack data, but does not forward any other data that resides in the cache.

Figure 6.8 shows the execution times for all of the applications when they execute with and without migration on cache configurations of 1KB and 32KB. We show results for multithreading levels of eight threads (four for *equake*). In making a thread migration decision, the migration strategy uses all three components: locality, computation demands, and communication demands.

raytrace and *equake* are two applications that benefited from thread migration

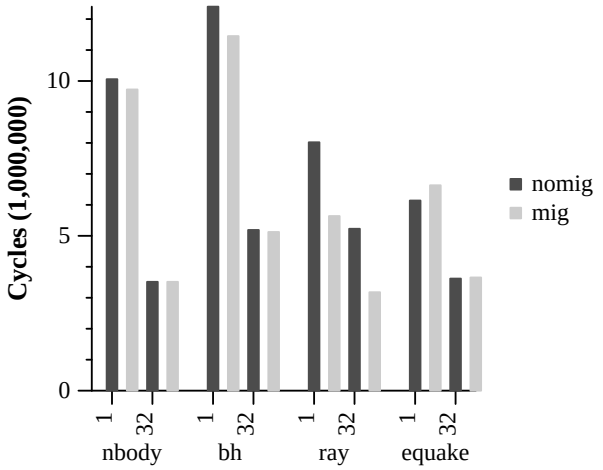


Figure 6.8: Execution times when thread migration is used with 1KB and 32KB caches.

in Chapter 4; *raytrace* obtained a large performance benefit from thread migration while *equake* obtained a relatively small benefit. As seen in Figure 6.8, the benefits of using idle computational cycles far outweigh the cost of having to reacquire the data that had been in threads' pre-migration caches for *raytrace*. However, *equake* does not obtain any benefits from thread migration. When the cache size is 1K, there is actually some performance degradation. This degradation is caused by the slow creation of new threads in *equake* when small caches are used. When threads finish, a master thread must create new threads. The small cache size means the master thread must re-cache all of its related cache lines before creating new threads. The delay in reacquiring those cache lines causes a delay in creating new threads. That delay results in the state exchanged between nodes not reflecting the fact that new threads are being created. Consequently, threads are moved to nodes that are creating new threads, causing those nodes to have overloaded processor resources. A better mechanism for detecting processor resource demands would alleviate this problem even with small cache sizes, eliminating this negative performance impact of thread migration.

nbody and *barnes-hut* were two applications that obtained little or no benefit from thread migration in isolation because their processor demands were well balanced.

Because caching reduces memory access latencies, it is possible that some threads with high cache hit rates could complete earlier than other threads with low cache hit rates, resulting in processor load imbalance. However, the resulting differences in thread lifetimes is small, and this effect results in small or no changes in execution times for these two applications.

6.4 Anchors and Caches

The goal of using the anchor technique is the same as the goal of using caches: to eliminate remote memory requests. Caches achieve this goal by retaining a copy of data locally at the accessing thread's location – in essence, moving data to computation. In contrast, the anchor technique moves computation to the location of the data it accesses. As shown in Chapter 5 and Section 6.2, both techniques can improve performance by reducing communication demands. In this section, we explore how these two approaches function together.

6.4.1 General Comparison

At first glance, caching and anchors appear to be competing techniques for reducing communication demands. In a system that uses caching, each remote memory access requires two messages: a request and a reply. If the data is retained in a cache, these request/reply pairs of messages are eliminated. When anchors are used, a message is sent to create a new thread at the location of the anchor, and when that thread finishes, a message is sent to the initiating computation to notify it of the subthread's completion. Similar to caching, all intermediate accesses to the anchor are eliminated.

The two techniques, however, are best suited for different situations; the key is determining each technique's favorable conditions. In order to understand in which scenarios each technique works well, we examine the number of messages created by systems using and not using caching and then evaluate how anchors affect the number of messages created.

We begin by looking at the messages created in a cacheless system. As discussed

in Section 6.1.1, requests to data residing on the same node as the requesting thread generate no messages. Requests to data located remotely generate three messages: processor to directory, directory to memory, and memory to processor. Assuming that N memory lines are each referenced R times and that the probability that a requestor's data resides locally is L, we can quantify the number of requests generated by NR references as

$$NR(1 - L)3.^{1} \tag{6.1}$$

In a system using caching, any requests that hit in the cache and do not require a state change generate zero messages. Otherwise, three messages are generated as in a cacheless system. If the data resides locally, the request to the directory must still be sent to update the cache line's state, but the message from the data's memory location to the processor is eliminated since they are the same node. Consequently, two messages are generated if the data resides locally and three messages are generated if the data resides remotely. Assuming cache hits occur with probability C, the total number of messages for NR requests is

$$NR((1 - C)(2L + (3(1 - L)))). \tag{6.2}$$

This equation does not include additional messages such as invalidates and writebacks needed to maintan cache state.

By setting Equations 6.1 and 6.2 equal to one another, we can determine a relationship between C and L,

$$C = (2L)/(3 - L). \tag{6.3}$$

Figure 6.9 graphs the number of messages sent in caching and cacheless systems when the probability that data is local varies and the probability of a cache hit varies from 25% to 75%.

Having determined the number of requests generated in each type of system,

[1] As discussed in Section 6.1.2, a local cache of memory locations would change the number of messages per request to two for those references that hit in the cache; we choose to use the conservative equation for our analysis.

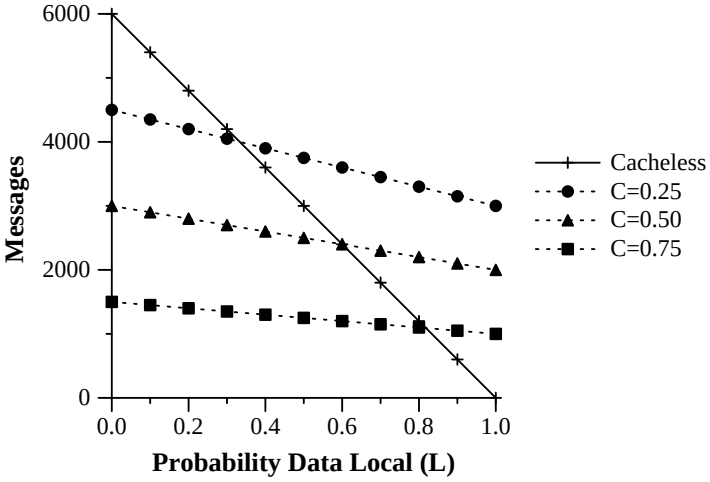


Figure 6.9: Number of messages sent in cacheless and caching systems when $N = 100$ and $R = 5$. N is the number of data and R is the number of requests to each data.

we can describe situations where each system performs well. Caching clearly works when cached data is reused before being evicted; this statement holds true regardless of where that data resides in memory. One of the clear benefits of caching is that it reduces communication to data retained in the cache even when that data resides at many different remote memory locations. If, however, a large fraction of the data lines accessed by computation resides locally in memory, less traffic will be generated by a cacheless system and the relative impact of the cache will be decreased.

The goal of our anchor technique is to make computation and data resident on the same nodes; in our equations, this corresponds to an increase in L's value. By increasing L, a cacheless system will generate fewer messages than a system using caching. In applications where large fractions of a computation's data resides at the same location, anchors will reduce communication demands more than caching.

The analysis above has left out several key factors in comparing the performance of caching and using anchors. We did not include communication demands created by caching, including invalidate and writeback messages. In addition to these messages, each cache line request transfers 64B of data, some of which may go unused. This additional communication can increase network contention, making a higher hit rate necessary for a caching system to perform better than a cacheless system.

Using anchors in a cacheless system also includes additional overhead. As discussed in Chapter 5, instructions that create remote subthreads increase the amount of computation performed. The decomposition of individual threads into multiple threads may increase the total number of stack lines used over the lifetime of the thread. Finally, processor demands are redistributed when anchors are used; this redistribution can either equalize processor demands or cause processor load imbalance.

6.4.2 Anchors versus Caching

Figure 6.10 shows the execution times for *nbody*, *raytrace*, and *barnes-hut* for different cache and multithreading configurations when anchors are not used. The figures also include execution times when caching is not used but anchors are used (*anch*). Additionally, they show execution times when anchors are used and processor demands are perfectly balanced (*anchbal*). Similarly, Figure 6.11 shows the total number of flits sent through the network for these applications, including a line for when anchors are used without caching.

As observed in Chapter 5, Figure 6.10 shows that using anchors without performing some form of processor load balancing can result in worse execution times than not using anchors; this is due to both increases in the number of instructions and redistributed processor demands. In a cacheless system, *nbody* and *barnes-hut*, two applications with initially well-balanced processor demands, incur increases in execution times when anchors are used compared to when anchors are not used. However, *raytrace* shows that increases in execution times are not always a consequence of using anchors. Using anchors improves *raytrace*'s distribution of processor demands, causing execution times to decrease. For the purposes of the remaining discussion, we assume the use of a processor load balancing technique that can be used with anchors to generate execution times bounded by the performance achieved using anchors and using anchors with perfect processor load balancing.

We examine the performance of anchors and of caches by looking at both communication demands and execution times. When caches hurt performance, such as at configurations with a 1KB cache and multithreading level of eight threads, the cache

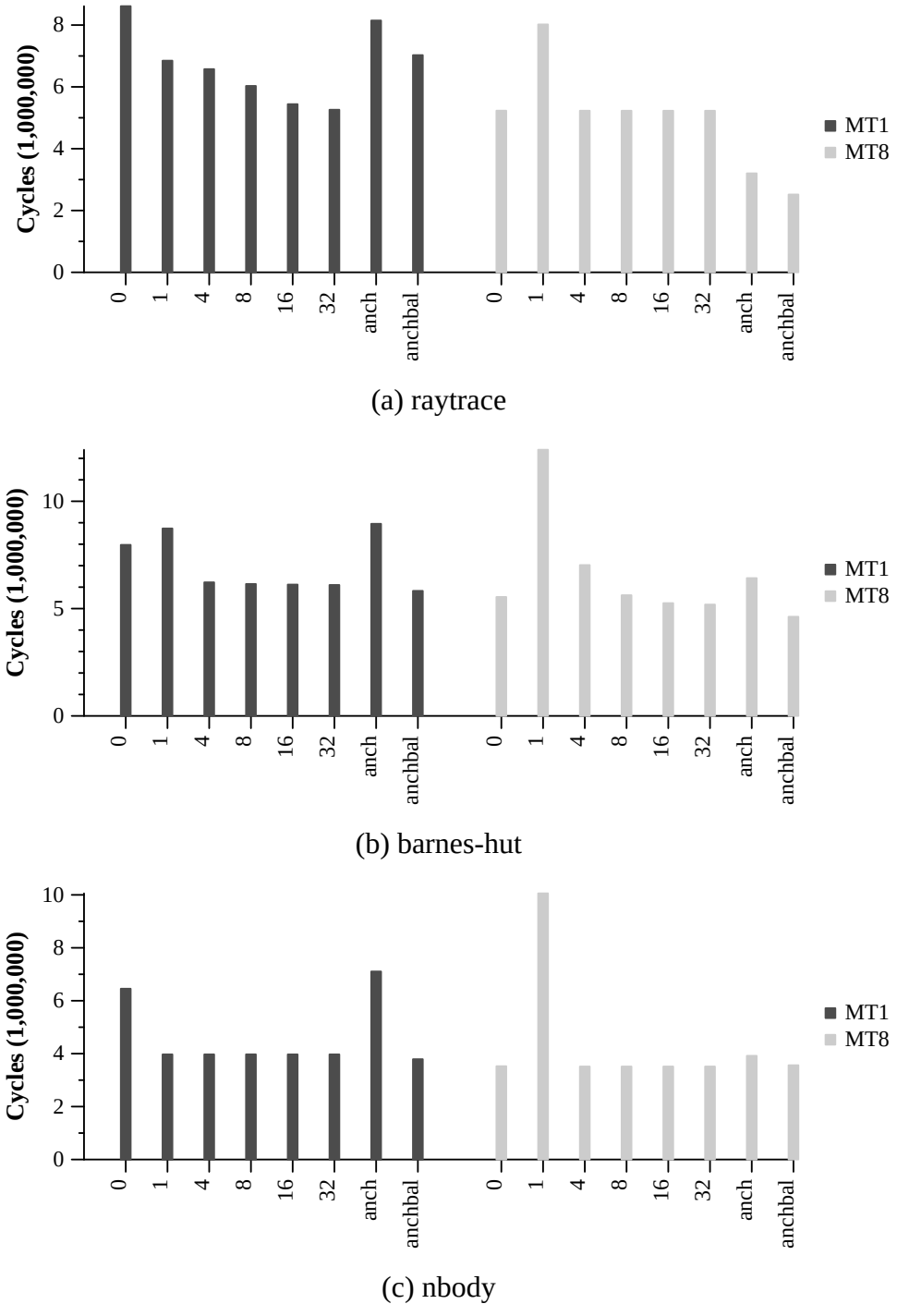


Figure 6.10: We show execution times when anchors are not used but caching is used. Additionally, we show execution times when anchors are used with no caching (*anch*) and when anchors are used with perfect processor load balancing (*anchbal*).

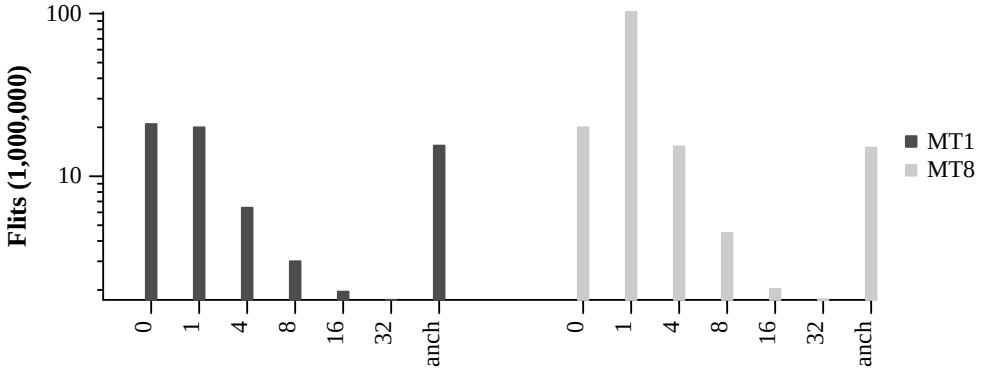
hit rate will be low. In these instances, the improved locality gained by using anchors reduces the number of flits transmitted below both the number generated in a system using caching as well as in a cacheless system. In these cases, the execution times, even without processor load balancing, are less than caching's execution times.

In the cases where caching helps performance, specific application data usage patterns impact the benefits obtained from the two techniques. Caching reduces communication more than using anchors when either a node's working set fits into the cache or constructive cache interference gained by multithreading reduces the number of requests sent off of the node. In the *raytrace* application, threads use small data sets, and threads residing on neighboring nodes use similar data sets. Consequently, caches can reduce the communication demands more than anchors. When multiple threads execute on each processor in *nbody*, constructive interference enables caching to reduce the number of flits below the number produced when using anchors.

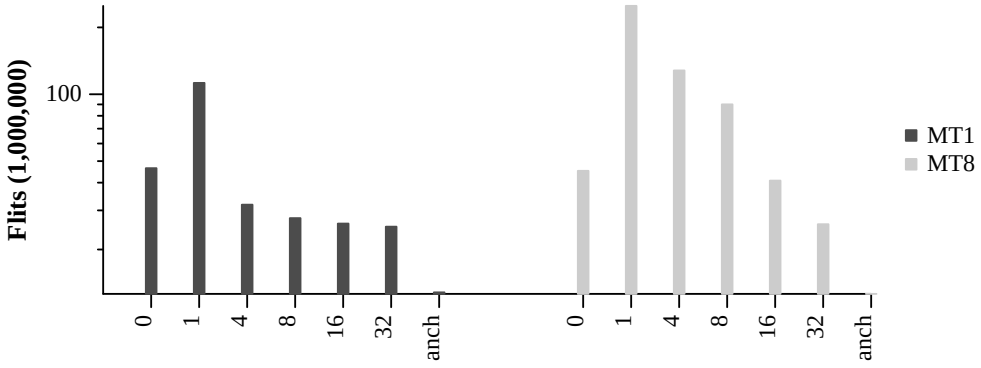
There exist situations, however, when caching cannot reduce communication demands as much as using anchors can reduce these demands. When a single thread executes per node in *nbody*, constructive cache interference cannot reduce communication demands. Many of the cache misses are compulsory misses. Additionally, a thread's entire working set cannot be retained in the cache for use by subsequent threads. Consequently, caching's ability to reduce communication demands is limited. Because each particle accessed by a thread fits into a single cache line and the amount of state transferred to create a new thread is slightly smaller than a cache line, using anchors can reduce the total number of flits sent into the on-chip network.

Regardless of the cache size used for *barnes-hut*, using anchors always generates at most half as many flits as caching. Not only are many of the cache misses compulsory misses like in *nbody*, but each particle accessed consists of multiple cache lines all resident at the same memory location. Thus, not only is the cache hit rate bounded, but using anchors causes requests to different data lines to hit in local memory.

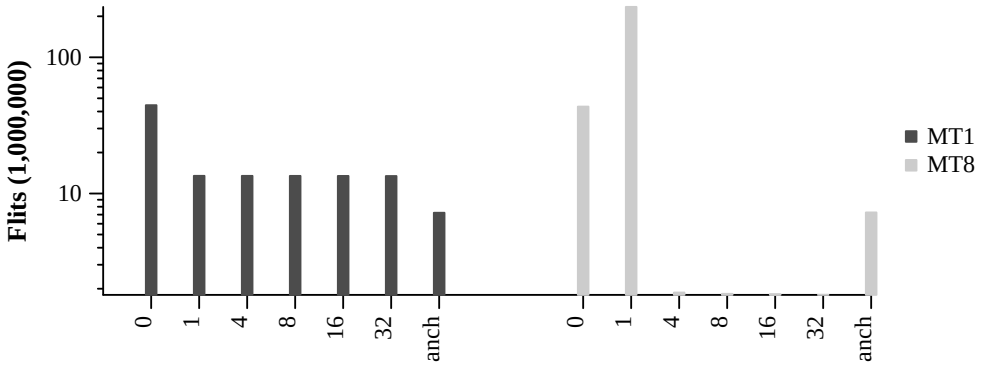
In summary, the anchor technique can reduce communication demands when cache hit rates are low, either because of conflict misses, as in the configurations with high



(a) raytrace



(b) barnes-hut



(c) nbody

Figure 6.11: We show the total number of flits when anchors are not used but caching is used. Additionally, we show the total number of flits when anchors are used with no caching (*anch*) and when anchors are used with perfect processor load balancing (*anchbal*).

levels of multithreading, or because of the compulsory misses. Anchors perform particularly well when multiple data lines that generate compulsory misses in a caching system actually reside at the same memory location.

6.4.3 Caching and Anchors

Having discussed the benefits of caching and of using anchors separately, we examine using the two techniques together. When the two are used together, the number of messages generated adheres to the equation for systems with caching. The use of anchors increases L; cache misses to data stored in local memory generate two messages instead of three. However, more messages must be created than when only caching is used to create new subthreads when anchors are used. Consequently, combining anchors with caches reduces the number of messages created only if the number of cache misses to data stored locally exceeds the number of messages used to create new subthreads.

In addition to the number of messages generated, redistribution of processor demands and total communication demands still impact performance. For each cache miss that our anchor technique turns into a local memory request, we eliminate the transmission of flits for an entire cache line. This reduction in total bytes sent through the network can reduce network congestion.

When a single thread executes on each node as seen earlier in Section 6.4.2, redistribution of processor demands caused by using anchors completely overwhelms any other benefits gained by using anchors. Caching alone should be used for these configurations unless some form of processor load balancing is performed. Therefore, we focus our attention on configurations where multiple threads execute on each node.

Figure 6.12 and 6.13 show the execution times and number of flits transmitted for the applications when a multithreading level of eight threads is used. The figures show results for a cacheless configuration (0KB) and for two cache configurations, 1KB and 32KB, when anchors are both used and not used. In cases where the cache cannot satisfy large fractions of the requests (1KB), using anchors can help reduce both execution times and the number of flits sent into the network. Both the execution

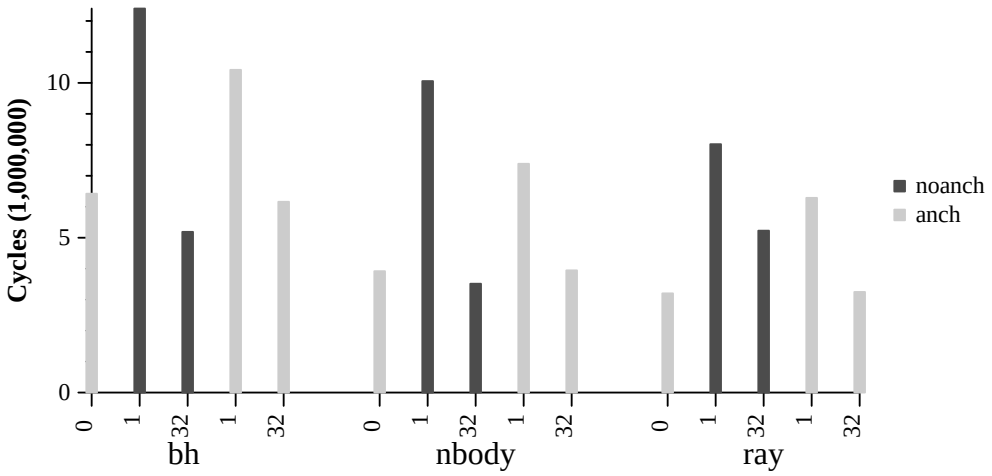


Figure 6.12: Execution times when caching and anchors are used on applications with a multithreading level of eight threads.

times and the number of transmitted flits, however, are still larger than when using anchors alone.

When a 32KB cache is used, the combination of caching and anchors increases execution times above the execution times for caching alone for *barnes-hut* and *nbody*. Because two messages are still sent on a cache miss that hits in local memory, the reduction in flits gained by adding anchors to caching is small. The *nbody* application experiences an increase in the number of flits with the addition of anchors due to the messages needed to create new threads remotely.

Unlike *barnes-hut* and *nbody*, *raytrace* experiences reductions in execution times when anchors are combined with a 32KB cache. The new execution time exceeds the execution time when only anchors are used; however, the increase is small. Looking at the number of flits sent through the network, we observe the number is both smaller than when caches are not used and larger than when only caches are used. Creating new threads increases the number of flits, but caching works to remove a number of the otherwise remaining remote requests.

In summary, the addition of anchors can reduce the negative impact of caching. However, choosing the single technique most suited for the application can obtain the best performance.

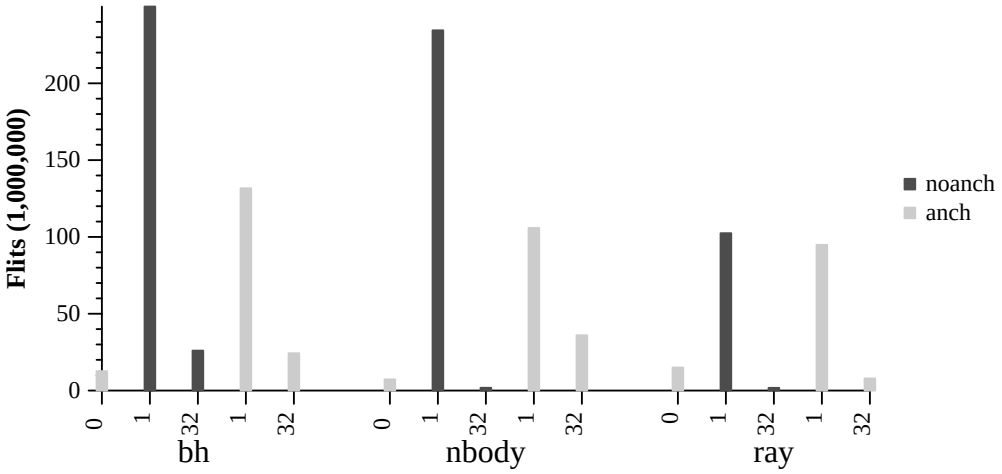


Figure 6.13: Total number of flits sent through network when caching and anchors are used on applications with a multithreading level of eight threads.

6.5 Application Summary

Table 6.2 summarizes the techniques we recommend for each of the four applications. In general, using caching with the anchor technique will obtain fewer performance benefits than using either technique alone. Additionally, when a cache cannot contain the application's data working set, it is better to use the anchor technique or migration without caching.

When applications suffer from large numbers of capacity or compulsory misses,

Table 6.2: Recommended techniques for each application

Application	Multithreading level	Technique
raytrace	1	Caching with Migration
	8	Caching with Migration
		Anchors
barnes-hut	Any	Anchors with Load Balancing
nbody	1	Anchors with Load Balancing
	8	Caching with Migration
equake	Any	Caching with Migration

such as in *barnes-hut* or *nbody* with a multithreading level of one thread, the anchor technique combined with some form of processor load balancing will perform as well as or better than caching. As computation becomes relatively faster than communication, the benefits of reducing communication demands by using anchors will grow.

Applications that experience constructive cache interference at higher multithreading levels, such *raytrace* and *nbody* with a multithreading level of eight threads, should use caching with migration to best improve their performance. Because *raytrace* also benefits from redistributed processor demands when anchors are used, we also recommend anchors be used to improve the application's execution times.

6.6 Conclusions

The limited size of per-node memory will limit cache capacity in future single-chip multiprocessors. Consequently, other techniques can help reduce on-chip communication. In this chapter, we showed that migration can reduce the average communication distance of memory requests that miss in the cache and can improve resource demand distribution. We showed that using our anchor technique in addition to caching can reduce communication demands. However, the use of anchors can perform significantly better than caching in scenarios where caches are either too small or applications exhibit large fractions of compulsory misses.

Chapter 7

Related Work

The work presented in this dissertation addresses resource management in a new environment: single-chip multiprocessors. Because single-chip multiprocessors are a recent development in computer architecture, we briefly discuss the motivation behind these designs. We also examine the different designs being explored in this new domain. Single-chip multiprocessors, however, are not the focus of this research; they are simply the platform for our research. Resource management and reductions in communication demands via migration techniques and strategic execution of computation at the data it uses (our anchor technique) are the areas of our contributions.

Because single-chip multiprocessors have existed for only a few years, little prior work about resource management has been explored within their specific domain. Consequently, we have used prior work in the domain of multi-chip multiprocessor systems as the foundation for our own research. In contrast to our work, most earlier work in migration has focused on either data or thread migration but not both; consequently, we discuss each body of work separately. Finally, we look at techniques that resemble our anchor technique or represent pieces of our anchor technique. We discuss prior research that examines the decomposition of computation as well as techniques that try to maximize the likelihood of computation executing at the location of the data it uses.

7.1 Technology Trends

Advances in technology have enabled the number of transistors placed on a single chip to exceed the billion transistor mark. Shrinking feature sizes have been accompanied by faster clock rates. These trends in the semiconductor industry have been coupled with an important constraint: growing wire delays. These growing wire delays combined with the increased number of transistors imply that the fraction of total chip area reachable in a single clock cycle will decrease [3].

The time it takes to transmit signals from one side of a chip to the other side will take an increasing number of clock cycles in future chips. The number of cycles required for across-chip communication can be kept manageable by dividing long wires into short wires connected by repeaters, where each short wire can be traversed

in a single clock cycle [23]. In this organization, the time for signal transmission is linear in terms of the number of small wires traversed.

Multiple cycle across-chip communications make global control difficult; large structures become less attractive for use in chip designs as it takes more than one cycle to reach all parts of a structure. At the same time, efforts to further exploit instruction level parallelism have seen diminishing returns, and the amount of thread level parallelism in server applications has been growing. This environment has given rise to the idea of placing multiple processing elements on a single chip in place of a single larger processing core.

7.2 Single-Chip Multiprocessors

Single-chip multiprocessors provide an organization that allows high computational throughput while limiting the effects of increasing wire delays [20]. A number of projects are currently exploring how best to design these microprocessors. This work can be broken into two strategies: placing a small number of complex processors on a single chip or placing many, simple processors on a single chip.

The logical first step in placing multiple processors on billion transistor chips is to place multiple complex processors and multiple levels of cache on them. Several chip manufacturers have taken this route. In 2001, IBM introduced the Power4 which contains two 1GHz 5-issue, superscalar cores as well as two levels of cache including a 1.41 MB shared L2 cache [14]. Intel and AMD both announced their impending dual-core chips in 2004; Intel's Montecito chip will contain two Itanium processors [25] and AMD's chip will contain two Opteron processors [4].

Researchers concerned about increasing wire delays and design complexity have looked beyond this logical next step. The premise behind chips like the Stanford Hydra chip [21] and Compaq's Piranha prototype [6] is that more thread level parallelism can be exploited if more, simpler processors replace the small number of large, complex processors. Hydra places four small MIPS cores on a single chip and creates fast communication among the processors via a shared, on-chip L2 cache. On Piranha, eight single-issue, in-order processors are connected via a logically shared

1MB L2 cache that is physically divided into eight banks.

As the fraction of reachable chip area continues to decrease, the number of these processing elements will need to grow [24][23]. Smart Memories [34], the Grid Processor Architecture [37], and RAW [46] are three projects exploring designs that connect large numbers of processing elements on a single chip with an interconnection network. Each processing element can be traversed in a single cycle and contains both an independent processor and a small amount of memory; the local memory decreases communication across the on-chip network [23][42]. The communication latency for these networks is linear in terms of the number of processing elements traversed.

The work in this dissertation applies to chips designed in this fashion. The key general features our work assumes are that the chip is populated by a large number of nodes that contain both a processor and small amounts of memory. These nodes function independently and are connected via an on-chip network where latency increases linearly with the number of nodes traversed.

7.3 Migration

Previous work in process and data migration has been based on two key assumptions reflecting the multi-chip environment from which they evolved. First, processor utilization affected performance more than usage of other resources, and, second, data locality was defined as either local or remote. We describe the insight gained by this research below despite differences from our work regarding the underlying assumptions about multiprocessor environments.

7.3.1 Thread Migration

Lin's Gradient Model [32] and Kale's Contracting Within Neighborhood (CWN) [27] techniques both used a distributed approach to migrate work away from heavily loaded processors. Lin's model dynamically migrated executing tasks based on a system-wide gradient surface, moving tasks towards lightly loaded processors. The gradient surface was created by dynamically propagating processor load throughout

the system gradually. CWN used a simpler approach which continually migrated newly created tasks towards a node's least loaded neighbor until either a node's load was less than it's neighbors or until the task had been migrated a predefined number of times. The limit on migrations constrained tasks to stay within a fixed radius of their parent, limiting global communication between tasks. Because this work was performed in a message-passing environment, the issue of data locality was not explored.

Load balancing research in shared-memory multiprocessors either ignored the issue of data locality, for example with the use of a central process queue [43], or included the effect of co-located data into scheduling decisions. Squillante and Lazowska [41] and Vaswani and Zahorjan [44] explored the impact of cache affinities on processor scheduling. The intuition is that executing processes develop an affinity for their current processor because their working set is resident in the processor's cache. By incorporating cache affinity information into scheduling decisions, performance could be improved by avoiding unnecessary cache refills. Chandra et. al [12] and Markatos and LeBlanc [35] extended the notion of data locality to include a node's entire memory hierarchy in their respective approaches.

7.3.2 Data Migration

The large latency difference between local and remote data gave rise to work that used either migration or replication to reduce remote memory accesses. In cache-coherent, non-uniform memory access architectures (CC-NUMA) such as DASH [30] and Alewife [1], programmers tried to partition applications so that data needed by a given process was allocated within the local memory of the cluster of processors the process was executing on. STiNG, another CC-NUMA machine, maintained a remote cache at each quad of processors; this remote cache maintained copies of data from remote memories in order to decrease the necessity of remote accesses [33]. Cache only memory architectures (COMA) use replication and migration of data to reduce remote accesses. In these architectures, data migrates to different parts of the distributed memory depending on which processors in the system are accessing

the data [19][18][17]. Researchers have compared the benefits of using each of these approaches separately [48][26] and together [16].

The techniques discussed above migrated and replicated cache blocks. Researchers have also examined ways to co-locate data with its accessing threads by either migrating or replicating pages in memory [9]. In systems incorporating process migration, page migration becomes especially important as processes are moved away from the initial location of their data in memory. Much of this work explores the different possible strategies for collecting state and making decisions. Bolosky et. al [10] examine migrating and replicating pages in the operating system based on page faults. Their work assumes two different access times, local versus global, simplifying their migration and replication strategies to either making pages resident at the accessing processor or not. In Verghese et. al [45] and Soundararajan et. al [39], the operating system dynamically migrated and replicated pages based on sampling of cache misses. LaRowe and Ellis [29] examine a mixture of migration and replication policies in conjunction with different mechanisms for triggering new decisions. Success in these approaches is measured in terms of making pages resident on the accessing nodes without creating large numbers of wasted page movements.

7.4 Thread Decomposition

7.4.1 Executing computation at data's location

The concept of executing computation at the location of the data it accesses has been explored both in shared-memory machines and distributed systems. Remote procedure calls [8] can be considered a mechanism for remotely invoking threads. Code annotations or program language extensions have been incorporated to allow the user to specify when code should execute at the location of specific data [22][12].

One of the key problems documented for these approaches is the movement of data. For example, in computation migration [22], all live variables move to the new location along with the current stack frame. Eager et al. [15] state that moving currently executing processes to improve processor load balance is not worth it because

of the state that has to be transferred. However, Eager et al. also state that remote invocation of processes are a viable option for balancing workloads.

7.4.2 Compiler optimizations

Existing compiler work improves data locality and/or reduces communication via code transformations [31][47][2][5][13]. Much of this work focuses on restructuring loops or tiling data in which data accesses are restructured. In an approach similar to our anchor approach, Maps [7] attempts to executes computation at the location of the data it uses. However, they achieve this by explicitly orchestrating instruction and data placement in the compiler.

7.5 Conclusions

Earlier work in multi-chip multiprocessors provides a foundation for the work to be performed on single-chip multiprocessors; however, it is important to keep in mind key distinctions between these different multiprocessor systems. Because the amount of local memory is small in single-chip multiprocessors, on-chip communication plays a much more prominent role in any techniques we create to minimize data access times than in earlier systems. The relatively short communication latencies and abundant processing resources permit us to incorporate lightweight techniques that were infeasible before. Understanding the differences enables us to retain mechanisms that will continue to work and create new ones that take advantage of single-chip multiprocessor's unique characteristics.

Chapter 8

Conclusions

As transistor capacities and wire delays continue to grow, chips will become increasingly organized with distributed control. Single-chip multiprocessors are one possible design point in this space. They consist of many independent nodes, containing both a processor and limited memory, connected by an on-chip network. The communication latency between nodes will be small for adjacent nodes (one or two processor cycles) and will increase linearly with the number of nodes traversed.

In this thesis, we explore the management of communication, computation, and memory resources in single-chip multiprocessors. Our work focuses on both reducing the distance and frequency of communication through the on-chip network and balancing resource demands. We present a reactive technique that migrates data and threads to both improve locality and reduce resource contention. Our proactive approach, called anchors, reduces communication frequency by executing computation at the same location as its most frequently referenced data.

8.1 Thesis Summary

Techniques attempting to reduce communication demands rely on an ability to detect which data and threads interact with one another. Our first contribution, therefore, is an architecture-independent framework for classifying applications based on the communication among their data and threads. We decompose a thread's execution over time into clusters based on the thread's data references. When looking across all threads' data clusters, the overlap among these clusters enables us to detect which threads use similar data and which data are used in conjunction with one another. This information can then be used to place data and threads on an architecture so that communication distance is minimized. It also illuminates ways of decomposing threads into smaller units of computation that access different data, aiding in the development of communication-sensitive placements of data and threads.

Resource demands and communication patterns cannot always be determined statically. Our second contribution focuses on both balancing resource demands and reducing communication distance during application execution. Our migration technique exploits a key feature of single-chip multiprocessors: communication to

neighboring nodes takes only slightly longer than communication on a single node. Consequently, data and threads can be moved away from nodes with highly utilized resources to neighboring nodes without incurring large latency penalties. Additionally, moving data and threads one node closer to the nodes with which they communicate reduces communication distance. Our technique models the competing demands of improving locality and equalizing resource demands as competing forces. To meet these goals, data and threads migrate gradually, one node at a time, based on recent communication patterns and resource demands for the small region of nodes surrounding their current locations. We show that, depending on application characteristics, our technique can reduce execution times by up to 41%.

Although not always possible, we can use static information about the interactions between computation and data to reduce communication demands. The third contribution of this dissertation is a technique called anchors; anchors enable the data most frequently used by a computation to specify the location where the computation executes. Communication between the data and the computation is eliminated because the two are located at the same node. We show that, by decomposing threads into independent computations based on the clusters of data identified by our framework and executing these smaller computations at the locations of their most frequently accessed data, we can decrease communication demands by up to 80% when compared to a cacheless system. As the average communication latency grows, either due to increases in the number of nodes on a chip or due to network contention, these communication reductions translate into larger performance improvements.

Finally, we explore how these techniques compare to and work with processor caches. Our results show that migration can still reduce communication distance when caching is unable to eliminate all remote data accesses. Migration can still redistribute resource demands, without incurring large performance penalties, when threads move away from data stored locally in their current nodes' caches. Our comparison of anchors and of caching identifies situations where each technique performs best. In situations where caches are unable to contain all of the requested data or where threads switch to using new clusters of data frequently, anchors can reduce communication demands and execution times more than caching alone.

8.2 Future Directions

While this dissertation provides insight into both the challenges of using single-chip multiprocessors and possible solutions for those challenges, it also provides the foundation for asking additional questions about this new environment.

The framework we have created for identifying relationships among data and threads permits us to manually decompose threads into smaller computations centered around distinct clusters of data. An open question is whether or not this process can be automated. Automation would require not only the identification of thread decomposition points, but it would also require the selection of the data used to anchor each computation. Concerns about processor load imbalance require an automated mechanism to incorporate some technique for avoiding all computations being assigned the same anchor.

An intuitive way of decreasing communication demands not explored in this dissertation is to compact data accessed together into contiguous memory addresses. Many communication patterns change over time, making it unlikely that the data used together reside in sequential memory locations. As communication becomes relatively more expensive than computation, it may become advantageous to use computation to copy data into sequential memory locations and, hence, improve data locality. This approach is similar to using scratchpad memory to store temporary data. Such a technique could reduce the data working set size and reduce the quantity of data transmitted through the on-chip network.

Finally, the work in this dissertation focuses on communication occuring on a single chip. The research does not study in detail how to move data on and off of the chip. Several issues need to be examined regarding off-chip communication. First, concerns about the amount of data to be moved onto the chip and the limitations of on-chip storage may suggest the need to reorganize data prior to it being fetched onto the chip. Second, a system may be composed of multiple chips connected and working together; a multi-chip environment creates different challenges from the ones explored in this thesis regarding data and thread placements.

Bibliography

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 2-13, June 1995.

[2] A. Agarwal, D. A. Kranz, and V. Natarajan, "Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared-Memory Multiprocessors", in *IEEE Transactions on Parallel and Distributed Systems*, pp. 943-962, Sept. 1995.

[3] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger, "Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 248-259, June 2000.

[4] AMD Press Release, "AMD Announces Technology Milestone With Its Multiple-Core Strategy," `http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~86455,00.html`, June 2004.

[5] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam, "Data and Computation Transformations for Multiprocessors," in *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 166-178, July 1995.

[6] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: A Scalable Architecture Based

on Single-Chip Multiprocessing," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 282-293, June 2000.

[7] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal, "Maps: A Compiler-Managed Memory System for Raw Machines," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[8] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," in *ACM Transactions on Computer Systems*, pp. 39-59, February 1984.

[9] D. Black, A. Gupta, and W. Weber, "Competitive Management of Distributed Shared Memory," in *COMPCON Spring '89, Digest of Papers*, pp. 184-190, March 1989.

[10] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott, "Simple But Effective Techniques for NUMA Memory Management," in *Proceedings of the 12th Symposium on Systems Principles*, pp. 19-31, December 1989.

[11] D. Burger and T. M. Austin. "The SimpleScalar Tool Set, Version 2.0", in *Computer Architecture News*, pp. 13-25, June 1997.

[12] R. Chandra, A. Gupta, and J. Hennessy, "Data Locality and Load Balancing in COOL," in *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, pp. 249-259, May. 1993.

[13] M. Cierniak and W. Li, "Unifying Data and Control Transformations for Distributed Shared-Memory Machines", in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pp. 205-217, June 1995.

[14] K. Diefendorff, "Power4 Focuses on Memory Bandwidth", in *Microprocessor Report*, 13(13):1-8, 1999.

[15] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "The Limited Performance Benefits of Migrating Active Processes for Load Sharing," in *ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pp. 63-72, 1988.

[16] B. Falsafi and D. A. Wood, "Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 229-240, June 1997.

[17] S. Frank, H. Burkhardt III, and J. Rothnie, "The KSR 1: Bridging the Gap Between Shared Memory and MPPs" in *COMPCON Spring '93, Digest of Papers*, pp. 285-294, Feb. 1993.

[18] E. Hagersten, A. Landin, and S. Haridi, "DDM - A Cache-Only Memory Architecture," in *IEEE Computer*, pp. 44-54, Sept. 1992.

[19] E. Hagersten, A. Saulsbury, and A. Landin, "Simple COMA Node Implementations," in *Proceedings of the 27th Hawaii International Conference on System Sciences*, Jan. 1994.

[20] L. Hammond, B. A. Hayfeh, and K. Olokotun, "A Single-Chip Multiprocessor," in *IEEE Computer*, pp. 79-85, Sept. 1997.

[21] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun, "The Stanford Hydra CMP," in *IEEE Micro Magazine*, March-April 2000.

[22] W. C. Hsieh, P. Wang, and W. E. Weihl, "Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems" in *PPOPP*, pp. 239-248, 1993.

[23] R. Ho, K. W. Mai, and M. A. Horowitz, "The Future of Wires," in *Proceedings of the IEEE*, 89(4):490-504, April 2001.

[24] J. Huh, D. Burger, and S.W. Keckler, "Exploring the Design Space of Future CMPs," in *International Symposium on Parallel Architectures and Compilation Techniques*, pp. 199-210, Sept. 2001.

[25] Intel Press Release, "Intel Silicon Innovation To Shape Direction Of The Digital World: Multi-Core Processors, Other Key Silicon Technologies Part of Platform Approach," `http://www.intel.com/pressroom/archive/releases/20040907corp.htm`, Sept. 2004.

[26] T. Joe and J. L. Hennessy, "Evaluating the Memory Overhead Required for COMA Architectures," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 82-93, 1994.

[27] L. V. Kale, "Comparing the Performance of Two Dynamic Load Distribution Methods," in *International Conference on Parallel Processing*, University Park, PA, pp. 8-11, Aug. 1988.

[28] S.W. Keckler, W.J. Dally, D. Maskit, N.P. Carter, A. Chang, and W.S Lee, "Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 306-317, 1998.

[29] R. LaRowe and C. Ellis, "Experimental Comparison of Memory Management Policies for NUMA Multiprcessors," in *ACM Transactions on Computer Systems*, pp. 319-363, Nov. 1991.

[30] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "The Stanford DASH Multiprocessor," in *IEEE Computer*, pp. 63-79, March 1992.

[31] A. W. Lim, G. I. Cheong, and M. S. Lam, "An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication," in *Proceedings of the 1999 Conference on Supercomputing, ACM SIGARCH*, pp. 228-237, June 1999.

[32] F. C. H. Lin and R. M. Keller, "The Gradient Model Load Balancing Method," in *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, pp. 32-38, Jan. 1987.

[33] T. Lovett and R. Clapp, "STiNG: A CC-NUMA Compute System for the Commercial Marketplace," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 308-317, May 1996.

[34] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," in *Proceedings of International Symposium on Computer Architecture*, pp. 161-171, June 2000.

[35] E. Markatos and T. LeBlanc, "Load Balancing vs. Locality Management in Shared-Memory Multiprocessors," in *International Conference on Parallel Processing*, St. Charles, Illinois, pp. 258-265, Aug. 1992.

[36] D. Matzke, "Will Physical Scalability Sabotage Performance Gains?" in *IEEE Computer*, vol. 30, pp. 37-39, Sept. 1997.

[37] R. Nagarajan, K. Sankaralingam, D. Burger, and S.W. Keckler, "A Design Space Evaluation of Grid Processor Architectures," in *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pp. 40-51, Dec. 2001.

[38] K. A. Shaw and W. J. Dally, "Migration in Single Chip Multiprocessors," in *IEEE Computer Architecture Letters*, Volume 1, Nov. 2002.

[39] 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 *Proceedings of the 25th International Symposium on Computer Architecture*, Barcelona, Spain, pp. 342-355, June 1998.

[40] SPEC OMP Benchmark Suite, http://www.specbench.org/omp.

[41] M. Squillante and E. Lazowska, "Using Processor-Cache Affinity Information in Shared Memory Multiprocessor Scheduling," Tech. Rep. FR-35, University of Washington Computer Science Department, Feb. 1990.

[42] M.B. Taylor et. al. "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-purpose Programs," in *IEEE Micro*, pp. 25-35, March/April 2002.

[43] A. Tucker and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed, Shared Memory Multiprocessors," in *Proceedings of the 12th Symposium on Operating Systems Principles*, Litchfield Park, AZ, pp. 159-166, Dec. 1989.

[44] R. Vaswani and J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors," in *Proceedings of ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, pp. 27-40, Oct. 1991.

[45] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, pp. 279-289, Oct. 1996.

[46] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to Software: Raw Machines," in *IEEE Computer*, pp. 86-93, Sept. 1997.

[47] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," in *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 30-44, June 1991.

[48] Z. Zhang and J. Torrellas, "Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA," in *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, pp. 272-281, Feb. 1997.