The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

SOFTWARE AND HARDWARE OPTIMIZATIONS FOR NOC-BASED CHIP MULTIPROCESSORS

A Thesis in

Computer Science and Engineering

by

Feihui Li

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2007

This thesis of Feihui Li was reviewed and approved* by the following:

Mahmut Kandemir
Associate Professor of Computer Science and Engineering
Thesis Adviser
Chair of Committee

Mary Jane Irwin
Evan Pugh Professor of Computer Science and Engineering
and A. Robert Noll Chair in Engineering

Yuan Xie
Assistant Professor of Computer Science and Engineering

Kenan Unlu
Professor of Mechanical and Nuclear Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

When semiconductor technology scales into the deep sub-micro regime, billions of transistors can pack into a single chip. It turns out that traditional monolithic processor architectures scale poorly with technology due to diminishing improvements in clock rates and the increasing interconnect delay. Such architectures cannot efficiently transform the fertile on-chip resources into computing capability. Chip Multiprocessors (CMPs), integrating multiple relatively simple processing cores on a single chip, are becoming the trend for microprocessor design, as witnessed by both industry and academia.

Processors, interconnection networks, and memories constitute the three major components of a CMP architecture. This thesis optimizes two of these components, namely, interconnection network and memory subsystem. When the number of processing nodes of CMPs scales up, a new type of interconnection network, Network-on-Chip (NoC), is normally employed. Thus, we study the emerging interconnection network for CMPs: NoC, and a critical component of the memory subsystem for CMPs: the on-chip, level-2 (L2) Non-Uniform Cache Architecture (NUCA). Targeting these components, this thesis proposes a set of hardware and software optimization schemes.

The first part of this thesis uses compiler-directed approaches to reduce the energy consumption of NoCs. Three compiler approaches are proposed, including proactive communication link turn-on/off, compiler-directed voltage selection for communication links, and profile-driven message rerouting. The experimental results with array/loop-intensive applications demonstrate that the compiler-directed approaches are more efficient in reducing the NoC energy consumption than pure hardware-based power management schemes.

The second part of this thesis targets the design of high-performance L2 NUCA design and optimization for CMPs. The contribution of this part includes both a novel 3D NoC-bus hybrid NUCA design and a migration-based NUCA design. We demonstrate, through extensive experiments, that the 3D circuit technology is quite efficient in shortening the wire delay and thus reduces the L2 NUCA access latency. The other NUCA proposal is a careful migration scheme (eviction-triggered migration and access-triggered migration), aiming at finding a proper physical location for each cache line in L2. The experimental results show that this scheme generates significant improvements in L2 cache performance.

Overall, this thesis demonstrates that it is possible to reduce power consumption and improve performance of NoC-based CMPs through hardware and software directed optimization schemes.

Table of Contents

List of Tables

List of Figures

Acknowledgments

My sincerest gratitude is to my thesis advisor, Prof. Mahmut Taylan Kandemir. He gave me the opportunity to explore the exciting research areas of computer architecture and optimizing compilers and guided my research with great enthusiasm. His constant encouragement, support and guidance were key for me to complete this thesis.

I am indebted to Prof. Mary Jane Irwin for her precious time devoted reading my research papers, my proposal, and this dissertation. I have benefited a lot from her broad vision and helpful advices. I am grateful to Dr. Vijaykrishnan Narayanan and Dr. Yuan Xie for their insightful suggestions, enlightening discussions, and their friendship. I would also like to thank my other committee member, Dr. Kenan Unlu, for his valuable comments on my Ph.D. work and for serving in my thesis committee.

Many thanks are to past and present MDL members for providing a supportive and productive environment during my stay at Penn State. Especially, I thank Guangyu Chen, Chrys Nicopoulos, Jie Hu, Guilin Chen, Madhu Mutyam, and Yuh-Fang Tsai for the happy time we worked together.

Finally, I owe my deepest gratitude to my family members. I thank my parents, my parents-law, my brother and my sister-in-law for their love, support and understanding. Also, this thesis would not happen without love, understanding and encouragement from my dear husband, Haisang Wu.

Chapter 1

Introduction

1.1 Chip Multiprocessors

The rapid scaling of semiconductor technology into the deep sub-micron regime has been accompanied by a dramatic increase in transistor density. Packing billions of transistors on a single chip is now a reality. The challenge for designing billion-transistor processors is how to efficiently utilize the fertile on-chip resources. Conventional monolithic superscalar architectures, which feature sophisticated microarchitectural functions such as dynamic scheduling, speculative execution, and dynamic branch prediction, scale poorly with technology [3] due to the diminishing improvements in clock rates and the continuing increase in interconnect delays. As a result, superscalar architectures cannot provide sustained performance growth that has been achieved during the past decades. Chip Multiprocessors (CMPs) [69, 33] emerge as a promising alternative for fully utilizing the increasing integration density of microprocessor chips. Compared to traditional monolithic architectures, CMPs integrate multiple relatively simple processing cores on a single die.

As stated by Hammond et al [33], CMPs are favorable (over conventional monolithic architectures) due to the advantages in speed, design complexity, execution parallelism, and other issues. First, CMPs dramatically reduce, and sometimes even completely avoid, the complexity of some microarchitectural components, such as the branch predictors and the issue queues. Consequently, processing cores of CMPs are relatively small and simple, and thus can operate

with quite high clock rates. Simple and fast processing cores also indicate accelerated design and validation cycles. Second, CMPs can employ thread-level and/or process-level parallelism. Having multiple threads of control in parallel is effective in hiding cache-miss penalties, since the execution of some threads can overlap with the stalls of others. superscalar architectures have only a single thread of control. They extract parallel instructions at the cost of complicated hardware, while the instruction level parallelism (ILP) is quite limited due to true dependencies and control dependencies. Simultaneous Multi-Threading (SMT) processors [99] support multiple concurrent threads on a single core. However, SMTs are still monolithic and suffer the same scaling problem as superscalar processors. For example, one important issue for SMTs is the contention among different threads to the single shared primary cache. CMPs normally assign independent primary caches to different processing cores and eliminate the contention. Besides these issues, CMPs are also preferable from the wire delay viewpoint. The wire delay is becoming dominant, since it scales much more slowly than the logic gate delay when technology scales. This fact affects the design of tradition superscalar processors. However, CMPs accommodate to such a trend very well because of their naturally partitioned structures. That is, frequently communicating components are clustered closely, while less relevant components are apart from each other. Finally, CMPs are arguably simpler than complex monolithic systems in terms of validation and verification.

Currently almost all major CPU manufacturers are producing commercial CMP chips. Sun's UltraSPARC T1 (formerly code named Niagara) [54] is an 8-core CMP, supporting up to four threads per core. This architecture targets workloads with high thread-level parallelism such as web server applications. Intel's Core Duo processor [42], a dual-core architecture, has versions for desktop, laptop and mobile applications. Orienting game/multimedia applications,

Sony, Toshiba and IBM (STI) [46] designed Cell processor. It incorporates a dual-threaded two-way-issue Power architecture and eight synergistic processing elements. Broadcom's BCM1480 [13], consisting of four Broadcom SB-1 MIPS64 CPUs, targets the next-generation computing, storage and networking applications. These CMP chips are clearly signaling the growing popularity of CMP architectures.

1.2 Network-on-Chip: Interconnection of Future CMPs

Most current CMPs employ conventional interconnects such as buses and crossbars for connecting processing cores. Buses, however, suffer from the resource contention issue. When the number of nodes on a bus increases, performance may degrade due to excessive conflicts. Therefore, buses are not considered appropriate for systems of more than 10 nodes [9]. Crossbars, although performing well with a reasonably large number of cores, become not favorable when the size of CMPs increases, due to their high costs.

A segmented network fabric, namely, the Network-on-Chip (NoC) [26, 10], is emerging as a solution to chip-level communications. The structure of an NoC resembles that of a traditional macro network. That is, an NoC is composed of on-chip routers and communication links. Each processing core is attached a router, which interfaces it to the entire network through neighboring routers. The data to be exchanged among on-chip processing cores (homogeneous or heterogeneous) are transmitted along routers and communication links. As in a traditional macro network, the transmitted data is assembled into packets. Each packet includes both the header field (containing destination information and etc.) and the actual data.

The adoption of NoC architecture is driven by its predictability, scalability, and parallelism. From the physical design point of view, NoCs can be built at the early design stage,

and their speed and power can be estimated as well, because of the regular, well controlled structure. NoC-based systems can also properly accommodate multiple asynchronous clocking through network protocols. From the perspective of microarchitecture, NoC-based systems are easy to expand by adding more routers, links and processing cores. The contention problem of centralized interconnects such as buses is alleviated in an NoC, as all the links in an NoC can operate simultaneously for transmitting different messages. Further, according to the computer community, "From a system design viewpoint, with the advent of multi-core processor systems, a network is a natural architectural choice. An NoC can provide separation between computation and communication, support modularity and IP reuse via standard interfaces, handle synchronization issues, serve as a platform for system test, and, hence, increase engineering productivity."[67] TeraFlop [40], a research CMP chip proposed by Intel, consists of 80 cores and connects these cores through a mesh-based NoC. Several CMP chips from academia, for example, RAW [94] and TRIPS [81] processors, also employ NoCs as their main communication fabric. In this thesis, our focus is on next-generation NoC-based CMP architectures.

1.3 Non-Uniform Cache Architecture: Memory Subsystem of CMPs

Besides the interconnection, memory subsystem is another critical factor for the success of CMP systems. With multiple cores residing on a single die, CMPs require higher memory bandwidth than traditional processors. Consequently, they usually need to incorporate large on-chip (level-2) L2 and/or (level-3) L3 caches. For example, Sun's Ultrasparc T1 contains 3MB of on-chip L2 cache, and Intel's Dual Core Itanium 2 includes 2.5MB of on-die L2 cache and 24MB of on-die L3 cache. When the number of processing cores on a CMP increases, the demand for memory bandwidth also increases, leading to even larger on-chip L2/L3 caches [38].

Traditional memory hierarchy design assumes each cache level has a constant access latency. However, the large size of on-chip L2 caches and the increasing wire delay make the L2 cache access latency not a constant any more. Instead, the latency is dependent on the distance between the accessing processor and the requested cache line. To employ this variance in L2 cache access latencies for improving L2 performance, Kim et al designed Non-Uniform Cache Architecture (NUCA) [48]. NUCA partitions the L2 cache into multiple individually addressable banks and connects these banks using Network-on-Chip. Cache banks exhibit different access latencies, which are determined by the relative locations to the processor. Data migration is employed in NUCA to move frequently accessed cache lines closer to the processor.

Prior NUCA designs for CMPs include a purely shared L2 cache [8], a purely private L2 cache [52] and a set of alternatives between purely shared and purely private caches [39, 22, 106, 7, 17]. Basically, all of these studies focus on minimizing the average L2 hit latency and/or maximizing the effective on-chip L2 capacity (i.e., minimizing the off-chip accesses), with an attempt to achieve an overall L2 performance enhancement.

1.4 Thesis Scope

This thesis focuses on optimizations targeting NoC-based CMPs. Specifically, we target two critical components of CMPs: interconnection (Network-on-Chip) and memory subsystem (shared NUCA-based L2).

Regarding Network-on-Chip, we focus on reducing the power consumption without affecting performance excessively. Prior studies [10, 100] indicate that, although an NoC architecture provides high communication bandwidth, it consumes a significant portion of the overall chip power due to the amount of NoC components (routers and links) and the communication

data volume. For example, in the case of the MIT RAW processor, the NoC connecting 16 on-chip tiles consumes 36% of the entire chip power, with each router consuming 40% of the per tile power [100, 94]. This observation motivates for research targeting the NoC power management.

Most prior NoC power management schemes are hardware based [90, 76]. That is, they adjust unused NoC components or NoC components under low utilization into low power modes based on the history utilization information. In contrast, in this thesis, we propose three compiler-directed approaches for reducing NoC power consumption. The first two approaches analyze the communication pattern of a parallel application and insert explicit network power control commands at proper points in the parallel code. The optimized parallel code will set NoC components to appropriate power modes at suitable time points during execution. The first approach we propose [57] assumes only two power modes (power-on and power off) and inserts proactive communication link showdown and activation commands. In comparison, our second approach [19] analyzes the critical paths within a parallel code in order to assign a suitable voltage level for each communication link. The third approach proposed [58] is profile-driven. Based on communication traces, the compiler reroutes messages by clustering messages into a small subset of communication links. The goal of this approach is to increase the idle periods of communication links and thus enhance the effectiveness of a pure hardware-based link shutdown scheme.

CMPs, with multiple cores on a single chip, impose a continuously increasing demand for memory bandwidth. Without sufficient memory bandwidth, different cores will compete with each other on using the limited set of pins and this might invalidate potential advantages of CMPs. In a CMP-based execution scenario, the latency of accessing off-chip memories is an order of magnitude higher than that of accessing on-chip memories. Therefore, the design of an

efficient on-chip cache system is crucial for achieving the potential performance of CMPs. The second part of this thesis focuses on the performance of the shared level-2 (L2) NUCA for CMPs. In this category, two micro-architectural schemes are proposed in this thesis. The first approach proposed employs the three-dimensional (3D) integration technology [27, 63] to design the high performance NUCA for CMPs, because the 3D technology decreases the lengths of cross-chip wires dramatically. The existing NUCA design [48, 8] is adapted to the 3D scenario. Our second approach focuses on a migration-based NUCA design for CMPs. This approach proposes two types of data migrations: eviction-triggered migration and access-triggered migration, aiming at finding a proper physical position for each cache line such that the average L2 access latency is minimized.

This thesis is organized as follows. Chapter 2 discusses three compiler-directed schemes for reducing NoC power consumption. We introduce the architecture abstraction and the NoC power model, explain the proposal details, and show the experimental results. In Chapter 3, a 3D NUCA architecture and a migration-based NUCA design for CMPs are presented. Further, we provide the details of these two schemes and the associated simulation results in this chapter. Finally, Chapter 4 concludes this thesis, and provides points for future research.

Chapter 2

Compiler-Directed Power Management for Network-on-Chip

2.1 Background

As the power consumption of interconnection networks is becoming an important concern [94, 81], we have witnessed several efforts devoted to optimizing the NoC energy consumption. We classify these related works into three categories: hardware-based, compiler-based, and task mapping.

The hardware-based NoC energy optimizations usually depend on adjusting link and buffer power modes according to NoC utilization. Based on the technique of voltage/frequency scalable links proposed by Kim and Horowitz [50], Shang et al [82] evaluated a history-based dynamic voltage scaling (DVS) scheme for communication links. The hardware is enhanced to lower down the voltage of the communication links in low utilization. Kim et al [49] designed a link shutdown scheme that minimized the number of power-on links while maintaining the connectivity of the network. They proposed an adaptive routing algorithm, and compared their scheme against a link voltage scaling approach. Soteriou and Peh [90] explored the design space for communication link turn-on/off. In their scheme, the decisions of turning off communication links are also based on the past message traffic observed. Besides these works, several low power NoC circuit design techniques [76] are also proposed.

Within the area of low power design, many compiler optimizations that target the functional units, register files, or cache/memory hierarchies were proposed [16, 47, 73, 2, 62]. Considering the general domain of optimizing communication and locality of communicated data in distributed-memory message-passing architectures, representative locality-oriented studies include [53, 15, 6, 32]. The main idea behind these studies is to minimize the inter-processor data communication. Until now, much less attention is paid to compiler works for low power CMPs. Prior compiler works [55, 56, 66] on CMPs target mainly at extracting and employing parallelism. Two closely related works include [18] and [89], where [18] optimized the energy consumption of NoCs using compiler-directed communication link allocation, and [89] is a software-directed link DVS technique based on off-line profiling.

Besides hardware-based and compiler-based schemes, another approach to reducing the NoC energy consumption was based on task mapping. Shin and Kim [84] used genetic algorithms to determine task assignment, tile mapping, routing path allocation, task scheduling, and link speed assignment for applications running on NoC-based systems. Asica et al. [4] proposed another genetic algorithm that allowed users to specify a particular optimization goal, such as performance or energy consumption. Hu and Marculescu [37] proposed an algorithm that mapped a given set of IP blocks onto a generic regular NoC and constructed a deadlock-free routing function such that the total communication energy consumption was minimized.

In addition, regarding NoC energy optimization, Benini and De Micheli [10] also identified possible approaches for energy savings, including node-centric and network-centric techniques. Simunic and Boyd [86] later implemented several of these techniques using a closed-loop control model.

Our proposed approaches are different from all the prior efforts in that they are compiler-directed NoC power management. Specifically, they use compiler-based (automated) communication analysis to identify the active/idle or communication slack patterns for communication links, and insert explicit link power management calls, i.e., link turn on/off or setting link voltage levels in the application code. Or we use profiling information to restrict the routing paths so as to shutdown more communication links. To the best of our knowledge, these are the first study of using compiler support to manage interconnection network power. In comparison, the prior hardware-based network power optimizations, cannot be as effective as our approaches for loop-intensive applications running on small size networks.

2.2 Architecture Abstraction

In the work aiming at managing the NoC power consumption, we assume that the application code is parallelized using message-passing directives. The resulting parallel program consists of a number of processes that are to be executed in parallel and communicate with each other through message passing. A message-passing based computing system (Figure 2.1) consists of a set of computing nodes ($\mathcal{N}$) and a set of switches ($\mathcal{S}$). Figure 2.2(a) shows the structure of a computing node ($N_i \in \mathcal{N}$), which contains a processor, a memory module, an outgoing message port (out-port), and an incoming message port (in-port). The out-port and in-port of a computation node are connected to an in-port and out-port of a switch, respectively. Figure 2.2(b) shows the structure of a switch. Each switch ($S_i \in \mathcal{S}$) has p in-ports and q out-ports. Each out-port of a switch is connected to an in-port of either another switch or a computing node; each in-port of a switch is connected to an out-port of either another switch or a computing node. A message sent by the application is first split into a set of fix-sized data packets. A

data packet flows into the switch through one of the p in-ports. The crossbar interconnect in the switch forwards this packet to an out-port of this switch based on the destination of this packet and the routing algorithm used by the network. From this out-port, the packet will be sent to an in-port of another switch, or the in-port of the destination computing node.

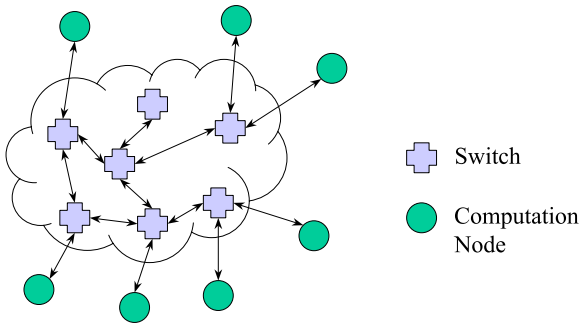
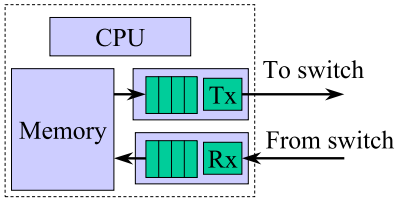
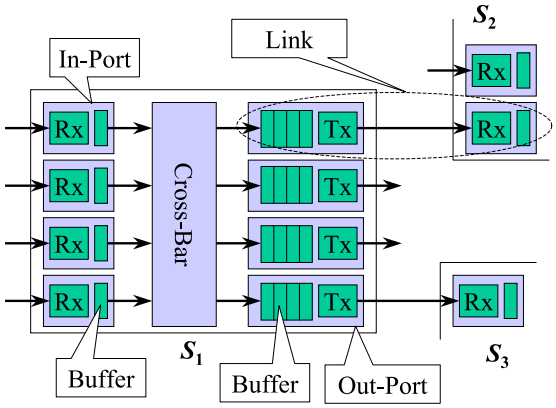


Fig. 2.1. A message-passing based parallel computing system consisting of computation nodes and a set of switches.

A packet received by an in-port is first stored in the buffer of this in-port. For a computation node, the in-port buffer can hold multiple packets. These packets in the buffer are first assembled into messages before being passed to the application. For a switch, however, an in-port buffer can hold only one packet at any given time. An in-port of a switch does not need a large buffer because the packet received by the in-port will be forwarded to an out-port immediately. The out-port that is trying to send a packet to an in-port is stalled if there is no free slot in the buffer of the targeted in-port. Each out-port has a FIFO (First-In-First-Out) buffer that can hold up to n packets.



(a) A computation node consisting of a CPU, a memory module, an in-port and an out-port.



(b) A switch with p ($p = 4$) in-ports and q ($q = 4$) out-ports. The buffer in each in-port can contain only one packet; the buffer in each out-port can contain up to n packets. The crossbar can forward packets from any in-port to any out-port.

Fig. 2.2. Internal structures of a computation node and a switch.

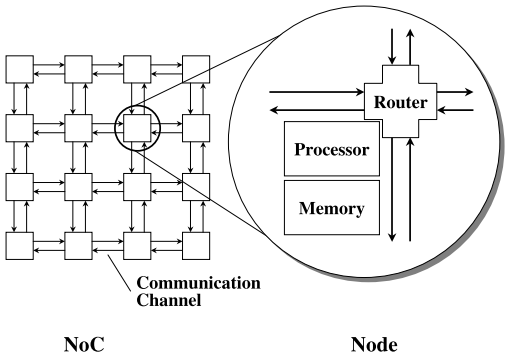


Fig. 2.3. A mesh-based network architecture.

We focus on an $M \times N$ (M rows, N columns) mesh architecture.[1] Figure 2.3 shows an example mesh. The nodes in the $M \times N$ mesh are numbered from 0 to $MN - 1$. We use p_i to denote the i^{th} node in the mesh. A pair of nodes, p_i and p_j, are adjacent to each other if and only if the following condition holds:

$$|i \mod N - j \mod N| + |\lfloor i/N \rfloor - \lfloor j/N \rfloor| = 1.$$

Each pair of adjacent nodes, p_i and p_j, are connected by a pair of links, $i \rightarrow j$ and $j \rightarrow i$. Link $i \rightarrow j$ ($j \rightarrow i$) transfers messages from p_i (p_j) to p_j (p_i). We see from Figure 2.3 that each computation node is bond with a switch in this mesh architecture. Without misunderstanding, when we mention a node in this work, we mean a node consisting of a processor, a memory module, and a switch (router).

We assume the system runs a single embedded application at a given time. This application consists of a set of parallel processes (i.e., it is parallelized to be executed over the mesh nodes). Each node in the mesh executes at most one process. A process running on a node, p_i, can send messages to a process running on another node, p_j, through connection $p_i \xrightarrow{*} p_j$. If p_i and p_j are adjacent, connection $p_i \xrightarrow{*} p_j$ contains only one link. On the other hand, if p_i and p_j are not adjacent, $p_i \xrightarrow{*} p_j$ contains multiple links, that is, a "multi-hop" connection. The set of links involved in a connection is determined by the specific routing algorithm used. In this work, we assume a static (deterministic) X-Y routing algorithm [30]. In this algorithm, a message is first continuously passed in x-dimension and then in y-dimension until it reaches its destination.

[1] Our approaches can be adapted to be used with other types of architectures.

2.3 Power Model

The increasing NoC energy consumption has motivated works on modeling NoC power consumption. Orion [101], an architectural-level power-performance simulator for interconnection networks, assesses different network architectures and the impact of different communication patterns on energy consumption. It enables the exploration for power-performance tradeoffs for interconnection network design. LUNA [31], a high-level power analysis framework for on-chip networks, gives the spatial and temporal power profile of the network by using link utilization as a high level power metric. Patel et al [70] have focused on the power constrained design of interconnection networks and proposed power models for routers and links in the network.

We use an interconnection network power model proposed in [31, 101], which can be represented as follows:

$$\begin{aligned} E_{network} &= E_{link} + E_{switch} \\ &= E_{link} + E_{crossbar} + E_{arbitration} + E_{buffer}, \end{aligned}$$

This means that the links and switches are the two main energy consuming components in an interconnection network. The energy consumed by the switches can be further classified into the energy consumption of crossbar, arbitration logic, and buffers. In this work for managing the network power, we focus only on the link energy consumption and model it in detail using the approach described in [90], which is to be elaborated in later analysis. The energy optimizations for the switches can be found in [49, 70, 100].

The energy consumed by the link circuitry turns out to be a significant portion of the total network energy consumption, which is pointed out by many previous studies [21, 82, 90]. In [21], the authors provide the on-chip link/router power numbers, showing that in many cases the link power consumption is larger than that of the router. The off-chip communication links consume an even higher portion of the total network power budget [90]. Therefore, one can expect significant power savings through optimizing the power consumption of the communication links.

2.4 Compiler-Directed Proactive Link Turnoff and Activation

Figure 2.4 illustrates our first proposal for optimizing power consumption of NoCs. The proposed compiler algorithm (proactive link energy optimizer) analyzes the communication pattern of parallel applications, and decides which links to shut down/activate and when to shut down/activate. Based on the decisions, the algorithm modifies the original parallel codes by inserting link turnoff and pre-activation instructions at proper positions and generates an energy optimized parallel program. Please note that the focus of this work is on the proactive link energy optimizer in stead of application parallelization and communication optimization.

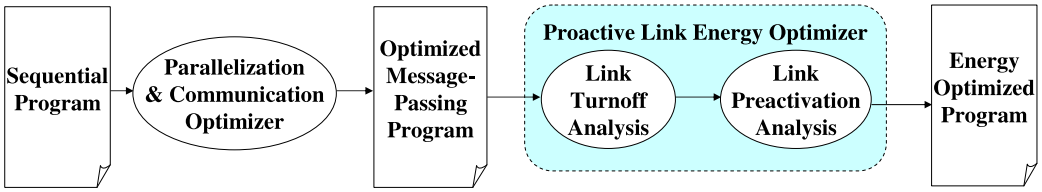


Fig. 2.4. High level view of proactive link energy optimization.

2.4.1 Hardware Support for Link Turnoff and Activation

Figure 2.5 depicts the structure of a link and associated switch components that supports link shutdown and activation. A time-out counter in the sender node monitors the link utilization by decreasing its value at each clock tick when the link is not in use (once the link is used, the counter is reset to the maximum value). When the value reaches 0, link components (Tx and Rx) are turned off to conserve energy. Power control logic inside the sender node is responsible for link activation as well. When the link reactivates, a small link state monitor in the receiver node detects it and turns on Rx.

To support compiler-directed link turnoff and activation, we extend both the hardware design and the message format. A 1-bit "SHARED" flag is added into power control logic. We use this flag to indicate whether multiple connections share this link. When SHARED is set to 0 (not shared), the associated link can be turned off by a passing message. Each message extends by adding two one-bit link control flags: "HOLD" and "LAST". Setting flag LAST to 1 indicates that the sending node will not send messages for a relatively long period of time after sending this message. Therefore, the links involved for transferring this message can turn off to conserve energy if SHARED is 0. The link turnoff in this situation is regardless of the time-out counter value. However, if SHARED is 1, indicating other messages need this link, a message with flag LAST as 1 cannot turn off the link. Setting flag HOLD of a message to 1 indicates that the sender will use the connection again in the near future. Thus, sending a message with HOLD $= 1$ sets SHARED flags of all links passed by the message to 1, in order to prevent other nodes from turning off these links. A link with SHARED $= 1$ still turns off when the time-out counter reaches zero. When a link is reactivated from the power-down mode, its SHARED

flag is initialized to 0. Table 2.1 summarizes how the flags of a message affect the state of a communication link.

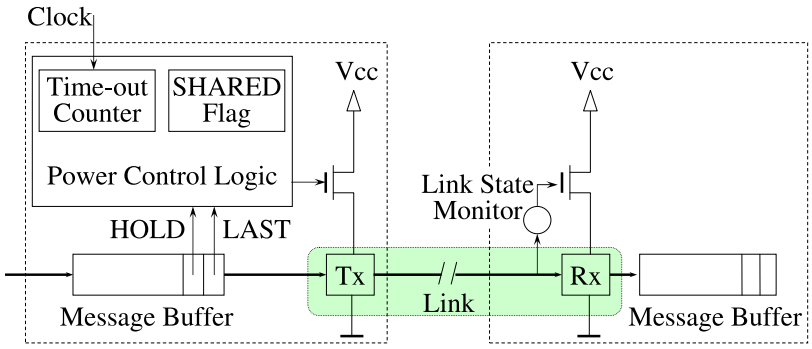


Fig. 2.5. The architecture supporting link shutdown and activation. Output buffer, Tx, Rx, and input buffer can turn off to conserve energy. HOLD and LAST are flags in the message header.

Using these control flags, we can control the states of links along the path from source node to destination node. This mechanism is especially important when the source and destination nodes of a given message are not adjacent to each other (when we have only the nearest-neighbor type of communication as in the case of stencil computations [80], the calculation of these flags can be significantly simplified). By making use of flag LAST, a program can turn off idle links earlier than a pure time-out based hardware mechanism would. Similarly, by utilizing flag HOLD in the message and SHARED associated with each link, the program can prevent a node from turning off links still needed by other nodes, and thus reduces potential performance/energy penalties.

Table 2.1. Controlling link state by using flags HOLD and LAST of a message and flag SHARED of the link. X: don't care; Reset: reset the time-out counter to the maximum value; $-$: the state (value) is not changed. Whenever there is a message coming and its targeting link is power-off, this link will reactivate.

Message Flags		**Current State**			**State After Message Transmission**		
HOLD	**LAST**	**SHARED**	**Counter**	**Link**	**SHARED**	**Counter**	**Link**
0	0	X	X	On	-	Reset	On
0	1	0	X	On	-	X	Off
0	1	1	X	On	-	-	On
1	0	X	X	On	1	Reset	On

2.4.2 Link Turnoff

Targeting loop-intensive, message-passing based parallelized embedded programs, the task of our compiler algorithm is to set flags HOLD and LAST properly for each message-sending operation inside the program. The input program is already parallelized and optimized using techniques such as [34, 98, 103]. For ease of discussion, we assume that the program contains only one loop nest $\mathcal{L}$. For a message-sending command "send$_k(p, m)$" (the k^{th} message-sending command m to processor p), if flag LAST message m is set to 1, links used to transfer message m turn off once m has been delivered. Besides, setting flag HOLD of message m to 1 prevents links passed by message m from being turned off by other nodes. Flags HOLD and LAST do not affect the correctness of the program, however, by setting their values properly, one can intelligently shut down the communication links without waiting until the time-out counter reaches 0.

The below example illustrates the code transformation performed by our compiler:

```
                                        for I⃗ = L⃗ to U⃗ {
                                            ...
for I⃗ = L⃗ to U⃗ {                          if(I⃗ ∈ H_{k,i}) m.HOLD = 1;
    ...                                     else m.HOLD = 0;
    send_k(p(I⃗), m);  ⟹                    if(I⃗ ∈ G_{k,i}) m.LAST = 1;
    ...                                     else m.LAST = 0;
}                                           send_k(p(I⃗), m);
                                            ...
                                        }
```

In this abstract code fragment, i is the ID of the node executing this code fragment, $\vec{L}$ and $\vec{U}$ are the lower and upper bound vectors for the loop nest, and $\vec{I}$ is the iteration vector.[2] At iteration $\vec{I}$, "send($p(\vec{I})$, m)" sends message m to the target node $p(\vec{I})$. $H_{k,i}$ is the set of loop iterations where "send$_k$($p(\vec{I})$, m)" sends messages with HOLD$=1$. Similarly, $G_{k,i}$ is the set of loop iterations where "send$_k$($p(\vec{I})$, m)" sends messages with LAST$=1$. The compiler computes iteration sets $H_{k,i}$ and $G_{k,i}$ for each message instruction executed on each node. In this work, we assume that both $H_{k,i}$ and $G_{k,i}$ can be expressed in Presburger formulas.

In stead of finding optimal $H_{k,i}$ and $G_{k,i}$, which is very hard if not impossible, we consider a heuristic. Before presenting the details, we define several auxiliary functions:

- $connection(i, j)$: the set of links used in the connection from node p_i to p_j. This function is determined by the used routing algorithm.

[2] Vector $\vec{I}$ keeps the loop indices from the outermost position to the innermost position. $\vec{L}$ and $\vec{U}$ are also defined as vectors and each contains an entry for each loop index, again from the outer most position to the inner most position.

- $targets(i, \vec{I})$: the set of nodes to which node p_i sends messages at iteration $\vec{I}$ of loop nest $\mathcal{L}$.

- $links(i, \vec{I})$: the set of links used by node p_i at iteration $\vec{I}$ of loop nest $\mathcal{L}$. This function can be computed as:

$$links(i, \vec{I}) = \bigcup_{j \in targets(i, \vec{I})} connection(i, j).$$

- $use(i, \vec{I}, \vec{\delta})$: the set of links used by node p_i during the period from iteration $\vec{I}$ to $\vec{I} + \vec{\delta}$ of loop nest $\mathcal{L}$. $\vec{\delta}$ is the threshold, which is determined by the power consumption of a link in different states and the energy penalty for turning on a power-off link. The time for executing all the loop iterations enclosed by range $[\vec{I}, \vec{I} + \vec{\delta}]$ is equal to T, the shortest idle period during which the energy saving by turning off the link can amortize the reactivation penalty. This function can be computed as:

$$use(i, \vec{I}, \vec{\delta}) \quad = \quad \bigcup_{\vec{J} = \vec{I}'}^{\vec{I} + \vec{\delta}} links(i, \vec{J}), \text{ where } \vec{I}' \text{ is the next iteration following } \vec{I}.$$

Based on the auxiliary functions, we give $H_{k,i}$ and $G_{k,i}$ for command "send$_k(p(\vec{I}), m)$":

$$H_{k,i} \quad = \quad \{\vec{I} \mid |connection(i, p(\vec{I}))| > 1 \wedge connection(i, p(\vec{I})) \subseteq use(i, \vec{I}, \vec{\delta})\};$$

$$G_{k,i} = \{\vec{I} \mid connection(i, p(\vec{I})) \cap use(i, \vec{I}, \vec{\delta}) = \phi\}.$$

The explanation for set $G_{k,i}$ is straightforward — flag LAST of message m is set to 1 if all the links for transferring m will not be used by source node i in the near future, i.e., during the

iterations $\vec{I}$ through $\vec{I} + \vec{\delta}$. Explaining set $H_{k,i}$, however, is a little involved. flag HOLD of message m is set to 1 only when both of the following criteria are satisfied: (1) transferring m requires multiple hops, and (2) links used for transferring m will be used by the source node of m in the near future, i.e., during the loop iterations $\vec{I}$ through $\vec{I} + \vec{\delta}$. The rationale behind these two criteria includes two parts. First, a connection connecting two communicating nodes is likely to share some links with another connection with another pair of nodes. If a node sends messages again along one of the connections soon, we do not want any link in this connection turned off by another node. A message with HOLD $= 1$ marks all the links along its way from the source node to the target node as "shared" (by setting SHARED flags of these links to 1) so that these links will not be mistakenly turned off. Second, we do not set flags HOLD to 1 for messages whose source and target nodes are neighbors. The reason is that, if a link used by two adjacent nodes is shared by a connection between two non-adjacent nodes, flag SHARED of this link will be set by the latter connection; if the link used by two adjacent nodes not shared by any other connections, keeping SHARED of this link at 0 allows us turn off the link immediately when we are sure that the source node will not use it for a certain period of time.

For a typical message-based parallel program, most communications take place between adjacent nodes. The compiler based scheme focuses on turning off idle links not shared by multiple connections promptly since the compiler can predict the behavior of these links accurately. As shared links has more complex behavior, which is harder for the compiler to predict, we do not turn off shared links in the compiler algorithm.

2.4.3 Link Pre-activation

A power-off communication link must be turned on (i.e., reactivated) before it can be used. Reactivation incurs both performance and energy penalties. We cannot avoid the energy cost due to reactivation, however, the performance penalty, however, can be hidden by *pre-activation,*. That is, we can turn on a power-off link a certain number of cycles before it is actually needed. The compiler performs pre-activation by inserting pre-activation instructions into the application program.

We first define $attach(i) = \{l_1, l_2, .., l_n\}$ as the set of links that connect node p_i with its neighbors. Since node p_i cannot directly control links not belonging to $attach(i)$, in this work, node p_i only pre-activates links in its own $attach(i)$ set. For ease of discussion, we define $Q_{j,i}$ is the set of iterations (executed on node i) using link l_j. The set of iterations in which we pre-activate link l_j is:

$$A_{j,i} = \{\vec{I} \mid [\vec{I} - \vec{\beta}_1, \vec{I} + \vec{\beta}_2] \cap Q_{j,i} = \phi\},$$

where $\vec{\beta}_1$ and $\vec{\beta}_2$ are two constant vectors.

Figure 2.6 explains the meaning of $\vec{\beta}_1$ and $\vec{\beta}_2$. Specifically, the overall execution time for loop iterations within $[\vec{I} - \vec{\beta}_1, \vec{I} + \vec{\beta}_2]$ is equal to T_D, the time-out period of the links; and, the overall execution time for the loop iterations within $[\vec{I}, \vec{I} + \vec{\beta}_2]$ is equal to T_P, the delay due to reactivating a turned off link. In this work, we express set $A_{j,i}$ in Presburger formulas. The below example shows the link pre-activation code inserted by our compiler:

```
                                    for I⃗ = L⃗ to U⃗ {
                                        if(I⃗ ∈ A_{1,i}) pre-activate link l_1;
for I⃗ = L⃗ to U⃗ {                    if(I⃗ ∈ A_{2,i}) pre-activate link l_2;
    ...                             ... ...
    send_k(p(I⃗), m);        ⟹      if(I⃗ ∈ A_{n,i}) pre-activate link l_n;
    ...                             ...
}                                   send_k(p(I⃗), m);
                                    ...
                                }
```

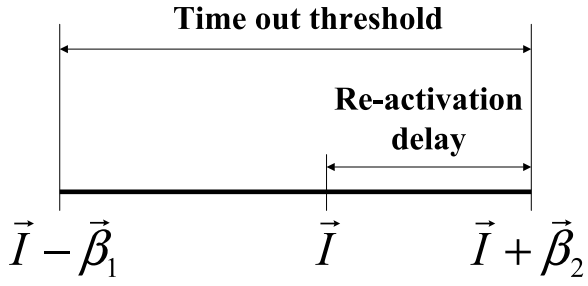


Fig. 2.6. Both $\vec{\beta}_1$ and $\vec{\beta}_2$ are constant vectors. The overall execution time for loop iterations within $[\vec{I} - \vec{\beta}_1, \vec{I} + \vec{\beta}_2]$ is equal to the time-out period of links; the overall execution time the loop iterations within $[\vec{I}, \vec{I} + \vec{\beta}_2]$ is equal to the delay due to activating a power-off link.

Pre-activation is in a sense similar to prefetching [64, 71], a commonly-used latency-hiding mechanism for cache memories. In prefetching, data/instructions are brought into cache memory before needed, in an attempt to hide memory access latency. Similarly, in pre-activation, a communication link is activated (actually reactivated) before it is needed to hide the associated

reactivation latency. We want to reiterate at this point that a wrong placement of the link shut-down or pre-activation commands by the compiler does *not* affect program correctness. It can only cause extra performance and/or power penalties.

2.4.4 Experiments

In this section, we present the experimental evaluation of our compiler-based, proactive approach for managing network power. We first introduce the experimental setup and then show experimental results.

We employed the link power model in [90]. Links consume constant power regardless of the utilization due to link signaling features. That is, even when a link is not transmitting data, it consumes the same power, P_{on}. When a link is turned off, its power consumption is assumed to P_{off}. When a link is reactivated from power-off state to power-on state, it incurs an energy penalty E_P during the transition period T_P. As in [90], the power during this transition period is equal to that in power-on state. Therefore, the total link energy consumption E_{link} can be expressed as:

$$E_{link} = \sum_{i=1}^{N} (P_{on} \cdot T_{on_i} + P_{off} \cdot T_{off_i} + n_i \cdot E_P),$$

where T_{on_i} and T_{off_i} are, respectively, the lengths of total power-on and power-off time periods for link i, n_i is the number of times link i has been reactivated, and N is the total number of links in the mesh network. In other words, the total energy consumption of links is obtained by aggregating the energy consumption of all the links in the network. Each link's energy contains energies spent in power-on state, power-off state, and the power-off to power-on transition

period. Based on the assumption of $P_{off} = 0$ as in [90, 49] , the expression above becomes:

$$E_{link} = \sum_{i=1}^{N} (P_{on} \cdot T_{on_i} + n_i \cdot E_P).$$

Our simulation framework is a SimpleScalar [5] based execution engine. Multiple simulation processes communicate with each other through a network simulation process. The network simulation process captures all the data communications across the network during the whole execution time, simulates the turn-on/off behavior of each link, and then calculates the overall link energy consumption. The network simulator is adapted from a high-level network power analysis framework, LUNA [31], which has been validated against Orion [101], a cycle-accurate network simulator.

As mentioned earlier, this approach targets loop-intensive applications. The benchmark codes used in our study are extracted from Spec and Perfect Club benchmarks. We give important characteristics of these codes in Table 2.2. Note that "jacobi", "lu", "mxm", and "red-black SOR" are frequently used codes in embedded multimedia processing. We hand-parallelized these codes. All the arrays in the benchmarks are decomposed into 9 parts (3×3) and distributed over the corresponding processors in the 3×3 mesh. The computation-to-processor assignment is performed using the owner-computes rule [79]. The last column of Table 2.2 gives the link energy consumption under the pure hardware-based link power management scheme. As mentioned earlier, in our experiments, we normalize all the energy results with respect to these hardware optimized values.

Table 2.3 gives the default values of important parameters used in our simulation. The power values of links is obtained from the power estimates for an on-chip router and its links

Table 2.2. Benchmark codes.

Name	Input array size	Number of messages	Volume of messages	Link energy
eflux	25.2MB	7546	220.6KB	330.6μJ
jacobi	180MB	18960	151.7KB	737.6μJ
lu	1.4MB	431233	280.8MB	232.8mJ
mxm	1.1MB	178200	142.6MB	149.1mJ
red-black SOR	90MB	18884	94.2KB	722.4μJ
tomcatv	25.2MB	14890	357.7KB	582.0μJ
tsf	112MB	18884	150.7KB	734.2μJ

(0.07μm, 1GHz) in [21]. We also present results when varying the values of some parameters. For most experiments, we assume a 3×3 two-dimensional mesh network.

Figure 2.7 shows the normalized link energy consumption values for the proposed compiler-based approach and the hardware-based approach. Each bar is broken into two components: "Link Power-on" and "Link Reactivation". "Link Power-on" represents the link energy consumed in the link power-on states, while "Link Reactivation" is the energy penalty for link reactivations. Each benchmark contains two bars, "HW" and "SW", which correspond to the hardware-based approach and the compiler-based approach, respectively. The results of "SW" are normalized with respect to the corresponding "HW" results.

Our main observation from this bar-chart is that the compiler-based scheme saves more energy than the hardware-based scheme for all the seven benchmarks. The reason is that the compiler can shut down a communication link proactively. First, when the compiler decides to turn off a link, the link does not need to wait for some time (e.g., T_D in the hardware-based approach). Thus, it can obtain further energy savings over the hardware-based approach. Note that, during this waiting period in the hardware scheme, the links still consume power. Second, since

Table 2.3. Default system configuration parameters.

Link frequency	1GHz
Power of the links for one on-chip switch	0.1446W
Link reactivation delay (T_P)	1000 cycles
Link reactivation energy (E_P)	36.2nJ
Link turnoff threshold for the compiler (T)	T_P
Processor frequency	1GHz
Packet header size (flits)	3
Flit size (bits)	39
Buffer size (flits)	64

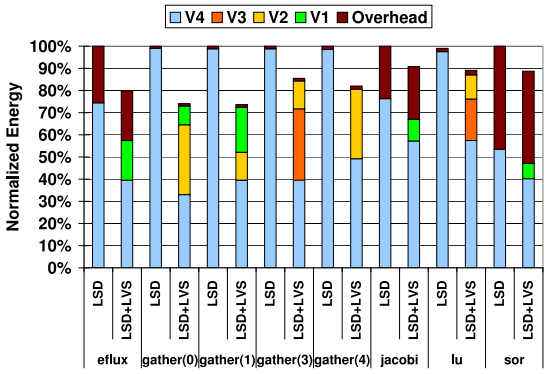


Fig. 2.7. Normalized link energy consumption. HW: hardware-based approach; SW: our compiler-based approach.

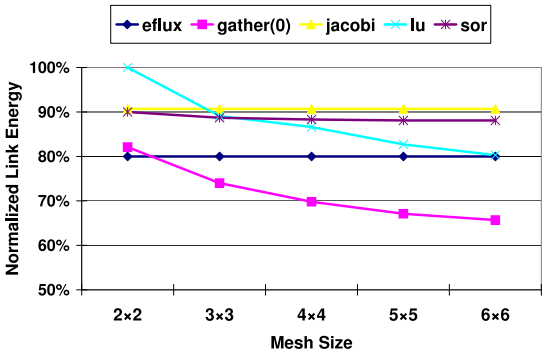


Fig. 2.8. Performance penalty of the hardware-based approach (over the case when no power optimization is performed).

the compiler is proactive instead of reactive, it can assess the benefits of a link turnoff more accurately. In contrast, the time-out based hardware approach cannot make sure that a link turnoff is really beneficial from the energy perspective. Consequently, sometimes, in the hardware-based link turnoff case, the energy spent for reactivating a link cannot be amortized by the energy saved through turning off this link. As a result, we see an average 18.3% link energy saving obtained by the compiler-based approach over the hardware-based approach. For benchmarks "lu" and "mxm", hardware-based and compiler-based approaches give similar power savings, because these two benchmarks feature frequent communications and large exchanged data volumes. Both approaches cannot execute many link turnoffs.

While our compiler-based scheme performs better than the pure hardware-based one from the power consumption angle, we also need to consider performance for a fair evaluation. We found in our experiments that, as expected, the compiler scheme does not cause any observable performance degradation. This is because it can pre-activate a link which is in the power-off state before it is actually needed. The hardware scheme, on the other hand, does incur some performance penalty, as illustrated by the bar-chart in Figure 2.8. The average communication latency increase it brings is 6.6%. Our base case hardware approach does not incorporate an adaptive routing algorithm when a packet requires a power-off link. This could be one reason why we observe this latency penalty for the hardware-based approach. However, even in [90] that exploits a fully adaptive routing algorithm Opt-Y, the hardware approach incurs an average network latency penalty of 3.5% under the same link reactivation delay. Therefore, we can conclude that the compiler-based scheme is preferable over the hardware based scheme from the performance angle as well. This is particularly true for embedded on-chip networks, where adaptive routing is not typically employed due to its high energy costs [37].

We next perform sensitivity studies by changing the default mesh size and the input data size. Figure 2.9 illustrates the results with different mesh sizes. Only results for benchmark "jacobi" are presented because the trends observed with other benchmarks are very similar. We see that, with different mesh sizes, link energy behavior is quite stable. One can expect our approach to be successful even with large meshes. Results with different data sizes are in Figure 2.10. Besides this default size, denoted as 1X, we made experiments with data sizes X/4, X/2, 2X, and 4X. In this figure, the link energy values achieved by the compiler-based approach are still normalized to that of the hardware scheme. We see that the relative energy savings achieved by the compiler-based approach over the hardware scheme are quite consistent.

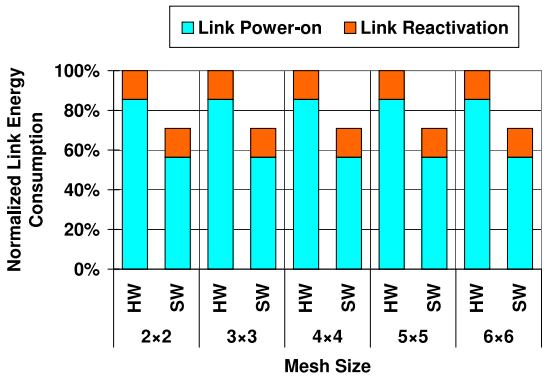


Fig. 2.9. The impact of mesh size (benchmark: jacobi).

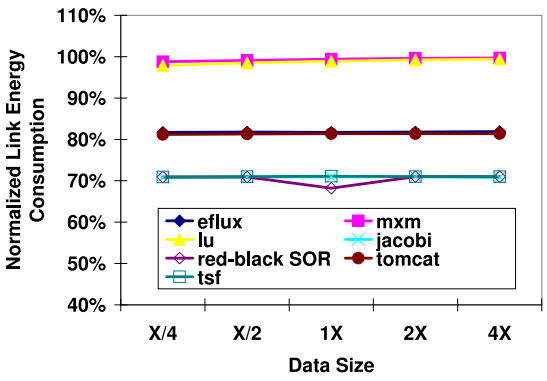


Fig. 2.10. The impact of data size.

2.5 Compiler-Directed Voltage Scaling on Communication Links

2.5.1 Motivation

With voltage scalable communication links inside NoCs, there exist chances to scale down the link voltage in a in a performance sensitive manner. Figure 2.11 depicts two example scenarios. In the first scenario (Figure 2.11(a)), a pair of processors in different NoC nodes communicate with each other using non-blocking send and blocking receive operations. Note that the amounts of data sent between them are different. Consequently, the communication from processor 2 to processor 1 can be performed more slowly than that from processor 1 to processor 2. One can potentially scale down the voltage and frequency on the communication channel from processor 2 to processor 1, thereby reducing power. To prevent any performance penalty, the scaling factor should be determined based on the difference between the magnitudes of the communication volumes. In the second scenario (Figure 2.11(b)), a pair of processors first send data to each other and subsequently perform some computation. Let us assume that, while the data volumes in the two communications are the same, the amount of computation performed by processor 2 is much larger than that performed by processor 1. As a result, one can scale down the voltage/frequency on the communication channel from processor 1 to processor 2. These two scenarios illustrate that the opportunities to scale down voltages can come from the differences between the communication volumes on channels and/or from the differences between the computation volumes on NoC nodes; and both of these variances can be exploited through voltage scaling to reduce NoC power consumption without significantly affecting performance. In this work, we do not apply voltage scaling to processors; our focus is on NoC communication links.

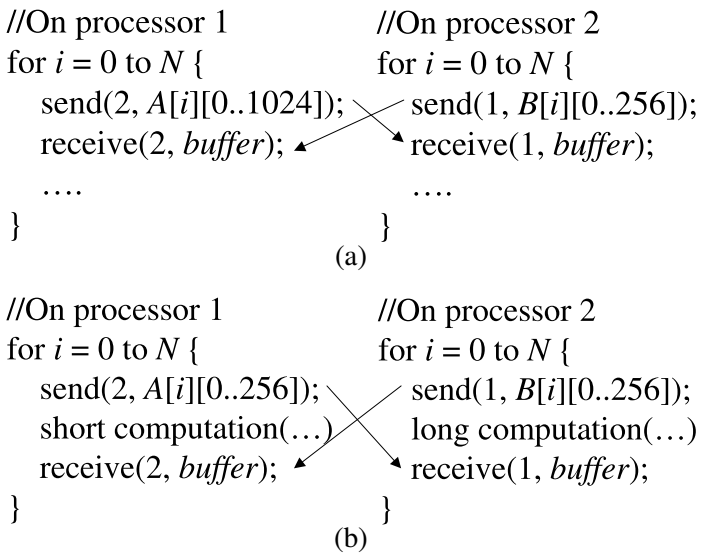


Fig. 2.11. Two example scenarios.

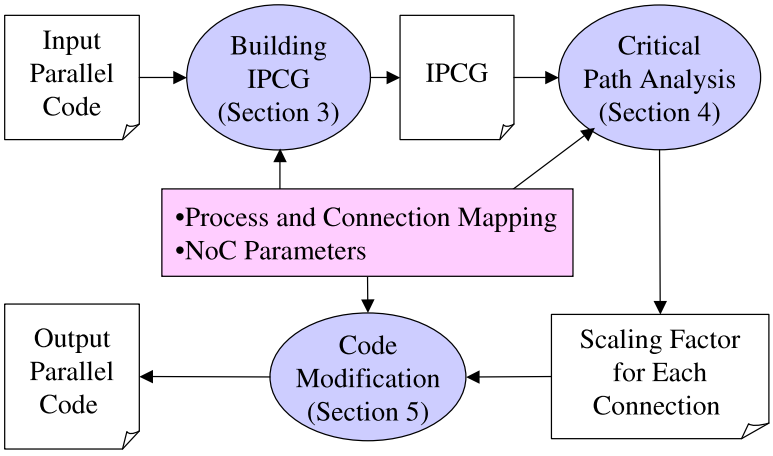


Fig. 2.12. High level view of compiler-directed channel voltage scaling.

The voltage/frequency scaling approach proposed in this section has three major components. The first component analyze the input code, builds a graph called the *Interprocess Communication Graph (IPCG)*, which captures the communication behavior of the parallel application at hand. The second component, a voltage scaling algorithm, apply critical path analysis to this graph, identifies the voltage scaling opportunities, and determines the channel frequencies and voltages. The last component modifies the input code to insert explicit voltage scaling instructions. Figure 2.12 shows the high level view of our approach. We assume that the input application code has already been parallelized (either manually or through a compiler) for message-passing communication, and inter-processor communications have been optimized using known techniques such as message vectorization and message coalescing [98]. We also that the process-to-node mapping has already been performed.

The rest of this section is structured as follows. The next section presents our graph-based representation. Section 2.5.3 illustrates our algorithm that analyzes the graph and identifies the opportunities for voltage scaling. Section 2.5.4 discusses the necessary code modifications. An experimental evaluation of our approach is presented in Section 2.5.5.

Let us assume, without loss of generality, that a communication channel can work at m different voltage levels (and corresponding frequencies), namely v_1, v_2, ..., v_m, such that $v_1 > v_2 > ... > v_m$. We further assume that it provides data rate λ when working at the highest voltage v_1 and the maximum frequency. The maximum data rate that can be provided at voltage v_i is assumed to be $k_i\lambda$, where $0 < k_i \leq 1$. We refer to k_i as the *scaling factor* for voltage v_i. Thus, $k_1 = 1$. When a communication channel works at higher voltage, it provides higher data rate, and consumes more per-bit energy and dissipates more heat. The upper bound of the performance of a given application can be achieved by operating all the communication channels

used by this application at the highest voltage level available. However, this can be an overkill for many applications. For these applications, some communication channels can be operated at lower voltages to reduce energy consumption while not affecting the overall performance significantly. Our goal is to find the the lowest voltage level for each communication channel used by a given application such that the overall performance degradation incurred is within a given bound, as compared to the performance achieved by operating all the communication channels in NoC at the highest available voltage level. To achieve this goal, we need to answer two important questions: (1) which communication channels are non-critical, and (2) to which extent their speeds (and thus voltages) can be reduced without exceeding a preset performance penalty bound.

2.5.2 Inter-Process Communication Graph

In this section, we introduce the *Inter-Process Communication Graph (IPCG)*, an abstract representation that captures the communication behavior of a message-passing based parallel program in a concise manner. In order to simplify our analysis, we make the following assumptions about the application code being analyzed:

- An application process uses "send(p, m)" to send message m to process p, and uses instruction "receive(p, m)" to receive message m from process p. For each send or receive instruction, the value p and the size of message, m, can be determined at compilation time.[3]

[3]We can handle the cases where the process id (p) in a send/receive instruction is expressed as a function of the id of the process which executes that send/receive instruction.

- The message send/receive relationships can be statically captured at compilation time. That is, for each message send instruction, the compiler can determine which receive instruction receives the messages sent by that send instruction; and for each receive instruction, the compiler can statically determine the send instruction that sends the corresponding messages.

- A send instruction is blocked if the previous message sent by the same process has not been delivered to the buffer of the target process.[4] A receive instruction is blocked if its message is not ready in the buffer of the receiver node.

An IPCG is a weighted directed graph, and can be defined for a given message-passing parallel program $\mathcal{P}$ as:

$$G(\mathcal{P}) = (V(\mathcal{P}), E(\mathcal{P}), \alpha, \beta),$$

where $V(\mathcal{P})$ is the set of vertices, $E(\mathcal{P}) \subseteq V(\mathcal{P}) \times V(\mathcal{P})$ is the set of edges, and α and β are the weight functions for the edges. Vertex set $V(\mathcal{P})$ can be expanded as:

$$V(\mathcal{P}) = X(\mathcal{P}) \cup B(\mathcal{P}) \cup S(\mathcal{P}) \cup D(\mathcal{P}) \cup R(\mathcal{P}),$$

where $X(\mathcal{P})$, $B(\mathcal{P})$, $S(\mathcal{P})$, $D(\mathcal{P})$, and $R(\mathcal{P})$ are defined as follows:

- $X(\mathcal{P})$: A vertex $x \in X(\mathcal{P})$ corresponds to a loop in $\mathcal{P}$; and x represents the entry point of this loop.

[4] To be accurate, a send instruction is blocked when the previous message sent by the same process has not left the buffer of the sender router. However, in an NoC that employs wormhole routing [68], the incurred propagation delay is typically negligible. Therefore, the difference between the time points when the last bit of a message leaves the buffer of the source node and when it is delivered to the buffer of its destination node is normally very small.

- $B(\mathcal{P})$: A vertex $b \in B(\mathcal{P})$ corresponds to a loop in $\mathcal{P}$; and b represents the back-jump point of this loop.

- $S(\mathcal{P})$: A vertex $s \in S(\mathcal{P})$ corresponds to a send instruction in $\mathcal{P}$; and s represents the point at which the message is sent.

- $D(\mathcal{P})$: A vertex $d \in D(\mathcal{P})$ corresponds to a send instruction in $\mathcal{P}$; and d represents the point at which the message is delivered to its destination.

- $R(\mathcal{P})$: A vertex $r \in R(\mathcal{P})$ corresponds to a receive instruction in $\mathcal{P}$; and r represents the point at which the message is used by the application.

Note that an inter-process communication in our NoC involves three stages. At the first stage, the sender invokes the send instruction, which copies the message into the buffer of the sender node. The sender process is blocked if this buffer is occupied. Our IPCG captures this using the vertices in $S(\mathcal{P})$. At the second stage, the NoC transfers the message to the receiver (the destination node). This stage completes when all the bits of of this message have been delivered to the receiver and stored in a temporary buffer. At the third stage, the receiver invokes a receive instruction to read the contents of the message in the buffer. Note that there may be a gap between the point when a message is delivered to the receiver node and the point when this message is actually accessed by the receiver process. In order to capture this, our IPCG denotes these two points using different vertices. Specifically, the vertices in $D(\mathcal{P})$ represent the point when a message is delivered to the receiver node, and the vertices in $R(\mathcal{P})$ represent the point when a message is accessed by the receiver process.

In this work, we refer to both message send and message receive instructions as *communication instructions*; both communication instructions and back-jumps as *instructions*, and

both instructions and loops as *execution units*. We use the term "execution unit v", where $v \in V(\mathcal{P})$, to refer to the execution unit in program $\mathcal{P}$ that corresponds to vertex v. Further, we use $b(x) \in B(\mathcal{P})$ to denote the back-jump instruction of loop x; $d(s) \in D(\mathcal{P})$, where $s \in S(\mathcal{P})$, to denote the delivery point of sending instruction s; $\psi(v)$, where $v \in V(\mathcal{P})$, to denote the id of the process to which execution unit v belongs.

We write "$x \models v$" (where $x \in X(\mathcal{P})$ and $v \in V(\mathcal{P})$) if and only if loop x directly encloses execution unit v. More specifically, if $v \in B \cup S \cup R$, instruction v is in the body of loop x and it is not enclosed by any other loop nested within loop x. If $v \in X$, loop v is nested within loop x and there is no other nested loop between loop x and v. In particular, if vertex v is not enclosed by any loop, we write "$\phi \models v$".

$E(\mathcal{P})$ is the edge set of IPCG $G(\mathcal{P})$. The edges of an IPCG can be classified into seven categories; therefore, we have:

$$E(\mathcal{P}) = E_1(\mathcal{P}) \cup E_2(\mathcal{P}) \cup E_3(\mathcal{P}) \cup E_4(\mathcal{P}) \cup E_5(\mathcal{P}) \cup E_6(\mathcal{P}) \cup E_7(\mathcal{P}),$$

where $E_1(\mathcal{P})$ through $E_7(\mathcal{P})$ are defined as follows:

- $E_1(\mathcal{P}) = \{(s, d(s)) | s \in S(\mathcal{P})\}$. An edge $(s, d(s)) \in E_1(\mathcal{P})$ is referred to as a *communication edge*.

- $E_2(\mathcal{P}) \subseteq X(\mathcal{P}) \times V(\mathcal{P})$. An edge (x, v) is in $E_2(\mathcal{P})$ if $x \models v$ and v is the first execution unit in the body of loop x.

- $E_3(\mathcal{P}) \subseteq V(\mathcal{P}) \times V(\mathcal{P})$. An edge (u, v) is in $E_3(\mathcal{P})$ if both u and v are directly enclosed by the same loop x (i.e., $\exists x \in X(\mathcal{P}) : x \models u, v$) and v is executed immediately after u at each iteration of loop x.

- $E_4(\mathcal{P}) = \{(b(x), x) | x \in X(\mathcal{P})\}$. An edge $(b(x), x) \in E_4(\mathcal{P})$ is referred to as the *back-jump edge*.

- $E_5(\mathcal{P}) \subseteq D(\mathcal{P}) \times S(\mathcal{P})$. An edge $(d(s), s')$ is in $E_5(\mathcal{P})$ if s and s' are directly enclosed by the same loop, and s' is the first send instruction that is executed after s at each loop iteration. Edge $(d(s), s')$ indicates that send instruction s' is blocked by send instruction s which is executed at same loop iteration. Such an edge is referred to as an *intra-iteration blocking edge*.

- $E_6(\mathcal{P}) \subseteq D(\mathcal{P}) \times S(\mathcal{P})$. An edge $(d(s), s')$ is in $E_6(\mathcal{P})$ if s and s' are directly enclosed by the same loop, and s and s' are the last and first send instructions in each loop iteration, respectively. Note that, if this loop contains only one send instruction, we have $s = s'$. Edge $(d(s), s')$ indicates that send instruction s' is blocked by send instruction s which is executed by a previous loop iteration. Such an edge is referred to as an *inter-iteration blocking edge*.

- $E_7(\mathcal{P}) \subseteq D(\mathcal{P}) \times R(\mathcal{P})$. An edge $(d(s), r)$ is in $E_7(\mathcal{P})$ if the messages sent by send instruction s is received by receive instruction r.

Each edge in $E_1(\mathcal{P})$ represents a *communication task*, while each edge in $E_2(\mathcal{P}) \cup E_3(\mathcal{P})$ represents a *computation task*. Therefore, the edges in $E_1(\mathcal{P}) \cup E_2(\mathcal{P}) \cup E_3(\mathcal{P})$ are referred to as the *task edges*. For a task edge (u, v), $\alpha(u, v)$ and $\beta(u, v)$ represent, respectively, the compiler-estimated lower and upper bounds of the length of task (u, v), i.e., the time it takes to complete

this task. The length of a computation task (u, v) is determined by the sum of the latencies of the instructions between the start points of execution units u and v. The values of $\alpha(u, v)$ and $\beta(u, v)$ can be obtained by either profiling or through a static analysis based approach such as the one proposed in [102]. In comparison, the length of a communication task $(u, d(u))$ is determined as follows:

$$\alpha(u, d(u)) = \frac{l_{min}}{\lambda} \text{ and } \beta(u, d(u)) = \frac{l_{max}}{\lambda},$$

where l_{min} and l_{max} are the minimum and maximum sizes of the messages sent by instruction u, respectively; and λ is the maximum available data rate of a communication channel in NoC. On the other hand, the edges in $E_4(\mathcal{P}) \cup E_5(\mathcal{P}) \cup E_6(\mathcal{P}) \cup E_7(\mathcal{P})$ do not represent any real task. They are simply introduced to enforce the timing constraints among instructions, and hence they are referred to as the *control edges*. Consequently, for a control edge (u, v), we have $\alpha(u, v) = \beta(u, v) = 0$.

An edge (u, v) indicates that the execution unit v is executed after execution unit u. Further, an edge $(u, v) \in E_4(\mathcal{P}) \cup E_6(\mathcal{P})$ indicates that u and v are executed in different loop iterations. So, we refer to the edges in set $E_4(\mathcal{P}) \cup E_6(\mathcal{P})$ as the *inter-iteration edges*. On the other hand, an edge $(u, v) \notin E_4(\mathcal{P}) \cup E_6(\mathcal{P})$ specifies that u and v are executed within the same loop iteration. Therefore, we call the edges not in set $E_4(\mathcal{P}) \cup E_6(\mathcal{P})$ the *intra-iteration edges*. Note that the graph obtained by eliminating the inter-iteration edges from $G(\mathcal{P})$ is acyclic. Therefore, the intra-iteration edges in $G(\mathcal{P})$ determine a partial order among the vertices.

When drawing an IPCG, we use different symbols to represent the different types of vertices, as shown in Figure 2.13. Also, we use dashed arrows to represent inter-iteration edges,

and solid arrows to represent intra-iteration edges. A task edge (captured by a thick solid arrow) from u to v is labeled with "$\alpha(u, v)/\beta(u, v)$". We omit the labels for the control edges.

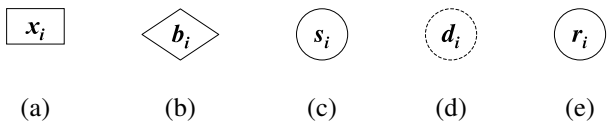


Fig. 2.13. Symbols used for different IPCG vertices. (a) Start point of loop x_i. (b) Back-jump instruction b_i. (c) Send instruction s_i. (d) Delivery point d_i. (e) Receive instruction r_i.

As an example, Figure 2.14(b) shows the IPCG for the message-passing parallel code given in Figure 2.14(a). One can observe from this IPCG that the parallel program consists of four loops: x_1, x_2, x_3, and x_4. Loop x_2 is nested within loop x_1. The execution of instruction r_2 follows that of loop x_2. We assume that loop x_2 contains at least one iteration; therefore, we have $\alpha(x_2, r_2) = 120$, where 120 is conservative estimation of the per-iteration execution time of loop x_2, which can be computed as the length of the longest path from x_2 to b_2, without using any inter-iteration edge. Since the number of iterations of loop x_2 is assumed to be unknown at compilation time, we have $\beta(x_2, r_2) = \infty$.

We say loops x_i and x_j ($\psi(x_i) \neq \psi(x_j)$) communicate with each other (denoted as $x_i \leftrightarrow x_j$) if an instruction directly enclosed by one of them sends message to an instruction directly enclosed by the other. Further, we write $x_i \overset{*}{\leftrightarrow} x_j$ if $x_i \leftrightarrow x_j$ or there exists an x_k such that $x_i \overset{*}{\leftrightarrow} x_k$ and $x_j \overset{*}{\leftrightarrow} x_k$. A vertex set H is a *parallel group* if $H \subseteq X(\mathcal{P})$ such that we have $x_i \leftrightarrow x_j$ for any pair of x_i and x_j in H, and, for any $H' \supset H$, there exits at least an

$x_i \in H'$ and an $x_j \in H'$ such that $x_i \leftrightarrow x_j$ does not hold. Note that a parallel group represents a set of loops that are executed in parallel and communicate with each other. Our voltage scaling optimization is carried out at the parallel group granularity; i.e., we process the parallel groups one by one. We define $L(H)$, the *loop communication graph (LCG)* for parallel group H, as the subgraph of IPCG $G(\mathcal{P})$ induced by H^*, where

$$H^* = \{v | \exists x \in H : x \models v\} \cup \{d(v) | \exists x \in H, v \in S(\mathcal{P}) : x \models v\}.$$

Similar to IPCG $G(\mathcal{P})$ which captures the communication behavior of parallel program $\mathcal{P}$, LCG $L(H)$ captures the communication behavior of the loops in H. For example, the IPCG shown in Figure 2.14(b) includes two parallel groups: $H_1 = \{x_1, x_3\}$ and $H_2 = \{x_2, x_4\}$, whose corresponding LCGs are shown in Figures 2.14(c) and (d), respectively.

2.5.3 Critical Path Analysis

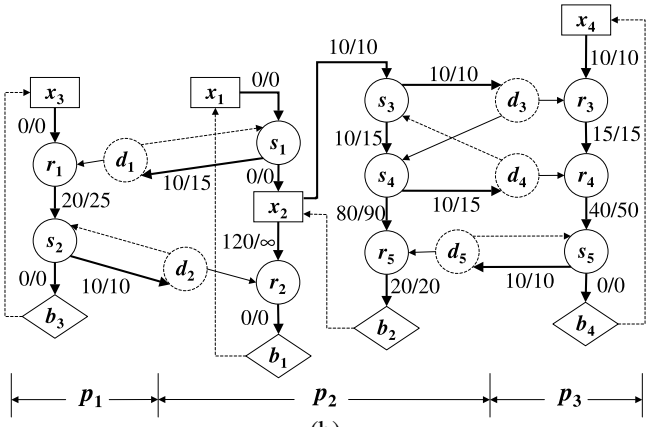
Recall that our approach assumes that the input code has already been parallelized and process-to-node mapping has been performed. Therefore, for a given process pair, we know the set of communication channels that will be used for transferring messages between them. Our goal now is to analyze the LCGs determined in the previous step one by one and identify the communication channels that can be voltage/frequency-scaled and the amount of scaling to be applied. The proposed approach reduces the NoC energy consumption by changing the frequencies/voltage levels of some communication channels and does *not* change the behavior of the application itself. Therefore, while inaccuracy in our analysis (e.g., in assigning $\alpha(u, v)$

```
//Process 1
x3: for(...) {
  r1: receive(2, ...)
  computing: 20∼25 cycles;
  s2: send(2, ...)
}
```

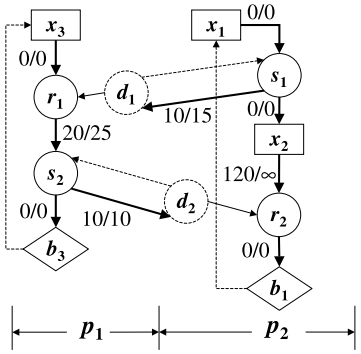
```
//Process 2
x1: for(...) {
  s1: send(1, ...);
  x2: for(...) {
    computing: 10 cycles;
    s3: send(3, ...);
    computing: 10∼15 cycles;
    s4: send(3, ...);
    computing: 80∼90 cycles;
    r5: receive(3, ...);
    computing: 20 cycles;
  }
  r2: receive(1, ...)
}
```

```
//Process 3
x4: for(...) {
  computing: 10 cycles;
  r3: receive(2, ...)
  computing: 15 cycles;
  r4: receive(2, ...)
  computing: 40∼50 cycles;
  s5: send(2, ...)
}
```

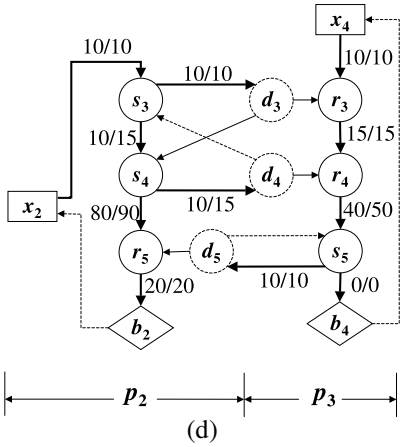
(a)



(b)



(c)



(d)

Fig. 2.14. (a) Code for a message-passing parallel program; "$x \sim y$" indicates that a computation task can take minimum x and maximum y cycles to complete. (b) The IPCG for the code shown in (a). (c) The LCG for parallel group $\{x_1, x_3\}$. (d) The LCG for parallel group $\{x_2, x_4\}$.

and $\beta(u, v)$ values) may affect the performance and the energy consumption of the application, it does *not* create any correctness issue.

For a given parallel group $H = \{x_1, x_2, ..., x_n\}$, let us use $t_{i,j}$ to represent the earliest time that the j^{th} iteration of loop x_i can start, and we have $t_{i,0} = 0$ ($i = 0, 1, ..., n$). q is the minimum number such that there exists a constant $R \geq 1$ such that:

$$t_{1,q+R} - t_{1,q} = t_{2,q+R} - t_{2,q} = ... = t_{n,q+R} - t_{n,q} = T. \tag{2.1}$$

The start time of the $(q + mR + k)^{\text{th}}$ iteration of loop x_i, where $0 \leq k < R$ and $m \geq 0$, is $t_{i,q+k} + mT$. Therefore, R can be thought as the re-occurring period of the timing behavior of parallel group H. The timing behavior of parallel group H during its entire execution time can be represented by the behavior during the period from the q^{th} iteration through the $(q + R - 1)^{\text{th}}$ iteration. Therefore, we refer to these iterations as *representative iterations*. Representative iterations are important because our approach uses them for determining scaling factors.

As an example, Figure 2.15 shows timing for four loops executed in parallel on four NoC nodes. The first iteration ($i = 0$) of each loop starts at the same time (i.e., $t_{1,0} = t_{2,0} = t_{3,0} = t_{4,0} = 0$). However, due to load imbalance and timing constraints exhibited by inter-process communications, the first iterations of these loops are not necessarily completed at the same time; consequently, the start times of the second iterations ($t_{1,1}$, $t_{2,1}$, $t_{3,1}$ and $t_{3,1}$) of these loops may differ from each other. In this example, constants q is 1 and R is 3 such that $t_{i,j+R} = t_{i,j} + T$ for all $j \geq q$, where T is a constant.

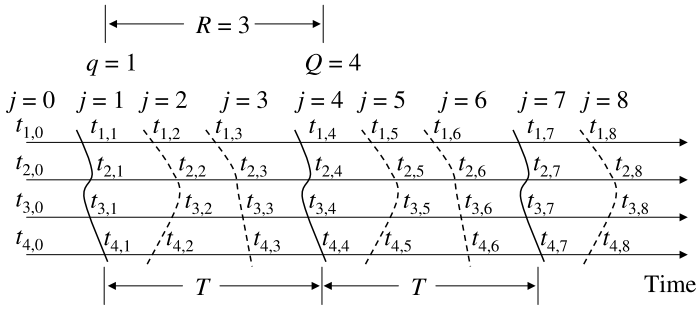


Fig. 2.15. Timing for an example parallel execution. In this particular example, there exist constants $q = 1$ and $R = 3$ such that $t_{i,j} = t_{i,j+R} + T$ for all $j \geq q$, where T is a constant.

Figure 2.16, Figure 2.17, and Figure 2.18 together present our algorithm for determining the scaling factors for connections used by a given parallel group H. For an application containing multiple parallel groups, we analyze these parallel groups individually. Our algorithm takes $L(H)$, the LCG for the given parallel group H, as input. It computes $k[i, j]$, the scaling factor for the communication channels in the connection from process i to process j, such that the overall percentage performance degradation due to voltage/frequency scaling does not exceed δ, where δ is a user-specified constant, as compared to the performance achieved by operating all the communication channels in NoC at the highest available voltage/frequency level.

The algorithm works in three phases. In the first phase, it computes $t_\alpha[i, j]$, the earliest time v_i at the j^{th} iteration can be reached, assuming all the tasks are finished in their shortest times. The values of $t_\alpha[i, j]$s are the minimum values that satisfy the following expressions:

$$\forall i : t_\alpha[i, 0] = 0, \tag{2.2}$$

$$\forall (k, i) \in E^+ : t_\alpha[i, j] \geq t_\alpha[k, j] + \alpha(k, i), \tag{2.3}$$

$$\forall (k, i) \in E^- : t_\alpha[i, j] \geq t_\alpha[k, j - 1], \tag{2.4}$$

where E^+ is the set of intra-iteration edges, and E^- is the set of inter-iteration edges. One can observe from Figure 2.16 that the first phase contains a repeat-until loop. At the Q^{th} iteration of this loop, we use Expression (2.3) to compute $t_\alpha[i, Q]$ for each vertex v_i in the LCG. After that, we check if there exist a constant q ($0 \leq q < Q$) and a constant T such that, for all $v_i \in H$, we have $t_\alpha[i, Q] - t_\alpha[i, q] = T$. Note that the q^{th} through $(Q-1)^{\text{th}}$ iterations are the representative iterations. If we cannot find such a q and T, we use Expression (2.4) to compute the initial values of $t_\alpha[i, Q+1]$ for the next iteration. We repeat this procedure until we find a suitable q and T. In order to limit compilation time, the algorithm terminates if we cannot not find suitable q and Q values within the first Q^* iterations. The computational complexity for each iteration of this phase is $O(m+n)$, where m and n are the number of edges and vertices in the LCG under consideration, respectively. Therefore, the computational complexity of this phase is $O(Q^*(m+n))$.

In the second phase, we compute the worst case execution time for representative iterations, assuming that each task takes the longest time to complete and all the communication channels work at the highest data rate. The worst case start time for each vertex in V is the minimum value of $t_\beta[i, j]$ that satisfies the following expressions:

$$\forall i : t_\beta[i, q] = t_\alpha[i, q], \tag{2.5}$$

$$\forall (k, i) \in E^+ : t_\beta[i, j] \geq t_\beta[k, j] + \beta(k, i), \tag{2.6}$$

$$\forall (k, i) \in E^- : t_\beta[i, j] \geq t_\beta[k, j-1], \tag{2.7}$$

The worst execution time for representative iterations of loop $v_i \in H$ can be computed as $t_\beta[i, Q] - t_\beta[i, q]$. The computational complexity for this phase is $O((m+n)(Q-q))$, where m

and n are the number of the edges and the vertices in the LCG under consideration, respectively, and the values of q and Q are as determined in the previous phase.

In the third phase, we try to maximize the scaling factor $k[i,j]$ $(0 < k[i,j] \leq 1)$ for each communication channel exercised by connection ⟦i,j⟧ under the following constraints:

$$\forall i : t[i,q] = t_{\alpha}[i,q], \tag{2.8}$$

$$\forall (k,i) \in E^{+} : t[i,j] \geq t[k,j] + \beta(k,i)/k[\psi(v_k), \psi(v_i)], \tag{2.9}$$

$$\forall (k,i) \in E^{-} : t[i,j] \geq t[k,j-1], \tag{2.10}$$

$$\forall v_i \in H : t[i,Q] \leq t[i,q] + \max\{(1+\delta)T, t_{\beta}[i,Q] - t_{\beta}[i,q]\} \tag{2.11}$$

In this phase, we assume all the tasks are finished in their longest time. Expression (2.11) means that, for a loop whose worst case overall execution time for the representative iterations does not exceed $(1+\delta)T$, we can tolerate a percentage performance degradation up to δ. On the other hand, for a loop whose worst case overall execution time for the representative iterations is already longer than $(1+\delta)T$, we do not allow any further performance degradation. As a result, the overall performance degradation due to scaling the voltage/frequency of communication channels is within δ.

Phase 3 shown in Figure 2.18 contains a "repeat-until" loop. At each iteration of this loop, we select a connection and reduce the data rate of the communication channels in this connection by one level. After that, we estimate the execution time for each loop. If the estimated performance degradation exceeds the limit set by Expression (2.11), the data rate of the selected connection cannot be reduced. We repeat this procedure until there is no connection whose data rate can be further reduced. Note that, at each step, instead of scaling down the voltage/frequency of the selected connection aggressively, we reduce its voltage/frequency by

only one level. This choice allows us to scale down the speeds of more connections. As will be discussed later, a group of connections working at similar speeds is more desirable than a group of connections whose speeds differ from each other significantly if these connections share some communication channels. The computational complexity of this phase is $O(v^c(m+n)(Q-q))$, where c is the number of connections used, v is the number of available frequency/voltage levels for a communication channel, m is the number of edges, n is the number of the vertices in the LCG under consideration, and the values of q and Q are as determined in phase 1. Therefore, the overall complexity of our algorithm is $O((m+n)(v^c(Q-q)+Q^*))$.

We now explain how our algorithm operates using an example. Figure 2.19(a) shows a parallel program with three processes and Figure 2.19(b) shows the corresponding IPCG. This IPCG contains only one LCG. Figure 2.20 shows how our algorithm computes t_α and t_β for each vertex in this LCG. At the beginning of phase 1, the values of all $t_\alpha[i,0]$s are initialized to 0, as shown in Figure 2.20(a). At the first iteration of phase 1, we compute the value of $t_\alpha[i,0]$ for each vertex, and then initialize the values of $t_\alpha[x_1,1]$, $t_\alpha[x_2,1]$, $t_\alpha[x_3,1]$, $t_\alpha[s_1,1]$, $t_\alpha[s_2,1]$, and $t_\alpha[s_3,1]$ based on the values of $t_\alpha[b_1,0]$, $t_\alpha[b_2,0]$, $t_\alpha[b_3,0]$, $t_\alpha[d_1,0]$, $t_\alpha[d_2,0]$, and $t_\alpha[d_3,0]$, respectively, as specified by Expression (2.4). After that, we start the second iteration. Figure 2.20(b) shows the values of t_αs at the beginning of this iteration. Similarly, we compute the values of $t_\alpha[i,1]$s and initialize the values of $t_\alpha[x_1,2]$, $t_\alpha[x_2,2]$, $t_\alpha[x_3,2]$, $t_\alpha[s_1,2]$, $t_\alpha[s_2,2]$, and $t_\alpha[s_3,2]$ for the third iteration. Figure 2.20(c) illustrates the state at the beginning of the third iteration. We repeat this procedure until we reach the state shown in Figure 2.20(d), where we have $T = t_\alpha[x_i,4] - t_\alpha[x_i,3] = 50$ for $i = 1,2,3$. At this point, we can also determine that $q = 3$, $Q = 4$, and $R = 1$.

Global Variables:
$L(H)$ — the LCG of parallel group H;
V — the set of vertices in LCG $L(H)$. The vertices in V has been sorted in the partial order determined by the intra-iteration edges, i.e., for all $v_i, v_j \in V$, we have $i < j$ if (v_i, v_j) is an intra-iteration edge.
$t_\alpha[i,j], t_\beta[i,j]$ — the best and the worst start times of vertex v_i in the j^{th} iteration.
$k[i,j]$ — the scaling factor for connection $[\![i,j]\!]$;
$0 < k[i,j] \leq 1$; particularly; $k[i,j] = 1$ if $i = j$.
q, Q, T — $\forall i \in H : T = t[i,Q] - t[i,q]$

// Initialization
set all $t_\alpha[i,j]$, $t_\beta[i,j]$, and $t[i,j]$ to 0; set all $k[i,j]$ to 1; $Q = 0$; $T = -1$;

// Phase 1: Computing $t_\alpha[i,j]$, q, Q, and T
repeat{
for $i = 1$ to $|V|$
for each intra-iteration edge (v_i, v_j) where $t_\alpha[j,Q] < t_\alpha[i,Q] + \alpha(v_i, v_j)$
$t_\alpha[j,Q] = t_\alpha[i,Q] + \alpha(v_i, v_j)$;
$Q = Q + 1$;
for each inter-iteration control edge (v_i, v_j) where $t_\alpha[j,Q] < t_\alpha[i,Q-1]$
$t_\alpha[j,Q] = t_\alpha[i,Q-1]$;
for $q = Q - 1$ to 0
if($\forall v_i, v_j \in H : t_\alpha[i,Q] - t_\alpha[i,q] \neq t_\alpha[j,Q] - t_\alpha[j,q]$) {
$T = t_\alpha[i,Q] - t_\alpha[i,q]$ where $v_i \in H$;
break; // the values of T, Q, and q have been determined.
}
} until ($Q \geq Q^*$ or $T > 0$);
if($T = -1$) {terminate with failure;}

Fig. 2.16. Algorithm for critical path analysis: Phase 1.

```
// Phase 2: Computing t_β[i, j]
for r = q to Q { for each v_i ∈ V { t_β[i, r] = t_α[i, r]; } }
for r = q to Q {
  for i = 1 to |V|
    for each intra-iteration edge (v_i, v_j) where t_β[j, r] < t_β[i, r] + β(v_i, v_j)
      t_β[j, r] = t_β[i, r] + β(v_i, v_j);
  for each inter-iteration control edge (v_i, v_j) where t_β[j, r + 1] < t_β[i, r]
    t_β[j, r + 1] = t_β[i, r];
}
```

Fig. 2.17. Algorithm for critical path analysis: Phase 2.

```
// Phase 3: Computing t[i, j] and Determining k[i, j]
for all connection ⟦i, j⟧ { g[i, j] = 0; }
repeat {
  for each connection ⟦i, j⟧ where g[i, j] = 0 {
    k' = k[i, j]; // back up k[i, j]
    decrease k[i, j] to the next scaling level;
    for r = q to Q { for each v_i ∈ V { t[i, r] = t_α[i, r]; } }
    for r = q to Q {
      for i = 1 to |V|
        for each intra-iteration edge (v_i, v_j)
          where t[j, r] < t[i, r] + α(v_i, v_j)/k[ψ(v_i), ψ(v_j)]
            t[j, r] = t[i, r] + α(v_i, v_j)/k[ψ(v_i), ψ(v_j)];
      for each inter-iteration control edge (v_i, v_j) where t[j, r + 1] < t[i, r]
        t[j, r + 1] = t[i, r];
    }
    if(∃v_i ∈ H : t[i, Q] − t[i, q] > max{(1 + δ)T, t_β[Q, i] − t_β[q, i]}) {
      k[i, j] = k'; // restore k[i, j] to its previous value.
      g[i, j] = 1; // this connection cannot be scaled any further.
    }
  }
} until g[i, j] = 1 for all connection ⟦i, j⟧;
```

Fig. 2.18. Algorithm for critical path analysis: Phase 3.

In phase 2, we compute the value of t_β for each vertex based on Expressions (2.5), (2.6), and (2.7). Figure 2.20(e) shows the result of this phase. Note that, since we have $q = 3$ and $Q = 4$, we only need to compute the values of $t_\beta[i, 3]$s. After that, we compute the values of $t_\alpha[x_1, 4]$, $t_\alpha[x_2, 4]$, and $t_\alpha[x_3, 4]$, based on the values of $t_\alpha[b_1, 3]$, $t_\alpha[b_2, 3]$, and $t_\alpha[b_3, 3]$, respectively, as captured by Expression (2.7).

With a performance degradation of 10%, i.e., $\delta = 10\%$, since in phase 1 we computed that $q = 3$, $Q = 4$, and $T = 50$, we have:

$$t[x_1, Q] \leq t[x_1, q] + \max\{(1+\delta)T, t_\beta[x_1, Q] - t_\beta[x_1, q]\} = 170,$$
$$t[x_2, Q] \leq t[x_2, q] + \max\{(1+\delta)T, t_\beta[x_2, Q] - t_\beta[x_2, q]\} = 210,$$
$$t[x_3, Q] \leq t[x_3, q] + \max\{(1+\delta)T, t_\beta[x_3, Q] - t_\beta[x_3, q]\} = 190.$$

That is, as specified by Expression (2.11), after scaling the voltages/frequencies of the communication channels, the start times of the Q^{th} iterations of loops x_1, x_2, and x_3 cannot be later than 170, 210, and 190, respectively.

Figure 2.21 shows how the third phase of our algorithm determines $k[1, 2]$, $k[2, 3]$, and $k[3, 1]$, the scaling factors for connections $[\![1, 2]\!]$, $[\![2, 3]\!]$, and $[\![3, 1]\!]$, respectively. At each step of this phase, we try to reduce the scaling factor of one connection. The tables in (a) through (f) show the values of k and t at each step. First, we try the combination where $k[1, 2] = 0.8$, $k[2, 3] = 1$, and $k[3, 1] = 1$. Based on these scaling factors, we compute the values of $t[i, 3]$ as shown in Figure 2.21(a). It is not difficult to verify that all the constraints specified by Expressions (2.8), (2.9), (2.10), and (2.11) are satisfied. In the next step, we try the combination of $k[1, 2] = 0.8$, $k[2, 3] = 0.8$, and $k[3, 1] = 1$. We see from the table in Figure 2.21(b) that, after scaling, we have $t[x_3, Q] = 196.25 > 190$, which violates the constraint captured by Expression

(2.11). Therefore, $k[2,3]$ is restored to 1. We next try the combination of $k[1,2] = 0.8$, $k[2,3] = 1$, and $k[3,1] = 0.8$, and find that it fails due to $t[x_1, Q] = 176.25 > 170$. Figures 2.21(d), (e), and (f) depict the next three steps. Phase 3 terminates after the step shown in (f) since we cannot find any other possible values for the scaling factors. Therefore, the scaling factors for connections $[\![1,2]\!]$, $[\![2,3]\!]$, and $[\![3,1]\!]$ are determined as $k[1,2] = 0.4$, $k[2,3] = 1$, and $k[3,1] = 1$, respectively.

2.5.4 Code Modification

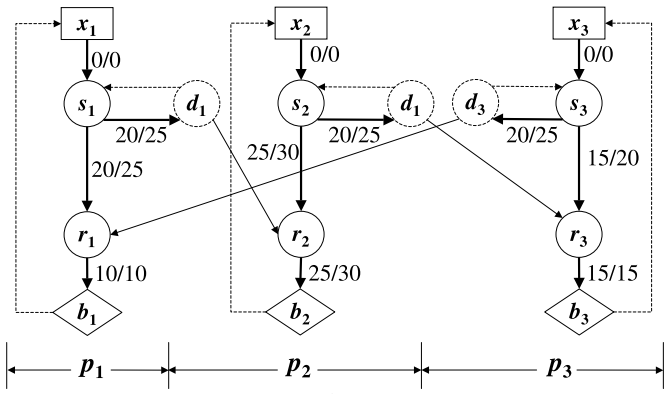
At compilation time, we assign an id to each loop such that loops executing simultaneously (not necessarily in the same parallel group) have the same id. In order to set voltage/frequency levels of communication channels used by connection $[\![i,j]\!]$, source node i sends a voltage/frequency control message to destination node j. This message contains the id of the loop and the desirable voltage/frequency level for connection $[\![i,j]\!]$, as determined by our algorithm. This message is transferred along the communication channels on the path from node i to node j. Each communication channel has a small memory space that keeps the loop id of the last voltage/frequency control message transferred. When the hardware of a communication channel detects that the message being transferred is a voltage/frequency message, it compares the loop id contained in this message against the one kept in the memory associated with this communication channel. If these two ids are different, the hardware of the communication channel sets its supply voltage to the voltage level specified in the message, and updates the id stored in the corresponding memory space. On the other hand, if these two ids are identical, the hardware of the communication channel compares its present voltage level (v) against the voltage level (v') specified in the message. If $v < v'$, the hardware adjusts its voltage level to v'; otherwise,

```
x1: for(...) {
  s1: send(2, ...);
  computing: 20∼25 cycles;
  r1: receive(3, ...);
  computing: 10 cycles;
}
```

```
x2: for(...) {
  s2: send(3, ...);
  computing: 25∼30 cycles;
  r2: receive(1, ...);
  computing: 25∼30 cycles;
}
```

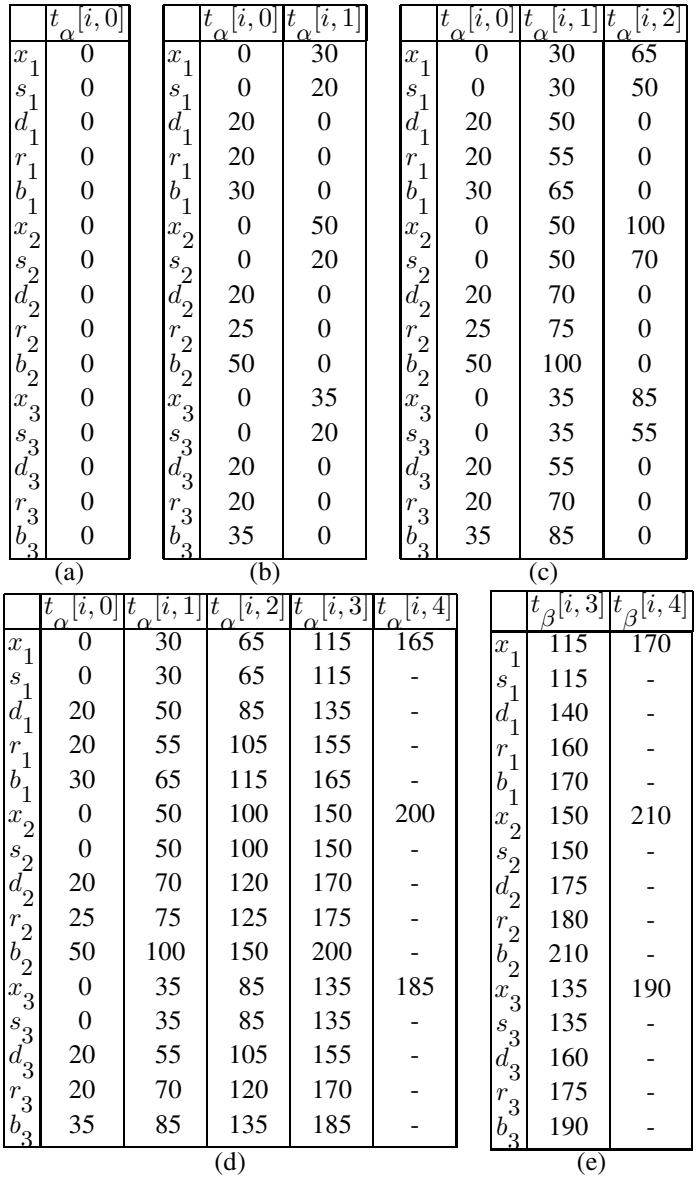
```
x3: for(...) {
  s3: send(1, ...);
  computing: 15∼20 cycles;
  r3: receive(2, ...);
  computing: 15 cycles;
}
```

(a)



(b)

Fig. 2.19. An example parallel code (a) and its IPCG (b).



	$t_\alpha[i,0]$
x_1	0
s_1	0
d_1	0
r_1	0
b_1	0
x_2	0
s_2	0
d_2	0
r_2	0
b_2	0
x_3	0
s_3	0
d_3	0
r_3	0
b_3	0

(a)

	$t_\alpha[i,0]$	$t_\alpha[i,1]$
x_1	0	30
s_1	0	20
d_1	20	0
r_1	20	0
b_1	30	0
x_2	0	50
s_2	0	20
d_2	20	0
r_2	25	0
b_2	50	0
x_3	0	35
s_3	0	20
d_3	20	0
r_3	20	0
b_3	35	0

(b)

	$t_\alpha[i,0]$	$t_\alpha[i,1]$	$t_\alpha[i,2]$
x_1	0	30	65
s_1	0	30	50
d_1	20	50	0
r_1	20	55	0
b_1	30	65	0
x_2	0	50	100
s_2	0	50	70
d_2	20	70	0
r_2	25	75	0
b_2	50	100	0
x_3	0	35	85
s_3	0	35	55
d_3	20	55	0
r_3	20	70	0
b_3	35	85	0

(c)

	$t_\alpha[i,0]$	$t_\alpha[i,1]$	$t_\alpha[i,2]$	$t_\alpha[i,3]$	$t_\alpha[i,4]$
x_1	0	30	65	115	165
s_1	0	30	65	115	-
d_1	20	50	85	135	-
r_1	20	55	105	155	-
b_1	30	65	115	165	-
x_2	0	50	100	150	200
s_2	0	50	100	150	-
d_2	20	70	120	170	-
r_2	25	75	125	175	-
b_2	50	100	150	200	-
x_3	0	35	85	135	185
s_3	0	35	85	135	-
d_3	20	55	105	155	-
r_3	20	70	120	170	-
b_3	35	85	135	185	-

(d)

	$t_\beta[i,3]$	$t_\beta[i,4]$
x_1	115	170
s_1	115	-
d_1	140	-
r_1	160	-
b_1	170	-
x_2	150	210
s_2	150	-
d_2	175	-
r_2	180	-
b_2	210	-
x_3	135	190
s_3	135	-
d_3	160	-
r_3	175	-
b_3	190	-

(e)

Fig. 2.20. Computing t_α and t_β for the IPCG shown in Figure 2.19. (a) At the beginning of phase 1, all $t_\alpha[i,0]$s are initialized to 0. (b) The situation at the beginning of the second iteration of phase 1. (c) The t_α value for each vertex the beginning of the third iteration of phase 1. (d) The t_α value at the end of phase 1. (e) The t_β value for each vertex computed in phase 2.

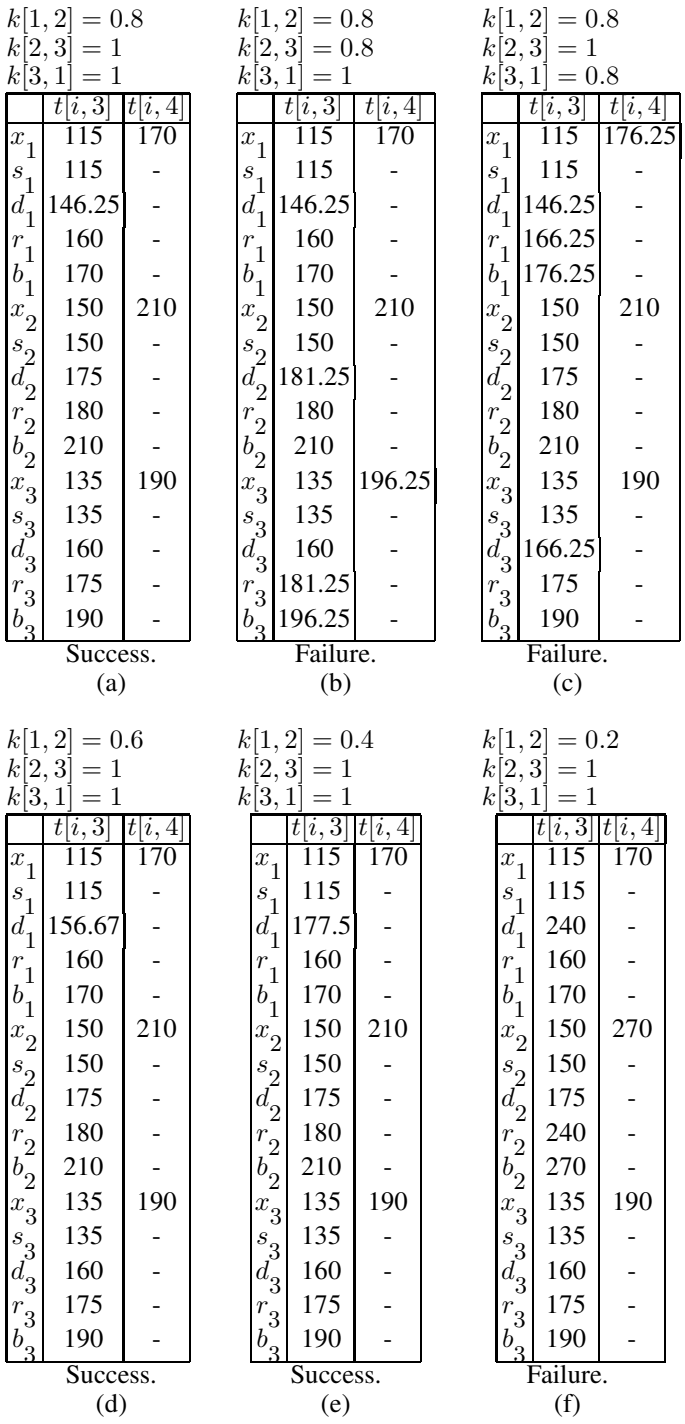


Fig. 2.21. Determining scaling factors $k[1,2]$, $k[2,3]$, and $k[3,1]$ assuming $\delta = 10\%$. At each step of Phase 3, we try to reduce the scaling factor of one connection. The tables in (a) through (f) show the values of k and t at each step.

it remains at its present voltage (v). A voltage/frequency control message is discarded upon its arrival at its destination node.

Based on the hardware support discussed above, the task of our code modification module is to insert instructions that send voltage/frequency control messages before the entry of each loop. Let us assume that loop x is in process p_0 and that it sends messages to processes p_1, p_2, ..., p_n. Our compiler inserts the following instructions before the entry of loop x:

send(p_1, {CTRL, $id(x)$, $f(k[p_0, p_1])$});
send(p_2, {CTRL, $id(x)$, $f(k[p_0, p_2])$});
... ...
send(p_n, {CTRL, $id(x)$, $f(k[p_0, p_2])$});
x: for(...) { ... }

The function $id(x)$ gives the id of loop x; the function $f(k[p_0, p_i])$ maps $k[p_0, p_i]$, the scaling factor of connection $[\![p_0, p_i]\!]$, to the corresponding frequency/voltage level; and the flag "CTRL" in the header of the message indicates that this is a voltage/frequency control message. As an example, Figure 2.22 gives the modified code for the program shown in Figure 2.19(a). This code contains only one parallel group whose id is 1.

send(2, {CTRL, 1, $f(0.4)$});	send(3, {CTRL, 1, $f(1)$});	send(1, {CTRL, 1, $f(1)$});
x_1: for(...) {	x_2: for(...) {	x_3: for(...) {
s_1: send(2, ...);	s_2: send(3, ...);	s_3: send(1, ...);
computing: 20∼25 cycles;	*computing: 25∼30 cycles*;	*computing: 15∼20 cycles*;
r_1: receive(3, ...);	r_2: receive(1, ...);	r_3: receive(2, ...);
computing: 10 cycles;	*computing: 25∼30 cycles*;	*computing: 15 cycles*;
}	}	}

Fig. 2.22. An example parallel code with voltage/frequency control instructions inserted.

2.5.5 Experiments

To test the effectiveness of our approach, we implemented it using the SUIF infrastructure [93] and performed experiments with 12 embedded benchmark codes. An important characteristic of these benchmarks is that we were able to parallelize them using our compiler and apply known optimizations such as message vectorization and message coalescing. Our parallelization module takes a sequential program and generates an MPI-based parallel code. The average increase in compilation time due to our voltage scaling approach (over the case when the application is parallelized but no power optimization is performed) was about 68% (when we set the value of Q^* to 50).

Table 2.4(a) gives the default simulation parameters used in our experiments and Table 2.4(b) lists the relevant values for the voltage/frequency levels used. Energy/performance values in Tables 2.4(a) and (b) are from [89, 49] and represent typical values for an NoC. The simulated NoC architecture is a two-dimensional mesh, though our approach can work with any other regular NoC topology.

We focus on two metrics: *energy consumption* and *execution cycles.* We obtain NoC energy consumption values using an enhanced version of LUNA [31] that supports voltage scaling. More specifically, we used the multi-processor system simulator SIMICS [1] to simulate parallel execution and enhance SIMICS with LUNA NoC power models. The energy consumption in memory components and CPUs has been estimated using Wattch based high-level power models [14]. To sum up, the processor execution is simulated using SIMICS and network traffic is simulated using LUNA. While we report only the energy consumption numbers for NoC channels and routers (with their buffers), our experimentation found that the energy consumption of these

components constitutes about 44% of the total energy consumption of an NoC node, assuming a two-issue embedded CPU with 20KB software-managed on-chip local memory per node (that stores both instructions and data).

Table 2.4. (a) Default values of our major simulation parameters. (b) Available link voltage/frequency levels.

Parameter	Value
NoC Topology	5×5 2D mesh
Idle Channel Energy Consumption	8.5 pJ/cycle
Overheads for Voltage Switch	1020pJ, 120 cycles
Processor	1 GHz, 2-issue
Local Node Memory	20KB
Packet Header Size	3 Flits
Flit Size	39 Bits

(a)

Voltage (V)	Rate (bps)	Energy (pJ/bit)
0.7	200M	4.21
0.9	660M	5.25
1.1	1.33G	6.49
1.3	1.93G	8.31
1.5	2.50G	10.21

(b)

Table 2.5 gives information on the benchmarks used in this study. The last three applications are the only codes in MediaBench and MiBench that we could parallelize automatically. The third column of Table 2.5 shows the number of LCGs for each benchmark. The fourth column gives the NoC energy consumption when no power optimization is applied. But, even in this case, if a communication channel is not used, it is kept in the off-state to save energy. Therefore, any savings our approach achieves over the default case comes from voltage/frequency scaling. The last column gives the total execution cycles (not just communication cycles) taken when no power optimizations (other than communication channel shutdown) are applied. The energy and performance results reported below are *normalized* with respect to the corresponding values in these two columns. The number of voltage/scaling instructions inserted in the benchmarks varied between 37 and 153, and their impact on execution cycles and power were negligible. In

Table 2.5. Benchmarks and their important characteristics.

Benchmark	Explanation [Source]	Number of LCGs	NoC Energy without Opt. (mJ)	Total Cycles (M)
Morph2	Morphological operations	54	231.4	446.6
Disc	Speech/music discriminator	49	194.4	380.9
Jpeg	Lossy compression	67	296.5	571.2
Viterbi	Viterbi decoder	78	308.0	761.5
Rasta	Speech recognition	34	187.8	322.1
3Step-log	Logarithmic search motion est.	28	126.6	209.3
Full-search	Full search motion est.	34	133.4	207.1
Hier	Hierarchical motion est.	28	130.7	211.7
Phods	Parallel hierarchical motion est.	34	119.0	203.4
Epic	Image data compression	44	188.7	366.5
Lame	MP3 encoder	37	173.4	352.0
FFT	Fast Fourier transform	41	192.1	372.3

any case, energy/performance results presented below include the overheads due to (1) the execution of these scaling instructions in the CPUs, (2) the latency incurred and extra power spent in the network during voltage/frequency scaling, and (3) increased code size which increases the number of accesses to the local memories.

The first two bars for each benchmark in Figure 2.23 give the normalized energy consumption results with a hardware-based voltage scaling scheme and our compiler-directed scheme. Our approach is run under $\delta = 0\%$, i.e., no performance penalty is allowed (however, we still incur some negligible performance penalty due to inaccuracies in determining voltage/frequency levels). The hardware scheme used is from [82], in which a hardware circuit is employed to constantly monitor the traffic in the NoC and dynamically adapt the voltage/frequency of the communication channels based on the collected traffic statistics. The results show that the compiler based approach saves on average 17.21% energy over the hardware based scheme. The main

reason for this is that the compiler accurately predicts allowable extra latencies for communication channels and selects appropriate voltage levels accordingly. In comparison, the hardware based approach relies on the history information which cannot capture the changes in usage frequency of different channels. Our experiments also showed that the hardware based approach incurs around 7.2% performance degradation on average when considering all 12 benchmarks. In comparison, the performance penalty incurred by our approach was less than 1% for all the benchmarks. Based on these results, we see that the compiler-based approach is preferable over the hardware-based scheme from both performance and power angles. The stacked bar chart in Figure 2.24 shows the percentage of energy spent at different voltage levels as well as the overhead energy incurred by doing voltage transitions. Our approach exercises all voltage levels available during execution.

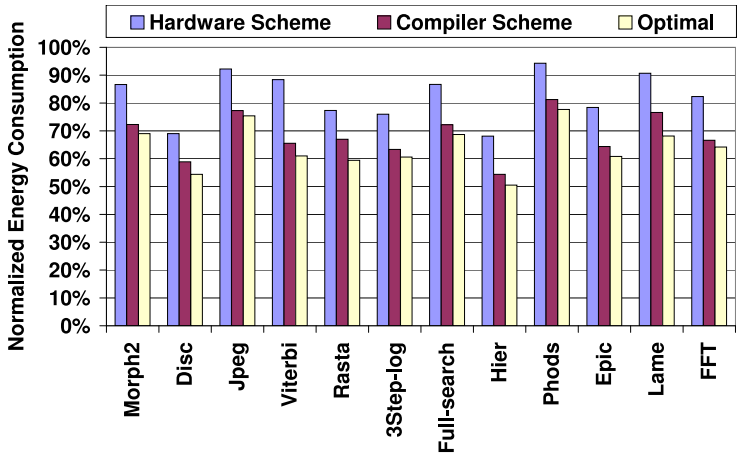


Fig. 2.23. Normalized NoC energy consumption.

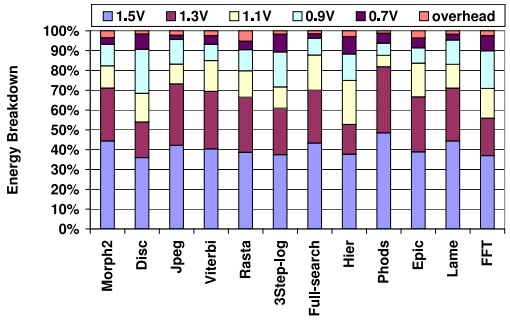


Fig. 2.24. NoC energy consumption breakdown.

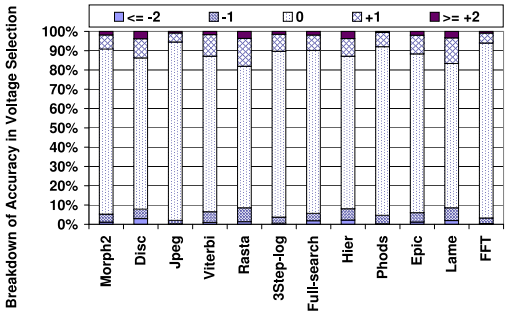


Fig. 2.25. Accuracy of voltage selection.

While the results discussed so far show the success of the compiler-based scheme, it is also important to measure how close our energy savings come to optimal savings. In order to obtain the values of the maximum savings, we analyzed the execution trace of each benchmark, and manually tuned the voltage level for each communication channel used by each parallel group in each benchmark. The normalized energy savings with this optimal scheme are given as the third bar for each benchmark in Figure 2.23 under $\delta = 0\%$. The difference between our compiler scheme and the optimal scheme is only 6.14% on average. To explain why our approach generates results that are very close to the optimal savings, we study in Figure 2.25 the accuracy of our approach in identifying the correct voltage levels for communication channels with respect to the voltage levels used by the optimal scheme. For ease of discussion, we refer to an instance where a communication channel transfers a data packet as a *communication activity*. The segment of a bar marked with 0 in Figure 2.25 captures the percentage of communication activities where the voltage level selected by the compiler is the same as the optimal level, whereas $\mp x$ corresponds to the case where the compiler-predicted voltage is x levels lower (higher) than the optimal level. These results clearly show that our approach is very accurate in

determining the optimal voltage levels to use most of the time, which explains why it gets so close to the optimal energy savings. Specifically, on average, 82.94% of the time the compiler-based approach selects the optimal voltage level. And, about 13.69% of the time, the voltage selected by the compiler is either one level below or above the optimal voltage level.

2.6 Profile-driven Message Rerouting

2.6.1 Motivation

Using communication link shutdown or voltage/frequency scaling can significantly reduce NoC power consumption as demonstrated by previous research and our two proposals presented in Section 2.4.4 and Section 2.5.5. Such techniques, while very effective in reducing power consumption in certain cases, work best when communication links have long idle periods, which allow compensation for performance/power overheads due to switching between voltage levels and between link shut-down/turn-on states. Specifically, long idle periods are preferable from the viewpoint of maximizing power savings through link shutdown.

This proposal focuses a *profile-driven compiler optimization* for increasing the length of idle periods of communication links for a two-dimensional, on-chip, mesh network. The proposed compiler-directed approach achieves its goal by maximizing communication link reuse. That is, this approach clusters the required data communications into a small set of links at any given time, increasing the idle periods for the remaining communication links in the network. Clearly, this scheme needs to occur in a performance-sensitive manner. Therefore the goal is to reduce network energy consumption as much as possible without causing extra link contention and significantly degrading network performance.

The targeted application domain is array/loop-intensive embedded programs, and the targeted NoC is a two-dimensional mesh used by a single application at a time. This paper proposes a profile-driven static message routing scheme that maximizes link reuse between different execution states of a given application. We introduce a novel data structure called "communication graph" to capture different network states during application execution and a new abstraction, the "link signature", to capture link utilization in a given network state.

The remainder of this section is organized as follows. Section 2.6.2 introduces the hardware support for compiler-directed message routing. Section 2.6.3 explains how link signatures and the communication graph derive from an automated compiler analysis and Section 2.6.4 describes the link reuse optimization algorithm. Section 2.6.6 presents the experimental results.

2.6.2 Hardware Support for Compiler-Directed Message Routing

We focus on a power-aware NoC that has a hardware-controlled link shutdown scheme. As discussed in 2.4.1, each communication link in the network, as well as its corresponding buffers, can be turned off when they remain idle for a certain period of time. The powered-off components re-activate on demand, i.e., they turn on only when needed.

Our goal is to determine the most appropriate routing for each message at compilation time thereby allowing maximized link reuse across different messages. Thus, the compiler must have a way of providing routing information, which may be different from the default X-Y routing, to each message. We propose to let the compiler attach routing information to each message-send operation in the code, requiring the packets of all the messages issued by a message-send operation to follow the same route (the message-send operations considered here are source

level communication commands such as MPI_Send in the MPI Library [95]). Please note that the selection of the communication library to use is orthogonal to the focus of this work.

To support the compiler-directed message routing, we extend the switch design to handle two types of routing schemes: *default X-Y routing* and *compiler-directed routing*. The header of each packet contains a flag bit, indicating which routing mechanism to use for a given packet (0: X-Y routing; 1: compiler-directed routing). A packet using the default X-Y routing has the identification (ID) of the destination node in its header, as shown in the upper part of Figure 2.26. When a switch receives such a packet, it forwards the packet according to the X-Y routing algorithm.

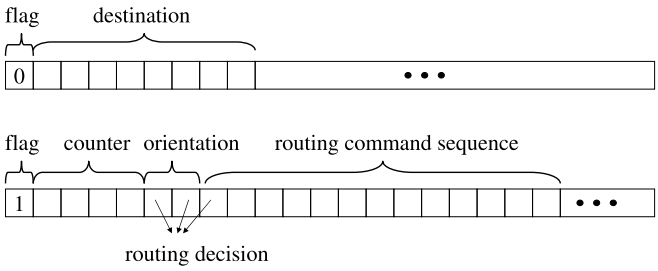


Fig. 2.26. Fields in the header of a packet (Top: default X-Y routing; Bottom: compiler-directed routing).

On the other hand, the header of a packet that employs the compiler-directed routing contains three fields (see the lower part of Figure 2.26): the hop counter (4 bits), the orientation (2 bits), and the routing command sequence (13 bits). Assuming that node, P_i, sends packet, p, to node, P_j, for each switch, S_k, on the path of this packet, a corresponding bit in the routing command sequence of the packet tells the switch to which output port to forward this packet.

Table 2.6. Routing decisions based on orientation and routing command bits (N: North; S: South; W: West; E: East).

Orientation	00	00	01	01	10	10	11	11
Routing command	0	1	0	1	0	1	0	1
Routing decision	N	E	N	W	S	E	S	W

The meaning of a routing command bit, however, is interpreted along with the value of the orientation field. This means that the compiler can only choose an alternate path from among the set of possible *shortest paths*. Once the orientation of a path is known (Northwestern: 01; Southwestern: 11; Northeastern: 00; and Southeastern: 10), only a single bit of the routing command, indicating the dimension (X: 1; Y: 0), can determine the routing decision (North, South, West, or East). Table 2.6 provides the meaning of routing commands for different values of the orientation field. The node sending a given packet sets the value of the hop counter of that packet. As the packet moves forward from one switch to another, the hop counter number decreases by one. When the counter value becomes zero, the packet has arrived at its destination. Due to the limited number of bits available in a packet header in the current implementation, the compiler-directed routing mechanism is not applicable for a packet whose source and destination nodes are more than 13 hops apart. For such a packet, the default X-Y routing mechanism applies.

2.6.3 Link Signature and Communication Graph

Assume that a parallel program executing on the mesh-based NoC architecture consists of n parallel threads, $\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_n$, and thread $\mathcal{P}_p$ is scheduled to run on the p^{th} mesh node. These

threads send messages to each other using communication commands (send operations). We denote the set of communication commands in thread $\mathcal{P}_p$ using $C_p = \{\mathcal{M}_{1,p}, \mathcal{M}_{2,p}, ..., \mathcal{M}_{k,p}, ..., \mathcal{M}_{q,p}\}$, where q is the total number of communication commands in the program code of thread $\mathcal{P}_p$ and $\mathcal{M}_{k,p}$ is the k^{th} communication command in the code of $\mathcal{P}_p$. For this study, all the messages sent by a given $\mathcal{M}_{k,p}$ follow the same route in the NoC. At a given point in execution, multiple messages may be undergoing transmission on the mesh. Representing the network state using a set of message-send operations, S_i, is:

$$S_i \quad = \quad \{\mathcal{M}_{k,p} \mid \text{A message sent by } \mathcal{M}_{k,p} \text{ is in transmission over the mesh}\}.$$

$S_0 = \phi$ represents a state in which no message is in transmission.

Given a specific network state, a further determination of link utilization at this state is necessary. The *link utilization vector* (LUV) for a given send operation, $\mathcal{M}_{k,p}$, is a vector $\vec{u}_{k,p}$, the j^{th} element of which gives the number of packets sent by $\mathcal{M}_{k,p}$ and transferred through the j^{th} communication link of the mesh. Thus, a *link signature* (LS), $\vec{s}_i$, to represent the link utilization at a network state S_i, is:

$$\vec{s}_i = \sum_{\mathcal{M}_{k,p} \in S_i} \vec{u}_{k,p},$$

where $\sum$ denotes element-wise vector addition operator.

Given a vector, $\vec{w}$ (which can be either an LUV or an LS), function $\theta(\vec{w})$ returns the set of links used by the message(s) captured by $\vec{w}$. Figure 2.27 gives an example link signature

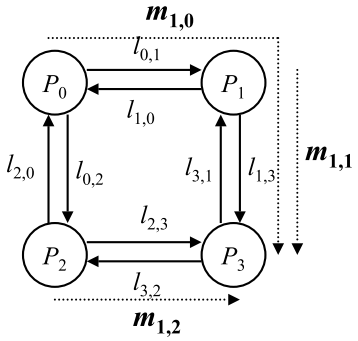
calculation. The network state $S_1 = \{\mathcal{M}_{1,0}, \mathcal{M}_{1,1}, \mathcal{M}_{1,2}\}$ in this example indicates a gather type of communication. Three concurrent 20-packet messages, $m_{1,0}$, $m_{1,1}$ and $m_{1,2}$, are sent by commands $\mathcal{M}_{1,0}$, $\mathcal{M}_{1,1}$, and $\mathcal{M}_{1,2}$, respectively, as shown in Figure 2.27(a). The first task is to obtain the LUVs for the corresponding send operations and then add them to compute the corresponding LS for this state, as shown in Figure 2.27(b). Applying function θ to this link signature, we obtain $\theta(\vec{s}_1) = \{l_{0,1}, l_{2,3}, l_{1,3}\}$, which means that this state has only three links in use. From the resulting signature, one can also see that link $l_{1,3}$ has the highest communication load (40 packets).

The network state changes during the course of execution. More specifically, the network transitions from a state, S_i, to another state, S_j, in two situations:

- A new message is sent by communication command, $\mathcal{M}_{k,p}$. In this case, $S_j = S_i \cup \{\mathcal{M}_{k,p}\}$.
- A message sent by communication command $\mathcal{M}_{k,p}$ arrives at its destination node. In this case, $S_j = S_i - \{\mathcal{M}_{k,p}\}$.

A *communication graph* (CG) captures the communication behavior of a program. A communication graph is an undirected graph, in which each vertex corresponds to a network state and, each edge (S_i, S_j) indicates the transition between states, S_i and S_j. $W_{i,j}$, the weight attached to edge, (S_i, S_j), gives the number of transitions taking place between states, S_i and S_j, during the execution of this program.

We use *profiling* to build the CG of a parallel program. Specifically, we instrument the target program to notify a profiler each time a node sends a message or when a message arrives its destination node. The profiler keeps track of the current network state, S_i. When



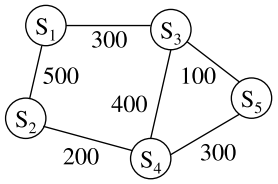
(a) A gather type of communication in a two-by-two mesh (Target node: P_3).

Links:	$l_{0,1}$	$l_{1,0}$	$l_{2,3}$	$l_{3,2}$	$l_{0,2}$	$l_{2,0}$	$l_{1,3}$	$l_{3,1}$
$\vec{u}_{1,0}$:	(20	0	0	0	0	0	20	0)
$\vec{u}_{1,1}$:	(0	0	0	0	0	0	20	0)
$\vec{u}_{1,2}$:	(0	0	20	0	0	0	0	0)
$\vec{s}_1$:	(20	0	20	0	0	0	40	0)

(b) Link utilization vectors ($\vec{u}_{1,0}$, $\vec{u}_{1,1}$, and $\vec{u}_{1,2}$) and link signature ($\vec{s}_1$) for the scenario in (a) (assuming all messages have a size of 20 packets).

Fig. 2.27. A link signature calculation example.

the profiler receives a notification from the instrumented program, it computes the new state, S_j, and increases the value of $W_{i,j}$, which represents the number of transitions between S_i and S_j. After the program completes its execution, we construct its CG based on the computed network states, the state transitions, and the values of $W_{i,j}$. Figure 2.28(a) illustrates an example communication graph.



(a) A communication graph.

step	vertex-pair considered	
1	S_1	S_2
2	S_1	S_3
3	S_3	S_4
4	S_4	S_5

(b) Processing order by Scheme I.

step	vertex-pair considered	
1	S_1	S_2
2	S_3	S_4
3	S_4	S_5

(c) Processing order by Scheme II.

Fig. 2.28. Two different approaches traversing a CG (a shaded vertex indicates the corresponding link signature NOT modified at the current step).

Based on the concepts of link signature and communication graph, we present a high level view of our compiler-based approach in Figure 2.29. This approach profiles the parallel application code to be optimized and builds a communication graph, which captures the communication pattern of the entire parallel program. Given a communication graph, a link reuse optimizer statically re-routes the pre-determined message routing paths to increase link reuse. The output of the link reuse optimizer is a modified communication graph. Subsequently, the

code rewriter module annotates each message-send operation in the application code with the determined message routing information and generates an optimized parallel code.

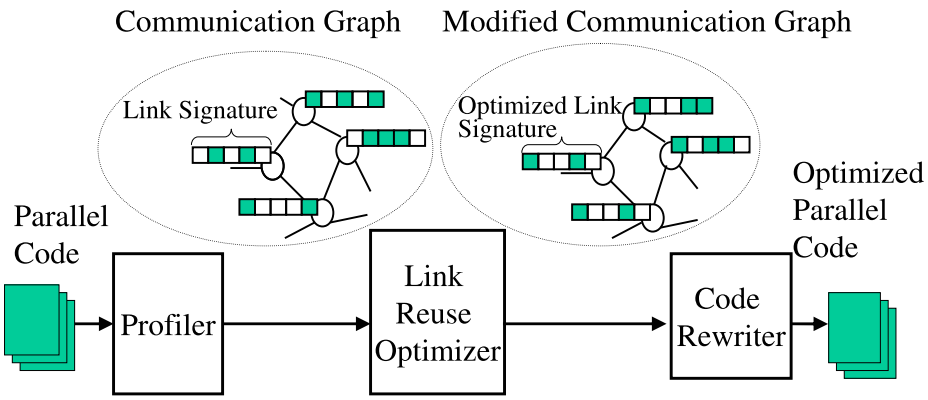


Fig. 2.29. Compiler-directed link reuse optimization scheme. Each vertex of the communication graph captures a network state.

2.6.4 Optimizing Link Reuse

The problem of increasing link reuse can express as one of maximizing link reuse between adjacent vertices in a communication graph. That is, when going from one state to another at runtime, the desire is to reuse the same set of links as much as possible. Each vertex in a CG has a default link signature, obtained using the default X-Y routing for messages sent by the communication commands in that vertex. The compiler's task is to re-assign link signatures to those vertices, with an attempt to maximize communication link reuse.

1) Traversing Communication Graph. It is necessary to determine an order in which we traverse network states to assign them new link signatures, as assigning a signature to a given

vertex (network state) will affect the selection of the signatures for its neighbors in the CG. At least two different ways of traversing a CG exist. The first approach starts with the edge having the largest weight and performs signature re-assignment to associated vertices. After that, this approach considers the edge with the next largest weight among the edges incident on selected vertices. Since one of the vertices of the edge under consideration has an assigned signature, signature assignment is for the other vertex only. This step repeats until all the vertices are processed. This approach, referred to as *Scheme I*, expands the selected set of edges at each step by considering only the neighbors. The second approach, referred to as *Scheme II*, starts the same way as Scheme I. However, after selecting the edge with the largest weight and assigning new signatures to corresponding vertices, the next edge selection considers all the remaining edges (i.e., not just those that are incident on the previously selected vertices). To illustrate the differences between Scheme I and Scheme II, let us consider the example CG shown in Figure 2.28(a). The pairs of vertices considered by Scheme I and Scheme II at each step (for signature re-assignment) appear in Figure 2.28(b) and Figure 2.28(c), respectively. Figure 2.30(a) and Figure 2.30(b) are the pseudo-codes for the compiler algorithms that implement Scheme I and Scheme II, respectively.

2) Routing Flexibility. Re-routing (the messages sent by) communication commands can achieve improvement in communication link reuse. In the present scheme, only the *shortest paths* are considered for re-routing messages since this typically causes less energy consumption than using longer paths. Even with this restriction, in many cases a certain re-routing flexibility is available. Consider a two-dimensional mesh where a message, m, is to be sent from a source node, (x_s, y_s), to a destination node, (x_d, y_d). If $m = |x_d - x_s|$ and $n = |y_d - y_s|$, this message has C^m_{m+n} possible, unique, shortest paths. Recall from Section 2.6.3 that the defined

Input:
A communication graph $CG(V, E, W)$;
Output:
$\vec{u}_{i,p}$ for each $\mathcal{M}_{i,p}$ in the program;

P — the set of network states that have been processed;
R — the set of communication commands whose LUVs have been determined;
C — the set of candidate edges;

$P = \phi$; $R = \phi$; $C = \phi$;
while($P \neq V$) {
  if($C = \phi$)
    $C = \{(S_x, S_y)\}$ if $W_{x,y}$ is maximum;
  select $(S_i, S_j) \in C$ with maximum $W_{i,j}$;
  call reroute(S_i, S_j, R);
  $P = P \cup \{S_i, S_j\}$; // processed S_i and S_j
  $R = R \cup S_i \cup S_j$; // determined LUVs for S_i and S_j
  $C' = \{(S_a, S_b) | S_a \in P \wedge S_b \in (V - P)\}$;
  $C = (C - \{(S_i, S_j)\}) \cup C'$;
}

(a) Scheme I.

Input:
A communication graph $CG(V, E, W)$;
Output:
$\vec{u}_{i,p}$ for each $\mathcal{M}_{i,p}$ in the program;

P — the set of network states that have been processed;
R — the set of communication commands whose LUVs have been determined;
C — the set of candidate edges;

$P = \phi$; $R = \phi$; $C = E$;
while($P \neq V$) {
  select a $(S_i, S_j) \in C$ if $W_{i,j}$ is maximum;
  call reroute(S_i, S_j, R);
  $P = P \cup \{S_i, S_j\}$; // processed S_i and S_j
  $R = R \cup S_i \cup S_j$; // determined LUVs for S_i and S_j
  $C = C - \{(S_i, S_j)\}$;
}

(b) Scheme II.

Fig. 2.30. Pseudo codes for two CG traversing schemes (Scheme I and Scheme II).

link utilization vector represents the path taken by a message. Now, *a set of alternate link utilization vectors* (ALUV), $A_{i,p}$, can represent all the alternate (shortest) paths available to a message sent by the communication command, $\mathcal{M}_{i,p}$. Therefore, re-routing a message can be thought of replacing the current LUV of an associated $\mathcal{M}_{i,p}$ with a new LUV selected from the corresponding ALUV set. The number of alternate link utilization vectors in an ALUV set (i.e., $|A_{i,p}|$) thus represents the *routing flexibility* for (the messages sent by) communication command, $\mathcal{M}_{i,p}$.

3) Problem Formulation. Formulating the problem of optimizing the communication link reuse between two neighboring vertices in a CG focuses on two vertices, S_a and S_b, as shown in Figure 2.31. Each communication command, e.g., $\mathcal{M}_{a3,p3}$ in state S_a, has a set of alternate link utilization vectors, which represent the alternate, shortest paths for the corresponding message. A single communication command is likely to appear in multiple network states. However, we can change the associated routing only once (i.e., all the messages sent by it are always transferred through the same path). Therefore, when optimizing states, S_a and S_b, the possibility exists that some communication commands have already been assigned new routes during the previous steps (such as $\mathcal{M}_{a4,p4}$ and $\mathcal{M}_{a5,p5}$ in state, S_a, and $\mathcal{M}_{b3,p3}$ in state, S_b) and these routes cannot be further changed. The goal is to choose a new LUV for each send operation (except those already assigned new LUVs) in S_a and S_b minimizing the number of unique links used in S_a and S_b (i.e., maximizing the link reuse).

Selecting the new utilization vectors should not degrade the performance of the default routing scheme. However, selecting alternate re-routings can increase the network contention. Therefore, some sort of *performance constraint* should be introduced for selecting the re-routings. In one network state, the communication link with the highest load often heavily

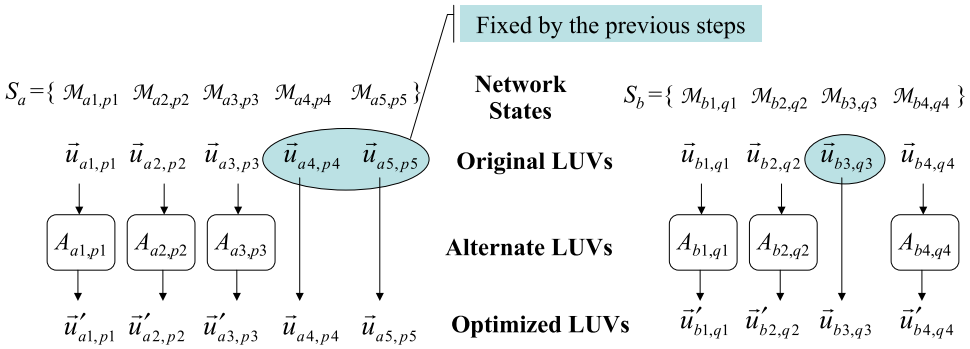


Fig. 2.31. Link reuse optimization between two network states, S_a and S_b.

influences the latency of transmitting messages. The link with the highest load corresponds to the largest entry in the link signature associated with a given state. The higher the value of the largest entry, the more likely there will be severe link contention. Therefore, in optimizing link reuse, and in order to avoid degrading network latency, increasing the value of the largest entry in any original link signature is undesirable (although the largest value may be permitted to shift to another link). For example, given the default link signature (10, 40, 10, 10, 0, 0, 0, 0) of a network state, from the performance perspective, an alternate signature such as (10, 50, 0, 10, 0, 0, 0, 0) is inexpedient. However, for another alternate signature (40, 20, 10, 0, 0, 0, 0, 0), the compiler has difficulty judging its impact on latency as compared to the default, (10, 40, 10, 10, 0, 0, 0, 0). The current implementation accepts this second alternate signature. Since the proposed approach is built within a compiler in a modular fashion, it is very flexible. That is, if desired, one can easily explore more strict performance constraints. We want to emphasize, however, that judging the latency behavior of a network state based on its signature at compile time is a very difficult problem in general. This is the reason for adopting a simple compile-time

heuristic, based on the assumption that the link with the largest load typically forms the main latency bottleneck.

4) Heuristic. We present a heuristic for calculating the routings for (the messages sent by) the communication commands. The pseudo code for our heuristic is given in Figure 2.32.

First, for each $\mathcal{M}_{i,p}$ unassigned with a new routing in network states S_a and S_b, we calculate its LUV and ALUV. Also, we obtain the link signatures for states, S_a and S_b. Based on the signatures, we compute num_links, the total number of the links used in S_a and S_b combined. The goal is to reduce the value of this variable as much as possible under performance constraints. We sort the communication commands in these two states into a sequence with ascending routing flexibilities (represented by $|A_{i,p}|$). We start with the communication command that has the lowest routing flexibility and assign a proper route to it. The reason for starting with the command with the lowest flexibility is that deciding the routing of this command early in the optimization process is ultimately more beneficial. Otherwise, due to its limited routing flexibility, difficulties may arise for assigning a new LUV to it after many other send operations have their routing paths fixed. We assign the appropriate routes to the communication commands, one-by-one, until processing all commands in S_a and S_b is complete.

The method for choosing a route for a communication command $\mathcal{M}_{i,p}$ (recall that all the messages sent by the same $\mathcal{M}_{i,p}$ follow the same path in the NoC) requires some explanation. Without losing generality, assuming that the send operation to be re-routed belongs to state S_a, the heuristic selects a new LUV for operation $\mathcal{M}_{i,p}$ by considering all the re-routing options captured in $A_{i,p}$. For each alternate re-routing, the heuristic algorithm checks whether the performance constraint is satisfied with respect to state S_a. If the performance constraint is met, the new link signature is computed for state S_a. Subsequently, using this new signature,

Input:
S_a, S_b — two network states;
R — the set of communication commands whose LUVs have been determined
Output:
$\vec{u}_{i,p}$ — LUV for each $\mathcal{M}_{i,p} \in ((S_a \cup S_b) - R)$.

procedure reroute(S_a, S_b, R) {
  for each $\mathcal{M}_{i,p} \in (S_a \cup S_b - R)$ {
    calculate $\vec{u}_{i,p}$, the LUV of $\mathcal{M}_{i,p}$, based on X-Y routing;
    calculate $A_{i,p}$, the ALUV of $\mathcal{M}_{i,p}$;
    calculate $\vec{s}_a$ and $\vec{s}_b$, the link signatures of state S_a and S_b;
    $num\_links = |\theta(\vec{s}_a) \cup \theta(\vec{s}_b)|$;
    if($\theta(\vec{s}_a) \subseteq \theta(\vec{s}_b) \vee \theta(\vec{s}_b) \subseteq \theta(\vec{s}_a)$) return;
    sort all $\mathcal{M}_{i,p} \in (S_a \cup S_b - R)$ by routing flexibility $|A_{i,p}|$
    for each $\mathcal{M}_{i,p} \in (S_a \cup S_b - R)$ {
      for each $\vec{v} \in A_{i,p}$ {
        if $\mathcal{M}_{i,p} \in S_a \wedge \mathcal{M}_{i,p} \in S_b$ {
          calculate $\vec{s}_{a\_new}$ and $\vec{s}_{b\_new}$ using $\vec{v}$ as LUV of $\mathcal{M}_{i,p}$
          if($\max(\vec{s}_{a\_new}) > \max(\vec{s}_a)$) continue;
          if($\max(\vec{s}_{b\_new}) > \max(\vec{s}_b)$) continue;
          if($|\theta(\vec{s}_{a\_new}) \cup \theta(\vec{s}_{b\_new})| \geq num\_links$) continue;
          if($|\theta(\vec{s}_{a\_new}) \cup \theta(\vec{s}_{b\_new})| = num\_links \wedge |\theta(\vec{s}_{a\_new}) \cap \theta(\vec{s}_{b\_new})| \leq |\theta(\vec{s}_a) \cap \theta(\vec{s}_b)|$)
            continue;
          replace $\vec{u}_{i,p}$ with $\vec{v}$;
          $\vec{s}_a = \vec{s}_{a\_new}$; $\vec{s}_b = \vec{s}_{b\_new}$;
          $num\_links = |\theta(\vec{s}_a) \cup \theta(\vec{s}_b)|$;
        } else {
          if($\mathcal{M}_{i,p} \in S_a$) { $x = a$; $y = b$; }
          else { $x = b$; $y = a$; }
          calculate $\vec{s}_{x\_new}$ by using $\vec{v}$ as LUV of $\mathcal{M}_{i,p}$
          if $max(\vec{s}_{x\_new}) > max(\vec{s}_x)$ continue;
          if($|\theta(\vec{s}_{x\_new}) \cup \theta(\vec{s}_y)| > num\_links$) continue;
          if($|\theta(\vec{s}_{x\_new}) \cup \theta(\vec{s}_y)| = num\_links \wedge |\theta(\vec{s}_{x\_new}) \cap \theta(\vec{s}_y)| \leq |\theta(\vec{s}_a) \cap \theta(\vec{s}_b)|$)
            continue;
          replace LUV of $\mathcal{M}_{i,p}$, i.e., $\vec{u}_{i,p}$, with $\vec{v}$;
          $\vec{s}_x = \vec{s}_{x\_new}$;
          $num\_links = |\theta(\vec{s}_x) \cup \theta(\vec{s}_y)|$;
        }
      }
    }
  }
}
function $\max(\vec{v})$ { return the value of the largest entry of $\vec{v}$; }

Fig. 2.32. Communication link reuse optimization heuristic.

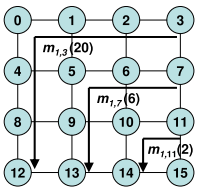
denoted $\vec{s}_{a\_new}$, and the current signature of state S_b ($\vec{s}_b$), the heuristic re-calculates the total number of links used by the messages in S_a and S_b. This total number of links is num_links. Among all the alternatives in the set, $A_{i,p}$, that satisfy the performance constraint, the heuristic selects the one that leads to the minimum num_links value. If num_links cannot be reduced with any alternate utilization vector, the choice is for the alternate LUV that maximizes the number of links reused by the two states (i.e., $|\theta(\vec{s}_a) \cap \theta(\vec{s}_b)|$ is maximized). The utilization vector for this communication command is then fixed, and the routing assignment for this command is complete at this point. Once a communication command is given a new LUV, this command is not considered again when processing the other vertex-pairs. When all the send operations have been assigned new routes, the thread codes are annotated with the corresponding LUVs.

The computational complexity of the heuristic is $O(N * K * C^m_{m+n})$, where N is the number of network states, K is the number of send operations, and C^m_{m+n} represents the largest routing flexibility in an $m \times n$ mesh, as mentioned earlier.

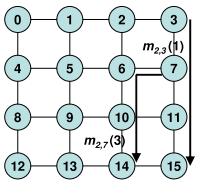
5) Example. We use an example here to illustrate how the link reuse optimization scheme works. Since the steps traversing a communication graph are relatively simple, we only present the link reuse optimization between two adjacent network states. The focus is on a four-by-four mesh network and two neighboring network states in a CG: S_a and S_b. The goal is to maximize link reuse between them, assuming that $S_a = \{\mathcal{M}_{1,3}, \mathcal{M}_{1,7}, \mathcal{M}_{1,11}\}$, and $S_b = \{\mathcal{M}_{2,3}, \mathcal{M}_{2,7}\}$. Figure 2.33(a) and Figure 2.33(b) illustrate the default routings of the messages sent by these communication commands in S_a and S_b, respectively. We assume that message $m_{i,j}$ is sent by the send operation $\mathcal{M}_{i,j}$. For example, message $m_{2,7}$ is sent by the send operation, $\mathcal{M}_{2,7}$, which is the second send operation in the code of thread $\mathcal{P}_7$ that runs on mesh node 7. The target node of this send operation is node 14. We further assume, for clarity of

presentation, the size of each message is 20 packets. One can calculate the LUV for each send operation and the LS for each network state, as shown in Figure 2.33(d), under the default routings. The ALUV sets for the send operations are also calculated, although they are not shown here due to space limitations. However, the routing flexibility, given within the parentheses associated with the corresponding message, appears in Figure 2.33(a) and Figure 2.33(b).

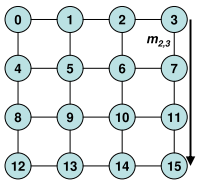
The task is to select new LUVs for send operations, with the assumption that no send operation in these two states has fixed its LUV in the previous optimization steps (i.e., when processing the other state pairs). Thus, considering all the operations in the two states, we start from $\mathcal{M}_{2,3}$, which has the lowest routing flexibility. With a flexibility of 1, it has no alternate LUV. Consequently, the route for this message is easily fixed, as shown in Figure 2.33(c) (this example uses the routings of the corresponding messages to represent the selected LUVs). Next, $\mathcal{M}_{1,11}$ has a routing flexibility of 2. However, no beneficial alternate LUV for this communication exists, and the approach maintains its default LUV, as is shown in Figure 2.33(e). The next send operation to process is $\mathcal{M}_{2,7}$. Since using any alternate LUV for it would violate the performance constraint in state S_b (for example, using either of the two alternate LUVs, link $l_{7,11}$ would overload), this operation is also fixed with its default LUV. This step completes the processing of all the send operations in state S_b. For each communication command in state, S_b, our approach decides to employ the default LUV, and the resulting routings are the same as those in Figure 2.33(b). Thus, we do not show the result of step III in Figure 2.33. In the following two steps, the heuristic returns beneficial re-routings for operations $\mathcal{M}_{1,7}$ and $\mathcal{M}_{1,3}$, as illustrated in Figure 2.33(f) and Figure 2.33(g), respectively. Each step reduces the total number of links used in the two network states (i.e., improves link reuse). Figure 2.33(g) gives the final routings for all the communication commands in state S_a. The modified LUVs and LSs returned by this



(a) Network state S_a.



(b) Network state S_b.

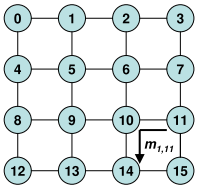


(c) Re-routing step I (state S_b).

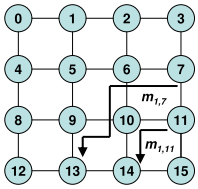
Links:	$l_{3,2}$	$l_{2,1}$	$l_{1,0}$	$l_{0,4}$	$l_{4,8}$	$l_{8,12}$	$l_{7,6}$	$l_{6,5}$	$l_{5,9}$	$l_{9,13}$	$l_{11,10}$	$l_{10,14}$	$l_{3,7}$	$l_{7,11}$	$l_{11,15}$	$l_{6,10}$	...
$\vec{u}_{1,3}$:	(20	20	20	20	20	20	0	0	0	0	0	0	0	0	0	0	...)
$\vec{u}_{1,7}$:	(0	0	0	0	0	0	20	20	20	20	0	0	0	0	0	0	...)
$\vec{u}_{1,11}$:	(0	0	0	0	0	0	0	0	0	0	20	20	0	0	0	0	...)
$\vec{s}_a$:	(20	20	20	20	20	20	20	20	20	20	20	20	0	0	0	0	...)
$\vec{u}_{2,3}$:	(0	0	0	0	0	0	0	0	0	0	0	0	20	20	20	0	...)
$\vec{u}_{2,7}$:	(0	0	0	0	0	0	20	0	0	0	0	20	0	0	0	20	...)
$\vec{s}_b$:	(0	0	0	0	0	0	20	0	0	0	0	20	20	20	20	20	...)

$$|\theta(\vec{s}_a) \cup \theta(\vec{s}_b)| = |\{l_{3,2}, l_{2,1}, l_{1,0}, l_{0,4}, l_{4,8}, l_{8,12}, l_{7,6}, l_{6,5}, l_{5,9}, l_{9,13}, l_{11,10}, l_{10,14}, l_{3,7}, l_{7,11}, l_{11,15}, l_{6,10}\}| = 16$$

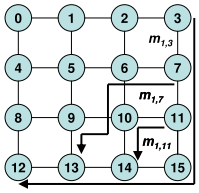
(d) LUVs and link signatures with default X-Y routing. Omitted LUV entries are zeros.



(e) Re-routing step II (state S_a).



(f) Re-routing step IV (state S_a).



(g) Re-routing step V (state S_a).

Links:	$l_{7,6}$	$l_{9,13}$	$l_{11,10}$	$l_{10,14}$	$l_{3,7}$	$l_{7,11}$	$l_{11,15}$	$l_{6,10}$	$l_{10,9}$	$l_{15,14}$	$l_{14,13}$	$l_{13,12}$	...
$\vec{u}'_{1,3}$:	(0	0	0	0	20	20	20	0	0	20	20	20	...)
$\vec{u}'_{1,7}$:	(20	20	0	0	0	0	0	20	20	0	0	0	...)
$\vec{u}'_{1,11}$:	(0	0	20	20	0	0	0	0	0	0	0	0	...)
$\vec{s}'_a$:	(20	20	20	20	20	20	20	20	20	20	20	20	...)
$\vec{u}'_{2,3}$:	(0	0	0	0	20	20	20	0	0	0	0	0	...)
$\vec{u}'_{2,7}$:	(20	0	0	20	0	0	0	20	0	0	0	0	...)
$\vec{s}'_b$:	(20	0	0	20	20	20	20	20	0	0	0	0	...)

$$|\theta(\vec{s}'_a) \cup \theta(\vec{s}'_b)| = |\{l_{7,6}, l_{9,13}, l_{11,10}, l_{10,14}, l_{3,7}, l_{7,11}, l_{11,15}, l_{6,10}, l_{10,9}, l_{15,14}, l_{14,13}, l_{13,12}\}| = 12$$

(h) LUVs and link signatures after re-routing.

Fig. 2.33. An example illustrating how our approach works. (a) and (g) are default routings and compiler-determined routings for S_a, respectively. (b) shows default routings of state S_b (routings of S_b not changed in this example).

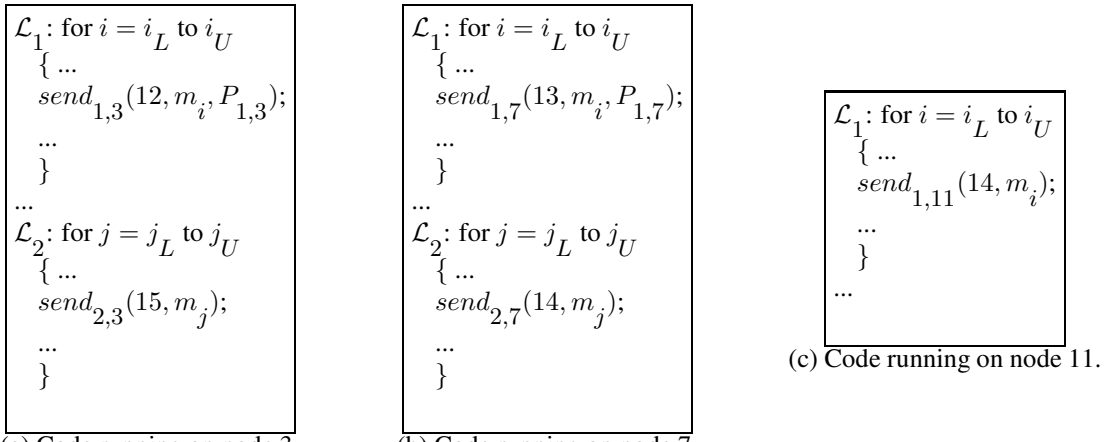
method are given in Figure 2.33(h). Clearly, the total number of links used in states S_a and S_b decreases from 16 to 12.

2.6.5 Code Rewriter

Code rewriter in our approach (see Figure 2.29) is responsible for providing a version of the message send operation, which incorporates the compiler-determined routing information. The code fragments shown in Figure 2.34 correspond to the example in Figure 2.33. After applying our algorithm, the default message send operations, $send_{1,3}(12, m_i)$ and $send_{1,7}(13, m_i)$, are replaced with the operations including specific routing information, i.e., $send_{1,3}(12, m_i, P_{1,3})$ and $send_{1,7}(13, m_i, P_{1,7})$, respectively. These versions of send operations assemble message headers by inserting routing paths according to Figure 2.26 and Table 2.6. Therefore, all the messages sent by operation $send_{1,3}$ have the message header: 10110110001110000000; whereas all the messages sent by operation $send_{1,7}$ have the header: 10100111010000000000. The other message send operations remain unchanged, i.e., for those remaining messages, the flags in their message headers are zeros, and the default X-Y routing determines the routing paths.

2.6.6 Experiments

To conduct the experiments, we implemented a flit-level on-chip interconnection network simulator. The network, parametrized similar to that in [21, 26], is in a five-by-five configuration. The link speed is set to 1Gb/sec. Each switch input port has a buffer that can hold 64 flits; each flit is 128 bits wide (packet size is 16 flits). The communication links in this network can be shutdown independently, using a time-out based mechanism as described in [90]. The time-out



(a) Code running on node 3. (b) Code running on node 7. (c) Code running on node 11.

Fig. 2.34. Code rewriting for the example in Figure 2.33.

counter threshold for the hardware-based scheme is set to 1.5μsec based on some preliminary analysis. The time taken to switch a link from the power-down state to the active state is set as 1μsec, and the energy overhead of this switching is 140μJ, based on prior research [21, 90]. When a link is turned off, it consumes zero leakage energy. Under the simulation parameters mentioned earlier, the leakage energy (which includes the leakage in the links as well as in the switches) contributes to about 41% of the total network energy consumption (leakage plus dynamic), on average, under the 65nm process technology. In order to accurately quantify the performance impact of this approach, we also connected the network simulator to SIMICS [1]. Each node of the architecture is an 800 MHz, embedded in-order, CPU with 32KB instruction and data caches.

The compiler component for this approach uses the Paradigm compiler infrastructure [96]. We modified the original front-end of the compiler to accept C codes (in addition to Fortran codes). Input code is optimized such that, for each loop nest, the outermost loop that does

not carry any loop-carried data dependencies is parallelized and the inter-processor communication is hoisted to the highest loop level possible using message vectorization. This is a well-known communication optimization. The communication library used for generating communication calls is MPI [95]. Having determined the code fragment that will be executed by each processor, invoking the approach proposed in this study follows. This approach determines link signatures, builds the communication graph, and performs message re-routing. Both communication graph traversal schemes (Scheme I and Scheme II), discussed in Section 2.6.4, are implemented. The experimental methodology includes performing experiments with three different versions for each benchmark. The first version is the one that employs the default routing, i.e., the X-Y routing and uses the underlying hardware-based link shutdown scheme, modeled after the schemes described in [90, 49, 21]. In this implementation, parameters are selected such that the energy savings achieved by link shutdown are maximized without unnecessarily hurting network latency. In the rest of this section, this scheme with the default routing and link shutdown hardware is the *base scheme*. The other two schemes evaluated for this study are Scheme I and Scheme II. Both schemes run on top of the same link shutdown hardware used in the base scheme, and the main goal in this experimental evaluation is discovering how much *additional energy savings* our compiler-directed re-routing approach generates over that of the hardware-based link shutdown approach.

The information about the applications used in this study appears in Table 2.7. A common characteristic of these benchmarks is their array/loop-intensive embedded application nature. The code sizes of these benchmarks range from 63 to 8,612 C lines, while their dataset sizes are within the range of 68.9KB-1,866.4KB. The third and fourth columns present the number of nodes and edges in the communication graph the proposed approach builds for each benchmark.

The table indicates that the number of nodes is not excessively large. The fifth column gives the leakage energy consumption in the network under the base scheme, as described earlier. The values within the parentheses show the leakage saving percentages achieved by this base scheme over an alternate scheme that does not perform any network power management. Finally, the sixth column indicates network latency of the base scheme (that is, the total number of cycles spent in the network). The values within parentheses in this column show the percentage degradation in network latency as compared to a case with no power optimization. The fifth and sixth columns show that the base scheme saves 52.2% leakage power on an average, and incurs 8.4% additional latency over a case with no power optimization.

Energy and performance results presented in the rest of this section are with respect to the absolute values listed in the fifth and sixth columns of Table 2.7, respectively. That is, results are normalized with respect to the corresponding results of the base case hardware-based link shutdown scheme. The presented performance and energy results include all extra network overheads incurred by the proposed approach (e.g., those due to augmented message headers). The increase in compilation time due to our optimization ranged between 89% (3Step-log) and Lame 236% (Lame), including time spent profiling. Since both profiling and compilation are essentially off-line activities, these increases are within acceptable range.

Figure 2.35 presents the average link utilization (the fraction of the cycles in which the links are used for transferring packets), which varies between 10.6% and 32.3%, averaging 21.4%. In other words, link utilization is not very high. The main reason for this is that applications in our experimental suite are optimized through several source-level communication optimizations that minimize inter-processor data communication. That is, the compiler is very

Table 2.7. Benchmarks from experiments and their important characteristics. Energy values are in mJ, and the latency values are in million cycles.

Benchmark Name	Brief Description	CG Size Node	CG Size Edge	Network Energy	Network Latency
Morph2	Morphological operations	338	1081	75.5(64.9%)	380.4(8.8%)
Disc	Speech/music discriminator	816	2937	99.2(46.3%)	123.6(6.9%)
Jpeg	Compression for still images	524	1729	92.7(55.8%)	445.1(10.3%)
Viterbi	A graphical Viterbi decoder	622	2239	72.5(32.9%)	150.8(9.8%)
Rasta	Speech recognition	498	1424	118.1(50.7%)	219.5(6.2%)
3Step-log	Logarithmic search motion est.	127	396	15.2(62.4%)	107.4(5.7%)
Full-search	Full search motion est.	136	448	13.5(48.0%)	95.6(12.3%)
Hier	Hierarchical motion est.	138	503	20.4(56.3%)	151.9(7.3%)
Phods	Parallel hierarchical motion est.	128	440	16.7(66.6%)	111.3(10.4%)
Epic	Image data compression	1144	4516	103.9(30.7%)	420.4(6.1%)
Lame	MP3 encoder	2062	7526	80.1(55.0%)	272.1(9.0%)
FFT	Fast Fourier transform	416	1747	87.2(55.9%)	253.3(7.4%)

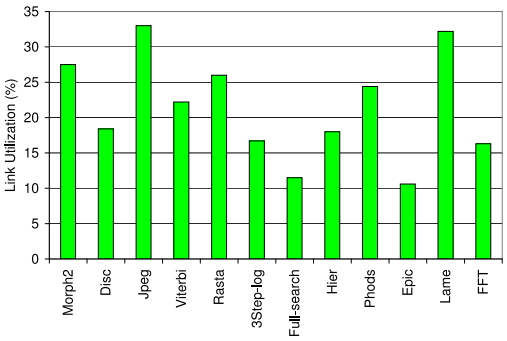


Fig. 2.35. Link utilization.

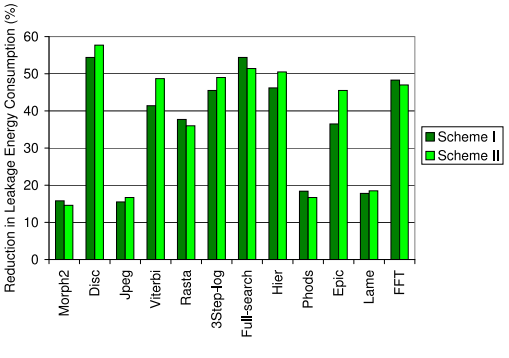
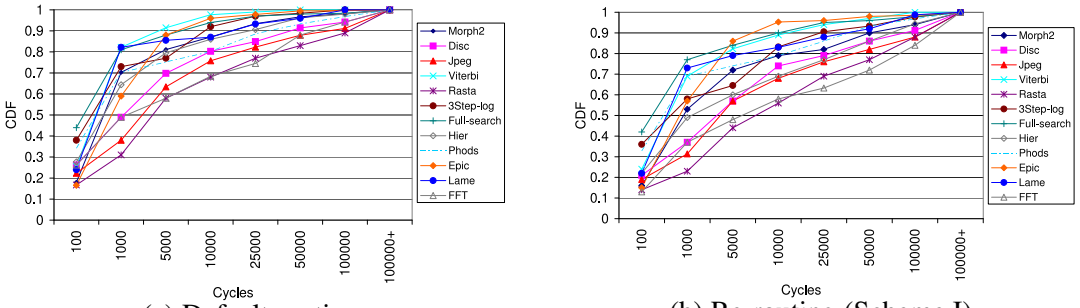


Fig. 2.36. Percentage reductions in leakage energy consumption.



(a) Default routing. (b) Re-routing (Scheme I).

Fig. 2.37. CDF for link idle periods.

successful in reducing the amount of inter-processor communication. This, in turn, reduces the average link utilization in the 5×5 mesh (a network that is not very large).

The next set of results, presented in Figure 2.36, show the percentage reduction in leakage energy consumption when using the proposed approach. Each bar in this bar-chart gives the leakage energy saving over the base scheme. Each application has two bars, one for each edge selection scheme: Scheme I and Scheme II. From these results, the average leakage energy savings, when applying the compiler-directed message re-routing, are 37.30% and 39.56% for Scheme I and Scheme II, respectively. This means that both edge selection schemes are successful in reducing the leakage energy consumption, with neither being clearly superior. These results clearly show that the compiler-directed link reuse optimization can improve the behavior of the hardware-based link shutdown scheme. To explain why message re-routing brings further savings over the base scheme alone, Figures 2.37(a) and (b) present the CDF (cumulative distribution function) curves for the link idle periods with the base scheme and the compiler-directed message re-routing approach (Scheme I). An (x,y) point on a given curve in these graphs indicates that y*100 percent of the total link idle periods are equal or less than x cycles. One can see

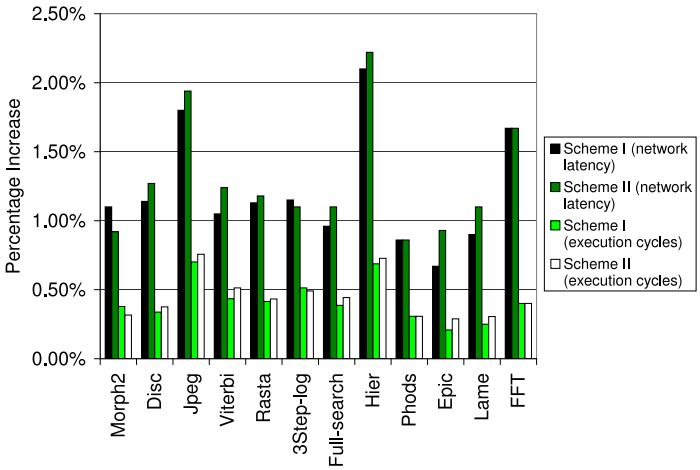


Fig. 2.38. Percentage increases in network cycles and overall execution time.

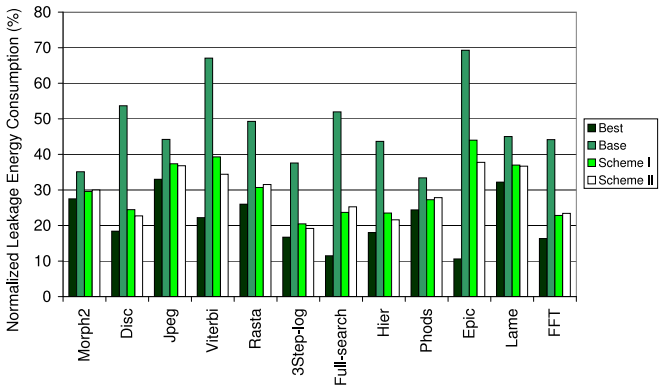


Fig. 2.39. Leakage energy consumptions.

that the message re-routing increases the link idle periods significantly. The resulting increase in idle times, in turn, allows the hardware-based link shutdown scheme to be used more effectively.

The percentage increases in the network cycles (network latency) and overall execution time over the base scheme are in Figure 2.38 (the network latency increases due to the base scheme itself appear in the last column of Table 2.7). The average network latency increase with Scheme I and Scheme II (over the base scheme) are 1.21% and 1.29%, respectively. In other words, the network overhead brought by the new approach, over the base scheme, is very small. This overhead is attributable to the link contention created by the approach during the optimization of the link signatures. A very small fraction of this increase is also due to the additional latency imposed by the augmented message headers. Also observable from Figure 2.38 is that the average increase in overall execution time is less than 0.5% for both Scheme I and Scheme II.

Figure 2.39 summarizes the normalized leakage energy consumptions with the different schemes. The results are normalized with respect to a scheme that does not employ any power management. For each application, the first bar in this graph gives the best (minimum) possible leakage consumption. "Best" in this context means that a link and the corresponding switch are turned off as soon as they become idle and turned on (without any penalty) upon the next request. The second bar for an application gives the normalized leakage consumption from the base scheme. The last two bars on the other hand are for this study's Scheme I and II. These results show that the average normalized leakage consumption values for the best case, the base scheme, Scheme I and Scheme II are 21.40%, 47.85%, 30.00% and 28.92%, respectively. That is, the base scheme, Scheme I and Scheme II reduce leakage energy consumption by 52.15%, 70.00% and 71.08%, respectively. This means that Scheme I and II save significant amounts of

leakage energy as compared to the base scheme. When considering the dynamic energy as well (in addition to leakage), we found that the total (average) energy savings achieved by the base scheme, Scheme I and Scheme II are 21.37%, 27.49% and 27.94%, respectively, including the impact of augmented message headers. These total energy savings resulting from the proposed schemes are quite significant, considering the fact that the best scheme can save, at most, 32.22% of the total network energy.

Besides the above results, we also conducted sensitivity studies either by varying the number of mesh nodes or by changing the input sizes. We tested mesh sizes from 15 nodes to 50 nodes and found that leakage energy savings obtained from different mesh sizes are similar. When the number of nodes increases, slight increases in savings occur. Exploring the effect of input size on energy savings is important because the proposed approach is profile-based and a different input set can generate different network states than those obtained by profiling. Our results indicate that energy savings are quite consistent as inputs change. This is because a different input does not significantly affect the inter-processor communication pattern of a compiler-parallelized application, although it can sometimes change the control flow of the application. As a result, little variance results from the input used to execute the application.

2.7 Summary

Reducing the power consumption of NoCs is an important optimization goal for NoC-based CMPs. Most of prior efforts on network power optimization are hardware-based schemes. These schemes are reactive by definition as they control communication link status based on the observations made in the past. The main contribution of this chapter are three compiler-directed communication power optimization. In the first two approaches, the compiler analyzes

a parallel application code, identifies the program points in the code for inserting link power control commands (either shut down/turn on or voltage scale up/down), and then modifies the code by inserting link power control function calls. Thus, the generated parallel code are power-optimized. The third approach is profile-driven, which reroutes messages to use only a subset of the links at a given time in order to increase idle periods of the remaining links. This approach enhances the effectiveness of a pure hardware-based link turn-on/off scheme dramatically. Our results demonstrate the success of these three approaches in reducing NoC energy.

Chapter 3

Non-Uniform Cache Architecture for Chip Multiprocessors

3.1 Background

Emerging CMPs contain large level-two (L2) and/or level-three (L3) caches on the processor die. For example, IBM's Power5 [87] has a 1.875MB on-chip L2 cache, Sun's Ultrasparc T1 contains 3MB of on-chip L2 cache, and Intel's dual core Itanium2 includes 2.5MB of on-die L2 cache and 24MB of on-die L3 cache. The increasing number of on-chip processing cores and the associated increasing demand for memory bandwidth will make the sizes of on-chip L2/L3 caches continue to increase [38].

Traditional memory subsystem design has assumed that each level in the memory hierarchy has a single, uniform access time. However, diminutive feature sizes exacerbate the impact of interconnect delay [35, 3, 78], making access times in large caches dependent on the physical location of the requested cache line. That is, cache access times will be transformed into variable latencies based on the distance between the requesting processor core and the target cache line.

Kim et al. [48] introduced the concept of Non-Uniform Cache Architectures (NUCA) based on the above observation. Under uniprocessor scenario, they divided a NUCA-typed L2 into multiple, individually-addressable banks in order to employ the property of variant cache access latencies for improving L2 performance. Different cache banks have different access latencies, which are dependent on the distances to the processor. They employed a gradual

migration policy to place frequently-accessed cache lines closer to the processor. A switched interconnection network (i.e., NoC) is found superior than private per-bank channels in connecting cache banks and the processor.

Designing an NoC-based NUCA architecture under CMP scenarios is more challenging, since multiple processing cores may share and/or contend the on-chip caches. Two premium L2 cache designs for CMPs are a purely shared L2 cache [8] and a purely private L2 cache [52]. For a purely shared L2 cache, the aggregate on-chip L2 caches can be accessed by any processor on chip. Each cache line inside L2 cache is unique, that is, no replicates exist. The purely shared L2 cache maximizes the on-chip cache capacity, and thus minimizes the number of off-chip memory accesses. However, with non-uniform cache accesses, a purely shared cache can exhibit a large average L2 hit latency, since frequently accessed cache lines might be placed far from the accessing processor(s). The increased L2 hit latency can offset the benefit from decreased off-chip accesses. Therefore, for a purely shared L2 cache design, some cache line placement or migration policies must be available to avoid the increase in L2 hit latencies. [8] is a representative purely shared L2 NUCA for CMPs. Beckmann and Wood adapted the NUCA design for a uniprocessor [48] to a shared L2 NUCA for a 8-core CMP. Specifically, they dispersed multiple ways within the same logic cache set across the entire chip and used cache line migration to move frequently accessed cache lines towards the requesting processor(s).

On the other hand, a purely private L2 cache [52] means that each processor has its own L2 cache, and the processor cannot directly access other processors' L2 caches. Such a purely private L2 cache design requires the maintenance of the coherence among different private L2 caches. The purely private L2 cache design provides relatively lower L2 hit latencies, since each processor has its data placed in its local L2 cache. However, the private L2 cache design does

not utilize on-chip cache capacity efficiently since each shared cache line may have a replicate inside the local L2 cache of each requesting processor. Such duplicates decrease the effective aggregate L2 cache size and leads to more off-chip memory accesses, which are detrimental to performance. Another problem with the private L2 cache is that it cannot accommodate imbalanced memory requirements from different processors, since the size of each private L2 cache is fixed at design time.

A set of research efforts studied alternate CMP NUCA designs between the purely shared and the purely private caches. These studies targeted minimizing the average L2 hit latency and/or maximizing the effective on-chip L2 capacity (i.e., minimizing the off-chip accesses). Huh et al [39] proposed a flexible CMP shared cache design by varying the degree of sharing, i.e., by changing the sizes of processor clusters, in which the processors share their local private L2 caches. Chishti et al [22] introduced CMP-NuRAPID, which employed controlled copy of read-only data, fast in-situ communication for read-write sharing and capacity stealing from neighbors. CMP-NuRapid architectures featured a flexible data placement at the cost of maintaining tag-data pointers. Zhang and Asanovic [106] designed a victim replication mechanism by duplicating L1 victims at local L2 spaces, aiming at reducing the L2 hit latency. [7] proposed a controlled adaptive selective replication mechanism with an attempt to replicate frequently accessed read-only data and thus reduce the L2 access latency without increasing L2 miss rate significantly. Chang and Sohi [17] presented a scheme of cooperative caching for CMPs based on private L2 caches through a central coherence engine. This scheme supported a spectrum of capacity sharing points between the purely private cache and the purely shared cache. To handle data sharing among processors, they modified the cache coherence protocol to maintain a data owner for each on-chip cache line and let the owner (rather than the off-chip memory hierarchy)

provide the cache line whenever another processor could not find this cache line in the local L2 space. Most of these studies replicated cache lines inside the shared L2 space, which required a proper L2 cache coherence mechanism.

Other works around caches in CMPs (not necessarily NUCA) were cache fair sharing and overall throughput maximization. Yeh and Reinman [104] presented the PDAS (Performance-Driven Adaptive Sharing) cache architecture for CMPs. This scheme dynamically partitioned a shared NUCA through monitoring the performance of each core. The objective was to maximize the throughput and to guarantee a minimum performance level for each thread at the same time. An OS level shared cache management scheme for CMPs was proposed in [75], which provided a set of cache quota management policies on top of a hardware cache quota enforcement scheme. In [52], the authors studied fair sharing and partitioning of a traditional shared L2 cache (with a constant access latency). Targeting the traditional shared cache as well, a framework, CQoS, was proposed in [43]. With an attempt to reduce the interferences among heterogeneous threads, this framework assigned different priorities to different memory access streams and employed three mechanisms (cache set partitioning, selective cache allocation, and heterogeneous cache regions) to enforce these priorities. Other works on CMP caches included the management of traffics between L2 and L3 caches [91] and the OS-directed data mapping from pages to L2 cache slices [23].

In this chapter, we focus on the NUCA design for CMPs and present two proposals. First, we observe that the introduction of three-dimensional (3D) circuits [27, 63] provides an opportunity to reduce wire lengths. Considering the characteristics of 3D integration technology, we propose the design of a 3D NoC-based NUCA architecture for CMPs in Section 3.2.4. We adapt the available NUCA [48] architecture into the new 3D scenario, and give shared L2 NUCA

management policies, which benefit from both [8] and [22]. Second, by tracing the access pattern of a shared L2 cache in a 8-core CMP, we observe that private cache lines (accessed by only one processor), dominate the L2 cache space, while accesses to shared cache lines (accessed by two or more processors) dominate the overall L2 accesses. Thus, in Section 3.3.4, we propose a migration-based L2 NUCA design, which employs an initial cache line placement policy and a migration scheme for hot shared cache lines.

3.2 Employing 3D Integration for CMP NUCAs

3.2.1 Motivation

A three dimensional (3D) chip is a stack of multiple device layers with direct vertical interconnects tunneling through them [27, 63]. The benefits of 3D ICs include: 1) higher packing density due to the addition of a third dimension to the conventional two-dimensional layout, 2) higher performance due to reduced average interconnect length, and 3) lower interconnect power consumption due to the reduction in total wiring length [44]. Joyner et al. [44] have shown that three-dimensional architectures reduce wiring length by a factor of the square root of the number of layers used. For example, a 4-layer 3D NoC would have, on average, $\approx \sqrt{4} = 2$ times shorter wiring length, as illustrated in Figure 3.1. Consequently, 3D technology can be useful in reducing the access latencies to remote cache banks of a NUCA architecture.

Researchers for 3D technology have so far focused on physical aspects, low-level process technologies, and developing automated design and placement tools [27, 29, 24]. Research at the architectural level has also surfaced [12, 97]. Specific to 3D memory design, [74, 105] have

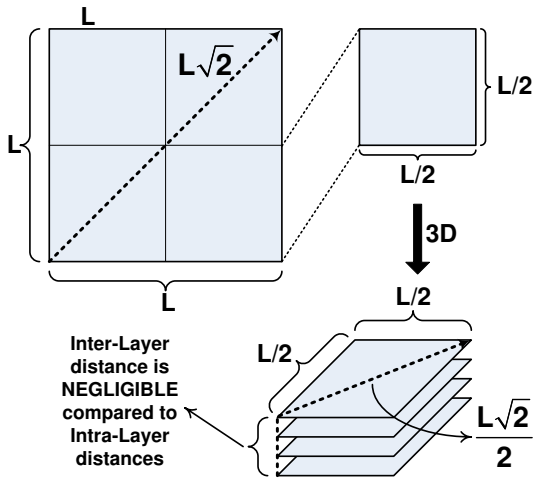


Fig. 3.1. Wiring scales in length as the square root of the number of layers in three dimensions.

studied multi-bank uniform cache structures. In our work, we focus our attention on the design of a 3D NoC-based non-uniform L2 cache architecture.

3.2.2 3D NoC-Bus Hybrid Architecture

There are currently various 3D technologies being explored in industry and academia, but the two most promising ones are Wafer-Bonding [27] and Multi-Layer Buried Structures (MLBS) [45]. Wafer-bonding technology processes each active device layer separately and then connects the layers in a single entity. For MLBS, the front-end processing is repeated on a single wafer to build multiple device layers, before the back-end process builds interconnects among the devices. Since MLBS is not compatible with current manufacturing processes, it is not as appealing as wafer bonding techniques [12, 41]. There are also two primary wafer orientation schemes, Face-To-Face [12] and Face-To-Back [41, 29]. While the former provides the greatest layer-to-layer via density, it is suitable for two-layer organizations, since additional layers would

have to employ back-to-back placement using larger and longer vias. Face-To-Back, on the other hand, provides uniform scalability to an arbitrary number of layers, despite a reduced inter-layer via density. Hence, to provide scalability and easy manufacturability, we assume in this work the use of Face-To-Back Wafer-Bonding.

Our proposed architecture for multiprocessor systems with large shared L2 caches involves placement of CPUs on several layers of a 3D chip with the remaining space filled with L2 cache banks. Most 3D IC designs observed in the literature so far have not exceeded 5 layers, mostly due to manufacturability issues, thermal management, and cost. As previously mentioned, the most valuable attribute of 3D chips is the very small distance between the layers. A distance on the order of tens of microns is negligible compared to the distance traveled between two network on-chip routers in 2D (1500 μm on average for a 64 KB cache bank implemented in 70 nm technology). This characteristic makes traveling in the vertical (inter-layer) direction very fast as compared to the horizontal (intra-layer).

One inter-layer interconnect option is to extend the NoC into three dimensions. This requires the addition of two more links (up and down) to each router. However, adding two extra links to an NoC router will increase its complexity (from 5 links to 7 links). This, in turn, will increase the blocking probability inside the router since there are more input links contending for an output link. Moreover, the NoC is, by nature, a multi-hop communication fabric, thus it would be unwise to place traditional NoC routers on the vertical path because the multi-hop delay and the delay of the router itself would overshadow the ultra fast propagation time.

It is not only desirable, but also feasible, to have single-hop communication amongst the layers because of the short distance between them. To that effect, we propose the use of dynamic Time-Division Multiple Access (dTDMA) buses as “Communication Pillars” between

the wafers, as shown in Figure 3.2. These vertical bus pillars provide single-hop communication between any two layers, and can be interfaced to a traditional NoC router for intra-layer traversal using minimal hardware, as will be shown later. Furthermore, hybridization of the NoC router with the bus requires only one additional link (instead of two) on the NoC router. This is the case because the bus is a single entity for communicating both up and down. Due to technological limitations and router complexity issues, not all NoC routers can include a vertical bus, but the ones that do form gateways to the other layers. Therefore, those routers connected to vertical buses have a slightly modified architecture, as to be explained in this section.

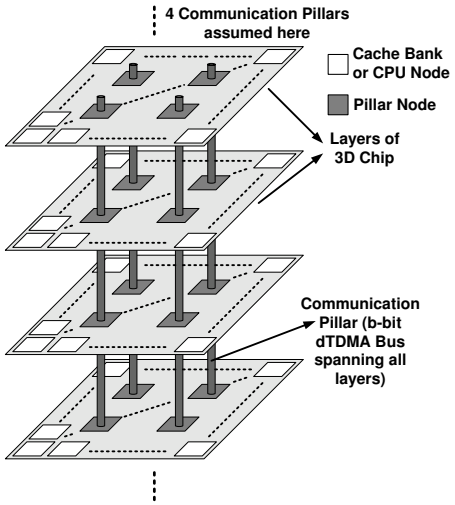


Fig. 3.2. Proposed 3D Network-in-Memory architecture

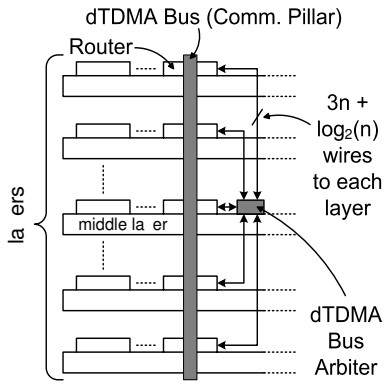


Fig. 3.3. Side view of the 3D chip with the dTDMA bus (pillar).

3.2.2.1 Using dTDMA Bus as Communication Pillars

The dTDMA bus architecture [77] eliminates the transactional character commonly associated with buses, and instead employs a bus arbiter which dynamically grows and shrinks the number of timeslots to match the number of active clients. Single-hop communication and transaction-less arbitrations allow for low and predictable latencies. Dynamic allocation always produces the most efficient timeslot configuration, making the dTDMA bus nearly 100% bandwidth efficient. Each pillar node requires a compact transceiver module to interface with the bus. Details of the operation of this module can be found in [77]. The total number of wires required by the control signals from the arbiter to each layer is $3n + log_2(n)$, for n layers. Because of its very small size, the dTDMA bus interface is a minimal addition to the NoC router.

The presence of a centralized arbiter is another reason why the number of vertical buses, or pillars, in the chip should be kept low. An arbiter is required for each pillar with control signals connecting to all layers, as shown in Figure 3.3. The arbiter should be placed in the middle layer of the chip to keep wire distances as uniform as possible. The area occupied by the

arbiter and the transceivers is much smaller compared to the NoC router, thus fully justifying our decision to use this scheme as the vertical gateway between the layers. The area and power numbers of the dTDMA components and a generic 5-port (North, South, East, West, local node) NoC router (all synthesized in 90 nm technology) are shown in Table 3.1. Clearly, both the area and power overheads due to the addition of the dTDMA components are orders of magnitude smaller than the overall budget. Therefore, using the dTDMA bus as the vertical interconnect is of minimal area and power impact. A 7-port NoC router was considered and eliminated in the design search due to prohibitive contention issues, multi-hop communication in the vertical direction, and substantially increased area/power overhead due to an enlarged crossbar and more complicated switch arbiters. The dTDMA bus was observed to be better than an NoC for the vertical direction as long as the number of device layers was less than 9 (bus contention becomes an issue beyond that).

The length of vertical interconnect between two layers is assumed to be 10 μm. According to [28], the parasitics of inter-tier vias have a small effect on power and delay, because of their small length (i.e. low capacitance) and large cross-sectional area (i.e. low resistance).

Table 3.1. Area and power overhead of dTDMA bus.

Component	**Power**	**Area**
Generic NoC Router (5-port)	119.55 mW	0.3748 mm^2
dTDMA Bus Rx/Tx (2 per client)	97.39 μW	0.00036207 mm^2
dTDMA Bus Arbiter (1 per bus)	204.98 μW	0.00065480 mm^2

Table 3.2. Area overhead of inter-wafer wiring for different via pitch sizes.

Bus Width	**Inter-Wafer Area (due to dTDMA Bus wiring)**			
	10 μm	**5 μm**	**1 μm**	**0.2 μm**
128 bits(+42 control)	62500 μm^2	15625 μm^2	625 μm^2	25 μm^2

The density of the inter-layer vias determines the number of pillars which can be employed. Table 3.2 illustrates the area occupied by a pillar consisting of 170 wires (128-bit bus + 3x14 control wires required in a 4-layer 3D SoC) for different via pitch sizes. In Face-To-Back 3D implementations, the pillars must pass through the active device layer [74], implying that the area occupied by the pillar translates into wasted device area. This is the reason why the number of inter-layer connections must be kept to a minimum. However, as via density increases, the area occupied by the pillars becomes smaller, and, at the state-of-the-art via pitch of 0.2 μm, becomes negligible compared to the area occupied by the NoC router (see Table 3.1 and Table 3.2). However, as previously mentioned, via densities are still limited by via pad sizes, which are not scaling as fast as the actual via sizes. As shown in Table 3.2, even at a pitch of 5 μm, a pillar induces an area overhead of around 4% to the generic 5-port NoC router, which is not overwhelming. These results indicate that, for the purposes of our 3D architecture, adding extra dTDMA bus pillars is feasible.

Via density, however, is not the only factor limiting the number of pillars. Router complexity also plays a key role. As previously mentioned, adding an extra vertical link (dTDMA bus) to an NoC router will increase the number of ports from 5 to 6, and since contention probability within each router is directly proportional to the number of competing ports, an increase in the number of ports increases the contention probability. This, in turn, will increase congestion

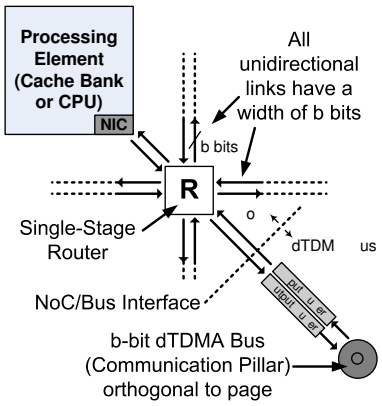


Fig. 3.4. A high-level overview of the modified router of the pillar nodes.

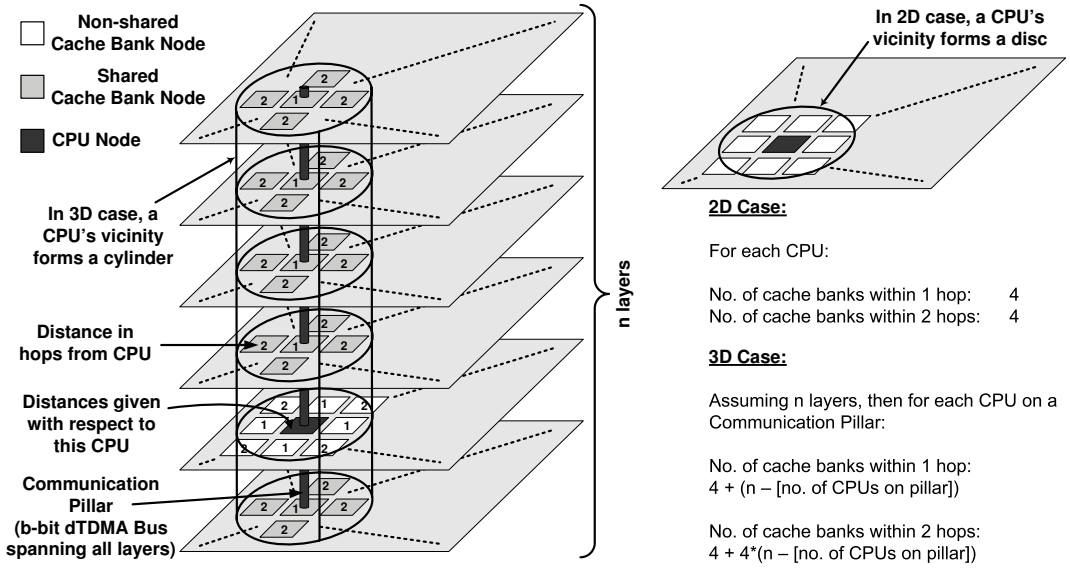


Fig. 3.5. A CPU has more cache banks in its vicinity in the 3D architecture.

within the router, since more flits will be arbitrating for access to the router's crossbar. Thus, arbitrarily adding vertical pillars to the NoC routers adversely affects the performance of each pillar router. Hence, the number of high-contention routers (pillar routers) in the network increases, thereby increasing the latency of both intra-layer and inter-layer communication. On the other hand, there is a minimum acceptable number of pillars. In this work, we place each CPU on its own pillar.

3.2.2.2 NoC Router Architecture

A generic NoC router consists of four major components: the routing unit (RT), the virtual channel allocation unit (VA), the switch allocation unit (SA), and the crossbar (XBAR). In the mesh topology, each router has five physical channels (PC): North, South, East, and West, and one for the connection with the local processing element (CPU or cache bank). Each physical unit has a number of virtual channels (VC) associated with it. These are first-in-first-out (FIFO) buffers which hold flits from different pending messages. In our implementation, we used 3 VCs per PC, each 1 message deep. Each message was chosen to be 4 flits long. The width of the router links was chosen to be 128 bits. Consequently, a 64B cache line can fit in a packet (i.e., 4 flits/packet x 128 bits/flit = 512 bits/packet = 64 B/packet).

The most basic router implementations are 4-stage ones, i.e., they require a clock cycle for each component within the router. In our L2 architecture, low network latency is of utmost importance, thereby necessitating a faster router. Lower-latency router architectures have been proposed which parallelize the RT, VA and SA using a method known as speculative allocation [72]. This method predicts the winner of the VA stage and performs SA based on that. Moreover, a method known as look-ahead routing can also be used to perform routing one step ahead

(perform the routing of node i+1 at node i). These two modifications can significantly improve the performance of the router. Two stage, and even single-stage [65], routers are now possible which parallelize the various stages of operation. In our proposed architecture, we use a single-stage router to minimize latency.

Routers connected to pillar nodes are different, as an interface between the dTDMA pillar and the NoC router must be provided to enable seamless integration of the vertical links with the 2D network within the layers. The modified router is shown in Figure 3.4. An extra physical channel (PC) is added to the router, which corresponds to the vertical link. The extra PC has its own dedicated buffers, and is indistinguishable from the other links to the router operation. The router only sees an additional physical channel.

3.2.2.3 CPU Placement

The dTDMA pillars provide rapid communication between layers of the chip. We have a dedicated pillar associated with each processor to provide fast inter-layer access, as shown in Figure 3.2. Such a configuration gives each processor instant access to the pillar, additionally providing them with rapid access to all cache banks that are adjacent to the pillar. By placing each processor directly on a pillar, its memory locality (the number of banks with low access latency) is increased in the vertical direction (potentially both above and below), in addition to the pre-existing locality in the 2D plane. This is illustrated in Figure 3.5. Such an increase in the number of cache banks with low access latency can significantly improve the performance of applications. The relative sizing of L2 cache banks and CPU+L1 cache, as shown in Figure 3.5, is meant to be illustrative. Our placement approach works even when a CPU+L1 cache span the size of multiple L2 cache banks.

Stacking CPUs directly on top of each other would give rise to thermal issues. Increased temperatures due to layer stacking is a major challenge in 3D design [12], and is often a major determining factor in component placement. Since the CPUs are expected to consume the overwhelming majority of power (they are constantly active, unlike the cache banks), it would be thermally-unwise to stack any two or more processors in the same vertical plane. Furthermore, stacking processors directly on top of each other on the same pillar would affect the performance of the network as well, as it would create high congestion on the pillar. Processors are the elements which generate most of the L2 traffic (there is also some traffic generated by the migration algorithm, as explained in Section 3.2.3); therefore, forcing them to share a single link would create excessive traffic. Our simulations in later sections will validate this argument. To avoid thermal and congestion problems, CPUs can be offset in all three dimensions (maximal offsetting), as shown in Figure 3.6.

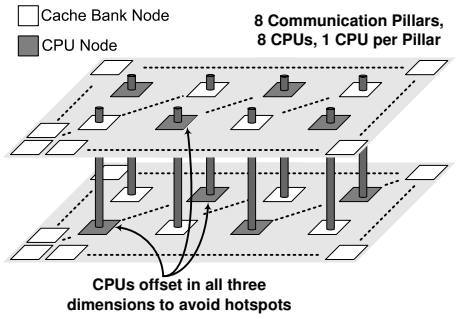


Fig. 3.6. Hotspots can be avoided by offsetting CPUs in all three dimensions.

3.2.3 Cache Management Policies

In this section, we present our organization of processor and L2 cache banks, and then detail our L2 cache management policies.

3.2.3.1 Processor and L2 Cache Organization

Figure 3.7 illustrates the organization of the processors and L2 caches in our design. Similar to CMP-DNUCA [8], we separate cache banks into multiple clusters. Each cluster contains a set of cache banks and a separate tag array for all the cache lines within the cluster. Some clusters have processors placed in the middle of them, while others do not. All the banks in a cluster are connected through a network-on-chip for data communication, while the tag array has a direct connection to the local processor in the cluster. Note that even though it is not explicitly shown in Figure 3.7, each processor has its own private L1 cache and an associated tag array for L2 cache banks within its local cluster. For a cluster without a local processor, the tag array is connected to a customized logic block which is responsible for receiving a cache line request, searching the tag array and forwarding the request to the target cache bank. This organization of processors and caches can be scaled by changing the size and/or number of the clusters.

3.2.3.2 Cache Management Policies

Based on the organization of processors and caches given in the previous subsection, we developed our cache management policies, consisting of a cache line search policy, a cache placement and replacement policy, and a cache line migration policy, all of which are detailed as follows.

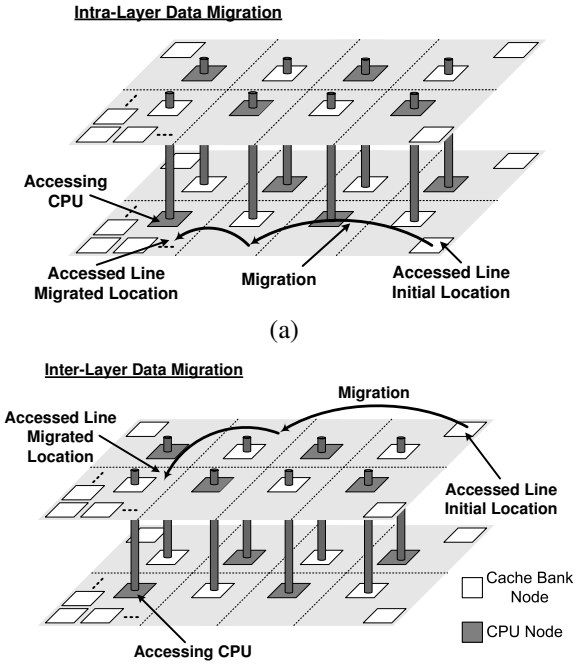


Fig. 3.7. Intra-layer and inter-layer data migration in the 3D L2 architecture. Dotted lines denote clusters.

Our cache line search strategy is a two-step process. In the first step, the processor searches the local tag array in the cluster to which it belongs and also sends requests to search the tag array of its neighboring clusters. All the vertically neighboring clusters receive the tag that is broadcast through the pillar. If the cache line is not found in either of these places, then the processor multicasts the requests to the remaining clusters. If the tag match fails in all the clusters, then it is considered as an L2 miss. On a tag match in any of the clusters, the corresponding data is routed to the requesting processor through the network on chip.

We use cache placement and replacement policies similar to those of CMP-DNUCA [8]. Initially a cache line is placed according to the low-order bits of its cache tag, that is, these bits determine the cluster in which the cache line will be placed initially. The low-order bits of the cache index indicate the bank in the cluster into which the cache line will be placed. The remaining bits of the cache index determine the location in the cache bank. The tag entry of the cluster is also updated when the cache line is placed. The placement policy can only be used to determine the initial location of a cache line as when cache lines start migrating, the lower order bits of the cache tag can no longer indicate the cluster location. Finally, we use a pseudo-LRU replacement policy to evict a cache line to service a cache miss.

Similar to prior approaches, our strategy attempts to migrate data closer to the accessing processor. However, our policy is tailored to the 3D architecture and migrations are treated differently based on whether the accessed data lies in the same or different layer as the accessing processor. For data located within the same layer, the data is migrated gradually to a cluster closer to the accessing processor. When moving the cache lines to a closer cluster, we skip clusters that have processors (other than the accessing processor) placed in them since we do not want to affect their local L2 access patterns and get the cache lines to the next closest cluster

without a processor. Eventually, if the data is accessed repeatedly by only a single processor, it migrates to the local cluster of the processor. Figure 3.7(a) illustrates this intra-layer data migration.

For data located in a different layer, the data is migrated gradually closer to the pillar closest to the accessing processor (see Figure 3.7(b)). Since clusters accessible through the vertical pillar communications are considered to be in local vicinity, we never migrate the data across the layers. This decision has the benefit of reducing the frequency of cache line migrations, which in turn reduces power consumption.

To avoid false misses (misses caused by searches for data in the process of migration), we employ a lazy migration mechanism as in CMP-DNUCA [8].

3.2.4 Experiments

We simulated the 3D CMP architecture by using Simics [61] interfaced with a 3D NoC simulator. A full-system simulation of an 8-processor CMP architecture running Solaris 9 was performed. Each processor uses in-order issue and executes the SPARC ISA. The processors have private L1 caches and share a large L2 cache. The default configuration parameters for processors, memories and Network-in-Memory are given in Table 3.3. Some of the parameters in this table are modified for studying different configurations. The shown cache bank and tag array access latencies are extracted using Cacti 3.2 [85].

To model the latency of the three-dimensional, hybrid NoC/bus interconnect, we developed a cycle-accurate simulator in C, based on an existing 2D NoC simulator [51]. For this work, the 2D simulator was extended to three dimensions, and the dTDMA bus was integrated as the

vertical communication channel. The 3D NoC simulator produces, as output, the communication latency for cache access.

In our cache model, private L1 caches of different processors are maintained coherent by implementing a distributed directory-based protocol. Each processor has a directory tracking the states of the cache lines within its L1 cache. L1 access events (such as read misses) cause state transitions and updates to directories, based on the MSI protocol. The traffic due to L1 cache coherence is taken into account in our simulation.

We simulated nine SPEC OMP benchmarks [92] with our simulation platform. These benchmarks are listed in Table 3.4. For each benchmark, we marked an initialization phase in the source code. The cache model is not simulated until this initialization completes. This is reflected as the fastforward cycles for each benchmark shown in the second row of Table 3.4. After that, each application runs 500 million cycles for warming up the L2 caches. We then collected statistics for the next 2 billion cycles following the cache warm-up period. The third row in Table 3.4 gives the total number of L2 cache accesses (including data read, data write, and instruction fetch) within the sampling period for each benchmark. We see that the benchmarks mgrid, swim and wupwise exhibit many more L2 accesses than the others, as a result of higher L1 miss rates.

We first introduce the schemes compared in our experiments. We refer to the scheme with perfect search from [8] as CMP-DNUCA. We name our 2D and 3D schemes as CMP-DNUCA-2D and CMP-DNUCA-3D, respectively. Note that our 2D scheme is just a special case of our 3D scheme discussed in the paper, with a single layer. Both of these schemes employ cache line migration. To isolate the benefits due to 3D technology, we also implemented our 3D scheme without cache line migration, which is called CMP-SNUCA-3D.

Table 3.3. Default system configuration parameters (L2 cache is organized as 16 clusters of size 16x64KB).

Processor Parameters	
Number of Processors	8
Issue Width	1
Memory Parameters	
L1 (split I/D)	64KB, 2-way, 64B line, 3-cycle, write-through
L2 (unified)	16MB (256x64KB), 16-way, 64B line, 5-cycle bank access
Tag Array (per cluster)	24KB, 4-cycle access
Memory	4GB, 260 cycle latency
Network Parameters	
Number of Layers	2
Number of Pillars	8
Routing Scheme	Dimension-Order
Switching Scheme	Wormhole
Flit Size	128 bits
Router Latency	1 cycle

Table 3.4. Our benchmarks.

Benchmarks	Fastforward (million cycles)	L2 Transactions
ammp	3,633	24,508,715
apsi	4,453	27,013,447
art	3,523	25,638,435
equake	21,538	27,502,906
fma3d	18,535	12,599,496
galgel	3,665	38,181,613
mgrid	3,533	204,815,737
swim	4,306	164,762,040
wupwise	18,777	141,499,738

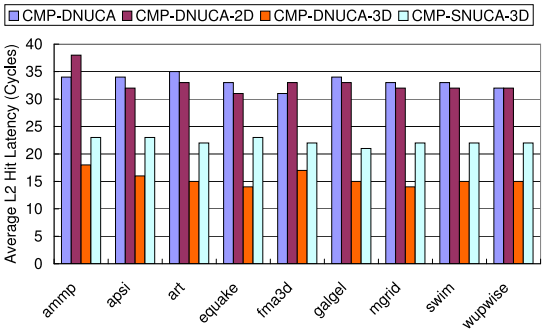


Fig. 3.8. Average L2 hit latency values under different schemes.

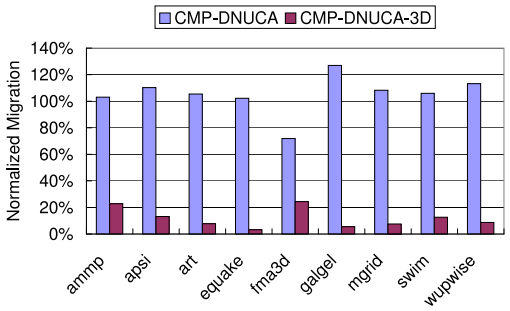


Fig. 3.9. Number of block migrations for CMP-DNUCA and CMP-DNUCA-3D, normalized with respect to CMP-DNUCA-2D.

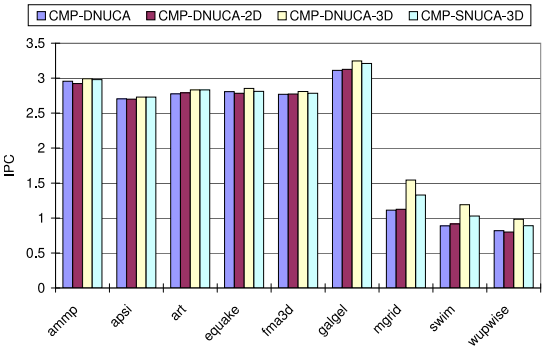


Fig. 3.10. IPC values under different schemes.

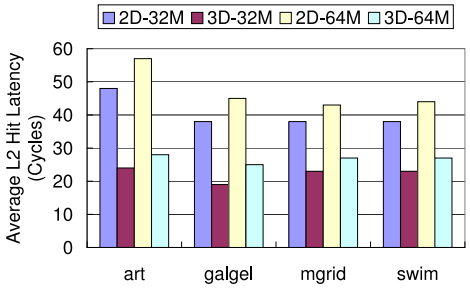


Fig. 3.11. Average L2 hit latency values under different schemes.

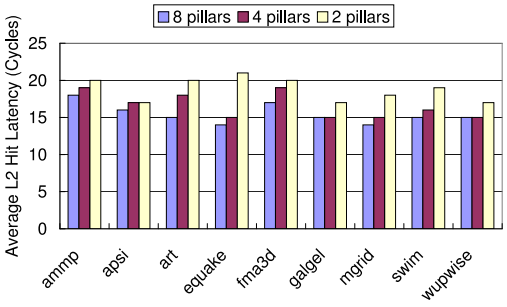


Fig. 3.12. Impact of the number of pillars (the CMP-DNUCA-3D scheme).

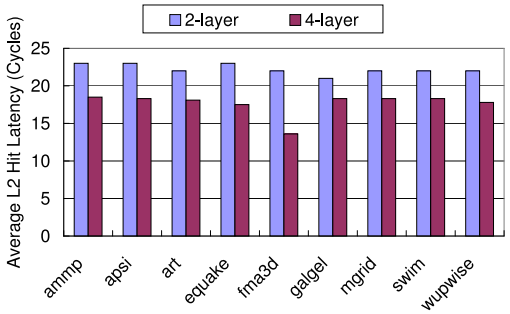


Fig. 3.13. Impact of the number of layers (the CMP-SNUCA-3D scheme).

Our first set of results give the average L2 hit latency numbers under different schemes. The results are presented in Figure 3.8. We observe that our 2D scheme (CMP-DNUCA-2D) generates competitive results with the prior 2D approach (CMP-DNUCA [8]). Our 2D scheme shows slightly better IPC results for several benchmarks because we place processors not on the edges of the chip, as in CMP-DNUCA, but instead surround them with cache banks as shown in Figure 3.7. Our results with 3D schemes reiterate the expected benefits from the increase in locality. It is interesting to note that CMP-SNUCA-3D, which does not employ migration, still outperforms the 2D schemes that employ migration. On the average, L2 cache latency reduces by 10 cycles when we move from CMP-DNUCA-2D to CMP-SNUCA-3D. Further gains are also possible in the 3D topology using data migration. Specifically, CMP-DNUCA-3D reduces average L2 latency by 7 cycles as compared to the static 3D scheme. Further, we note that even when employing migration, as shown in Figure 3.9, 3D exercises it much less frequently compared to 2D, due to the increased locality (see Figure 3.5). The reduced number of migrations in turn reduces the traffic on the network and the power consumption. These L2 latency savings translate to IPC improvements commensurate with the number of L2 accesses. Figure 3.10 illustrates that the IPC improvements brought by CMP-DNUCA-3D (CMP-SNUCA-3D) over our 2D scheme are up to 37.1% (18.0%). The IPC improvements are higher with mgrid, swim and wupwise since these applications exhibit higher number of L2 accesses.

We next study the impact of larger cache sizes on our savings using CMP-DNUCA-2D and CMP-DNUCA-3D. When we increase the size of the L2 cache, we increase the size of each cluster, while maintaining the 16-way associativity. Figure 3.11 shows the average L2 latency results with 32MB and 64MB L2 caches for four representative benchmarks (art and galgel with low L1 miss rates and mgrid and swim with high L1 miss rates). We observe that L2 latencies

increase with the large cache sizes albeit at a slower rate with the 3D configuration (on average 7 cycles for 2D versus 5 cycles for 3D), indicating that 3D topology is a more scalable option when we move to larger L2 sizes.

Next we make experiments by modifying some of the parameters in the underlying 3D topology. The results with the CMP-DNUCA-3D scheme using different numbers of pillars to capture the effect of the different inter-layer via pitches are given in Figure 3.12. As the number of pillars reduces, the contention for the shared resource (pillar) increases to service inter-layer communications. Consequently, average L2 latency increases by 1 to 7 cycles when we move from 8 to 2 pillars. Also, when the number of layers increases from 2 to 4, the L2 latency decreases by 3 to 8 cycles, primarily due to the reduced distances in accessing data, as illustrated in Figure 3.13 for the CMP-SNUCA-3D scheme.

3.3 Migration-based NUCA design

3.3.1 Motivation

3.3.1.1 Shared L2 Access Pattern

To characterize the usage characteristics of a large on-chip shared L2 cache, we simulate the SPECOMP benchmarks [92] targeting a CMP platform with 8 processors. Each processor is modeled as a simple in-order architecture executing the SPARC ISA. There are separate private L1 data/instruction caches associated with each processor. All 8 processors share one large on-chip L2 cache. Section 3.3.4 provides a more detailed description of our major simulation parameters and experimental setup.

Figure 3.14 presents the distribution of the private and shared L2 cache lines. If a cache line is only accessed by one processor during its lifetime (from its fetch from the off-chip memory to its eviction from the on-chip L2), we call it a *private* cache line; otherwise, it is tagged as *shared*. A shared cache line may have a different number of requestors, which we classified into four categories as illustrated in Figure 3.14. One can observe from this graph that, for all our benchmarks, the percentage of private cache lines dominates, which is within the range 66.6%-99.9%, averaging in 92.2%. Besides the cache line distribution, we also record the cache line *access* distribution, and the results are given in Figure 3.15. We see that, although the private cache lines dominate the requested lines (see Figure 3.14), most of the line accesses are to shared ones (see Figure 3.15); that is, shared lines are used more frequently than the private ones. When averaged across our benchmark codes, 62.1% of the L2 accesses are to the shared cache lines. We also need to mention that the trends captured by these two graphs are consistent with the observations made by previous research [59, 7]. To summarize, shared lines play an important role in determining the access patterns exhibited by the L2 cache. Consequently, their placement within the L2 space (in a NUCA L2 architecture) can be an important factor in determining overall performance.

One approach along this direction is to try to replicate the shared cache lines around the vicinity of the requesting processors so that the access latency to frequently accessed shared cache lines can be reduced [7, 106]. However, maintaining multiple replicas inside the L2 space requires the implementation of an L2 cache coherence scheme (to organize parallel accesses to shared data), which complicates the cache design and impacts the performance/power characteristics of the L2 cache. Note that, for a NUCA-based L2 organization, which keeps only one

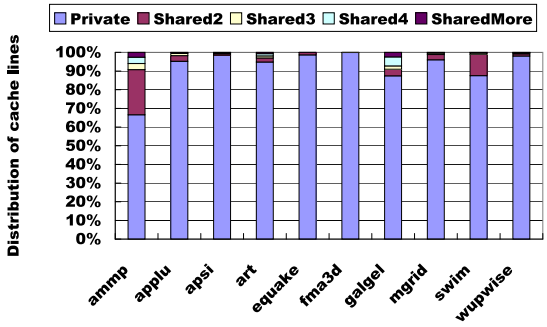


Fig. 3.14. Distribution of different types of L2 cache lines (Private: private L2 cache lines; Shared$N$: L2 cache lines shared by N processors).

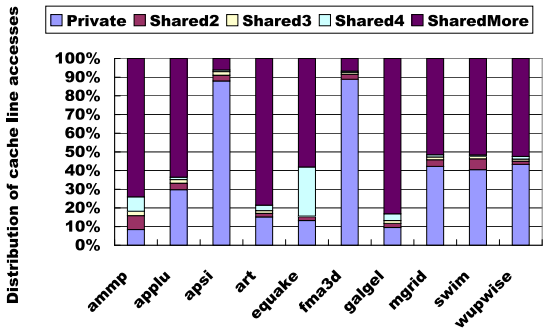


Fig. 3.15. Distribution of different types of L2 cache accesses (Private: accesses to private L2 cache lines; Shared$N$: accesses to L2 cache lines, which are shared by N processors).

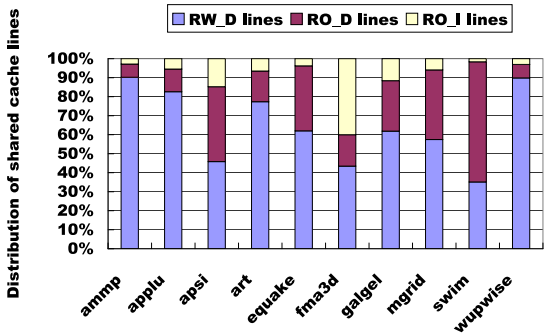


Fig. 3.16. Distribution of shared L2 cache lines (RW_D: read-write data; RO_D: read-only data; RO_I: instructions).

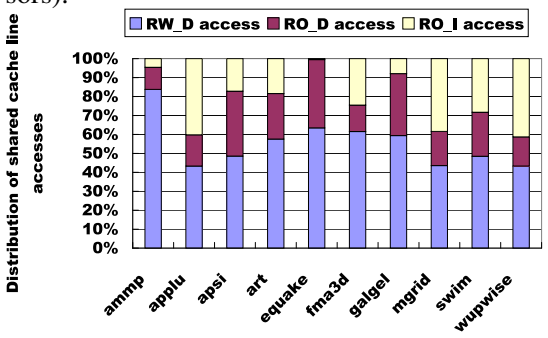


Fig. 3.17. Distribution of accesses to shared L2 cache lines (RW_D: accesses to read-write data; RO_D: accesses to read-only data; RO_I: accesses to instructions).

copy of a cache line in L2, there is no coherence issue for L2, although L1 private caches need to be kept coherent with each other.

Prior studies also proposed duplicating read-only L2 caches lines [7, 106] to avoid the L2 coherence problem based on the observation that the shared read-only accesses are a significant portion of the shared accesses. This can be observed from our collected statistics as well, as shown in Figures 3.16 and 3.17. However, such a scheme would work well only if we could identify the read-only cache lines correctly and quickly, which is not an easy task in practice. From Figures 3.16 and 3.17, we see that, although the read-only accesses to shared lines are frequent (on average 44.7%), a large percentage of them are to data rather than instructions. Without significant extra effort, it would be very hard (if not impossible) to determine whether a data cache line will remain as read-only during its entire lifetime or not. This and similar concerns motivate us for searching for alternate solutions to managing shared L2 lines.

Without duplicating shared L2 cache lines, the only remaining option is to determine an *ideal position* (location) for each shared line within the L2 space so that the overall access cost (performance/power) to this line is optimized. We assume that, for each L2 miss, we place the newly-fetched cache line close to its first (original) requestor. This placement would be suitable for a private cache line as its first requesting processor is also its only requestor. The question is whether this initial position is also appropriate for the shared cache lines. Figure 3.18 provides the percentage of the cases in which the first requestor is also the most active requestor (issuing the most requests among all requestors of that line) for all the shared cache lines. If this percentage is large, the initial position might be good enough for the shared cache lines as well. Otherwise, we need to explore different methods for deciding where the shared cache lines should be placed at any given time during execution. Unfortunately, according to Figure 3.18,

this percentage is only 52.9% on average. This means that, after a processor fetches an off-chip line (data block) to the on-chip L2 cache, with around 50% probability, there is another processor (or multiple other processors) that uses this cache line more heavily than this processor (the original requestor). Based on above observations, in the remainder of this paper, we propose and evaluate a scheme that determines the position of each cache line (either private of shared).

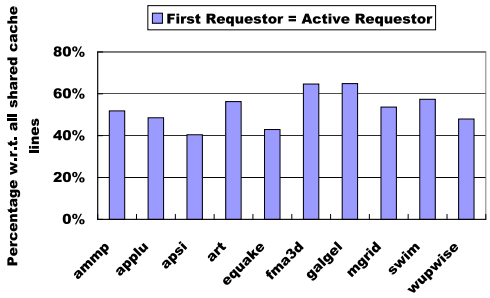


Fig. 3.18. Percentage of shared cache lines for which the first requestor is also the most active requestor.

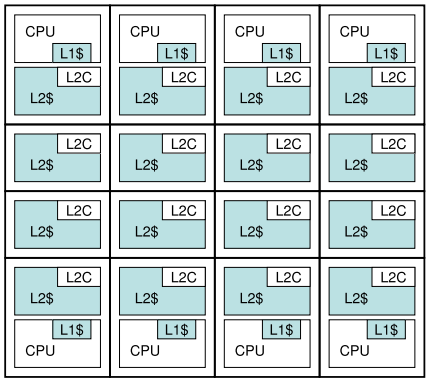


Fig. 3.19. A NUCA-based mesh CMP (L2C: L2 controller).

3.3.1.2 Motivation for Migration-based NUCAs

As originally proposed in [48], the main idea behind a NUCA organization is to distribute the multiple "ways" (cache lines) of a given cache set across the entire chip in such a way that they have different distances from the viewpoint of a given processor and thus exhibit different access latencies. Figure 3.20 illustrates this proposal in the context of a single CPU system. In this case, a large L2 cache is divided into multiple clusters based on the distance from the

processor. Usually, the cache associativity is chosen to be equal to the number of clusters so that each cache set has exactly one "way" mapped into each cluster (it is also possible to increase the cache associativity to map two or more ways into one cluster).

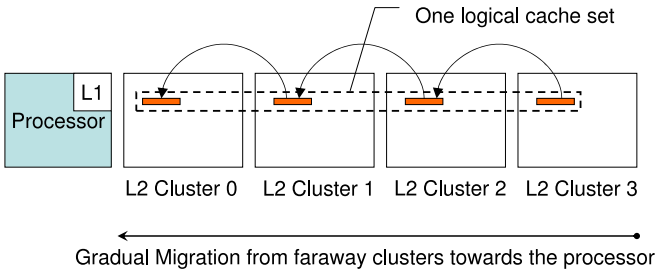


Fig. 3.20. NUCA for a uniprocessor architecture.

In order to place the frequently accessed cache lines closer to the processor, cache line migration can be used. Once a cache line is accessed, it can be migrated from its current cluster to another one, which is closer to the processor. A migration in this context essentially involves swapping two cache lines. For example, if the processor in Figure 3.20 accesses a cache line in cluster 2, it can be migrated to cluster 1, which means a cache line needs to be evicted from cluster 1. Consequently, we need to select a victim line in cluster 1. The NUCA migration policy in [48] chooses to put this evicted cache line back to cluster 2. We see that, when the number of clusters is equal to the cache associativity, such a migration scheme arranges the cache lines within the same set (across the clusters) as an LRU queue, with more recently accessed lines being placed closer to the processor. Thus, when we fetch a new cache line from the off-chip

memory, the corresponding cache line in cluster 3 (the least recently used cache line in the specific set) becomes the natural victim and can be evicted from the on-chip L2 cache.

Beckmann and Wood [8] adapt the NUCA concept to Chip Multiprocessors (CMPs). This work inherits the main idea from the NUCA concept in the single processor case. Specifically, multiple ways within the same set are still placed across the entire chip and cache line migration is employed in moving the frequently accessed cache lines towards their requesting processors. There is an important difference between the single CPU based and CMP based NUCA schemes though. In a CMP NUCA, line migration can become very tricky, as it may not always improve access latencies due to data sharing among multiple processors. When multiple processors request the same cache line and pull this cache line in different directions (in the 2D space), a ping-pong effect can occur and this can easily lead to excessive migrations. As stated in [8], this is also the main reason for false misses (a requested cache line is in migration). Lazy migration [8] is proposed to eliminate false sharing, which lets the cache line wait for 1000 cycles before really migrating it. If, during this time period, there is a request from some other processor, the pending migration is canceled. Lazy migration eliminates many shared cache line migrations and can significantly reduce the false miss rate.

We observe that there are three issues with the cache migration scheme described in [8]. First, sharing pattern among different processors may lead to excessive cache line migrations. Proposed lazy migration [8] may not be an acceptable solution, because it eliminates many migrations of shared cache lines and focuses on improving the access latencies of private cache lines. As illustrated in Section 3.3.1.1 however, the access percentage of the private cache lines is much lower than that of the shared cache lines. Second, tracking the access pattern of each cache line (the accessing processors and the elapsed cycles for triggering lazy migration) increases

the hardware implementation complexity, and this can incur significant overheads in terms of both memory requirements and power consumption. Third, [8] does not study in detail the replacement policy of NUCA for CMPs. In the case of NUCA for a single processor, the cache line at the farthest cluster is the natural victim as the migration gradually puts the least recently used cache lines in there. In the context of CMPs, however, there is no such a natural LRU queue. A pseudo-LRU replacement policy [88] may not be appropriate for CMP NUCA, as cache sets are spread across the whole L2 space. Maintaining global LRU bits for each set requires many L2 cache accesses to traverse the whole chip to update the access pattern (LRU bits). It is easy to see that this can be very costly from the performance angle. Beckmann and Wood [8] use the low order bits of the cache tag to select an initial position (a victim) for the cache line. This policy clearly gives an equal treatment to the physically-spread cache lines of a logical set. Although this seems to work reasonably well for their workloads, there is no study targeting the evaluation of the replacement policies for CMP NUCA.

The issues raised above on NUCA design for CMPs as well as the access pattern of shared L2 cache motivate us to explore a way of determining an optimal position for each L2 cache line to reduce the average memory access latency. Our proposal consists of two components: The first component includes our baseline NUCA organization and placement/replacement policies, which are explained in the next section (Section 3.3.2); the second component is an optimization scheme built upon our baseline NUCA design, targeting the frequently accessed shared cache lines, which is detailed in Section 3.3.3.

3.3.2 Migration-Based NUCA Baseline Design

Our baseline NUCA organization for CMPs builds upon the prior efforts [8, 48]. As shown in Figure 3.19, the processors and cache banks are organized in a mesh topology. We classify the L2 cache banks into multiple clusters and choose the number of the clusters such that the shared L2 cache associativity is a multiples of the number of clusters. Therefore, we map a fixed number of cache lines from a logical set to each physical cluster, making it possible for any cache line to reside in any cluster in the chip. In the architecture shown in Figure 3.19, each processor has one closest L2 cache cluster, which we call this processor's local L2 cache space. We call a processor together with its local L2 cache cluster a *processor-cache cluster*, and we refer to an L2 cache cluster not attached to any processor as the *pure cache cluster*. These two types of clusters represent two different types of mesh nodes, which are connected through a Network-on-Chip (NoC). The pure cache clusters are controlled by a local L2 controller (also referred to as *agent* in this paper). This controller contains the tag array for all the cache lines within the cluster. It is in charge of managing all the accesses to the cache lines in this cluster by sequentially inquiring the local tag array and local data banks. The sequential access speeds up the decision for an L2 hit or miss, since, at the tag access step, the local L2 controller knows whether the requested cache line is currently in its space.

The main difference between our baseline NUCA design and the previous proposals is on the cache management policy, which includes both initial placement of lines into the L2 space and replacement policies. Regarding the initial placement for a newly-fetched line, we choose to place it into the *local cache space* of the requesting processor. Since we know that most of the cache lines are private ones (see Figure 3.14), it makes sense to place a private cache line

closer to its only requestor, rather than a randomly-chosen location (and then gradually migrate it towards its requestor), as has been suggested in [8]. We expect that this careful initial placement can save most of the potential migrations of the private cache lines. And, it also eliminates the need for tracking the access pattern of each L2 cache line, as required by lazy migration.

It needs to be noted that the initial placement also involves a replacement policy, which may in turn trigger *cascade migrations*. Within a processor's local cache space, there are only a limited number of possible locations (= L2 cache associativity / number of L2 cache clusters) for a cache line. Placing a cache line in a cache cluster requires the eviction of a local cache line from these allowable locations. There are at least two questions to be answered to solve this replacement problem. (1) How are we going to determine such a victim? and (2) Are we going to evict this victim from the entire on-chip cache space?

Compared to an equal treatment of the physically-scattered lines in a logical set and random selection of a victim from among them, we maintain LRU bits for each subset[1] in the local cache cluster (of course, only when the subset size is larger than 1). Thus, we can employ the conventional pseudo-LRU replacement policy [88] to select a local victim. Therefore, for the victim determination, the local cache cluster works like a private cache, as it does not consider all the remaining remote lines in the same logical set as potential victim candidates.

The next question to address is whether we directly evict such a victim to the off chip memory? If the answer is yes, the CMP NUCA behaves like multiple private L2 caches, as, for each processor, its private data can only reside within its local L2 space (or off-chip). This clearly reduces the utilization of the aggregate L2 space, e.g., a processor with a large working set cannot

[1] We define a cache subset as a part of a cache set, which consists of the local cache lines belonging to the same logical set.

make use of the cache capacity of a neighboring node, which happens to run a less memory-intensive process. Therefore, we need to design a scheme to handle such victims carefully. Our proposal is to determine the location of the victim based on the *L2 miss intensities*, i.e., how many L2 misses occur during a unit time, of both the local processor and other processors, as we believe that the metric of L2 miss intensity of a processor represents the degree of the memory access intensity of its current running thread. Based on the L2 cache size, the number of processors, and the characteristics of the workloads, we determine a threshold for the L2 miss intensity, T. When the L2 miss intensity of a processor is higher than T, we set its cluster priority as the highest one (which is 2). If the L2 miss intensity is in the range of $[0, T)$, the priority level is set to 1. Since we have several pure cache clusters without an attached processor, we propagate the priorities of these processor-cache clusters to the remaining pure cache clusters. We use a waveform-like algorithm to set a priority for each cluster; the pseudo-code of this algorithm is given in Figure 3.21. Figure 3.23 illustrates an example priority setting. Once we know the priorities of the processor-cache clusters (the top row and the bottom row according to Figure 3.19), we can propagate them and set a priority for each pure cache cluster, in the hope of specifying whether it can accommodate a victim from another cluster.

The combination of the local and remote cluster priorities determines the migration target for a victim cache line. The pseudo-code for our migration policy is given in Figure 3.22. The idea is that, if the cluster evicting a cache line has higher priority, it has higher ability of keeping its victim on chip by checking more clusters' status and asking one of them to accommodate this victim if possible. If the evicting cluster has the lowest priority 0, its victim will be evicted directly from the entire on-chip cache (accompanied with a possible write back to the off-chip memory if it has been modified while residing in L2).

```
clear priority, flags set and sent, setClusters
if (there exists a local processor p_j) {
   set priority according to p_j.L2miss;
   set = 1;
   broadcast(set);
}
repeat {
   msg = receive();
   if (msg = a broadcast set message) {
      setClusters++;
   }
   if (msg = neighborPriority) {
      if (set = 0) {
         priority = neighborPriority > 0 ?
                  neighborPriority - 1 : 0;
         broadcast(set);
      }elsif (priority < neighborPriority - 1) {
         priority = neighborPriority - 1;
      }
   }
   if ( set = 1 and sent = 0 ) {
      send priority to all neighbors;
      sent = 1;
   }
} until (setClusters = n)
```

Fig. 3.21. A distributed algorithm for updating the cluster priority (n is the total number of clusters).

```
Input: Cluster C_s that throws the victim
Output: Destination cluster C_d for the victim
if (C_s.priority = 0) {
   C_d = NULL; // evict to off-chip
} elsif(C_s.priority = 1) {
   if (∃C_i, d(C_i - C_s) = 1 ∧ C_i.priority = 0) {
      C_d = C_i;
   } elsif(∃C_i, C_i.priority = 0) {
      C_d = C_i;
   } else {
      C_d = NULL; // evict to off-chip
   }
} elsif(C_s.priority = 2) {
   if (∃C_i, d(C_i - C_s) = 1 ∧ C_i.priority = 0 ) {
      C_d = C_i;
   } elsif(∃C_i, d(C_i - C_s) = 1 ∧ C_i.priority = 1) {
      C_d = C_i;
   } elsif(∃C_i, C_i.priority = 0) {
      C_d = C_i;
   } elsif (∃C_i, C_i.priority = 1) {
      C_d = C_i;
   } else {
      C_d = NULL; // evict to off-chip
   }
}
```

Fig. 3.22. Migration policy implementation for a victim cache line (function d($C_i - C_j$) gives the Manhattan Distance between C_i and C_j).

Maintaining a priority for each cluster is not expected to incur excessive overhead. We only need to reserve a two-bit flag for each cluster to store its priority level. The updates to this flag are triggered by the local L2 cache miss intensity monitor for a processor, which can be built using the performance counters available today in many modern CPUs [36]. We set a large monitoring time window to avoid frequently invocation of the whole-chip priority setting algorithm shown in Figure 3.21.

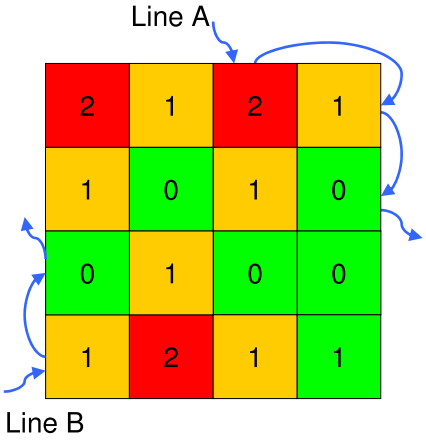


Fig. 3.23. An example of setting the cluster priority levels and determining the eviction-based cache line migration according to the algorithm in Figure 3.22.

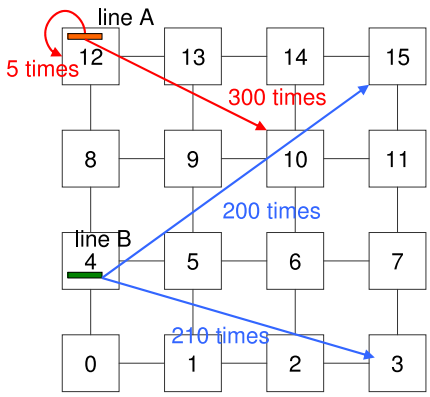


Fig. 3.24. Motivating examples for determining suitable locations for the hot shared cache lines.

How to locate a cache line in the large shared on-chip L2 cache is an important issue. Since a cache line can reside in any cluster, we need a multi-step checking scheme to first check the local and neighboring clusters and then send requests (if necessary) to remote clusters until we determine whether we have an L2 cache hit or miss. With our initial placement policy and

replacement scheme (triggered by the initial placement) presented above, we know that most private cache lines are close to their requestors and the requests for such lines can be satisfied quickly. Consequently, the average access latency for the private cache lines is expected to be reasonably small. The issue, however, is more problematic with the shared lines, as their initial locations (inside the first requestor's local cache cluster) might be far away from other dominating requestors (as discussed in Section 3.3.1.1). The problem is critical as we know that accesses to shared lines dominate private accesses (see Figure 3.15). Aiming at reducing the average access latencies to the shared cache lines, we next investigate how to determine an appropriate position for the hot shared cache lines.

3.3.3 Migration Algorithm for Shared Cache Lines

3.3.3.1 Problem Formulation

Proper management of the frequently-accessed shared cache lines is critical in extracting good performance from large shared L2 caches. Without duplicating hot shared L2 cache lines, there is only one option left to optimize the accesses to them, which is to place such hot lines into ideal positions within the cache space. As shown in Figure 3.24, if a shared cache line A residing in cluster 12 is also accessed by the processor inside cluster 10 (we call it P10) and P10 exhibits a much higher access frequency compared to the first requestor, P12, e.g., 300 vs. 5, we may want to migrate cache line A to cluster 10 to reduce average access latency to that cache line in the future. In another scenario, if a cache line B in cluster 4 is also accessed by both P3 and P15 with comparable access frequencies, it would be beneficial to move the line into cluster 7, an acceptable position for both the requestors.

For a given shared cache line, its average access latency can be estimated as:

$$lat = \sum_{i \in R} c * f_i * d(i,j) / \sum_{i \in R} f_i,$$

where j is the ID of the cache cluster where this cache line currently resides, i is the ID of the requesting cluster (which belongs to a set, R), and f_i is the number of requests to this cache line issued by cluster i. Function $d(i,j)$ gives the Manhattan Distance between requestor i and cluster j, and c represents the per hop access latency.

Although this equation does not consider the effect of network contention (since c is assumed to be constant) and could not give a very accurate average access latency value, it can be used as a first-degree optimization metric for the shared cache lines. Note that, if we can reduce the value of this metric, the real average access latencies will also be decreased most of the time.

Migrating a shared cache line to an appropriate position has the potential of reducing the L2 access latencies. This means, if the cache line owner cluster is changed from j to j', the average access latency for this cache line becomes:

$$lat' = \sum_{i \in R} c * f_i * d(i,j') / \sum_{i \in R} f_i,$$

under the assumption of no contention in the network.

Note that lat' might be higher or lower than the original lat. Thus, our goal is to find a proper target cluster j' for this cache line so that lat' is minimized. Since the constant c and the total access requests $\sum_{i \in R} f_i$ do not change with different j's, we can remove these two terms

and obtain:

$$Cost = \sum_{i \in R} f_i * d(i, j').$$

Our optimization goal then becomes one of determining a cluster j' so that $Cost$ is minimized. It turns out that this problem is a *post-office location* problem: there are n input points p_1, p_2, ..., p_n with associated weights w_1, w_2, ..., w_n and we need to find a point p (not necessarily one of the input points) that minimizes the sum $\sum_{i=1}^{n} w_i d(p, p_i)$, where $d(p, p_i)$ is the distance between points p and p_i. According to [25], for the one-dimensional post-office location problem, that is, the n distinct points are at x_1, x_2, ..., x_n, and their weights are w_1, w_2, ..., w_n, respectively, the optimal point p is the *weighted median* of these n numbers (weights). The important property of weighted median, x_k, is that x_k satisfies both $\sum_{x_i < x_k} w_i < \sum_{i=1}^{n} w_i/2$ and $\sum_{x_i > x_k} w_i \leq \sum_{i=1}^{n} w_i/2$.

Our optimization problem corresponds to the two-dimensional post office problem. Let us assume that there are n requestors, from p_1 to p_n, each of which can be represented by its X and Y coordinates, that is, node p_i is represented by (x_i, y_i). Each node p_i has an associated weight w_i, which is equal to the number of access requests issued by it, f_i. Our goal is to find a node $p_{j'}$, which minimizes $Cost = \sum_{i=1}^{n} f_i * d(i, j')$. Since our CMP architecture is based on a two-dimensional mesh topology, the distance between two nodes i and j' is the Manhattan Distance, i.e., $d(p_i, p_{j'}) = |x_i - x_{j'}| + |y_i - y_{j'}|$. Therefore, we can express our optimization target as:

$$Cost = \sum_{i=1}^{n} f_i * |x_i - x_{j'}| + \sum_{i=1}^{n} f_i * |y_i - y_{j'}|.$$

Both parts on the right hand side of this equation are non-negative. Thus, we can determine the location of migration target j' by searching the weighted medians at the X and Y directions

separately and independently. The two weighted medians, x and y, constitute the coordinates of the final migration target. For each direction, a simple linear algorithm given in Figure 3.25 is employed to obtain the weighted median in one dimension, as we know the exact locations of all the nodes in a mesh. Note that, since the searching for weighted medians is done separately and the X and Y coordinates are independent, the obtained migration target may not necessarily be one of the requestors of the line in question. For example, if there are three requestors (3,4), (4,3) and (8,8) and their access frequencies are the same, the algorithm in Figure 3.25 determines the final target as (4,4), which is not one of the three requestors.

3.3.3.2 Hardware Implementation

We know that the weighted medians of both coordinates (X and Y) together determine a *theoretically* ideal position for the shared cache line. Implementing it directly in a real NUCA L2 cache hardware however would involve saving all the numbers of accesses (to the cache line in question) from the different requestors and performing a long sequence of calculations. This would typically require considerable storage and incur significant computational overhead.

Since the percentage of shared cache lines is small (see Figure 3.14), we propose to utilize the available cache space to store their access patterns. As shown in the upper part of Figure 3.26, we extend each cache line of L2 by adding two 1-bit flags: Pattern and Shared, and a 4-bit migration limit. If Pattern flag is set to 1, this means that the current cache line does not store any data, but is used to represent the access patterns of a shared cache line. The location of a shared cache line's pattern line is determined by $(index + set\_num/2)\%set\_num$. As an example, let us consider Figure 3.27. If there are 16 sets in the cache and there is a shared line at

index 2, its access pattern line will reside at index 10. However, it can be any line in the subset with index 10, which will be determined by the replacement policy of the local cluster.

Input: The number of clusters per row (X direction) m, access array $f[0 \ldots m-1]$, the overall access number F
Output: X coordinate X_d of the migration target cluster
$half = F / 2$;
$w = 0$;
for ($i = 0$ to $m-1$) {
  if ($w < half$ and $w + f[i] \geq half$) {
    X_d = i;
    break;
  }else {
    $w = w + f[i]$;
  }
}

Fig. 3.25. The algorithm for computing the X coordinate of the target cluster for a shared cache line. The Y coordinate can be computed similarly.

When a cache line is first fetched, its Pattern and Shared flags are both set to zeros, indicating that it is a private line. When this cache line is later accessed by a remote requestor, its Shared flag is changed to 1, indicating that it has become a shared cache line (Shared flag will remain as 1 until the replacement). At the same time, we choose a cache line at the position $(index + set\_num/2)\%set\_num$ inside the same cluster for allocating an access pattern line for this shared line.[2] The selected line will have its Pattern flag set to 1 and replace its tag with the tag of the corresponding shared cache line. The data portion of this selected pattern line is used

[2]Clearly, allocating a pattern line may result in a victim, which is handled in the same way as that for the victim when placing a new cache line.

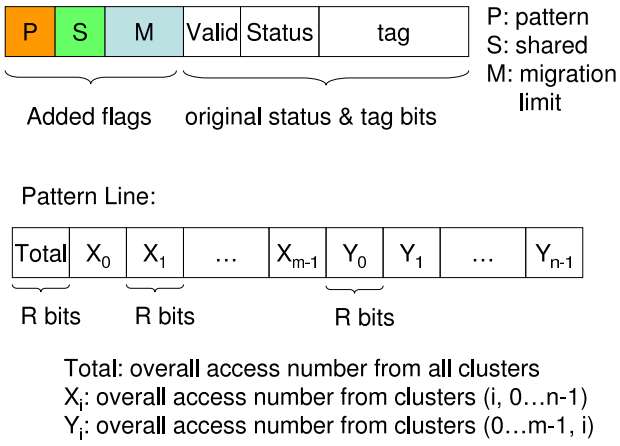


Fig. 3.26. Hardware support for tracking the access patterns of shared cache lines.

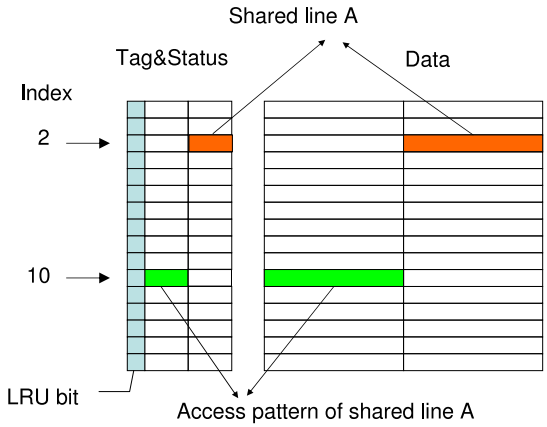


Fig. 3.27. Allocating a pattern line for a shared cache line.

for storing the access patterns. Note that both the shared cache line and its pattern line belong to a local cluster and can be evicted by the local replacement policy. We also enforce an additional rule, which states that when a shared cache line is evicted, its pattern line must be invalidated to avoid a dangling pattern line. A pattern line however can be evicted independently.

So far we discussed where we store the access patterns for the shared cache lines. There are still two questions to address: (1) How can we track the access patterns such that the calculations of migration targets are simplified? and, (2) How should the migration of a shared cache line be triggered?

Based on Figure 3.25, we need to track the number of accesses made from each cluster. Consequently, we maintain the access pattern as shown in Figure 3.26. The widths for all segments are the same, i.e., R bits. The maximum value of R we can maintain is $LineSize/(1 + m + n)$, where m/n are the row/column sizes for an $m \times n$ mesh. For example, in our default configuration, the cache line size is 64-byte and the mesh is 4x4, which means that we can reserve a maximum of 7 bytes for each segment (sufficient for tracking the number of accesses). Every time when a processor P_i requests this cache line, the three corresponding segments, "Total", "X_i", "Y_i", are increased by 1. This format of access pattern is amicable for applying the algorithm in Figure 3.25. Once the value of segment "Total" reaches a preset "Hot" threshold, the computation for migration target is triggered. If the calculated migration target is not the current host cluster, this cache line will be migrated.

A shared cache line can be accessed very frequently. Therefore, keeping track of its access pattern during its entire lifetime would cause significant overheads in terms of counter sizes and power consumption. We propose an adaptive tracking and migrating scheme for the hot shared cache lines. The idea is that we use a *migration limit* for the shared cache lines.

Every time when the computation for migration target is triggered, its migration limit number is reduced by one whether it is practically migrated or not (after the computation, all the access numbers in the pattern lines are cleared to zeros). When this number becomes zero, we disable the further access pattern tracking and any future migration for that line. When a shared cache line is migrated, its access pattern line in the local cache cluster is invalidated. If it does not reach its migration limit, a new access pattern line is allocated in its migration target. By adjusting this preset migration limit, we can capture the access patterns of shared cache lines during their lifetimes without incurring significant overhead.

3.3.4 Experiments

We built our simulation platform on top of Simics [61], which is a commercial full-system simulator. An 8-processor CMP architecture with private L1 caches and a shared L2 NUCA (shown in Figure 3.19) is simulated. The default configuration parameters are listed in Table 3.5, though we also perform several sensitivity studies by varying the default values of some of our simulation parameters. The proposed migration-based NUCA design is implemented as an extended cache module in Simics.

We tested our approach using ten SPECOMP benchmarks [92]. For each benchmark code, we simulated 3 billion cycles. The important statistics for these benchmarks are shown in Table 3.6.

To compare our proposal with prior studies, we also made experiments with the scheme named CMP-DNUCA proposed by [8], which we apply to our mesh based topology shown in Figure 3.19. We refer to our migration-based NUCA design as M-NUCA. The number that follows "M-NUCA" indicates the migration limit (see the last paragraph of Section 3.3.3.2).

Table 3.5. Default system configuration parameters.

Processor	
8 single-issue processors	
L1 Cache	
Split I/D, each 32KB, 2-way, 64B line, 3-cycle latency, write-back	
L2 Cache	
Unified, 16MB, 32-way, 64B line, 22-cycle per cluster latency	
Number of Clusters	16
Priority Setting Threshold	T: 350 (misses per Million cycles)
Hot Threshold	40 accesses
Main Memory	
4GB, 300-cycle latency	
Network	
Per Hop Latency	4 cycles

Table 3.6. Our benchmarks.

Benchmarks	**ammp**	**applu**	**apsi**	**art**	**equake**	**fma3d**	**galgel**	**mgrid**	**swim**	**wupwise**
L2 accesses / million cycle	7593	16202	29092	18658	12879	2607	8505	19300	28229	10447
L2 misses / million cycle	436	2413	3291	1298	893	1394	305	3582	4456	2277
L2 miss rates	5.7%	14.9%	11.3%	7.0%	6.9%	53.5%	3.6%	18.6%	15.8%	21.8%

For example, M-NUCA-0 means the baseline M-NUCA design discussed in Section 3.3.2, in which we neither track accesses to shared cache lines nor perform access-triggered migrations as described in Section 3.3.3. Note that, in M-NUCA-0, there are still migrations triggered by cache line placement/replacement, which we call eviction-triggered migrations. When the number attached to "M-NUCA" is larger than zero, the access-triggered migrations, in addition to the eviction-triggered migrations may take place; in this case, the access-triggered migrations have a migration limit for each shared cache line. For example, M-NUCA-1 means we can migrate a shared cache line based on its access pattern by at most once and M-NUCA-4 indicates that a shared line can change its location in L2 cache by responding to processor requests by up to 4 times.

Both our baseline M-NUCA policies and access-triggered migrations for the hot shared cache lines target at choosing a suitable location for each cache line for reducing access latencies. Therefore, the first set of results we are interested in are on the average L2 hit latencies, and are given in Figure 3.28. We observe a significant reduction in L2 cache hit latencies when moving from CMP-DNUCA to M-NUCA-0. This improvement is achieved by the placement policy adopted by M-NUCA, in which a cache line is initially placed around its first requestor. This favors the private cache lines from the access latency perspective. For a shared cache line, there is still around 50% probability that its first requestor is a dominating requestor (see Figure 3.18). Thus, this initial position is also suitable for many shared cache lines as well. When we move from M-NUCA-0 to M-NUCA-1, we observe further reduction on the average L2 hit latency for most benchmarks, which is brought by the access-triggered migrations of the hot shared cache lines. In two benchmarks, "equake" and "galgel", however, we observe a slight increase in average L2 hit latency under M-NUCA-1, compared to M-NUCA-0. We believe that the

reason for this behavior is the potential contention at the migration target when access-triggered migrations are used. When a shared cache line reaches the "Hot" threshold, the computation of migration target is triggered and this line is then sent to the migration target cluster. Since there is no handshaking between the local cluster and the target cluster, it may happen that all the victim candidates at the migration target are accessed frequently by a nearby processor. When this happens, one of these candidates is evicted to a farther location in the L2 space, which may result in an access latency increase if the benefits brought by shared cache line migration are not large enough. We also observe that there is no specific trend when we change the migration limit for hot shared cache lines. This indicates in our opinion that the access pattern of a shared cache line does not vary significantly along with time. Therefore, setting the maximum allowable number for the access-triggered migrations to 1 or 2 is sufficient for our parallel benchmarks.

The average L2 hit latency itself cannot determine the performance of an L2 NUCA design, as, from the L2 point of view, the average access latency of L2 cache is:

$$AverageL2HitLatency + L2MissRate \times MemoryAccessLatency.$$

M-NUCA design modifies the placement and replacement policies; thus, we also need to study its effect on L2 miss rate. The L2 miss rate results are presented in Figure 3.29. We see that M-NUCA-0 brings consistent and impressive improvements in L2 miss rates across all the benchmarks. This improvement is achieved through our cache line placement and replacement policies. Specifically, at each L2 miss, we do not randomly choose a victim and directly kick it out from on-chip L2 space. Instead, we choose a victim from the requestor's local L2 space. If the requestor has a non-zero priority (not the lowest priority), we try to keep this victim on chip

by asking other available clusters to hold it. With respect to allowing only eviction-triggered migrations in M-NUCA-0, adding the access-triggered migrations for the hot shared cache lines does not increase the L2 miss rate significantly. This is because our access-triggered migrations actually swap two cache lines in question; there is no cache line evicted from the on-chip cache space. Also, since we know that the percentage of shared cache lines is small, the capacity needed for maintaining the access patterns (see Figure 3.27) is limited, having no significant impact on L2 miss rate. In some benchmarks, e.g., "applu", "equake", "galgel", and "wupwise", we even observe a slight reduction in the L2 miss rates when moving from M-NUCA-0 to M-NUCA-1. We think that this is due to the interference between the eviction-triggered migration and the access-triggered migration, as the latter changes the local victim candidates for placement/replacement. To sum up, we see from Figure 3.29 that our scheme does not affect the L2 miss rate negatively (none of our schemes exhibits a higher L2 miss rate than CMP-DNUCA). As a result, in Figure 3.30 (which gives the average L2 access latency values including both hits and misses), we see that our M-NUCA scheme consistently yields a better overall L2 performance. M-NUCA-0, M-NUCA-1, M-NUCA-2, M-NUCA-4, and M-NUCA-8 generate the reduction in the L2 access latency of 25.8%, 27.7%, 28.5%, 29.3%, and 24.1%, respectively, when averaging over all benchmarks.

For a migration based CMP NUCA architecture, both the migration frequency and the migration distance are important metrics, as they directly affect the power consumption (while some of the migration latency can be hidden by parallel execution, power consumed due to migrations cannot be). Figure 3.31 presents the normalized migration counts (the base case is CMP-DNUCA) with the different schemes. We can observe that the number of migrations is reduced considerably with respect to CMP-DNUCA (M-NUCA-1 has only 22.9% migrations on

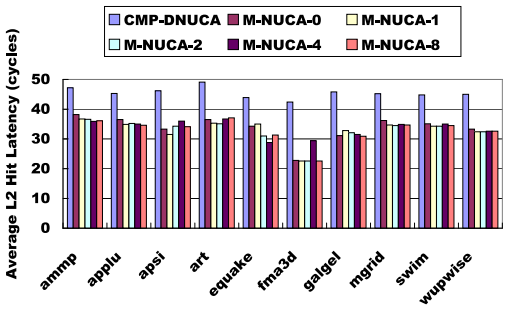


Fig. 3.28. Average L2 hit latency.

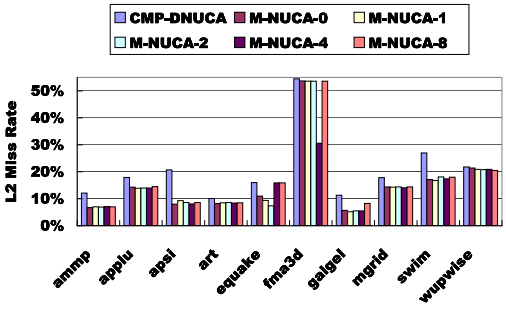


Fig. 3.29. L2 cache miss rate under the different schemes.

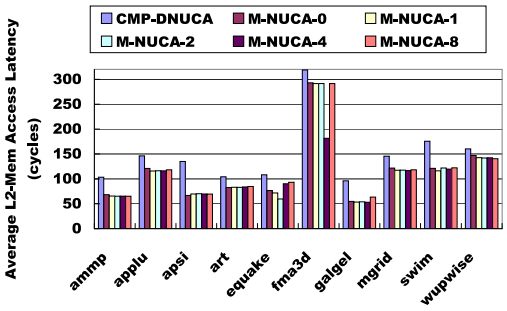


Fig. 3.30. Average L2 access latency (including both hits and misses).

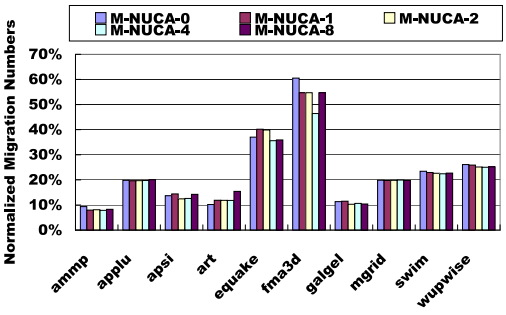


Fig. 3.31. Number of migrations performed (normalized with respect to the number of migrations under the CMP-DNUCA scheme).

average compared to CMP-DNUCA). The reason for this is that we only migrate cache lines at L2 misses (placement/replacement) and at times when a cache line becomes actively shared by multiple requestors. In the latter case, we set a migration limit to prevent the future migrations. Recall from our first set of results on the L2 performance (Figure 3.30) that setting the migration limit to 1 or 2 is sufficient for most of the cases. Another interesting metric to study is the average number of hops traversed per migration, since our migration policy can move a local cache line to any cluster within the L2 space, which might be far away. Figure 3.32 gives the average number of hops traversed per migration, which is less than 1.5 hops for every benchmark (note that CMP-DNUCA always migrates between neighbors, so the value of this metric under CMP-DNUCA is always 1). Overall, we see that our schemes do not incur excessive global movements of cache lines across the chip, and thus, they are not expected to cause significant power consumption.

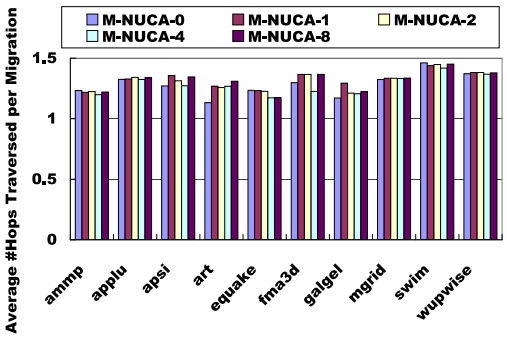


Fig. 3.32. Average number of hops a migration traverses.

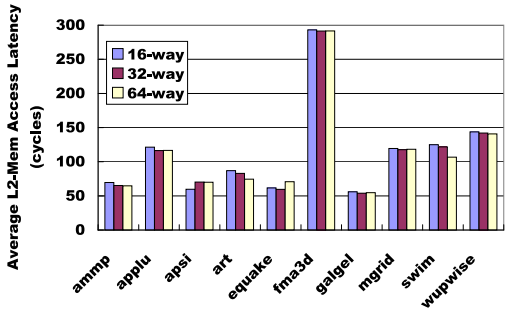


Fig. 3.33. Average L2 access latency under different cache associativities (migration limit is 2).

In addition to the results above, we also conducted several sensitivity studies to evaluate our proposal under varying configuration parameters. The first parameter we study is the associativity of the L2 NUCA. Since our scheme can perform local replacements, it is possible to design a globally high set-associative NUCA, which is in essence composed of multiple locally low set-associative clusters. To study the impact of associativity, we changed its value from the default globally 32-way set-associative to globally 16-way and 64-way. The results are shown in Figure 3.33. Note that, in order to focus on the effects of the different organizations, we assume the same local cluster access latency, i.e., 22 cycles, for all the three configurations.[3] We observe that, although no specific associativity works best for all the benchmarks, many benchmarks exhibit improvements on average L2 access latency when associativity is increased. This is because a higher associativity provides more flexibility for cache line allocation and migration in our scheme, and this helps to reduce the L2 latency. Another reason is that high associative caches usually have lower miss rates due to reduced conflict misses. On average, we achieved 27.2% and 29.1% reduction in the average L2 access latency over CMP-DNUCA, with 16-way and 64-way set associative caches, respectively.

As stated in Section 3.3.2, our scheme employs the concept of priority, trying to differentiate the memory-intensive thread phases with other concurrent thread phases. The L2 miss intensity of each processor is used to classify its cluster priority. Our default threshold for setting the priority of a processor-cache cluster to the highest value (i.e., 2) was 350 L2 misses per million cycles. We also conducted experiments with the threshold values of 35 and 3500 L2 misses per million cycles. Figure 3.34 gives the average L2 access latencies under the different thresholds. We observe that, the performance of M-NUCA does not change much really with

[3] From a circuit angle, changing the associativity may lead to a different access latency.

the different priority setting thresholds. This is because our benchmarks are multi-threaded parallel applications. The threads have similar behavior and do not compete much for the shared L2 cache. For heterogeneous applications, in which threads exhibit different memory requirements, changing the priority setting thresholds or directly setting the priority levels would be potential options for adjusting cache sharing among those threads. It will be studied in as a part of our future work. Another metric of interest in this work is the threshold for triggering a shared cache line, as it may affect the access pattern prediction accuracy. As illustrated in Figure 3.35, increasing this trigger threshold has a minor impact on the overall performance for most benchmarks.

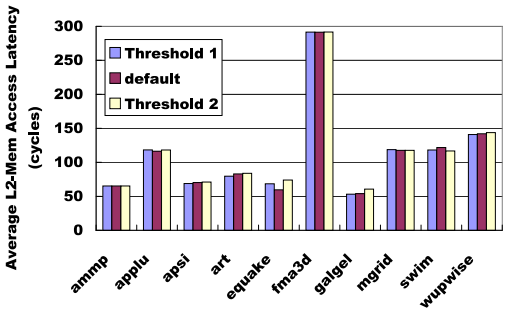


Fig. 3.34. Average L2 access latency under different priority setting thresholds (migration limit: 2; default: 350 L2 misses per million cycles; Threshold 1: 35 L2 misses per million cycles; Threshold 2: 3500 L2 misses per million cycles).

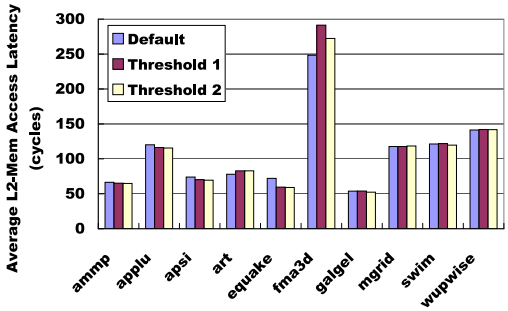


Fig. 3.35. The average L2 access latency with different thresholds for triggering the migration of shared cache line migration (migration limit: 2; default: 40 accesses; Threshold 1: 400 accesses; Threshold 2: 4000 accesses).

3.4 Summary

Design a high performance on-chip memory hierarchy for CMPs is an open challenge. This chapter presented two proposals aiming to achieve a high performance shared NUCA for CMPs. The first proposal was a 3D NUCA design based on a novel hybrid bus/NoC fabric. The hybrid interconnect fabric efficiently exploited the fast vertical interconnects in 3D circuits. We further discussed processor placement and L2 data management issues, and presented an extensive experimental evaluation of the proposed architecture as well as its comparison to 2D L2. Our experiments showed that the proposed 3D architecture reduced the average L2 access latency significantly over 2D topologies and this in turn brought IPC benefits. In addition, our results showed that moving from a 2D topology to a 3D topology can provide more latency reductions than incorporating sophisticated data migration strategies into the 2D topology. The results indicated the importance of considering 3D technology in designing future chip multiprocessors.

The second proposal was motivated by the sharing pattern of multi-threading applications. Our statistics showed that most of cache lines in a shared L2 cache are accessed by only one processor, but a large percentage of L2 accesses are to the small set of shared cache lines. We proposed two types of migrations, eviction-triggered migration and access-triggered migration, for private and shared cache lines, respectively. The goal was to find a proper physical location for each cache line such that the average L2 access latency was minimized. The experimental results showed that the migration schemes separated the data sets for different processors successfully and generated up to 29.3% L2 latency improvement.

Chapter 4

Concluding Remarks and Future Work

Chip Multiprocessors (CMPs), integrating multiple cores on a single die, become the trend of microprocessor design, as witnessed by both industrial and academic worlds. Designing relatively simple cores for CMPs indicates higher clock rates, lower per core design and verification costs, and better scalability, compared to designing a single monolithic, sophisticated processor. However, CMPs face other challenges. First, the multiple cores on chip need to communicate with each other and require an efficient on-chip interconnection network. Second, having multiple cores on a sing chip exacerbates the demand for memory bandwidth. Designers must carefully design the memory subsystem for CMPs in order to achieve the potential computing capability of CMPs. Motivated by these observations, this thesis focused on the two critical components of CMPs: interconnection and memory subsystem. Specifically, we explored the optimizations for Network-on-Chip, an emerging choice of interconnection for CMPs, and the optimizations for the shared L2 Non-Uniform Cache Architecture (NUCA).

The first part of this thesis aimed at reducing the energy consumption of Network-on-Chip. We proposed and evaluated three compiler-directed schemes:

- Compiler-directed proactive communication link turn-on/off
- Compiler-directed voltage level selection for communication links
- Profile-driven message re-routing

These schemes targeted array/loop-intensive embedded applications. A common feature of these schemes was that they all modified the parallel code to achieve NoC energy savings. The first approach, based on static program analysis, inserted communication link turn-on or turn-off commands into parallel codes. The generated parallel code was able to proactively set the power states of communication links. Thus, this scheme was able to immediately shut down communication links that were not needed any more and to pre-activate communication links before they were required. This approach achieved an average of 18.3% link energy saving over a pure hardware-based link on/off approach, while with negligible performance penalties. The second approach proposed a novel graph-based representation of the parallel program, Inter-Process Communication Graph (IPCG). An IPCG included both computations and communications within the parallel code. We designed algorithms to analyze the critical paths in a IPCG and to determine a appropriate frequency/voltage level for each communication link. The code modification module inserted voltage control functions before the entry of each loop. The experimental results showed that this scheme was much more effective than a pure hardware-based link voltage scaling scheme and came very close to an optimal voltage scaling scheme. In the third approach, we proposed the concept of Communication Graph (CG) to capture the communication behavior of a parallel program based on profiling information. We developed an algorithm for determining the routing path of each message-sending operation in an attempt to maximize the link reuse between neighboring network states of a CG. With this algorithm, the link usage was restricted to a small set of links at a given time and the remaining links were turned off to save power. The experimental results showed that this approach enhanced the effectiveness of a

hardware-based link turn-on/off scheme significantly (on average about 35% more energy savings). The success of these three compiler-directed approaches demonstrated that the compiler can play an important role in reducing the NoC energy consumption.

The second part of this thesis focused on the shared level-2 NUCA for CMPs. We built and simulated the following two NUCAs:

- A 3D NoC-bus hybrid L2 NUCA design
- A migration-based L2 NUCA design

The first NUCA design adapted the shared L2 NUCA design for 2D topologies to 3D architectures. Specifically, a novel hybrid bus/NoC fabric was proposed to efficiently employ the fast vertical interconnects in 3D circuits. We studied the placement of processors and cache banks in the 3D scenario. We conducted extensive experiments and demonstrated that a 3D L2 memory architecture generated much better results than the conventional 2D design under different numbers of layers and vertical (inter-wafer) connections. In particular, a 3D architecture without data migration gave better performance than a 2D architecture with dynamic data migration. The second NUCA design was motivated by the sharing pattern of multi-threading applications, that is, private cache lines dominated the L2 cache space while accesses to shared cache lines dominated the accesses to L2 cache. We presented a migration-based NUCA design, including both a careful initial placement policy (eviction-triggered migration) and sharing-aware data location determination (access-triggered migration). The two migration schemes determined a proper physical location for each cache line in L2 cache such that the average L2 access latency was reduced. We demonstrated, through experiments, that the migration-based NUCA design generated up to 29.3% better results than the previous NUCA design with gradual migrations.

4.1 Future Work

This thesis work has initiated several new ideas regarding the design and optimization of NoC-based Chip Multiprocessors. Specifically, we believe that the three problems listed below are very interesting and may worth paying attention to.

First, we can extend the compiler-directed NoC voltage scaling scheme to handle both processor cores and NoCs if processors are voltage scalable as well. According to [60], the energy reduction rates of both processors and communication links show a slowing down trend when the voltage level continues to scale down. That means, instead of aggressively scaling down the voltage of processors only or NoCs only, we prefer coordinated voltage scaling of processors and NoCs in order to achieve maximum energy reduction. The coordinated voltage scaling involves properly allocating the slack time to processors and communication links such that the energy reduction is maximized without significant performance degradation. To accurately estimate the slack time, we can employ static program analysis and/or program profiling to obtain the execution time (both computation and communication) of a parallel program.

Second, reliability is now an important design issue for NoCs, just as for other components of a computer system. This is due to the continuously scaling-down feature size and the coming along vulnerability to different types of noises [83, 20]. Normally, reliability is not the only design constraint for a system. It needs to be considered together with performance and power consumption. In the scenario of NoCs, researchers found that combining message retransmission and an error detection scheme is an energy-efficient way to design reliable NoCs [11]. This interesting observation motivates us to think about several questions: what can we do if a message is lost and does not arrive the destination? which routing path should the retransmitted

message go through? For example, on the one hand, if a duplicated message chooses the same routing path as that of the original message, the remaining communication links are more likely to remain idle and to be powered off. However, if a failed link exists along the routing path, this scheme cannot deliver this message correctly. Thus, we see that this scheme is favored from the perspective of power consumption, but not from the reliability angle. On the other hand, if a duplicate message selects an entirely non-overlapped routing path with that of the original message, it decreases the opportunities of link shutdown, but strengthens the NoC error resilience. Besides considering the reliability issue of NoCs only, NoC-based CMPs have the redundancy for providing the reliability of the entire architecture. For example, classifying the critical portion of a program, dynamically detecting the underutilized processors, and scheduling replicated threads to these underutilized processors might be a promising way to improve the reliability of an NoC-based CMP.

Third, regarding the on-chip storage, the latency is still a concern due to the increasing on-chip memory capacity and the long wire delay. Besides the data placement, replacement and migration policies proposed in this thesis, importing the idea of prefetch into on-chip data management may be a promising method to reduce the access latency. Since there is spatial locality in memory accesses, we can predict cache lines that are needed in the near future and proactively migrate these cache lines toward the requesting processors. In this way, we expect most of the data accesses to be satisfied by the local on-chip memories. In addition, compared to optimizing the access latency, less attention has been paid to the energy consumption of on-chip memory design for CMPs. It would be interesting to evaluate the energy behavior of different on-chip cache organizations and management policies.

References

[1] Virtutech Simics. http://www.virtutech.com/products/.

[2] N. AbouGhazaleh, B. Childers, D. Mosse, R. Melhem, and M. Craven. Energy management for real-time embedded applications with compiler support. In *Proc. the ACM SIGPLAN conference on Language, Compiler, and Tool for Embedded Systems*, 2003.

[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 *Proc. the 27th International Symposium on Computer Architecture*, June 2000.

[4] G. Ascia, V. Catania, and M. Palesi. Multi-objective mapping for mesh-based NoC architectures. In *Proc. the International Conference on Hardware/Software Codesign and System Synthesis*, 2004.

[5] T. M. Austin. The SimpleScalar/ARM toolset. http://www.eecs.umich.edu/ taustin/simplescalar.

[6] E. Ayguade, J. Garcia, M. Girones, M. Luz Grande, and J. Labarta. A research tool for automatic data distribution in HPF. *Scientific Programming*, 6(1):73–95, 1997.

[7] B. M. Beckmann, M. R. Marty, and D. A. Wood. ASR: Adaptive selective replication for CMP caches. In *Proc. the International Symposium on Microarchitecture*, 2006.

[8] B. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *Proc. the International Symposium on Microarchitecture*, 2004.

[9] L. Benini and G. Micheli. Networks on chips: a new SOC paradigm. *IEEE computer*, 35(1), January 2002.

[10] L. Benini and G. D. Micheli. Powering networks on chips: energy-efficient and reliable interconnect design for SoCs. In *Proc. the 14th International Symposium on Systems Synthesis*, 2001.

[11] D. Bertozzi, L. Benini, and G. D. Micheli. Low power error resilient encoding for on-chip data buses. In *Proc. the conference on Design, Automation and Test in Europe*.

[12] B. Black et al. 3D Processing technology and Its Impact on IA32 Microprocessors. In *Proc. the International Conference on Computer Design*, 2004.

[13] Broadcom. BCM1480 product brief. http://www.broadcom.com/.

[14] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proc. the International Symposium on Computer Architecture*, 2000.

[15] F. Catthoor, K. Danckaert, K. K. Kulkarni, E. Brockmeyer, T. van Achteren P. G. Kjeldsberg, and T. Omnes. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Academic Publisher, 2002.

[16] L. N. Chakrapani, P. Korkmaz, III V. J. Mooney, K. V. Palem, K. Puttaswamy, and W. F. Wong. The emerging power crisis in embedded processors: what can a poor compiler do? In *Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2001.

[17] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *Proc. the International Symposium on Computer Architecture*, 2006.

[18] G. Chen, F. Li, and M. Kandemir. Compiler-directed channel allocation for saving power in on-chip networks. In *Proc. Symposium on Principles of Programming Languages*, 2006.

[19] G. Chen, F. Li, M. Kandemir, and M. J. Irwin. Reducing NoC energy consumption through compiler-directed channel voltage scaling. In *Proc. Conference on Programming Language Design and Implementation*, 2006.

[20] H. H. Chen and D. D. Ling. Power supply noise analysis methodology for deep-submicron VLSI chip design. In *Proc. the 34th Conference on Design Automation*, 1997.

[21] X. Chen and L-S. Peh. Leakage power modeling and optimization in interconnection networks. In *Proc. the International Symposium on Low Power and Electronics Design*, 2003.

[22] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimization replication, communication, and capacity allocation in CMPs. In *Proc. the International Symposium on Computer Architectures*, 2005.

[23] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-Level page allocation. In *Proc. the Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[24] J. Cong and Y. Zhang. Thermal-Driven Multilevel Routing for 3-D ICs. In *Proc. the Asia South Pacific Design Automation Conference*, January 2005.

[25] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[26] W. J. Dally and B. Towles. Route packets, not wires: on-chip inteconnectoin networks. In *Proc. the 38th Conf. on Design Automation*, 2001.

[27] S. Das et al. Technology, Performance, and Computer Aided Design of Three-Dimensional Integrated Circuits. In *Proc. International Symposium on Physical Design*, 2004.

[28] W. R. Davis et al. Demystifying 3D ICs: The Pros and Cons of going vertical. *IEEE Design and Test of Computers*, 22(6), November 2005.

[29] Y. Deng et al. 2.5D System Integration: A Design Driven System Implementation Schema. In *Proc. the Asia South Pacific Design Automation Conference*, 2004.

[30] J. B. Duato, S. Yalamanchili, and L. Ni. Interconnection networks. *Morgan Kaufmann (pubs)*, 2002.

[31] N. Eisley and L-S. Peh. High-level power analysis of on-chip networks. In *Proc. the 7th Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, September 2004.

[32] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):179–193, 1992.

[33] L. Hammond, B. Nayfeh, and K. Olukotun. A Single-Chip Multiprocessor. *IEEE Computer Special Issue on "Billion-Transistor Processors"*, September 1997.

[34] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8), August 1992.

[35] R. Ho, K.W. Mai, and M.A. Horowitz. The Future of Wires. *Proc. the IEEE*, 89(4), April 2001.

[36] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing memory operations: providing memory performance feedback in modern processors. In *Proc. the International Symposium on Computer Architecture*, 1996.

[37] J. Hu and R. Marculescu. Energy- and performance-aware mapping for regular NoC architectures. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 24(4), April 2005.

[38] J. Huh, D. Burger, and S. W. Keckler. Exploring the design space of future CMPs. In *Proc. the 10th International Conference on Parallel Architecture and Compilation Techniques*, 2001.

[39] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *Proc. the International Conference on Supercomputing*, 2005.

[40] http://www.intel.com/idf/.

[41] M. Ieong et al. Three Dimensional CMOS Devices and Integrated Circuits. In *Proc. IEEE Custom Integrated Circuits Conference*, 2003.

[42] http://www.intel.com/products/processor/coreduo/.

[43] R. Iyer. A framework for enabling QoS in shared caches of CMP platforms. In *Proc. the Annual International Conference on Supercomputing*, 2004.

[44] J.W. Joyner, P. Zarkesh-Ha, and J.D. Meindl. A stochastic global net-length distribution for a three-dimensional system-on-a-chip (3D-SoC). In *Proc. 14th Annual IEEE International ASIC/SOC Conference*, September 2001.

[45] S.M. Jung et al. The Revolutionary and Truly 3-Dimentional 25F2 SRAM Technology with the Smallest S3 Cell, 0.16um2 and SSTFF for Ultra High Density SRAM. In *VLSI Technology Digest of Technical Papers*. 2004.

[46] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4-5), 2005.

[47] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Proc. of the 37th conference on Design automation*, 2000.

[48] C. Kim, D. Burger, and S.W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *Proc. the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[49] E. J. Kim, K. H. Yum, G. Link, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, M. Yousif, and C. R. Das. Energy optimization techniques in cluster interconnects. In *Proc. the Int. Symp. on Low Power Electronics and Design*, August 2003.

[50] J. Kim and M. Horowitz. Adaptive supply serial links with sub-1V operation and per-pin clock recovery. In *Proc. Int. Solid-State Circuits Conference*, February 2002.

[51] J. Kim, D. Park, C. Nicopoulos, N. Vijaykrishnan, and C. Das. Design and analysis of an NoC architecture from performance, reliability and energy perspective. In *Proc. the Symposium on Architecture for Networking and Communications Systems*, October 2005.

[52] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proc. the International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[53] I. Kodukula, N. Ahmed, and K Pingali. Data-centric multi-level blocking. *SIGPLAN Not.*, 32(5):346–357, 1997.

[54] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE MICRO Magazine*, April 2005.

[55] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a RAW machine. In *Proc. the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[56] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *Proc. the 35th International Symposium on Microarchitecture*, November 2002.

[57] F. Li, G. Chen, M. Kandemir, and M. J. Irwin. Compiler-directed proactive power management for networks. In *Proc. Conference on Compilers, Architectures and Synthesis of Embedded Systems*, 2005.

[58] F. Li, G. Chen, M. Kandemir, and I. Kolcu. Profile-driven energy reduction in network-on-chips. In *Proc. Conference on Programming Language Design and Implementation*, June 2007.

[59] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. J. Irwin. Enhancing L2 organization for CMPs with a center cell. In *Proc. the International Parallel and Distributed Processing Symposium*, 2006.

[60] J. Luo, L.-S. Peh, and N. Jha. Simultaneous dynamic voltage scaling of processors and communication links in real-time distributed embedded systems. In *Proc. DATE*, 2003.

[61] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, and G. Hallberg. Simics: A full system simulation platform. *IEEE Computer*, 35(2), February 2002.

[62] G. Memik and W. H. Mangione-Smith. Increasing power efficiency of multi-core network processors through data filtering. In *Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2002.

[63] P. Morrow, M.J. Kobrinsky, S. Ramanathan, C-M. Park, M. Harmes, V. Ramachandrarao, H. Park, G. Kloster, S. List, and S. Kim. Wafer-Level 3D Interconnects Via Cu Bonding. In *Proc. the 21st Advanced Metallization Conference*, October 2004.

[64] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proc. the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[65] R. Mullins, A. West, and S. Moore. Low-latency virtual-channel routers for on-chip networks. In *Proc. the 31st Annual International Symposium on Computer Architecture*, June 2004.

[66] R. Nagarajan and et al. Static placement, dynamic issue (SPDI) scheduling for EDGE architectures. In *Proc. International Conference on Architectures and Compilation Techniques*, October 2004.

[67] http://en.wikipedia.org/wiki/Network_On_Chip.

[68] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, 1993.

[69] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K-Y. Chang. The Case for a Single-Chip Multiprocessor. In *Proc. the 7th International Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[70] C. S. Patel. Power constrained design of multiprocessor interconnection networks. In *Proc. the International Conference on Computer Design*, Washington, DC, USA, 1997.

[71] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[72] L-S. Peh and W.J. Dally. A delay model and speculative architecture for pipelined routers. In *The Seventh International Symposium on High-Performance Computer Architecture*, January 2001.

[73] P. Petrov and A. Orailoglu. Compiler-based register name adjustment for low-power embedded processors. In *Proc. the IEEE/ACM International Conference on Computer-aided design*, 2003.

[74] K. Puttaswamy and G.H. Loh. Implementing Caches in a 3D Technology for High Performance Processors. In *Proc. the International Conference on Computer Design*, October 2005.

[75] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *Proc. the International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[76] V. Raghunathan, M. B. Srivastava, and R. K. Gupta. A survey of techniques for energy efficient on-chip communication. In *Proc. the 40th Conference on Design Automation*, 2003.

[77] T. Richardson, C. Nicopoulos, D. Park, V. Narayanan, Y. Xie, C. Das, and V. Degalahal. A Hybrid SoC Interconnect with Dynamic TDMA-Based Transaction-Less Buses and On-Chip Networks. In *Proc. VLSI Design*, 2006.

[78] P. Rickert. Problems or opportunities? Beyond the 90nm frontier. ICCAD Keynote Address, 2004.

[79] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proc. the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1989.

[80] G. Roth, J. Mellor-Crummey, K. Kennedy, and R. G. Brickner. Compiling stencils in high performance fortran. In *Proc. the ACM/IEEE Conference on Supercomputing*, 1997.

[81] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proc. the 30th Annual Int. Symp. on Computer Architecture*, 2003.

[82] L. Shang, L-S. Peh, and N. K. Jha. Dynamic voltage scaling with links for power optimization of interconnection networks. In *Proc. the International Symposium on High-Performance Computer Architecture*, February 2003.

[83] K. L. Shepard and V. Narayanan. Noise in deep submicron digital design. In *Proc. the IEEE/ACM Int. Conf. on Computer-Aided Design*, 1996.

[84] D. Shin and J. Kim. Power-aware communication optimization for networks-on-chips with voltage scalable links. In *Proc. the International Conference on Hardware/Software Codesign and System Synthesis*, September 2004.

[85] P. Shivakumar and N.P. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. Technical report, Compaq Computer Corporation, August 2001.

[86] T. Simunic and S. Boyd. Managing power consumption in networks on chip. In *Proc. the conference on Design, automation and test in Europe*. IEEE Computer Society, 2002.

[87] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM J. Res. Dev.*, 49(4/5):505–521, 2005.

[88] K. So and R. N. Rechtschaffen. Cache operations by MRU change. *IEEE Trans. Comput.*, 37(6):700–709, 1988.

[89] V. Soteriou, N. Eisley, and L.-S. Peh. Software-directed power-aware interconnection networks. In *Proc. Conference on Compilers, Architecture and Synthesis for Embedded Systems*, September 2005.

[90] V. Soteriou and L.-S. Peh. Design space exploration of power-aware on/off interconnection networks. In *Proc. the 22nd International Conference on Computer Design*, October 2004.

[91] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. Adaptive mechanisms and policies for managing cache hierarchies in chip multiprocessors. In *Proc. the Annual International Symposium on Computer Architecture*, 2005.

[92] Standard Performance Evaluation Corporation. SPEC OMP. http://www.spec.org/hpg/omp2001/, December 2005.

[93] SUIF. Stanford university intermediate format. http://suif.stanford.edu/.

[94] M. B. Taylor and et al. The RAW microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2), 2002.

[95] http://www-unix.mcs.anl.gov/mpi/.

[96] http://www.ece.northwestern.edu/cpdc/Paradigm/Paradigm.html.

[97] Y-F. Tsai, Y. Xie, N. Vijaykrishnan, and M.J. Irwin. Three-Dimensional Cache Design Exploration Using 3D Cacti. In *Proc. the International Conference on Computer Design*, October 2005.

[98] C.-W. Tseng. *An optimizing Fortran D compiler for MIMD distributed-memory machines*. PhD thesis, Dept. of Computer Science, Rice University, January 1993.

[99] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 533–544, 1998.

[100] H.-S. Wang, L.-S. Peh, and S. Malik. Power-driven design of router microarchitectures in on-chip networks. In *Proc. the 36th International Symposium on Microarchitecture*, San Diego, USA, 2003.

[101] H-S. Wang, X. Zhu, L-S. Peh, and S. Malik. Orion: A power-performance simulator for interconnection networks. In *Proc. the 35th Int. Symp. on Microarchitecture*, November 2002.

[102] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proc. International Symposium on Microarchitecture*, 1996.

[103] M. Wolfe. Parallelizing compilers. *ACM Computer Survey*, 28(1), 1996.

[104] T. Y. Yeh and G. Reinman. Fast and fair: data-stream quality of service. In *Proc. the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2005.

[105] A. Zeng, J. Lu, K. Rose, and R.J. Gutmann. First-Order Performance Prediction of Cache Memory with Wafer-Level3D Integration. *IEEE Design and Test of Computers*, 22(6), June 2005.

[106] A. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Proc. the 32nd International Symposium on Computer Architecture*, 2005.

Vita

Feihui Li is a Ph.D. candidate in the Department of Computer Science and Engineering, the Pennsylvania State University, University Park. She received her B.S. degree in 1999, and M.S. degrees in 2002, both from the Department of Electrical Engineering, Tsinghua University, Beijing, China. In 2002, she enrolled in the Ph.D. program in the Department of Computer Science and Engineering at the Pennsylvania State University. She is a student member of IEEE. She co-authored so far 18 conference papers. She served as a technical referee for numerous journals and conferences including IEEE Transactions on Computer, Int'l Symposium on High-Performance Computer Architecture, and Int'l Conference on Parallel and Distributed Processing Symposium and etc.